Network of Excellence on
Engineering Secure Future Internet
Software Services and Systems

Network of Excellence

Deliverable D8.2

# Initial Solutions for Secure Programming Environments and Composable Services

| | | |
|---|---|---|
| **Project Number** | : | 256980 |
| **Project Title** | : | NESSoS |
| **Deliverable Type** | : | Report |

| | | |
|---|---|---|
| **Deliverable Number** | : | D8.2 |
| **Title of Deliverable** | : | Initial Solutions for Secure Programming Environments and Composable Services |
| **Nature of Deliverable** | : | R |
| **Dissemination Level** | : | Public |
| **Internal Version Number** | : | |
| **Contractual Delivery Date** | : | September 30, 2011 |
| **Actual Delivery Date** | : | |
| **Contributing WPs** | : | WP 8 |
| **Editor(s)** | : | Wouter Joosen |
| **Author(s)** | : | Gilles Barthe, Marianne Busch, Manuel Clavel, Gabriele Costa, Jorge Cuellar, Philippe De Ryck, Lieven Desmet, Tom Goovaerts, Valerie Issarny, Bart Jacobs, Wouter Joosen, Boris Köpf, Fabio Martinelli, Wannes Meert, Nick Nikiforakis, Pieter Philippaerts, Frank Piessens, Jan Smans, Steven Van Acker, Dimitri Van Landuyt, Yves Younan |
| **Reviewer(s)** | : | Nikolaos Georgantas (INRIA), Artsiom Yautsiukhin (CNR) |

# Abstract

*The main objective of this work package is to extend and create improved programming platforms for web-based service-oriented applications of the Future Internet. This theme includes enhanced development and run-time environments. The first goal is to support secure service composition in service creation languages. The second goal is to develop platform support for security enforcement, both in language execution environments as well as in middleware. The third goal is to enhance the mainstream languages that are used in the core of service platforms. The main focus of this third element is to enhance analysis and verification capabilities. These results will thus underpin new assurance methods that will be developed in WP9.*



# Keyword List

*security,web services,run-time enforcement,component integration,secure service composition,program verification,information flow*

# Document History

| Version | Type of Change | Author(s) |
|---|---|---|
| 0.1 | initial setup | KUL |
| 0.2 | added contents to 'Secure Service Composition and Secure Service Platforms' | CNR |
| 0.3 | added contents to 'Security Enforcement in the Web Services Run-time' | KUL |
| 0.4 | added contents to 'Information Flow for Secure Services' | IMDEA |
| 0.5 | Added future work (secure navigation paths) | LMU & SIEMENS |
| 0.6 | added contents to 'Enhanced programming language support' | KUL |
| 0.7 | added contents to 'Secure Service Composition and Secure Service Platforms' | INRIA |
| 0.8 | Finalization of deliverable | KUL |

# Document Review

| Date | Version | Reviewer | Comment |
|---|---|---|---|
| | | | *(For the editor: please, list a resume of the most relevant reviewers' comments)* |

# Table of Contents

# 1 Introduction

Security support in programming environments is not new; still it remains a grand challenge, especially in the context of Future Internet (FI) services. Securing future internet services is inherently a matter of secure software and systems. The context of the future internet services sets the scene in the sense that (1) specific service architectures will be used, that (2) new types of environments will be exploited, ranging from small embedded devices ("things") to service infrastructures and platforms in the cloud, and that (3) a broad range of programming technologies will be used to develop the actual software and systems.

The search for and development of security support in programming environments has to take this context into account. The requirements and architectural blueprints that will be produced in earlier stages of the software engineering process cannot deliver the expected security value unless the programs (code) respect these security artifacts that have been produced in the preceding stages. Supporting security requirements in the programming - code - level requires a comprehensive approach. Three essential facets must be covered: service creation means must be improved and extended to deal with security needs. Service creation means both aggregating and (1) composing services from pre-existing building blocks (services and more traditional components), as well as (2) programming new services from scratch using a state-of-the-art programming language. The service creation context will typically aim for techniques and technologies that support compile-time and build-time feedback. One could argue that security support for service creation must focus on and enable better static verification. Then (3) the service execution support must be enhanced to deal with hooks and building blocks that facilitate effective security enforcement at run-time. Dependent on the needs and the state-of-the-art this may lead to interception and enforcement techniques that "simply" ensure that the application logic consistently interacts with underpinning security mechanisms such as authentication or audit services. Otherwise, the provisioning of the underpinning security mechanisms and services (e.g. supporting attribute based authorization in a cloud platform etc.) will be required as well for many of the typical FI service environments.

In this report, we build on the state-of-the-art in secure programming environments from the three perspectives listed above: execution support, service composition support and programming support respectively. Rather than being exhaustive on all aspects of this huge domain, we have focused on a set of topics that we believe to be critical for the near and long term research agenda.

This deliverable presents some of the results that have been achieved by the NESSoS partners of this work package during the first year of the project. The report is structured as follows.

Chapter 2 discusses three new contributions to the field of run-time security enforcement. The focus, as planned, has been on web services. A first contribution extends the new sandbox attribute present in HTML5 and improves upon it by making it more fine-grained. The second contribution protects users from cross-site request forgery attacks, and the third contribution protects users from session-stealing attacks. Ideally, many of these solutions would function under the hood, i.e. embedded in the execution environment of a (web) service.

Chapter 3 addresses service composition. We describe work that extends the state-of-the-art in securing web service composition. First, a new mechanism is introduced to securely orchestrate services in open networks. Secondly, new solutions for establishing trust management are summarized. Third, we sketch collaborative work between network partners in search for intelligent configuration of complex security services.

Chapter 4 addresses programming support. The main focus is on the extension of programming environments through annotations, rather than creating new programming languages. This part addresses the extension and improvement of VeriFast, which now becomes part of collaborative work within the NESSoS NoE.

Chapter 5 presents an effective collaboration towards achieving end-2-end properties in secure services. The current focus has been on information flow. The report sketches two new advances in information-flow analysis: dynamic information-flow analysis and quantitative information-flow analysis. Dynamic information-flow analysis aims to make information-flow analysis more permissive by leveraging the information available at run-time, whereas quantitative information-flow analysis enables one to certify the security of programs that leak only negligible amounts of information. Both tracks are promising and this work has triggered new collaborations within the NESSoS NoE.

The collection is based on work that has occurred at CNR, IMDEA, INRIA, KUL, Siemens and UNITN. The contributions in chapter 2 focus on web application security. This work has been conducted by KUL and inspired and triggered by input from Siemens. Also this part covers some collaboration with SAP, one of the partners in the NESSoS IAB. The consortium aims for organizing an open competition to gather and evaluate similar work and results from the broader research community. The contributions in Chapter 3 focus on service composition and service configuration. The work on service composition includes contributions from CNR and INRIA, who will collaborate

on some of these topics in the sequel of this project. The work on service configuration is based on collaboration between KUL and UNITN. Chapter 4 addresses language extensions that can enable formal verification. This work has been delivered by KUL; the related prototypes are part of the NESSoS Workbench. Chapter 5 presents an effective collaboration towards achieving secure information flow. This part sets the scene for collaboration between IMDEA and KUL.

Chapter 6 briefly discusses further interactions between this work package and other work packages in the project. We also highlight the new and upcoming initiatives and future collaborations within this work package. Finally, we conclude in chapter 7.

# 2 Web Application Security

Over the past decade, users have witnessed a functional expansion of the Web, where many applications that used to run on the desktop are now accessible through the browser. With this expansion, websites evolved from simple static HTML pages to dynamic Web applications, i.e. content-rich resources accessible through the Web. Unfortunately, this new strength and expressive power in web applications has significantly increased the potential attack surface of a website. New attacks have already emerged, and it is clear that the current standards are lacking in terms of security.

Our focus on run-time environments has been on web service security, given the current popularity of these platforms and given the need for solutions by many practical, business and societal applications.

This section gives an overview of three new technologies that have been developed to counter threats on services in the future internet and the web. Section 2.1 introduces a novel client-side security architecture to enable least-privilege integration of components into web mash-ups [78]. Section 2.2 proposes a new request filtering algorithm that can prevent CSRF attacks under one specific assumption about the way "good" sites should interact in a cross-origin setting [33]. Finally, section 2.3 presents a lightweight client-side protection mechanism against session hijacking [66]. All these contributions are also partially supported by the FP7-project WebSand.

## 2.1 Least-privilege Integration of Third-party Components in Web Mashups

The Internet has seen an explosion of dynamic websites in the last decade, not in the least because of the power of JavaScript. With JavaScript, web developers gain the ability to execute code on the client-side, providing for a richer and more interactive web experience. The popularity of JavaScript has increased even more since the advent of Web 2.0.

Web mashups are a prime example of Web 2.0. In a web mashup, data and functionality from multiple stakeholders are combined into a new flexible and lightweight client-side application. By doing so, a mashup generates added value, which is one of the most important incentives behind building mashups. Web mashups depend on collaboration and interaction between the different mashup components, but the trustworthiness of the service providers delivering components may strongly vary.

The two most wide-spread techniques to integrate third-party components into a mashup are via script inclusion and via (sandboxed) iframe integration. The script inclusion technique implies that the third-party component executes with the same rights as the integrator, whereas the latter technique restricts the execution of the third-party component according to the Same-Origin Policy. More fine-grained techniques (such as Caja [64] or FBJS [82]) require (some form of) ownership over the code to transform or restrict the component to a known safe subset before delivery to the browser. This makes these techniques less applicable to integrate third-party components directly from their service providers.

To enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components, web mashups should integrate components according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. Unfortunately, least-privilege integration of third-party mashup components can not be achieved with the current script-inclusion and frame-integration techniques. Moreover, the need for least-privilege integration becomes highly relevant, especially because of the augmented JavaScript capabilities of the upcoming HTML5 APIs [86] (such as access to local storage, geolocation, media capture and cross-domain communication).

In this section, we propose WebJail, a novel client-side security architecture to enable the least-privilege integration of third-party components in web mashups. The security restrictions in place are configurable via a high-level composition policy under control of the mashup integrator, and allow the use of legacy mashup components, directly served by multiple service providers. This section is a shortened version of [78]. The full version of this paper is included in the appendix (section A).

### 2.1.1 Problem Statement

This section specifies the attacker model, as well as two typical attack vectors. Next, the increasing impact of insecure mashup composition is discussed in the context of the upcoming set of HTML5 specifications. Finally, the security assessment is concluded by identifying the requirements for secure mashup composition, namely the least-privilege integration of third-party mashup components.

### Attacker model

Our attacker model is inspired by the definition of a *gadget attacker* in Barth *et al.* [9]. The term gadget in their definition should, in the context of this discussion, be read as "third-party mashup component".

We describe the attacker in scope as an attacker that is a malicious principal owning one or more machines on the network. The attacker is able to trick the integrator in embedding a third-party component under his control.

We assume a mashup that consists of multiple third-party components from several service providers, and an honest mashup consumer (i.e. an end-user). A malicious third-party component provider attempts to steal sensitive data outside its trust boundary (e.g. reading from origin-specific client-side storage), impersonate other third-party components or the integrator (e.g. requesting access to geolocation data on behalf of the integrator) or falsely operate on behalf of the end-user towards the integrator or other service providers (e.g. requesting cross-application content with XMLHttpRequest).

We have identified two possible ways in which an attacker could present himself as a malicious third-party component provider: he could offer a malicious third-party component towards mashup integrators (e.g. via a malicious advertisement, or via a malicious clone of a popular component), or he could hack into an existing third-party component of a service provider and abuse the prior existing trust relationship between the integrator and the service provider.

In this discussion, we consider the mashup integrator as trusted by the mashup consumer (i.e. end-user), and an attacker has no control over the integrator, except for the attacker's ability to embed third-party components of his choice. In addition, we assume that the attacker has no special network abilities (such as sniffing the network traffic between client and servers), nor special browser abilities (e.g. extension under control of the attacker or client-side malware) and is constrained in the browser by the Same-Origin Policy.

### Security-sensitive JavaScript operations

The impact of running arbitrary JavaScript code in an insecure mashup composition is equivalent to acquiring XSS capabilities, either in the context of the component's origin, or in the context of the integrator. For instance, a malicious third-party component provider can invoke typical security-sensitive operations such as the retrieval of cookies, navigation of the browser to another page, launch of external requests or access and updates to the Document Object Model (DOM).

However, with the emerging HTML5 specification and APIs, the impact of injecting and executing arbitrary JavaScript has massively increased. Recently, JavaScript APIs have been proposed to access geolocation information and system information (such as CPU load and ambient sensors), to capture audio and video, to store and retrieve data from a client-side datastore and to communicate between windows as well as with remote servers.

As a result, executing arbitrary JavaScript becomes much more attractive to attackers, even if the JavaScript execution is restricted to the origin of the component, or a unique origin in case of a sandbox.

### Least-privilege integration

Taking into account the attack vectors present in current mashup composition, and the increasing impact of such attacks due to newly-added browser features, there is clearly a need to limit the power of third-party mashup components under control of the attacker.

Optimally, mashup components should be integrated according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. This would enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components.

Unfortunately, least-privilege integration of third-party mashup components can not be achieved with the current script-inclusion and iframe-integration techniques. These techniques are too coarse-grained: either no restrictions (or only the Same-Origin Policy) are imposed on the execution of a third party component, implicitly inviting abuse, or JavaScript is fully disabled, preventing any potential abuse but also fully killing desired functionality.

To make sure that attackers described in Section 2.1.1 do not exploit the insecure composition attack vectors and multiply their impact by using the security sensitive HTML5 APIs described in Section 2.1.1, the web platform needs a security architecture that supports least-privilege integration of web components. Since client-side mashups are composed in the browser, this architecture must necessarily be implemented in the browser. It should satisfy the following requirements:

**R1 Full mediation.** The security-sensitive operations need to be fully mediated. The attacker can not circumvent the security mechanisms in place.

**R2 Remote component delivery.** The security mechanism must allow the use of legacy third-party components and the direct delivery of components from the service provider to the browser environment.

**R3 Secure composition policy.** The secure composition policy must be configurable (and manageable) by the mashup integrator. The policy must allow fine-grained control over a single third-party component, with respect to the security-sensitive operations in the HTML5 APIs.

**R4 Performance** The security mechanism should only introduce a minimal performance penalty, unnoticeable to the end-user.

Existing technologies like e.g. Caja [64] and FBJS [82] require pre-processing of mashup components, while ConScript [60] does not work in a mashup context because it depends on the mashup component to load and enforce its own policy.

### 2.1.2 WebJail Architecture

To enable the least-privilege integration of third-party mashup components, we propose WebJail, a novel client-side security architecture. WebJail allows a mashup integrator to apply the least-privilege principle on the individual components of the mashup, by letting the integrator express a secure composition policy and enforce the policy within the browser by building on top of the deep advice approach of ConScript [60].

The secure composition policy defines the set of security-sensitive operations that the component is allowed to invoke. Each particular operation can be allowed, disallowed, or restricted to a self-defined whitelist. Once loaded, the deep aspect layer will ensure that the policy is enforced on every access path to the security-sensitive operations, and that the policy can not be tampered with.

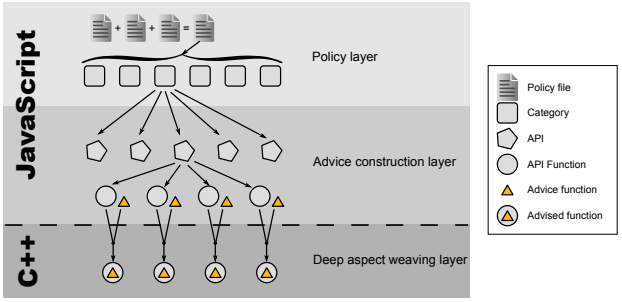



**Figure 2.1: The WebJail architecture consists of three layers: The policy layer, the advice construction layer and the deep aspect weaving layer.**

The WebJail architecture consists of three abstraction layers as shown in Figure 2.1. The upper layer, the *policy layer*, associates the secure composition policy with a mashup component, and triggers the underlying layers to enforce the policy for the given component. The lower layer, the *deep aspect weaving layer*, enables the deep aspect support with the browser's JavaScript engine. The *advice construction layer* in between takes care of mapping the higher-level policy blocks onto the low-level security-sensitive operations via a 2-step policy refinement process.

In this section, the three layers of the WebJail will be described in more detail.

#### Policy layer

The policy layer associates the secure composition policy with the respective mashup component. In this section, an analysis of security-sensitive operations in the HTML5 APIs is reported and discussed, as well as the secure composition policy itself.

**Security-sensitive JavaScript operations** As part of this research, we have analyzed the emerging specifications and browser implementations, and have identified 86 security-sensitive operations, accessible via JavaScript APIs. We have synthesized the newly-added features of these specifications in Figure 2.2, and we will briefly summarize each of the components in the next paragraphs. Most of these features rely on (some form of) user-consent and/or have origin-restrictions in place.

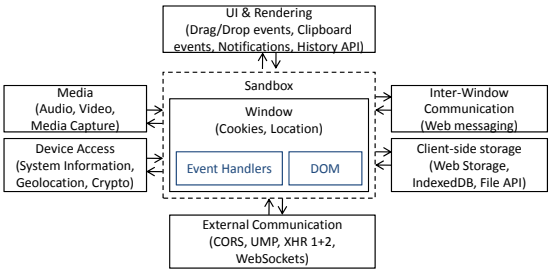



**Figure 2.2: Synthesized model of the emerging HTML5 specifications**

Central in the model is the *window* concept, containing the document. The window manifest itself as a browser window, a tab, a popup or a frame, and provides access to the location and history, event handlers, the document and its associated DOM tree. Event handlers allow to register for a specific event (e.g. being notified of mouse clicks), and access to the DOM enables a script to read or modify the document's structure on the fly. Additionally, a *sandbox* can impose coarse-grained restrictions on an iframe.

*Inter-frame communication* allows sending messages between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (postMessage).

*Client-side storage* enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

*External communication* features such as CORS, UMP, XMLHttpRequest level 1 and 2, and websockets allow an application to communicate with remote websites, even in cross-origin settings.

*Device access* allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information and ambient sensors.

*Media* features enable a web application to play audio and video fragments, as well as capture audio and video via a microphone or webcam.

*The UI and rendering* features allow subscription to clipboard and drag-and-drop events, issuing desktop notifications and populating the history via the History API.

**Secure composition policy** The policy layer associates the secure composition policy with a mashup component and deploys the necessary security controls via the underlying layers. As composition granularity, we have chosen the iframe level, i.e. mashup components are each loaded in their separate iframe.

In particular, within WebJail the secure composition policy is expressed by the mashup integrator, and attached to a particular component via a newly-introduced *policy* attribute of the iframe element of the component to be loaded.

```
<iframe src="http://untrusted.com/compX/"
policy="https://integrator.com/compX.policy"/>
```

We have grouped the identified security-sensitive operations in the HTML5 APIs in nine disjoint categories, based on their functionality: DOM access, Cookies, External communication, Inter-frame communication, Client-side storage, UI & Rendering, Media, Geolocation and Device access.

For a third-party component, each category can be fully disabled, fully enabled, or enabled only for a self-defined whitelist. The whitelists contain category-specific entries. For example, a whitelist for the category "DOM Access" contains the ids of the elements that might be read from or updated in the DOM. The nine security-sensitive categories are listed in Table 2.1, together with their underlying APIs, the number of security-sensitive functions in each API, and their WebJail whitelist types.

The secure composition policy expresses the restrictions for each of the security-sensitive categories, and an example policy is shown below. Unspecified categories are disallowed by default, making the last line in the example policy obsolete.

| Categories and APIs (# op.) | Whitelist |
|---|---|
| **DOM Access** | ElemReadSet, ElemWriteSet |
| DOM Core (17) | |
| **Cookies** | KeyReadSet, KeyWriteSet |
| cookies (2) | |
| **External Communication** | DestinationDomainSet |
| XHR, CORS, UMP (4) | |
| WebSockets (5) | |
| Server-sent events (2) | |
| **Inter-frame Communication** | DestinationDomainSet |
| Web Messaging (3) | |
| **Client-side Storage** | KeyReadSet, KeyWriteSet |
| Web Storage (5) | |
| IndexedDB (16) | |
| File API (4) | |
| File API: Dir. and Syst. (11) | |
| File API: Writer (3) | |
| **UI and Rendering** | |
| History API (4) | |
| Drag/Drop events (3) | |
| **Media** | |
| Media Capture API (3) | |
| **Geolocation** | |
| Geolocation API (2) | |
| **Device Access** | SensorReadSet |
| System Information API (2) | |
| **Total number of security-sensitive operations: 86** | |

**Table 2.1: Overview of the sensitive JavaScript operations from the HTML 5 APIs, divided in categories.**

```
{ "framecomm" : "yes",
  "extcomm" : [ "google.com", "youtube.com" ],
  "device" : "no" }
```

It is important to note that WebJails or regular frames can be used inside WebJails. In such a case, the functionality in the inner frame is determined by the policies imposed on enclosing frames, in addition to its own policy (if it has one, as is the case with a WebJail frame). Allowing sensible cascading of policies implies that "deeper" policies can only make the total policy more strict. If this was not the case, a WebJail with a less strict policy could be used to "break out" of the WebJail restrictions.

The semantics of a policy entry for a specific category can be thought of as a set. Let $\mathcal{V}$ be the set of all possible values that can be listed in a whitelist. The "allow all" policy would then be represented by the set $\mathcal{V}$ itself, a whitelist would be represented by a subset $w \subseteq \mathcal{V}$ and the "allow none" policy by the empty set $\phi$. The relationship "$x$ is at least as strict as $y$" can be represented as $x \subseteq y$. Using this notation, the combined policy $p$ of 2 policies $a$ and $b$ is the intersection $p = a \cap b$, since $p \subseteq a$ and $p \subseteq b$.

After loading, parsing and combining all the policies applicable to the WebJail-protected iframe, the policy is enforced via the underlying layers.

### Advice construction layer

The task of the advice construction layer is to build advice functions based on the high-level policy received from the policy layer, and apply these advice functions on the low-level security-sensitive operations via deep aspect technology in the deep advice weaving layer.

To do so, the advice construction layer applies a 2-step refinement process. For each category of the secure composition policy, the set of relevant APIs is selected. Next for each API, the individual security-sensitive operations are processed. Consider for instance that a whitelist of type "KeyReadSet"[1] is specified for the client-side storage in the composition policy. This is first mapped to the various storage APIs in place (such as Web Storage and File API), and then advice is constructed for the security-sensitive operations in the API (e.g. for accessing the *localStorage* object). The advice function decides, based on the policy, whether or not the associated API function will be called: if the policy for the API function is "allow all", or "allow some" and the whitelist matches, then the advice function allows the call. Otherwise, the call is blocked.

On successful completion of its job, the advice construction layer has advice functions for all the security-sensitive operations across the nine categories relevant for the specific policy. Next, the advices are applied on the original operations via the deep advice weaving layer.

[1] Such a whitelist contains a set of keys that may be read

#### Deep aspect weaving layer

The *(advice, operation)* pairs received from the advice construction layer are registered into the JavaScript engine as deep advice. The result of this weaving is that the original API function is replaced with the advice function, and that all accesspaths to the API function now go through the advice function. The advice function itself is the only place where a reference to the original API function exists, allowing it to make use of the original functionality when desired.

### 2.1.3 Evaluation

WebJail has been implemented as a prototype by modifying Mozilla Firefox. The prototype currently supports the security-sensitive categories external and inter-frame communication, client-side storage, UI and rendering (except for drag/drop events) and geolocation. More details about the implementation can be found in [78]. The remainder of this section will evaluate this prototype.

#### Performance

We performed micro-benchmarks on WebJail to evaluate its performance overhead with regard to page load-time and function execution. The prototype implementation is built on Mozilla Firefox 4.0b10pre, and compiled with the GNU C++ compiler v4.4.4-14ubuntu5. The benchmarks were performed on an Apple MacBook Pro 4.1, with an Intel Core 2 Duo T8300 CPU running at 2.40GHz and 4GB of memory, running Ubuntu 10.10 with Linux kernel version `2.6.35-28-generic`.

**Page load-time overhead** To measure the page load-time overhead, we created a local webpage (`main.html`) that embeds another local page (`inner.html`) in an iframe with and without a local policy file. `inner.html` records a timestamp (`new Date().getTime()`) when the page starts and stops loading (using the body `onload` event). WebJail was modified to record the starttime before anything else executes, so that policy retrieval, loading and application is taken into account. After the results are submitted, `main.html` reloads.

We averaged the results of 1000 page reloads. Without WebJail, the average load-time was 16.22ms ($\sigma$ = 3.74ms). With WebJail, the average is 23.11ms ($\sigma$ = 2.76ms).

**Function execution overhead** Similarly, we used 2 local pages (`main.html` and `inner.html`) to measure function execution overhead. `inner.html` measures how long it takes for 10000 iterations of a piece of code to execute. We measured 2 scenarios: a typical XMLHttpRequest invocation (constructor, `open` and `send` functions) and a localStorage set and get (`setItem` and `getItem`). Besides measuring a baseline without WebJail policy, we measured each scenario when restricted by 3 different policies: "allow all", "allow none" and a whitelist with 5 values. The averages are summarized in Table 2.2.

| | XMLHttpRequest | localStorage |
|---|---|---|
| Baseline | 1.25 ms | 0.37 ms |
| "Allow all" | 1.25 ms (+ 0%) | 0.37 ms (+ 0%) |
| "Allow none" | 0.07 ms (- 94.4%) | 0.04 ms (- 89.2 %) |
| Whitelist | 1.33 ms (+ 6.4%) | 0.47 ms (+ 27%) |

**Table 2.2: Function execution overhead**

To conclude, we have registered a negligible performance penalty for our WebJail prototype: a page load-time of 7ms, and an execution overhead in case of sensitive operations about 0.1ms.

#### Security

The `registerAdvice` function disconnects an available function and makes it available only to the advice function. Because of the use of deep aspects, we can ensure that no other references to the original function are available in the JavaScript environment, even if such references already existed before `registerAdvice` was called. We have successfully verified this full mediation of the deep aspects using our prototype implementation.

Because advice functions are written in JavaScript and the advice function has the only reference to the original function, it would be tempting for an attacker to attack the WebJail infrastructure. The retrieval and application of a WebJail policy happens before any other code is executed in the JavaScript context. In addition, the

`registerAdvice` function is disabled once the policy has been applied. The only remaining attack surface is the advice function during its execution. We know of 3 attack vectors: prototype poisoning of `Array.prototype.indexOf` and `Function.prototype.apply`, and `toString` redefinition on `vp[0]`. By introducing the readonly copies `ROindexOf` and `ROapply`, we prevent an attacker from exploiting the first 2 attack vectors. The third vector, `toString` redefinition, was verified in our prototype implementation and is not an issue because `toString` is never called on the argument `vp[0]`.

**Applicability**

To test the applicability of the WebJail architecture, we have applied our prototype implementation to mainstream mashup platforms, including iGoogle and Facebook. As part of the setup, we have instrumented responses from these platforms to include secure composition policies. Next, we have applied both permissive composition policies as well as restricted composition policies and verified that security-sensitive operations for the third-party components were executed as usual in the first case, and blocked in the latter case. For instance, as part of the applicability tests, we applied WebJail to control Geolocation functionality in the Google Latitude[41] component integrated into iGoogle, as well as external communication functionality of the third-party Facebook application "Tweets To Pages"[48] integrated into our Facebook page.

## 2.2 Automatic and Precise Client-Side Protection against CSRF Attacks

From a security perspective, web browsers are a key component of today's software infrastructure. A browser user might have a session with a trusted site A (e.g. a bank, or a webmail provider) open in one tab, and a session with a potentially dangerous site B (e.g. a site offering cracks for games) open in another tab. Hence, the browser enforces some form of isolation between these two origins A and B through a heterogeneous collection of security controls collectively known as the *same-origin-policy* [92]. An *origin* is a (protocol, domain name, port) triple, and restrictions are imposed on the way in which code and data from different origins can interact. This includes for instance restrictions that prevent scripts from origin B to access content from origin A.

An important known vulnerability in this isolation is the fact that content from origin B can initiate requests to origin A, and that the browser will treat these requests as being part of the ongoing session with A. In particular, if the session with A was authenticated, the injected requests will appear to A as part of this authenticated session. This enables an attack known as *Cross Site Request Forgery (CSRF)*: B can initiate effectful requests to A (e.g. a bank transaction, or manipulations of the victim's mailbox or address book) without the user being involved.

CSRF has been recognized since several years as one of the most important web vulnerabilities [8], and many countermeasures have been proposed. Several authors have proposed server-side countermeasures [8, 22, 54]. However, an important disadvantage of server-side countermeasures is that they require modifications of server-side programs, have a direct operational impact (e.g. on performance or maintenance), and it will take many years before a substantial fraction of the web has been updated.

Alternatively, countermeasures can be applied on the client-side, as browser extensions. The basic idea is simple: the browser can strip session and authentication information from malicious requests, or it can block such requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures [53, 32, 62, 63, 71, 93] are typically too strict: they block or strip all cross-origin requests of a specific type (e.g. GET, POST, any). This effectively protects against CSRF attacks, but it unfortunately also breaks many existing websites that rely on authenticated cross-origin requests. Two important examples are sites that use third-party payment (such as PayPal) or single sign-on solutions (such as OpenID). Hence, these existing client-side countermeasures require extensive help from the user, for instance by asking the user to define white-lists of trusted sites or by popping up user confirmation dialogs. This is suboptimal, as it is well-known that the average web user can not be expected to make accurate security decisions.

This section proposes a novel client-side CSRF countermeasure, that includes an automatic and precise filtering algorithm for cross-origin requests. It is *automatic* in the sense that no user interaction or configuration is required. It is *precise* in the sense that it distinguishes well between malicious and non-malicious requests. More specifically, through a systematic analysis of logs of web traffic, we identify a characteristic of non-malicious cross-origin requests that we call the *trusted-delegation assumption*: a request from B to A can be considered non-malicious if, earlier in the session, A explicitly delegated control to B in some specific ways. Our filtering algorithm relies on this assumption: it will strip session and authentication information from cross-origin requests, unless it can determine

that such explicit delegation has happened. This section is a condensed version of [33]. The full version of this paper is included in the appendix (section A).

### 2.2.1 Automatic and Precise Request Stripping

The core idea of our new countermeasure is the following: client-side state (i.e. session cookie headers and authentication headers) is stripped from all cross-origin requests, except for *expected* requests. A cross-origin request from origin A to B is *expected* if B previously (earlier in the browsing session) *delegated* to A. We say that B *delegates* to A if B either issues a POST request to A, or if B redirects to A using a URI that contains parameters.

The rationale behind this core idea is that (1) non-malicious collaboration scenarios follow this pattern, and (2) it is hard for an attacker to trick A into delegating to a site of the attacker: forcing A to do a POST or parametrized redirect to an evil site E requires the attacker to either identify a cross-site scripting (XSS) vulnerability in A, or to break into A's webserver. In both these cases, A has more serious problems than CSRF.

Obviously, a GET request from A to B is not considered a delegation, as it is very common for sites to issue GET requests to other sites, and as it is easy for an attacker to trick A into issuing such a GET request.

Unfortunately, the elaboration of this simple core idea is somewhat complicated by the existence of HTTP redirects. A web server can respond to a request with a *redirect* response, indicating to the browser that it should resend the request elsewhere, for instance because the requested resource was moved. The browser will follow the redirect automatically, without user intervention. Redirects are used widely and for a variety of purposes, so we cannot ignore them. In addition, attacker-controlled websites can also use redirects in an attempt to bypass client-side CSRF protection. Akhawe et al. [3] discuss several examples of how attackers can use redirects to attack web applications, including an attack against a CSRF countermeasure. Hence, correctly dealing with redirects is a key requirement for security.

The flowgraph in Figure 2.3 summarizes our filtering algorithm. For a given request, it determines what session state (cookies and authentication headers) the browser should attach to the request. The algorithm differentiates between simple requests and requests that are the result of a redirect.

**Simple Requests** Simple requests that are not cross-origin, as well as expected cross-origin requests are handled as unprotected browsers handle them today. The browser automatically attaches the last known client-side state associated with the destination origin (point 1). The browser does not attach any state to non-expected cross-origin requests (point 3).

**Redirect Requests** If a request is the consequence of a redirect response, then the algorithm determines if the redirect points to the origin where the response came from. If this is the case, the client-side state for the new request is limited to the client-side state known to the previous request (i.e. the request that triggered this redirect) (point 2). If the redirect points to another origin, then, depending on whether this cross-origin request is expected or not, it either gets session-state automatically attached (point 1) or not (point 3).

**When Is a Request Expected?** A key element of the algorithm is determining whether a request is *expected* or not. As discussed above, the intuition is: a cross-origin request from B to A is expected if and only if A first delegated to B by issuing a POST request to B, or by a parametrized redirect to B. Our algorithm stores such trusted delegations, and an assumption that we rely on (and that we refer to as the *trusted-delegation assumption*) is that sites will only perform such delegations to sites that they trust. In other words, a site A remains vulnerable to CSRF attacks from origins to which it delegates. Section 2.2.3 provides experimental evidence for the validity of this assumption.

The algorithm to decide whether a request is expected goes as follows.

For a simple cross-origin request from site B to site A, a trusted delegation from site A to B needs to be present in the delegation store.

For a redirect request that redirects a request to origin Y (light gray) to another origin Z (dark gray) in a browsing context associated with some origin $\alpha$, the following rules apply.

1. First, if the destination (Z) equals the source (i.e. $\alpha = Z$) (Figure 2.4(a)), then the request is expected if there is a trusted delegation from Z to Y in the delegation store. Indeed, Y is effectively doing a cross-origin request to Z by redirecting to Z. Since the browsing context has the same origin as the destination, it can be expected not to manipulate redirect requests to misrepresent source origins of redirects (cfr next case).

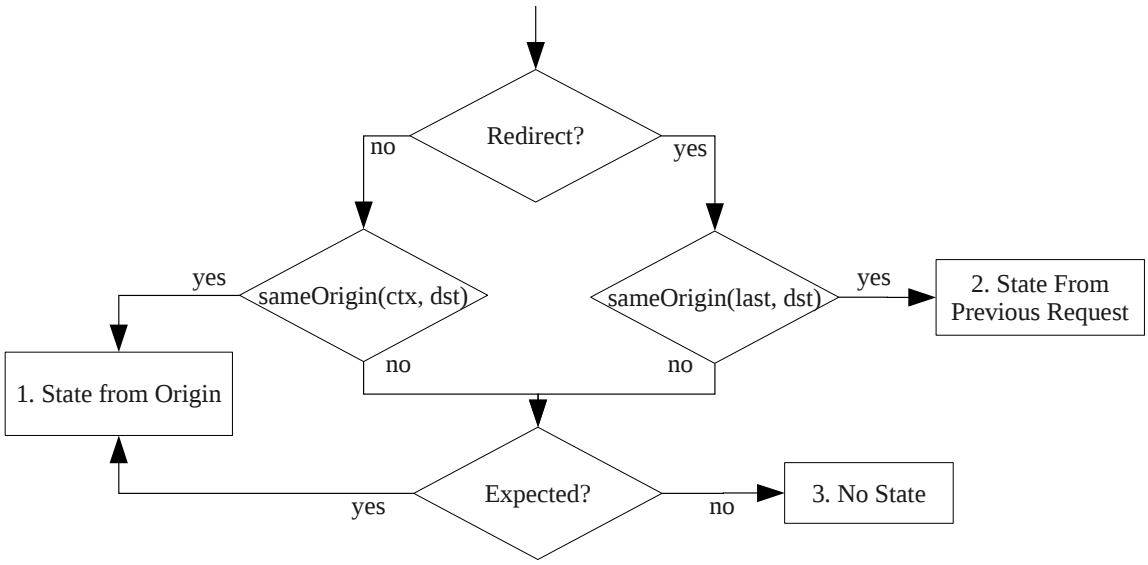
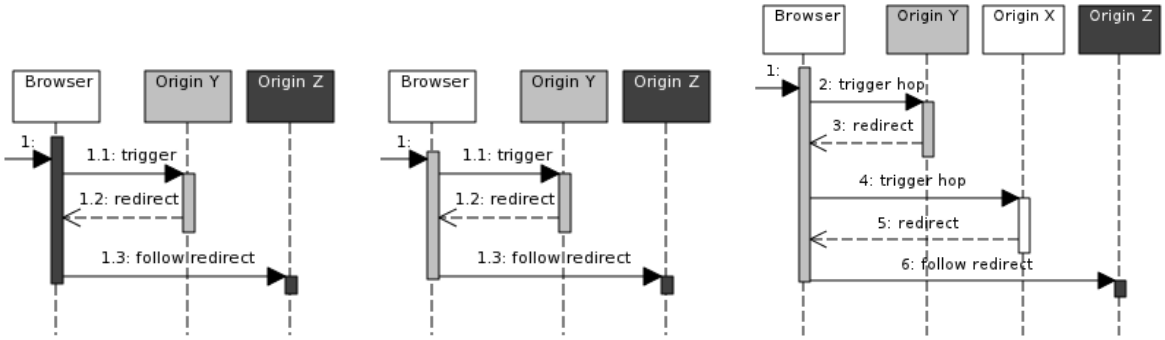



**Figure 2.3: The request filtering algorithm**

2. Alternatively, if the destination (Z) is not equal to the source (i.e. $\alpha \neq Z$) (Figure 2.4(b)), then the request is expected if there is a trusted delegation from Z to Y in the delegation store, since Y is effectively doing a cross-origin request to Z. Now, the browsing context might misrepresent source origins of redirects by including additional redirect hops (origin X (white) in Figure 2.4(c)). Hence, our decision to classify the request does not involve X.




(a) (b) (c)

**Figure 2.4: Complex cross-origin redirect scenarios**

Finally, our algorithm imposes that expected cross-origin requests can only use the GET method and that only two origins can be involved in the request chain. These restrictions limit the potential power an attacker might have, even if the attacker successfully deceives the trusted-delegation mechanism.

### 2.2.2 Formal Modeling and Checking

The design of web security mechanisms is complex: the behaviour of (same-origin and cross-origin) browser requests, server responses and redirects, cookie and session management, as well as the often implicit threat models of web security can lead to subtle security bugs in new features or countermeasures. In order to evaluate proposals for new web mechanisms more rigorously, Akhawe et al. [3] have proposed a model of the Web infrastructure, formalized in Alloy.

The base model is about 2000 lines of Alloy source code, describing (1) the essential characteristics of browsers, web servers, cookie management and the HTTP protocol, and (2) a collection of relevant threat models for the web. The Alloy Analyzer – a bounded-scope model checker – can then produce counterexamples that violate intended security properties if they exist in a specified finite scope.

In this section, we briefly introduce Akhawe's model and present our extensions to the model. We also discuss how the model was used to verify the absence of attack scenarios and the presence of functional scenarios.

### Modeling our countermeasure

The model of Akhawe et al. defines different principals, of which `GOOD` and `WEBATTACKER` are most relevant. `GOOD` represents an honest principal, who follows the rules imposed by the technical specifications. A `WEBATTACKER` is a malicious user who can control malicious web servers, but has no extended networking capabilities.

The concept of `Origin` is used to differentiate between origins, which correspond to domains in the real world. An origin is linked with a server on the web, that can be controlled by a principal. The browsing context, modeled as a `ScriptContext`, is also associated with an `origin`, that represents the origin of the currently loaded page, also known as the referrer.

A `ScriptContext` can be the source of a set of `HTTPTransaction` objects, which are a pair of an `HTTPRequest` and `HTTPResponse`. An HTTP request and response are also associated with their remote destination origin. Both an HTTP request and response can have headers, where respectively the `CookieHeader` and `SetCookieHeader` are the most relevant ones. An HTTP request also has a `method`, such as `GET` or `POST`, and a `queryString`, representing URI parameters. An HTTP response has a `statusCode`, such as `c200` for a content result or `c302` for a redirect. Finally, an HTTP transaction has a cause, which can be none, such as the user opening a new page, a `RequestAPI`, such as a scripting API, or another `HTTPTransaction`, in case of a redirect.

To model our approach, we need to extend the model of Akhawe et al. to include (a) the accessible client-side state at a certain point in time, (b) the trusted delegation assumption and (c) our filtering algorithm. We discuss (a) and (b) in detail, but due to space constraints we omit the code for the filtering algorithm (c), which is simply a literal implementation of the algorithm discussed in Section 2.2.1.

**Client-Side State** We introduced a new signature `CSState` that represents a client-side state (Listing 2.1). Such a state is associated with an `Origin` and contains a set of `Cookie` objects. To associate a client-side state with a given request or response and a given point in time, we have opted to extend the `HTTPTransaction` from the original model into a `CSStateHTTPTransaction`. Such an extended transaction includes a `beforeState` and `afterState`, respectively representing the accessible client-side state at the time of sending the request and the updated client-side state after having received the response. The `afterState` is equal to the `beforeState`, with the potential addition of new cookies, set in the response.

```
1  sig CSState {
2     dst: Origin,
3     cookies: set Cookie
4  }
5
6  sig CSStateHTTPTransaction extends HTTPTransaction {
7     beforeState : CSState,
8     afterState : CSState
9  } {
10    //The after state of a transaction is equal to the before state + any additional cookies set in the response
11    beforeState·dst = afterState·dst
12    afterState·cookies = beforeState·cookies + (resp·headers & SetCookieHeader)·thecookie
13
14    // The destination origin of the state must correspond to the transaction destination origin
15    beforeState·dst = req·host
16 }
```

**Listing 2.1: Signatures representing our data in the model**

**Trusted-delegation Assumption** We model the trusted-delegation assumption as a fact, that honest servers do not send a POST or parametrized redirect to web attackers ((Listing 2.2).

```
1  fact TrustedDelegation {
2     all r : HTTPRequest | {
3        (r·method = POST || some (req·r)·cause & CSStateHTTPTransaction)
4           &&
```

```
5       ((some (req·r)·cause & CSStateHTTPTransaction && getPrincipalFromOrigin[(req·r)·cause·req·host] in GOOD)
            || getPrincipalFromOrigin[transactions·(req·r)·owner] in GOOD)
6       implies
7          getPrincipalFromOrigin[r·host] not in WEBATTACKER
8    }
9 }
```

**Listing 2.2: The fact modeling the trusted-delegation assumption**

#### Using Model Checking for Security and Functionality

We formally define a CSRF attack as the possibility for a web attacker (defined in the base model) to inject a request with at least one existing cookie attached to it (this cookie models the session/authentication information attached to requests) in a session between a user and an honest server (Listing 2.3).

```
1  pred CSRF[r : HTTPRequest] {
2       //Ensure that the request goes to an honest server
3       some getPrincipalFromOrigin[r·host]
4       getPrincipalFromOrigin[r·host] in GOOD
5
6       //Ensure that an attacker is involved in the request
7       some (WEBATTACKER·servers & involvedServers[req·r]) || getPrincipalFromOrigin[(transactions·(req·r))·owner
            ] in WEBATTACKER
8
9       // Make sure that at least one cookie is present
10      some c : (r·headers & CookieHeader)·thecookie | {
11         //Ensure that the cookie value is fresh (i·e· that it is not a renewed value in a redirect chain)
12         not c in ((req·r)·*cause·resp·headers & SetCookieHeader)·thecookie
13      }
14 }
```

**Listing 2.3: The predicate modeling a CSRF attack**

We provided the Alloy Analyzer with a universe of at most 9 HTTP events and where an attacker can control up to 3 origins and servers (a similar size as used in [3]). In such a universe, no examples of an attacker injecting a request through the user's browser were found. This gives strong assurance that the countermeasure does indeed protect against CSRF under the trusted delegation assumption.

We also modeled the non-malicious scenarios, and the Alloy Analyzer reports that these scenarios are indeed permitted. From this, we can also conclude that our extension of the base model is consistent.

Space limitations do not permit us to discuss the detailed scenarios present in our model, but the interested reader can find the complete model available for download at [34].

### 2.2.3 Evaluating the Trusted-Delegation Assumption

Our countermeasure drastically reduces the attack surface for CSRF attacks. Without CSRF countermeasures in place, an origin can be attacked by any other origin on the web. With our countermeasure, an origin can only be attacked by another origin to which it has delegated control explicitly by means of a cross-origin POST or redirect. We have already argued in Section 2.2.1 that it is difficult for an attacker to cause unintended delegations. In this section, we measure the remaining attack surface experimentally.

We conducted an extensive traffic analysis using a real-life data set of 4.729.217 HTTP requests, collected from 50 unique users over a period of 10 weeks. The analysis revealed that 1.17% of the 4.7 million requests are treated as delegations in our approach. We manually analyzed all these 55.300 requests, and classified them in the interaction categories summarized in Table 2.3.

For each of the categories, we discuss the resulting attack surface:

**Third party service mashups.** This category consists of various third party services that can be integrated in other websites. Except for the single sign-on services, this is typically done by script inclusion, after which the included script can launch a sequence of cross-origin GET and/or POST requests towards offered AJAX APIs. In addition, the service providers themselves often use cross-origin redirects for further delegation towards content delivery networks.

As a consequence, the origin A including the third-party service S becomes vulnerable to CSRF attacks from S. This attack surface is unimportant, as in these scenarios, S can already attack A through script inclusion, a more powerful attack than CSRF.

| | # requests | | POST | redir. |
|---|---|---|---|---|
| Third party service mashups | 29.282 | (52,95%) | 5.321 | 23.961 |
| *Advertisement services* | *22.343* | *(40,40%)* | *1.987* | *20.356* |
| *Gadget provider services (appspot, mochibot, gmodules, ...)* | *2.879* | *(5,21%)* | *2.757* | *122* |
| *Tracking services (metriweb, sitestat, uts.amazon, ...)* | *2.864* | *(5,18%)* | *411* | *2.453* |
| *Single Sign-On services (Shibboleth, Live ID, OpenId, ...)* | *1.156* | *(2,09%)* | *137* | *1.019* |
| *3rd party payment services (Paypal, Ogone)* | *27* | *(0,05%)* | *19* | *8* |
| *Content sharing services (addtoany, sharethis, ...)* | *13* | *(0,02%)* | *10* | *3* |
| Multi-origin websites | 13.973 | (25,27%) | 198 | 13.775 |
| Content aggregators | 8.276 | (14,97%) | 0 | 8.276 |
| *Feeds (RSS feeds, News aggregators, mozilla fxfeeds, ...)* | *4.857* | *(8,78%)* | *0* | *4.857* |
| *Redirecting search engines (Google, Comicranks, Ohnorobot)* | *3.344* | *(6,05%)* | *0* | *3.344* |
| *Document repositories (ACM digital library, dx.doi.org, ...)* | *75* | *(0,14%)* | *0* | *75* |
| False positives (wireless network access gateways) | 1.215 | (2,20%) | 12 | 1.203 |
| URL shorteners (gravatar, bit.ly, tinyurl, ...) | 759 | (1,37%) | 0 | 759 |
| Others (unclassified) | 1.795 | (3,24%) | 302 | 1.493 |
| **Total number of 3rd party delegation initiators** | **55.300** | **(100%)** | **5.833** | **49.467** |

**Table 2.3: Analysis of the trusted-delegation assumption in a real-life data set of 4.729.217 HTTP requests**

In addition, advertisement service providers P that further redirect to content delivery services D are vulnerable to CSRF attacks from D whenever a user clicks an advertisement. Again, this attack surface is unimportant: the delegation from P to D correctly reflects a level of trust that P has in D, and P and D will typically have a legal contract or SLA in place.

**Multi-origin websites.** Quite a number of larger companies and organizations have websites spanning multiple origins (such as *live.com - microsoft.com* and *google.be - google.com*). Cross-origin POST requests and redirects between these origins make it possible for such origins to attack each other. For instance, *google.be* could attack *google.com*. Again, this attack surface is unimportant, as all origins of such a multi-origin website belong to a single organization.

**Content aggregators.** Content aggregators collect searchable content and redirect end-users towards a specific content provider. For news feeds and document repositories (such as the ACM digital library), the set of content providers is typically stable and trusted by the content aggregator, and therefore again a negligible attack vector.

Redirecting search engines register the fact that a web user is following a link, before redirecting the web user to the landing page (e.g. as Google does for logged in users). Since the entries in the search repository come from all over the web, our CSRF countermeasure provides little protection for such search engines. Our analysis identified 4 such origins in the data set: *google.be*, *google.com*, *comicrank.com*, and *ohnorobot.com*.

**False positives.** Some fraction of the cross-origin requests are caused by network access gateways (e.g. on public Wifi) that intercept and reroute requests towards a payment gateway. Since such devices have man-in-the-middle capabilities, and hence more attack power than CSRF attacks, the resulting attack surface is again negligible.

**URL shorteners.** To ease URL sharing, URL shorteners transform a shortened URL into a preconfigured URL via a redirect. Since such URL shortening services are open, an attacker can easily control a new redirect target. The effect is similar to the redirecting search engines; URL shorteners are essentially left unprotected by our countermeasure. Our analysis identified 6 such services in the data set: *bit.ly*, *gravatar.com*, *post.ly*, *tiny.cc*, *tinyurl.com*, and *twitpic.com*.

**Others(unclassified)** For some of the requests in our data set, the origins involved in the request were no longer online, or the (partially anonymized) data did not contain sufficient information to reconstruct what was happening, and we were unable to classify or further investigate these requests.

In summary, our experimental analysis shows that the trusted delegation assumption is realistic. Only 10 out of 23.592 origins (i.e. 0.0042% of the examined origins) – the redirecting search engines and the URL shorteners – perform delegations to arbitrary other origins. They are left unprotected by our countermeasure. But the overwhelming majority of origins delegates (in our precise technical sense, i.e. using cross-origin POST or redirect) only to other origins with whom they have a trust relationship.

## 2.3 Lightweight Protection against Session Hijacking

In the modern Web, JavaScript has proven its usefulness by providing server offloading, asynchronous requests and responses and in general improving the overall user experience of websites. Unfortunately, the de facto support of browsers for JavaScript opened up the user to a new range of attacks, of which the most common is Cross-site scripting (XSS[2]).

In XSS attacks, an attacker convinces a user's browser to execute malicious JavaScript code on his behalf by injecting this code in the body of a vulnerable webpage. Due to the fact that the attacker can only execute JavaScript code, as opposed to machine code, the attack was initially considered of limited importance. Numerous incidents though, such as the Sammy worm that propagated through an XSS vulnerability on the social network MySpace [73] and the XSS vulnerabilities of many high-impact websites (e.g., Twitter, Facebook and Yahoo [91]) have raised the awareness of the security community. More recently, Apache released information about an incident on their servers where attackers took advantage of an XSS vulnerability and by constant privilege escalation managed to acquire administrator access to a number of servers [1].

Today, the Open Web Application Security Project (OWASP) ranks XSS attacks as the second most important Web application security risk [67]. The Web Hacking Incident Database from the Web Application Security Consortium states that 13.87% of all attacks against Web applications are XSS attacks [29]. These reports, coupled with more than 300,000 recorded vulnerable websites in the XSSed archive [91], show that this problem is far from solved.

In this section, we present SessionShield, a lightweight countermeasure against session hijacking. Session hijacking occurs when an attacker steals the session information from a legitimate user for a specific website and uses it to circumvent authentication to that website. Session hijacking is by far the most popular type of XSS attack since every website that uses session identifiers is potentially vulnerable to it. Our system is based on the observation that session identifiers are strings of data that are intelligible to the Web application that issued them but not to the Web client who received them. SessionShield is a proxy outside of the browser that inspects all outgoing requests and incoming responses. Using a variety of methods, it detects session identifiers in the incoming HTTP headers, strips them out and stores their values in its own database. In every outgoing request, SessionShield checks the domain of the request and adds back the values that were previously stripped. In case of a session hijacking attack, the browser will still execute the session hijacking code, but the session information will not be available since the browser never received it.

Our system is transparent to both the Web client and the Web server, it operates solely on the client-side and it doesn't rely on the Web server or trusted third parties. SessionShield imposes negligible overhead and doesn't require training or user interaction making it ideal for both desktop and mobile systems. This section is a trimmed down version of the text published in [66].

### 2.3.1 SessionShield Design

SessionShield is based on the idea that session identifiers are data that no legitimate client-side script will use and thus should not be available to the scripting languages running in the browser. Our system shares this idea with the `HTTP-Only` mechanism but, unlike `HTTP-Only`, it can be applied selectively to a subset of cookie values and, more important, it doesn't need support from Web applications. This means, that SessionShield will protect the user from session hijacking regardless of the security provisioning of Web operators.

The idea itself is founded on the observation that session identifiers are strings composed by random data and are unique for each visiting client. Furthermore, a user receives a different session identifier every time that he logs out from a website and logs back in. These properties attest that there can be no legitimate calculations done by the client-side scripts using as input the constantly-changing random session identifiers. The reason that these values are currently accessible to client-side scripts is because Web languages and frameworks mainly use the cookie mechanism as a means of transport for the session identifiers. The cookie is by default added to every client request by the browser which aleviates the Web programmers from having to create their own transfer mechanism for session identifiers. JavaScript can, by default, access cookies (using the `document.cookie` method) since they may contain values that the client-side scripts legitimately need, e.g., language selection, values for boolean variables and timestamps.

[2]Cross-site scripting is commonly abbreviated as XSS to distinguish it from the acronym of Cascading Style Sheets (CSS)

| Session Framework | Name of Session variable |
|---|---|
| PHP | phpsessid |
| ASP/ASP.NET | asp.net_sessionid<br>aspsessionid*<br>.aspxauth*<br>.aspxanonymous* |
| JSP | jspsessionid<br>jsessionid |

**Table 2.4: Default session naming for the most common Web frameworks**

### Core Functionality

Our system acts as a personal proxy, located on the same host as the browser(s) that it protects. In order for a website or a Web application to set a cookie to a client, it sends a `Set-Cookie` header in its HTTP response headers, followed by the values that it wishes to set. SessionShield inspects incoming data in search for this header. When the header is present, our system analyses the values of it and attempts to discover whether session identifiers are present. If a session identifier is found, it is stripped out from the headers and stored in SessionShield's internal database. On a later client request, SessionShield queries its internal database using the domain of the request as the key and adds to the outgoing request the values that it had previously stripped.

A malicious session hijacking script, whether reflected or stored, will try to access the cookie and transmit its value to a Web server under the attacker's control. When SessionShield is used, cookies inside the browser no longer contain session identifiers and since the attacker's request domain is different from the domain of the vulnerable Web application, the session identifier will not be added to the outgoing request, effectively stopping the session hijacking attack.

In order for SessionShield to protect users from session hijacking it must successfully identify session identifiers in the cookie headers. Our system uses two identification mechanisms based on: a) common naming conventions of Web frameworks and of custom session identifiers and b) statistical characteristics of session identifiers.

### Naming Conventions of Session Identifiers

**Common Web Frameworks** Due to the popularity of Web sessions, all modern Web languages and frameworks have support for generating and handling session identifiers. Programmers are actually adviced not to use custom session identifiers since their implementation will most likely be less secure from the one provided by their Web framework of choice. When a programmer requests a session identifier, e.g., with `session_start()` in PHP, the underlying framework generates a random unique string and automatically emmits a `Set-Cookie` header containing the generated string in a `name=value` pair, where `name` is a standard name signifying the framework used and `value` is the random string itself. Table 2.4 shows the default names of session identifiers according to the framework used[3]. These naming conventions are used by SessionShield to identify session identifiers in incoming data and strip them out of the headers.

**Common Custom Naming** From the results of an experiment, we observed that "sess" is a common keyword among custom session naming and thus it is included as an extra detection method of our system. In order to avoid false-positives we added the extra measure of checking the length and the contents of the value of such a pair. More specifically, SessionShield identifies as session identifiers pairs that contain the word "sess" in their name and their value is more than 10 characters long containing both letters and numbers. These characteristics are common among the generated sessions of all popular frameworks so as to increase the value space of the identifiers and make it practically impossible for an attacker to bruteforce a valid session identifier.

[3] On some versions of the ASP/ASP.NET framework the actual name contains random characters, which are signified by the wildcard symbol in the table.

#### Statistical Characteristics of session identifiers

Despite the coverage offered by the previous mechanism, it is beyond doubt that there can be sessions that do not follow standard naming conventions and thus would not be detected by it. In this part we focus on the fact that session identifiers are long strings of symbols generated in some random way. These two key characteristics, length and randomness, can be used to predict if a string, that is present in a cookie, is a session identifier or not. This criterion, in fact, is similar to predicting the strength of a password.

Three methods are used to predict the probability that a string is a session identifier (or equivalently the strength of a password):

1. **Information entropy**: The strength of a password can be measured by the information entropy it represents [37]. If each symbol is produced independently, the entropy is $H = L \cdot \log_2 N$, with $N$ the number of possible symbols and $L$ the length of the string. The resulting value, $H$, gives the entropy and represents the number of bits necessary to represent the string. The higher the number of necessary bits, the better the strength of the password in the string. For example, a pin-code consisting out of four digits has an entropy of 3.32 bits per symbol and a total entropy of 13.28.

2. **Dictionary check**: The strength of a password reduces if it is a known word. Similarly, cookies that have known words as values are probably not session identifiers.

3. $\chi^2$: A characteristic of a random sequence is that all symbols are produced by a generator that picks the next symbol out of a uniform distribution ignoring previous symbols. A standard test to check if a sequence correlates with a given distribution is the $\chi^2$-test [56], and in this case this test is used to calculate the correlation with the uniform distribution. The less the string is correlated with the random distribution the less probable it is that it is a random sequence of symbols. The uniform distribution used is $1/N$, with $N$ the size of the set of all symbols appearing in the string.

Every one of the three methods returns a probability that the string is a session identifier. These probabilities are combined by means of a weighted average to obtain one final probability. SessionShield uses this value and a threshold to differentiate between session and non-session values (see Section 2.3.2 for details).

### 2.3.2 Implementation

We decided to prototype SessionShield using Python. We used an already implemented Python proxy, TinyHTTP-Proxy [47], and added the session detection mechanisms that were described in Section 2.3.1. The advantage of implementing SessionShield as a stand-alone personal proxy instead of a browser-plugin relies on the fact that cookies residing in the browser can still be attacked, e.g. by a malicious add-on [61]. When sensitive data are held outside of the browser, in the data structures of the proxy, a malicious add-on will not be able to access them. On the other hand, a browser plugin can transparently support HTTPS and provides a more user-friendly install and update procedure.

The threshold value of SessionShield was obtained by using the known session identifiers from an `HTTP-Only` experiment as input to our statistical algorithm and observing the distribution of the reported probabilities.

### 2.3.3 Evaluation

#### False Positives and False Negatives

SessionShield can protect users from session hijacking as long as it can successfully detect session identifiers in the incoming HTTP(S) data. In order to evaluate the security performance of SessionShield we conducted the following experiment: we seperated the first 1,000 cookies from one of our experiments and we used them as input to the detection mechanism of SessionShield. SessionShield processed each cookie and classified a subset of the values as sessions identifiers and the rest as benign data. We manually inspected both sets of values and we recorded the false positives (values that were wrongly detected as session identifiers) and the false negatives (values that were not detected as session identifiers even though they were). SessionShield classified 2,167 values in total (average of 2.16 values/cookie) with 70 false negatives (3%) and 19 false positives (0,8%).

False negatives were mainly session identifiers that did not comply to our session identifier criteria, i.e a) they didn't contain both letters and numbers or b) they weren't longer than 10 characters. Session identifiers that do not

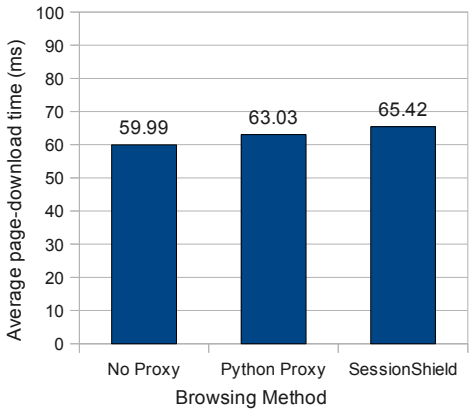



**Figure 2.5: Average download time of the top 1,000 websites when accessed locally without a proxy, with a simple forwarding Python-proxy and with SessionShield**

comply to these requirements are easily brute-forced even if SessionShield protected them. With regard to false positives, it is important to point out that in order for a website to stop operating correctly under SessionShield, its legitimate client-side scripts must try to use values that SessionShield classified as session identifiers. Thus the actual percentage of websites that wouldn't operate correctly and would need to be white-listed is less than or equal to 0,8%.

### Performance Overhead

In an effort to quantify how much would SessionShield change the Web experience of users, we decided to measure the difference in page-download time when a page is downloaded: a) directly from the Internet; b) through a simple forwarding proxy [47] and c) through SessionShield. Using `wget`, we downloaded the top 1,000 Internet websites [83] and measured the time for each.

In order to avoid network inconsistencies we downloaded the websites locally together with the HTTP headers sent by the actual Web servers. We used a fake DNS server that always resolved all domains to the "loopback" IP address and a fake Web server which read the previously-downloaded pages from disk and replayed each page along with its original headers. This allowed us to measure the time overhead of SessionShield without changing its detection technique, which relies on the cookie-related HTTP headers. It is important to point out that SessionShield doesn't add or remove objects in the HTML/JavaScript code of each page thus the page-rendering time isn't affected by its operation. Each experiment was repeated five times and the average page-download time for each method is presented in Fig. 2.5. SessionShield's average time overhead over a simple Python proxy is approximately 2.5 milliseconds and over a non-proxied environment is 5.4 milliseconds. Contrastingly, popular Web benchmarks show that even the fastest websites have an average page-download time of 0.5 seconds when downloaded directly from the Internet [88].

Since our overhead is two orders of magnitute less than the fastest page-download times we believe that SessionShield can be used by desktop and mobile systems without perceivable performance costs.

# 3 Secure Service Composition and Secure Service Platforms

Business compositions of the Future Internet are very dynamic in nature, and span multiple trust domains. We research service composition languages and service-oriented platforms for secure business compositions, as well as monitoring and analysis methods to enable assurance about the running business compositions.

In the course of the first year, partners have been working on three major elements, ranging from (1) general purpose service programming support, over (2) new types of security services that support trust establishment to (3) optimizations in the deployment and configuration of security services. These three themes establish the structure of this chapter.

In section 3.1, we discuss the work that creates a foundation for robust service composition, which can be a basis for services compositions with a reduced number of vulnerabilities. In fact this work describes an extended version of the call-by-value lambda-calculus. In section 3.2, we discuss new work that improves the potential to support trust management in open service systems and in section 3.3, we present new work on the optimal configuration and deployment of security services.

## 3.1 Towards enhanced type systems for service composition

We are developing a type system for open service networks. This type system can be characterized as a service oriented version of the call by value lambda-calculus. The focus on enabling open service networks is the main differentiator between this approach and related work.

This work (driven by CNR) is elaborated upon in the deliverable, as it is work-in-progress that has not been published elsewhere. Our longer term aim is to make the presented theory more practical and applicable to realistic examples.

### 3.1.1 Introduction

In the last decade, *history-based security* [2] has emerged as a powerful approach. Histories, i.e. execution traces, are sequences of relevant actions generated by running programs. Hence, a *security policy* consists in a distinction between accepted and rejected traces. Two main techniques using histories for deciding whether a program is secure or not are *run-time enforcement* and *static verification*. Briefly, policy enforcement is obtained by applying a monitor to the running program. Whenever, the monitored target tries to perform an action, the current trace is checked against the enforced policy. A security violation, i.e. an attempt to extend the actual trace to a forbidden one, activates some emergency operation, e.g. the program termination. Instead, static verification aims to prove the program safety for all possible execution scenarios. Roughly, this step is obtained by proving that, according to its structure, the program can only produce safe, i.e. policy-compliant, traces. Both of the above mentioned approaches have been widely studied (e.g. see [75, 19, 72, 59]).

Mixed approaches to security are also feasible. It has been shown that *local policies* [13] can be successfully exploited for this purpose. These policies are defined through *usage automata*, namely a variant of non-deterministic finite-state automata (NFAs) parameterized over system's resources. Policies compose each other through scope nesting and apply also to higher-order terms (e.g. mobile applications and service composition). The compliance with policies is statically verified through model checking. Programs passing this static check can run with no security monitors. Otherwise, the necessary policies are automatically turned into reference monitors to be associated with the guarded code (e.g. via instrumentation). In [12] Java has been extended with local policies showing the feasibility of this approach on a real-world system.

The benefits deriving from using local policies extend to service orchestration in closed networks [14, 15]. Indeed, service composition can be treated as a single, large scale service. If this composition is statically proved to comply with the existing policies, then the resulting service is secure.

In many cases we would like to also analyse *open* networks, i.e. networks having unspecified participants. The main motivation is that open networks seems to be everywhere. Additionally, we would like to build services bottom-up and incrementally to avoid having unspecified components. As a matter of fact, service-oriented paradigms aim to guarantee compositional properties and should be independent from the actual implementation of (possibly unknown) parties. Moreover, closed networks orchestrated by a global, composition plan require to be completely reorganized whenever a service rises or falls.

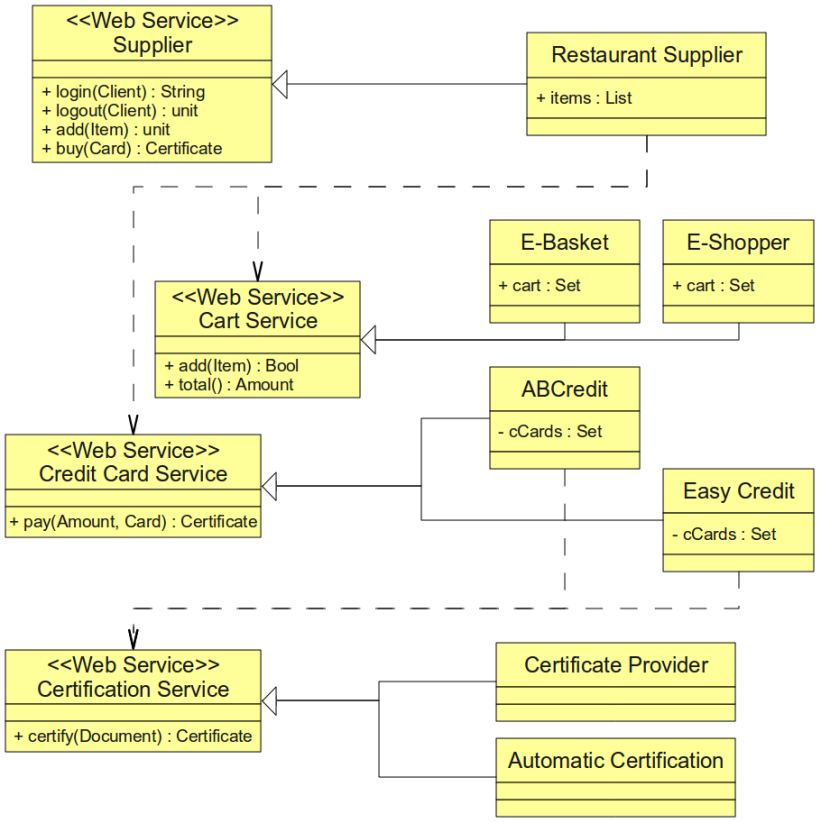



**Figure 3.1: The "Buy Something" scenario**

In [31] we extended the results of [14, 15] on networks that are open in the aforementioned sense. In particular, we singled out *partial* plans that involve parts of the known network and that can be safely adopted within any operating context. We say that a plan is partial if it is partially defined over the set of existing requests. For simplicity, we restricted our analysis to *stateless* security policies, i.e. they do not depend on the previous execution history, but only speak about the intended behaviour of called services.

A further improvement of this approach has been proposed in [30]. The main novelty was the application of partial evaluation techniques to the verification of security policies against composition plans. If enough information is available, this process exploits the partial plan of the network for verifying the security policies. Otherwise, the system turns the security policies into service *prerequisites*, i.e. simplified policies, which are checked dynamically.

### 3.1.2 A case study

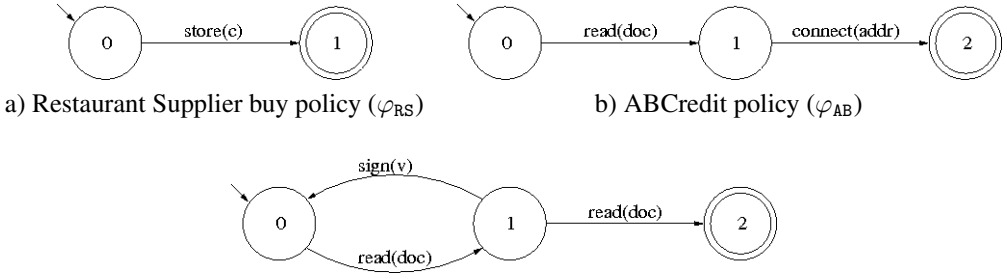
Below we present our working example which models the case study proposed in [87].

We specify it as the UML class diagram in Fig. 3.1. We consider a service network composed by two different class types. On the left, web services (*Web Service* stereotype) are specified through an operating interface, that is a complete specification of the service operations signatures.

The right side shows instances of services. Instances provide the actual implementation (solid line) of a service interface, possibly using some internal resource. When an instance implements a service it has to provide an implementation, namely a method, for each of the operations declared by the interface.

Dashed lines denote service composition relations, i.e. they connect a service instance to a service interface whenever one or more of the source operations implementations invoke (an operation of) the pointed service. Note that these relationships are not defined by the web service interface. Indeed, each service instance implementing a given interface must simply guarantee the interface functionalities.

The service interfaces involved in this case study are: *Supplier*, *Cart Service*, *Credit Card Service* and *Certification Service*. We briefly comment on them.




a) Restaurant Supplier buy policy ($\varphi_{\mathtt{RS}}$) b) ABCredit policy ($\varphi_{\mathtt{AB}}$)

c) Easy Credit policy ($\varphi_{\mathtt{EC}}$)

**Figure 3.2: Security policies**

A *Supplier* is a service offering operations for web-based selling services. These operations are:

- *login*: a client provides its identity (Client), receives back a welcome message (String) and starts a new session.
- *logout*: a client provides its identity (Client) and closes its current session.
- *add*: a client adds an item (Item) to its current set of items.
- *buy*: a client uses its credit card (Card) to buy the set of items that have been added during the session and receives back a certificated receipt.

A *Cart* Service provides basic support for a client electronic basket. Its operations are:

- *add*: an item (Item) is added to the cart.
- *total*: the value (Amount) of the current cart is returned.

The *Credit Card Service* offers a single, on-line payment operation returning a certified receipt (Certificate) whenever a certain amount (Amount) is charged on the client credit card (Card). Finally, the *Certification Service* simply takes a document (Document) and returns a signed version of it (Certificate).

The *Restaurant Supplier* is the only implementation of the *Supplier* class present in our network. Intuitively, it offers a public catalogue of items (items:List). Moreover, its methods invoke other services as part of their implementation. In particular, the *Restaurant Supplier* needs a *Cart Service* to handle the client's shopping sessions details and a *Credit Card Service* to perform payments.

The two implementations of the *Cart Service* are straightforward. Note that both of them have a local resource (cart) and they refer to no external service in their implementation.

Two alternative implementations are also present for the *Credit Card Service*. Again, these two instances have a local resource (Client). Moreover, these instances both require a *Certification Service* to sign their receipts.

Finally, two instances of the *Certification Service* are present.

Clearly, different service instances can focus on different security aspects and can require customised security policies. For instance, *Restaurant Supplier* can be interested in a policy saying "never store credit card details". Similarly, the instances of the *Credit Card Service* can specify requirements on the *Certifying Service* behaviour. Typical policies of this kind are "never open connections after reading the target document" or "do not read more than what is actually signed".

The usage automata in Fig. 3.2 correspond to the properties informally described above. Essentially, usage automata are finite state automata whose final states are considered as offending. These automata accept the execution traces that violate the policy they represent. Usage automata symbol alphabet changes according to which actions are considered security-relevant. Intuitively, usage automata define regular properties over events traces. A precise characterisation of usage automata and of their features is given in [16].

The network is intrinsically incomplete because there are no clients. However, the mentioned security policies do not involve any aspect of the possible clients behaviour. Indeed, all the services only define security constraints over their own traces and on those of the services they invoke. In particular, each service could be interested in verifying whether the interaction with others satisfies the existing policies or not.

A *usage automaton* $A_\varphi$ is a 5-tuple $\langle \mathsf{Ev}, Q, \imath, F, T \rangle$, where

- $\mathsf{Ev}$ is the input alphabet,
- $Q$ is a finite set of states,
- $\imath \in Q$ is the initial state,
- $F \subseteq Q$ is a set of final states,
- $T \in Q \times \mathsf{Ev} \times Q$ is a set of labelled transitions.

**Table 3.1: Definition of Usage Automaton.**

### 3.1.3 Service structure

Our programming model for service composition is based on $\lambda^{req}$, which was introduced in [14]. The syntax of $\lambda^{req}$ extends the classical call-by-value $\lambda$-calculus with two main differences: *security framing* and call-by-contract *service request*. Syntactically, security framing embraces a term and it represents the scope of a security policy. Instead, a request denotes the invocation to a remote service. In this section, we provide the reader with a detailed description of the $\lambda^{req}$ syntax.

Service networks are sets of services. A service $e$ is hosted in a location $\ell$, e.g. denoting a network address. We assume that there exists a trusted, public *service repository* `Srv` that contains references to available services. Abstractly, an element of `Srv` has the form $e_\ell : \tau \xrightarrow{H} \tau'$, where $e_\ell$ is the code of the service, $\ell$ is the unique location that hosts the service, $\tau \to \tau'$ is the type of $e_\ell$ and $H$ is its effect. Types represent a functional signature of the service in terms of input/output values. Clients requiring a service must specify its type and `Srv` returns an instance satisfying it. Instead, an effect $H$ represents the behaviour of the associated service mainly expressing the security-relevant events. In other words, the effect $H$ provides the clients with a behavioural contract of the side effects produced by a service that may affect security.

**Usage policies.** A usage policy governs the access to resources that we may wish to protect. A policy $\varphi$ is defined through a corresponding *usage automaton* $A_\varphi$. Usage automata are much like non-deterministic finite state automata (NFA). Briefly, a usage automaton consists of an input alphabet of events $\mathsf{Ev}$, a finite set of states $Q$, an initial state $\imath$, a set of final states $F$ and a set $T$ of transitions labelled by events.

In order to define the events, we assume as given a denumerable set of variables $\mathsf{Var}$, ranged over by $x, y, ...$, a finite set of resources $\mathsf{Res}$, ranged over by $\{r, r', r_1, ...\}$, a finite set of actions $\mathsf{Act}$, ranged over by $\alpha, \beta, \ldots$ These sets are pairwise disjoint. Also, we let $\dot{x}, \dot{y}$ range over $\mathsf{Res} \cup \mathsf{Var}$. Then, the events belong to $\mathsf{Ev} \subseteq \mathsf{Act} \times (\mathsf{Res} \cup \mathsf{Var})$, ranged over by $\alpha(\dot{x}), \beta(\dot{y}), \ldots$

The formal definition of usage automaton in given in Table 3.1.

Transitions are labelled with an event $\alpha(\dot{x})$, and we feel free to write $q \xrightarrow{\alpha(\dot{x})} q'$ instead of $\langle q, \alpha(\dot{x}), q' \rangle$). We say that usage automata are parametric over resources, because $\dot{x}$ can be a variable that will eventually be bound to an actual resource, so giving rise to an actual policy. Differently from [17], here we will allow for partial instantiations, because we wish to deal with open systems, and so some resource can be still unknown. A usage automaton is (partially) instantiated through a *name binding function* or *substitution*

$$\sigma : \mathsf{Res} \cup \mathsf{Var} \to \mathsf{Res} \cup \mathsf{Var}, \quad \text{such that } \forall\, r \in \mathsf{Res}.\, \sigma(r) = r$$

We understand that $\sigma$ is homomorphically applied to the usage automaton $A_\varphi$. We will sometimes write binding functions according to the following grammar

$$\sigma ::= \{\} \;\mid\; \{x \mapsto \dot{y}\} \cup \sigma'$$

where $x$ is a variable and $\dot{y}$ can be either a resource or a variable.

| | | |
|---|---|---|
| $e, e' ::=$ | $*$ | unit |
| | $r$ | resource |
| | $x$ | variable |
| | $\alpha(e)$ | access event |
| | $\texttt{if}\, g\, \texttt{then}\, e\ \texttt{else}\ e'$ | conditional |
| | $\lambda_z x.e$ | abstraction |
| | $e\, e'$ | application |
| | $\varphi[e]$ | security framing |
| | $\texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau'$ | service request |

**Table 3.2: The syntax of $\lambda^{req}$.**

Instantiating an automaton $A_\varphi$ with $\sigma$ gives back the automaton $A_\varphi(\sigma)$. Essentially, $A_\varphi(\sigma)$ has the same structure of $A_\varphi$ but its transitions set $T(\sigma)$ is defined to be

$$T(\sigma) = \{q \xrightarrow{\alpha(\sigma\, \dot{x})} q' \mid q \xrightarrow{\alpha(\dot{x})} q' \in T\}$$

A sequence of access events $\eta$ violates an instance of a usage automaton if it leads to a final state, also called *offending*. Hence, the accepted language is made of violating traces. So a trace $\eta$ *violates* $\varphi$ (in symbols $\eta \not\models \varphi$) whenever there exists a $\sigma$ such that $A_\varphi(\sigma)$ accepts $\eta$; otherwise, we say that $\eta$ *complies* with $\varphi$ (in symbols $\eta \models \varphi$).

From a language-theoretic point of view, we say that every instance of usage automata defines the upper bound of a class of regular languages over the parametric events alphabet. In symbols

$$\mathcal{L}(A_\varphi) = \bigcup_\sigma \mathcal{L}(A_\varphi(\sigma)) = \{\eta \mid \exists \sigma : \eta \in \mathcal{L}(A_\varphi(\sigma))\}$$

Hence, the compliance check between a trace $\eta$ and a usage automaton $A_\varphi$ corresponds to verifying whether $\forall\, \sigma.\, \eta \notin \mathcal{L}(A_\varphi(\sigma))$.

The following proposition states that instantiating a usage automaton we obtain a new automaton defining a less restrictive policy.

**Proposition.** For each usage automaton $A_\varphi$, mapping $\sigma$ and trace $\eta$

$$\eta \in \mathcal{L}(A_\varphi(\sigma)) \Longrightarrow \eta \in \mathcal{L}(A_\varphi)$$

**Proof** Straightforward from the definition of $\mathcal{L}(A_\varphi)$. □

**Syntax of services.** We introduce in Table 3.2 the syntax of $\lambda^{req}$. Similarly to the standard $\lambda$-calculus, syntactically correct $\lambda^{req}$ terms are the closed expressions, i.e. with no free variables, respecting the following grammar. We borrow most constructs from [14] and [17], but for simplicity we do not have a construct for creating resources like in [17].

The expression $*$ represents a distinguished, closed, event-free value. Resources, ranged over by $r, r'$, belong to finite set Res. Access actions $\alpha, \beta$ operate on resources giving rise to events $\alpha(r), \beta(r'), \ldots$ that actually are side effects.

Function abstraction (where $z$ in $\lambda_z x.e$ denotes the abstraction itself inside $e$) and application are standard. Note that here we use an explicit notation for conditional branching, the guards of which are defined below. This point will be further clarified in Section 3.1.4. Security framing applies the scope of a policy $\varphi$ to a program $e$. Service request requires more attention. We stipulate that services cannot be directly accessed by using a public name or address. Instead, clients invoke services through their public interface, i.e. their type and effect (see Section 3.1.4). A policy $\varphi$ is attached to the request in a call-by-contract fashion: the invoked service must obey the policy $\varphi$. Since both $\tau$ and $\tau'$ can be higher-order types, we can model simple value-passing interaction, as well as mobile code scenarios. Finally, the label $\rho$ is a unique identifier associated with the request, to be used while planning services.

$$g, g' ::= true \mid [\dot{x} = \dot{y}] \mid \neg g \mid g \wedge g' \qquad (\dot{x},\dot{y} \text{ range over variables and resources})$$

**Table 3.3: The syntax of guards.**

We use $v$ to denote values, i.e. resources, variables, abstractions and requests. Moreover, we introduce the following standard abbreviations: $\lambda x.e = \lambda_z x.e$ with $z \notin fv(e)$, $\lambda.e = \lambda x.e$ with $x \notin fv(e)$ and $e; e'$ for $(\lambda.e')e$.

The abstract syntax of guards is reported in Table 3.3.

Basically, guards can be either the constant $true$, a syntactic equality check between the variables and/or resources ($[\dot{x} = \dot{y}]$), a negation or a conjunction. All the variables of a guard are bound within it. This implies that the free variables of an expression do not include variables that only occur in guards. We use $false$ as an abbreviation for $\neg true$, $[\dot{x} \neq \dot{y}]$ for $\neg\,[\dot{x} = \dot{y}]$ and $g \vee g'$ for $\neg(\neg g \wedge \neg g')$. We also define an *evaluation function* $\mathcal{B}$ mapping guards into boolean values, namely $\{tt, ff\}$, as follows

$$\mathcal{B}(true) = tt \qquad \mathcal{B}([\dot{x} = \dot{x}]) = tt \qquad \mathcal{B}([\dot{x} = \dot{y}]) = ff \quad (\text{if } \dot{x} \neq \dot{y})$$

$$\mathcal{B}(\neg g) = \begin{cases} tt & \text{if } \mathcal{B}(g) = ff \\ ff & \text{otherwise} \end{cases} \qquad \mathcal{B}(g \wedge g') = \begin{cases} tt & \text{if } \mathcal{B}(g) = \mathcal{B}(g') = tt \\ ff & \text{otherwise} \end{cases}$$

Note that $\mathcal{B}$ is total, and that also guards containing variables can be evaluated (according to the second and third rules). In our model we assume resources to be uniquely identified by their (global) name, i.e. $r$ and $r'$ denote the same resource if and only if $r = r'$. In the following, we will use $[\dot{x} \in D]$ for $\bigvee_{d \in D}[\dot{x} = d]$.

**Operational semantics.** Clearly, the run-time behaviour of a network of services depends on the way they interact. As already mentioned, requests do not directly refer to the specific services that will be actually invoked during the execution, but to their abstract behaviour, i.e. to their type and effect (defined below), only. A *plan* resolves the requests by associating them with locations hosting the relevant services. Needless to say, different plans lead to different executions. A plan is said to be *valid* if and only if the executions it drives comply with all the active security policies. Of course, a service network can have many, one or even no valid plans.

A computational step of a program is a transition from a source configuration to a target one. In our model, configurations are pairs $\eta, e$ where $\eta$ is the execution *history*, that is the sequence of events done so far ($\varepsilon$ being the empty one), and $e$ is the expression under evaluation. Actually, the syntax of histories and expressions is slightly extended with markers $[^m_\varphi$ as explained below in the comment to the rule for framing. The automaton for a policy $\varphi$ will simply ignore these markers.

Formally, a plan is a (partial) mapping from request identifiers ($\rho, \rho', \ldots$) to service locations ($\ell, \ell', \ldots$) defined as

$$\pi, \pi' ::= \emptyset \mid \{\rho \mapsto \ell\} \mid \pi; \pi'$$

An empty plan $\emptyset$ is undefined for any request, while a singleton $\{\rho \mapsto \ell\}$ is only defined for request $\rho$ to be served by the service hosted at location $\ell$. Plan composition $\pi; \pi'$ combines two plans. It is defined if and only if for all $\rho$ such that $\rho \in dom(\pi) \cap dom(\pi') \Rightarrow \pi(\rho) = \pi'(\rho)$, i.e. the same request is never resolved by different services. Two such plans are called *modular*.

Given a plan $\pi$ we evaluate $\lambda^{req}$ expressions, i.e. services, according to the rules of the operational semantics given in Table 3.4. Actually, a transition should be also labelled by the location $\ell$ hosting the expression under evaluation. For readability, we omit this label.

Briefly, an event $\alpha(r)$ is appended to the current history (possibly after the evaluation of its parameter), a conditional branching chooses between two possible executions (depending on its guard $g$) and application works as usual (recall that $v$ is a value, i.e. either a resource, a variable, an abstraction or a request). The rule ($\mathtt{S{-}Frm_1}$) opens an instance of framing $\varphi[e]$ and records the activation in the history with a marker $[^m_\varphi$, and in the expression with $\varphi^m[e]$ (to keep different instantiations apart we use a fresh $m$ not occurring in the past history $\eta$). The rule ($\mathtt{S{-}Frm_2}$) simply deactivates the framing and correspondingly adds the proper marking $]^m_\varphi$ to the history. The rule ($\mathtt{S{-}Frm_3}$) checks whether the history at the right of the $m$-th instantiation of $\varphi$ respects this policy; in other words if the history after the activation of that specific instance does not violate $\varphi$. Recall that all "opening" and "closing" markers ($[^m_\varphi$

$$(\texttt{S}-\texttt{Ev}_1)\quad \frac{\eta, e \to_\pi \eta', e'}{\eta, \alpha(e) \to_\pi \eta', \alpha(e')} \qquad (\texttt{S}-\texttt{Ev}_2)\quad \eta, \alpha(r) \to_\pi \eta\alpha(r), *$$

$$(\texttt{S}-\texttt{If})\quad \eta, \texttt{if}\ g\ \texttt{then}\ e_{tt}\ \texttt{else}\ e_{ff} \to_\pi \eta, e_{\mathcal{B}(g)} \qquad (\texttt{S}-\texttt{App}_1)\quad \frac{\eta, e_1 \to_\pi \eta', e_1'}{\eta, e_1 e_2 \to_\pi \eta', e_1' e_2}$$

$$(\texttt{S}-\texttt{App}_3)\quad \eta, (\lambda_z x.e)v \to_\pi \eta, e\{v/x, \lambda_z x.e/z\} \qquad (\texttt{S}-\texttt{App}_2)\quad \frac{\eta, e_2 \to_\pi \eta', e_2'}{\eta, ve_2 \to_\pi \eta', ve_2'}$$

$$(\texttt{S}-\texttt{Frm}_1)\quad \eta, \varphi[e] \to_\pi \eta[^m_\varphi, \varphi^m[e] \quad m \text{ fresh} \qquad (\texttt{S}-\texttt{Frm}_2)\quad \eta, \varphi^m[v] \to_\pi \eta]^m_\varphi, v$$

$$(\texttt{S}-\texttt{Frm}_3)\quad \frac{\eta[^m_\varphi \eta', e \to_\pi \eta[^m_\varphi \eta'', e' \quad \eta'' \models \varphi}{\eta[^m_\varphi \eta', \varphi^m[e] \to_\pi \eta[^m_\varphi \eta'', \varphi^m[e']}$$

$$(\texttt{S}-\texttt{Req})\quad \frac{e_{\bar\ell} : \tau \xrightarrow{H} \tau' \in \texttt{Srv}\rfloor_\ell \qquad \pi(\rho) = \bar\ell \qquad H \models \varphi}{\eta, (\texttt{req}_\rho\ \tau \xrightarrow{\varphi} \tau')v \to_\pi \eta, e_{\bar\ell} v}$$

**Table 3.4: The operational semantics of $\lambda^{req}$.**

| | | |
|---|---|---|
| $H, H' ::=$ | $\varepsilon$ | empty |
| | $h$ | variable |
| | $\alpha(\dot{x})$ | access event |
| | $H \cdot H'$ | sequence |
| | $H + H'$ | choice |
| | $\varphi[H]$ | security framing |
| | $\mu h.H$ | recursion |
| | $gH$ | guard |

**Table 3.5: The syntax of history expressions.**

and $]^m_\varphi$) within a history are all distinct and that the usage automata skip them all. This is the way we implement our right-bounded local mechanism.

A service request firstly retrieves the service $e_{\bar\ell}$ that the current plan $\pi$ associates with $\rho$ within the repository $\texttt{Srv}$. We assume that services can not be composed in a circular way. This condition amounts to saying that there exists a partial order relation $\prec$ over services: read $\ell \prec \bar\ell$ as $\ell$ can see $\bar\ell$. So the rule says that the selected service must be within the sub-network that can be seen from the client (hosted at location $\ell$, the implicit and omitted label of the transition). This is rendered by the check $e_{\bar\ell} \in \texttt{Srv}\rfloor_\ell = \{e_{\ell'} : \tau \in \texttt{Srv} \mid \ell \prec \ell'\}$. Additionally, the effect of the selected service is checked against the policy $\varphi$ required by the client. In other words, the client verifies if its requirement $\varphi$ is met by the "contract" $H$ offered by the service. If successful, the service is finally applied to the value provided by the client, so implementing our "call-by-contract".

### 3.1.4 Type and effect system

We now introduce our type and effect system for $\lambda^{req}$. Our system builds upon [14, 15], aiming at better approximating service behaviour through more precise history expressions. For that, we introduce two new elements: effects with guards and a new typing rule that handles them. A key point is that guards generate invariants that we can exploit for validating the service network even when one or more resources are unspecified.

**History expressions** We statically check services to comply with given security policies. To do that we soundly over-approximate the behaviour of $\lambda^{req}$ programs by history expressions, that denote sets of histories. Table 3.5 contains the abstract syntax of history expressions, which extends those of [14] in the explicit treatment of guards.

$$\alpha(\dot{x}) \xrightarrow{\alpha(\sigma\dot{x})}_\sigma \varepsilon \qquad \frac{H \xrightarrow{a}_\sigma H'}{H \cdot H'' \xrightarrow{a}_\sigma H' \cdot H''} \qquad \frac{H \xrightarrow{a}_\sigma \varepsilon}{H \cdot H' \xrightarrow{a}_\sigma H'}$$

$$\frac{H \xrightarrow{a}_\sigma H'}{gH \xrightarrow{a}_\sigma H'} \ \sigma \models g \qquad \frac{H \xrightarrow{a}_\sigma H''}{H + H' \xrightarrow{a}_\sigma H''} \qquad \frac{H' \xrightarrow{a}_\sigma H''}{H + H' \xrightarrow{a}_\sigma H''}$$

$$\varphi[H] \xrightarrow{[_\varphi}_\sigma H\cdot]_\varphi \qquad ]_\varphi \xrightarrow{]_\varphi}_\sigma \varepsilon \qquad \frac{H\{\mu h.H/h\} \xrightarrow{a}_\sigma H'}{\mu h.H \xrightarrow{a}_\sigma H'}$$

$$[\![H]\!]^\sigma = \{a_1 \cdots a_n \mid H \xrightarrow{a_1}_\sigma \cdots \xrightarrow{a_n}_\sigma H_n\} \qquad n \geqslant 0;\ a_i \in \mathsf{Ev} \cup \{[_\varphi, ]_\varphi\}$$

**Table 3.6: The semantics of history expressions.**

A history expression can be empty ($\varepsilon$), a single access event to some resource ($\alpha(r)$ or $\alpha(x)$). Also, a history expression can be obtained through sequential composition ($H \cdot H'$) where we assume $H \cdot \varepsilon = \varepsilon \cdot H = H$. History expressions can be combined through non-deterministic choice ($H + H'$) and we feel free to consider "+" associative and to use sometimes the standard abbreviation $\sum_{i \in \{1,\ldots,k\}} H_i$, for a non-deterministic choice among $k$ history expressions. Moreover, we use safety framing $\varphi[H]$ for specifying that all the execution histories represented by $H$ are under the scope of the policy $\varphi$. Additionally, $\mu h.H$ (where $\mu$ binds the free occurrences of $h$ in $H$) represents recursive history expressions. Finally, we introduce guarded histories $gH$ (where the guard $g$ is defined according to the syntax of Definition 3.3).

The *operational semantics* function in Table 3.6 maps a history expression $H$ to a set of histories $\mathcal{H}$. Intuitively, the semantics of a history expression $H$ contains all the prefixes of all the traces that $H$ may perform. The traces of a history expression are produced according to the rules of a Labelled Transition System. We write $H \xrightarrow{a}_\sigma H'$ to denote that, under the substitution $\sigma$, $H$ performs $a$ and reduces to $H'$.

We now introduce the semantics of history expressions. Basically, each history expression denotes a set of histories.

The semantics operator $[\![\cdot]\!]^\cdot$ maps a history expression $H$ into a set of histories $\mathcal{H}$, where substitutions are the same as in Section 3.1.3.

As expected, an expression $\alpha(\dot{x})$ can transit to $\varepsilon$ while firing the event $\alpha(\sigma(\dot{x}))$ (recall that $\dot{x}$ stands for either a resource $r$ or a variable $x$). Sequential composition $H \cdot H'$ and non-deterministic choice $H + H'$ behave in the usual way. A policy framing $\varphi[H]$ adds a special event ($[_\varphi$) to denote the policy activation point and appends a corresponding deactivation event, i.e. $]_\varphi$, at the end of the scope of the policy. The history expression $gH$ behaves as $H$ if $\sigma$ *satisfies* $g$ (in symbols $\sigma \models g$). Satisfiability of a guard $g$ through a substitution $\sigma$ is equivalent to check whether $\mathcal{B}(\sigma g) = tt$ (remember that $\mathcal{B}$ is total). Finally, $\mu h.H$ behaves as $H$ where all the free instances of $h$ have been replaced by $\mu h.H$.

### 3.1.5 Typing relation

We define types and type environments in the usual way. Type environments are defined in a standard way as mappings from variables to types. Types can be either base types, i.e. unit or resources, or higher-order types $\tau \xrightarrow{H} \tau'$ annotated with the history expression $H$. Resources belong to subsets $R, R', \ldots$, e.g. $Bool$ and $Card$ in several examples in this paper. Note that, as expected, service interfaces (see Section 3.1.3) published on the service repository `Srv` are simply higher-order types annotated with a location identifier $\ell$.

A typing judgement has the form $\Gamma, H \vdash_g e : \tau$ and means that the expression $e$ is associated with the type $\tau$ and the history expression $H$. The guard $g$ labelling the $\vdash$ records information about the branchings passed through during the typing process. Table 3.7 shows the definitions of types, type environment and the type and effect system.

We require the repository to be trusted — this is the only trust requirement we put on our framework. Consequently, to be included in `Srv`, services must be well-typed with respect to our type and effect system, i.e.

$$e_\ell : \tau \xrightarrow{H} \tau' \in \mathtt{Srv} \Longrightarrow \emptyset, \varepsilon \vdash_{tt} e_\ell : \tau \xrightarrow{H} \tau'$$

$$\tau, \tau' ::= \mathbf{1} \mid R, R' \subseteq \mathsf{Res} \mid \tau \xrightarrow{H} \tau' \qquad \Gamma, \Gamma' ::= \emptyset \mid \Gamma; x : \tau$$

---

$$(\texttt{T-Unit})\ \Gamma, \varepsilon \vdash_g * : \mathbf{1} \qquad (\texttt{T-Res})\ \Gamma, \varepsilon \vdash_g r : R \subseteq \mathsf{Res} \qquad (\texttt{T-Var})\ \Gamma, \varepsilon \vdash_g x : \Gamma(x)$$

$$(\texttt{T-Abs})\ \frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash_g e : \tau'}{\Gamma, \varepsilon \vdash_g \lambda_z x.e : \tau \xrightarrow{H} \tau'} \qquad (\texttt{T-Ev})\ \frac{\Gamma, H \vdash_g e : R}{\Gamma, H \cdot \sum_{r \in R} \alpha(r) \vdash_g \alpha(e) : \mathbf{1}}$$

$$(\texttt{T-App})\ \frac{\Gamma, H \vdash_g e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash_g e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash_g e\, e' : \tau'} \qquad (\texttt{T-Frm})\ \frac{\Gamma, H \vdash_g e : \tau}{\Gamma, \varphi[H] \vdash_g \varphi[e] : \tau}$$

$$(\texttt{T-Wkn})\ \frac{\Gamma, H \vdash_g e : \tau \quad \sigma \models g \quad [\![H]\!]^\sigma \subseteq [\![H']\!]^\sigma}{\Gamma, H' \vdash_g e : \tau}$$

$$(\texttt{T-If})\ \frac{\Gamma, H \vdash_{g \wedge g'} e : \tau \quad \Gamma, H \vdash_{g \wedge \neg g'} e' : \tau}{\Gamma, H \vdash_g \texttt{if}\ g'\ \texttt{then}\ e\ \texttt{else}\ e' : \tau} \qquad (\texttt{T-Str})\ \frac{\Gamma, H \vdash_g e : \tau \quad g \Rightarrow g'}{\Gamma, g' H \vdash_g e : \tau}$$

$$(\texttt{T-Req})\ \frac{I = \{H \mid e_{\ell'} : \tau \xrightarrow{H} \tau' \in \texttt{Srv}\rfloor_\ell \wedge H \models \varphi\}}{\Gamma, \varepsilon \vdash_g \texttt{req}_\rho\, \tau \xrightarrow{\varphi} \tau' : \tau \xrightarrow{\sum_{H \in I} H} \tau'}$$

**Table 3.7: Types, type environment and typing relation.**

We briefly comment on the typing rules. We borrow most of them from [14], except for those dealing with guards, namely (T-If), (T-Str) and (T-Wkn). The rules (T-Unit, T-Res, T-Var) for $*$, resources and variables are straightforward. An event has type $\mathbf{1}$ and produces a history that is the one obtained from the evaluation of its parameter increased with the event itself (T-Ev). Note that, since the set of resources is finite, an event only has finitely many instantiations.

An abstraction has an empty effect and a functional type carrying a *latent effect*, i.e. the effect that will be produced when the function is actually applied (T-Abs). The application moves the latent effect to the actual history expression and concatenates it with the actual effects according the call-by-value semantics (T-App). Security framing extends the scope of the property $\varphi$ to the effect of its target (T-Frm).

The rule for conditional branching says that if we can type $e$ and $e'$ to the same $\tau$ generating the same effect $H$, then we can extend $\tau$ and $H$ to be the type and effect of the whole expression (T-If). Moreover, in typing the sub-expressions we take into account the guard $g'$ and its negation, respectively. Indeed, the rule requires that the expression $e$ has got type $\tau$ under the guard $g \wedge g'$, as $e$ will only be executed if $g'$ is true. Of course also the conditions collected so far in $g$ should be true, which in turn enable the whole $\texttt{if}\ g'\ \texttt{then}\ e\ \texttt{else}\ e'$ to occur at run-time. Symmetrically for $e'$.

Similarly to abstractions, service requests have an empty effect (T-Req). However, the type of a request is obtained as the composition of all the types of the possible servers. In particular, the resulting latent effect is the (unguarded) non-deterministic choice among them. Note that we only accept exact matching for input/output types. A more general approach would require to define a sub-typing relation. However, such an extension is out of the scope of this work and it is easy, following the proposal of [14], to which we refer the interested reader for details.

The last two rules are for weakening and strengthening. The first (T-Wkn) states that is always possible to make a generalisation of the effect inferred from an expression $e$, possibly losing precision. In particular, we say that in typing $e$ a history expression $H$ can be replaced by $H'$ provided that $H'$ denotes a superset of the histories denoted by $H$ under any possible environment $\sigma$, provided that it satisfies $g$. Finally, (T-Str) applies a guard $g'$ to an effect $H$ provided that $\forall\, \sigma . \sigma \models g \Rightarrow \sigma \models g'$, abbreviated by $g \Rightarrow g'$. This rule says that we can use the information stored in $g$ for wrapping an effect in a more precise guarded context.

Our type and effect system produces history expressions that approximate the run-time behaviour of programs.

The soundness of our approach relies on producing *safe* history expressions, i.e. any trace produced by the execution of an expression $e$ (under any valid plan) is denoted by the history expression obtained typing $e$. To prove type and effect safety we need the following results. The former states that type and effect inference is closed under logical implication for guards, the latter says that history expressions are preserved by programs execution.

**Property.** If $\Gamma, H \vdash_g e : \tau$ and $g' \Rightarrow g$ then $\Gamma, H \vdash_{g'} e : \tau$

The histories denoted by history expressions are slightly different from those produced at runtime. To compare histories of these two kinds, we introduce below the operator $\partial$, that removes labels from markers of framing events:

$$\varepsilon^\partial = \varepsilon \qquad (\alpha(\dot{x})\eta)^\partial = \alpha(\dot{x}) \cdot (\eta^\partial) \qquad ([^m_\varphi \eta)^\partial = [_\varphi \eta^\partial \qquad (]^m_\varphi \eta)^\partial = ]_\varphi \eta^\partial$$

**Property.** Let $\Gamma, H \vdash_g e : \tau$ and $\eta, e \rightarrow^*_\pi \eta', e'$. For each $g'$ such that $g' \Rightarrow g$, there exists $H'$ such that $\Gamma, H' \vdash_{g'} e' : \tau$ and $\forall \sigma . \sigma \models g' \Longrightarrow (\eta' [\![H']\!]^\sigma)^\partial \subseteq (\eta [\![H]\!]^\sigma)^\partial$

Now, the soundness of our approach is established by the following theorem, where we overload $\eta$ to denote both a history generated by the operational semantics of an expression $e$ (i.e. possibly containing framing markers), and a history belonging to the denotational semantics of a history expression $H$ (i.e. without markers).

**Theorem.** If $\Gamma, H \vdash_{true} e : \tau$ and $\varepsilon, e \rightarrow^*_\pi \eta', v$, then $\forall\ \sigma.\ \exists\ \eta \in [\![H]\!]^\sigma$ such that $\eta = (\eta')^\partial$.

We now define the notion of validity for a history expression $H$. The validity of $H$ guarantees that the expression $e$, from which $H$ has been derived, will raise no security violations at runtime.

**Definition.** A history $\eta$ is *balanced* iff it is produced by the following grammar.

$$B ::= \varepsilon \ \mid\ \alpha(\dot{x}) \ \mid\ [_\varphi B]_\varphi \ \mid\ BB'$$

$H$ is *valid* iff $\forall\, \sigma$ and $\forall\, \eta [_\varphi \eta' ]_\varphi \eta'' \in [\![H]\!]^\sigma$ (with $\eta'$ balanced) then $\eta' \models \varphi$.

The type and effect system of [14] has no rule for strengthening like our rule (T-Str). The presence of this rule in our system makes it possible to discard some of the denoted traces. These traces correspond to executions that, due to the actual instantiation of formal parameters, can not take place. Consequently, our type and effect system produces more compact and precise history expressions than those of [14]. This enables the verification algorithm to run faster and to produce fewer false positives, so improving both the efficiency and effectiveness of the original method.

As a matter of fact, the validity of a history expression $H$ is established through model-checking. Roughly, the model-checking procedure extracts each policy instance from a program. Then, the model-checker verifies whether the set of traces violating the policy and the set of histories denoted by expression in its scope have an empty intersection. If they share no traces, the program cannot violate the policy. This problem is known to be polynomial in the size of the history expression $H$. We refer the interested reader to [18].

### 3.1.6 Conclusion

We extended the type system of $\lambda^{req}$, a service-oriented version of the call-by-value $\lambda$-calculus, with rules for inferring guarded history expressions. Our approach produces a finer, but still safe, static approximation of the possible run-time behaviours than those of [13]. Although our type system performs similarly to others in case of closed service networks, we showed that it can be successfully applied to partially undefined networks of services. This assumption seems to be realistic. Indeed an "a priori" knowledge of clients is not always available to services at deploy-time. On the contrary, a service could be interested in verifying which compositions are viable only considering the part of the service network it is aware of at a certain time.

This research line seems to be promising and we are currently working further improvements. Mainly, we are investigating what are the minimal conditions under which partial plans are fully compositional. Indeed, such a result would allow for applying "divite et impera" strategies to the orchestration problem. Distributing the complexity of finding valid compositions over (small groups of) services would make the orchestration problem scale allowing for a faster verification of dynamic compositions.

## 3.2 Security Support in Trusted Services

### 3.2.1 Support for interoperable trust management

So far, few researchers seem to have investigated interoperability between heterogeneous trust models. The work in [80] describes a trust management architecture that enables dealing with multiple trust metrics and that supports

the translation of such metrics. However, the (limited) solution only deals with the composition at the level of trust measures (i.e. values). The problem of dealing with a variety of trust models has not been addressed. For example, it would be useful to differentiate between direct trust values and reputation-based values. There is a need to formalize heterogeneous trust models and their composition. Such a concern is in particular addressed in [55, 84], which introduce the notion of trust meta-models, starting from state of the art trust management systems. Nevertheless, it remains unclear how the proposed meta-model facilitates composing heterogeneous trust models and how it enables interoperability. Dealing with the heterogeneity of trust models is also investigated in [38, 81] but limited to reputation-based models. In short, the literature is increasingly rich when it comes to discussing and presenting trust models, yet dealing with their composition remains a challenge.

This work introduces a comprehensive approach based on the definition of a reference trust meta-model. Specifically, based on the state of the art, the trust meta-model formalizes the core entities of trust management systems, i.e., trust roles, metrics, relations and operations. The trust meta-model then serves specifying the composition of trust models in terms of mapping rules between roles, from which trust mediators can be synthesized. Trust mediators transparently implement mapping between respective trust relations and operations of the composed models. While this work introduces the composition approach from a theoretical perspective, next steps address the implementation, partially as part of the CONNECT project[1]. The current results have been created by INRIA; this work has been published in [69]. In the second project year, the consortium will evaluate how these facilities can be integrated in extensive service architecture for Future Internet.

### 3.2.2 Embracing social networks

A second important activity in this space is addressing the support of social networks in an open service ecosystem. Clearly concerns such as trust establishment, and also the expression and enforcement of access control policies are posing new challenges. Initial results that are driven by these observations have been published in [44] and [74].

## 3.3 Service Engineering: from Composition to Configuration

In the context of large-scale distributed service systems, security and performance turn out to be two competing requirements. They are conflicting and cannot be optimized simultaneously. Assuming we have appropriate security services in places (be it standard, well known facilities such as authentication services, or more advanced facilities such as the one referred to in the previous section). No matter what the policies are and what the stakeholder concerns are, we need to make trade-offs when deploying and configuring such service.

For example, consider the enforcement of authorization policies in a compound distributed service. The caching of data that are needed for security decisions can lead to security violations when these data changes fast without the cache being able to refresh it in time. Retrieving fresh data without leveraging on caching techniques impacts performance while combining fresh data with cached data can affect both security and performance. In principle, and probably in the ideal (hypothetical) case, performance and security are addressed separately (aiming for separation of concerns). This will usually leads to fast systems with security holes, rather than secure systems with poor performance.

We have examined how to dynamically configure an authorization system to an application that needs to be fast and secure. We have studied data caching, attribute retrieval and correlation. We have incepted a runtime management tool that can find and enact the configurations that enhance both security and performance needs. This obviously requires the input from experts. Our initial work illustrates that it takes a few seconds to find a configuration solution in a large-scale setting with over one thousand authorization components. This work is based on collaboration between UNITN and KUL. Initial architectures have been published in [42]. The new results will appear in [40].

[1] http://connect-forever.eu/

# 4 Enhanced programming language support

Being able to securely compose services is an important requirement to have a safe future internet, but it only solves half of the problem. An additional prerequisite is that the individual services must also be secure. If an attacker can exploit an individual service, composing this service with other services can break the security of the entire aggregate. It is therefore of crucial importance to make sure that services can be implemented in a secure way.

Thus, in order to solve the bigger picture, the consortium also works on securing the native components that make up the different services. In this section, we focus on popular languages such as Java and C, for which a number of common security vulnerabilities are known. However, our work is universally applicable. This section focuses on VeriFast, work in progress towards a sound and modular program verifier for common programming languages such as C and Java. VeriFast takes as input a number of source files, annotated with method contracts written in separation logic, inductive data type and fixpoint definitions, lemma functions and proof steps. The verifier checks that (1) the program does not perform illegal operations such as dividing by zero or illegal memory accesses, and (2) that the assumptions described in method and contracts hold in each execution.

Section 4.1 sets the scene and stresses why software verification is very useful, if not necessary. Section 4.2 defines the different building blocks that are used in VeriFast. Section 4.3 summarizes an evaluation of the current implementation of VeriFast, and finally Section 4.4 gives an overview of the ongoing and future work. In this context, we also want to highlight the potential to use this technology for relational logic proofs, i.e. proofs that aim for guarantees that pairs of program executions obey certain properties. This can be relevant - if successful - when information leakage is at stake.

## 4.1 Introduction

One absolute truth about software is that programmers make mistakes during the development process. These software bugs can give rise to subtle problems, but may sometimes result in disastrous catastrophes. One example is the crash of the Ariane 5 rocket, where a simple integer overflow lead to the destruction of this $370 million dollar machine. Another example is the Toyota recall due to a software bug in the car computer system. Some of these bugs are also so-called vulnerabilities. A vulnerability is a bug that can be exploited by an attacker to make the software do something it's not supposed to do. Most software contains such vulnerabilities, as demonstrated by Microsoft's monthly security updates.

VeriFast[49] (partially) solves these problems by proving that software does not contain errors such as unallocated memory access, null pointer dereferences, out of bound errors, data races, and ensures compliance with API contracts. In addition, compliance with the application's own specifications can also be verified. However, these advantages do not come for free. VeriFast tries to reason about the application as much as it can, but some programmer interaction is required in order for VeriFast to be able to fully verify an application. Programmers must *annotate* their source code with method pre- and postconditions, invariants, mathematical datatype and function definitions, recursive memory structure definitions, inductive proofs, and potentially some proof steps. This annotated source code can then be processed by VeriFast, which will result in one of two outcomes. Either the program verifies, and then a mathematical proof exists that the application does not have any of the issues mentioned before. However, in the case that the program does not verify, an explanation of why the proof could not be constructed is given to the programmer.

A feature that proved to be crucial in understanding failed verification attempts is VeriFast's symbolic debugger. As shown in Figure 4.1, the symbolic debugger can be used to diagnose verification errors by inspecting the symbolic states encountered on the path to the error. For example, if the tool reports an array indexing error, one can look at the symbolic states to find out why the index is incorrect. This stands in stark contrast to most verification condition generation-based tools that simply report an error, but do not provide any help to understand the cause of the error.

The symbolic debugger consists of a number of smaller windows that help the programmer determine where the verification failed. The main subwindow contains the source code of the program that is being verified. VeriFast indicates where the verification of the application failed by selecting and coloring the relevant parts of the source code. All the steps that were required to reach the result are shown in the *Steps* window. The programmer can use this window to select prior verification steps, and thus step back into the symbolic history of the verification algorithm. The *assumptions* window lists all the assumptions that VeriFast can deduce at the selected verification step. The *heap chunks* window offers a view on the symbolic heap; it contains all the elements that are allocated in symbolic memory at the selected verification step. Finally, there is the *locals* window that shows all the local

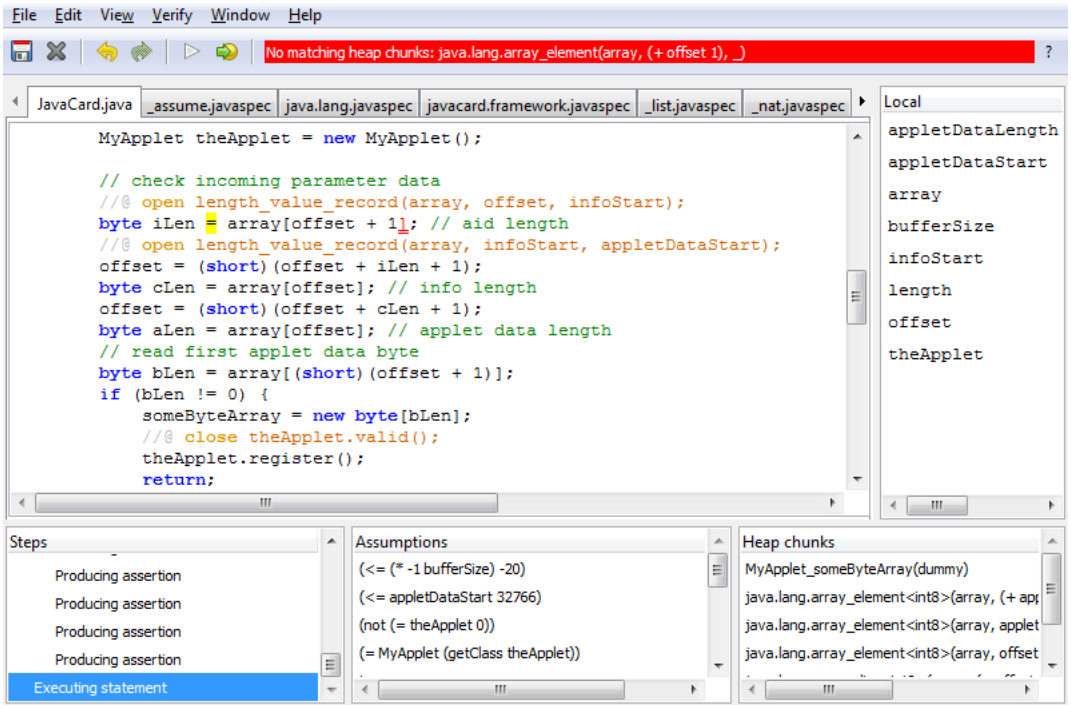



**Figure 4.1: The symbolic debugger of VeriFast**

variables and their symbolic values.

## 4.2 Building Blocks

In order to be able to successfully verify an application, the programmer must write annotations in his source code. These annotations can describe one of several *building blocks* that is used by the VeriFast approach. This section gives a short overview of these building blocks. More information can be found in [50, 51].

**Method Contracts** Developers can specify the behavior of a function via a method contract consisting of two assertions, a precondition and a postcondition. Both assertions are written in a form of separation logic. When VeriFast tries to verify a method, it will start by populating the symbolic state with everything that is described in the precondition of the method. VeriFast will then symbolically execute the function and will verify that for each possible execution the postcondition of the method is reached.

**Inductive Data Types** VeriFast supports inductive data types to allow developers to specify rich properties. To describe recursive data structures and to allow for information hiding, VeriFast supports separation logic predicates. A predicate is a named assertion. Predicates can be precise, which means that their input parameters uniquely determine (1) the structure of the heap described by those predicates, and (2) the values of the output parameters. Input and output parameters are separated by a semicolon. VeriFast tries to automatically fold and unfold precise predicates whenever necessary. Developers can also insert explicit fold (**close**) and unfold (**open**) proof steps in the form of ghost commands for non-precise predicates or when the automatic (un)folding does not suffice.

**Fixpoint Functions** VeriFast also supports fixpoint functions. Just like predicates and inductive data types, fixpoint functions can only be mentioned in specifications, not in the source code itself. The body of a fixpoint function must be a switch statement over one of the fixpoint's inductive arguments. To ensure soundness of the encoding of fixpoints, VeriFast checks that fixpoints terminate. In particular, VeriFast enforces that whenever a fixpoint $g$ is called in the body of a fixpoint $f$ that either $g$ appears before $f$ in the program text or that the call decreases the size of an inductive argument.

**Lemma Functions** Lemma functions allow developers to prove properties of their inductive data types, fixpoints and predicates, and allow them to use these properties when reasoning about programs. A lemma is a function without side-effects marked **lemma**. The contract of a lemma function corresponds to the property itself, its body to the proof and a lemma function call corresponds to an application of the property. VeriFast has two types of lemma functions: pure and spatial lemmas. A pure lemma is a function whose contract only contains pure assertions, and whose body proves that the precondition implies the postcondition. Spatial lemmas can mention spatial assertions such as predicates and points-to assertions. A spatial lemma with precondition $P$ and postcondition $Q$ states that the program state described by $P$ is equivalent to the state described by $Q$. A spatial lemma call does not modify the underlying values in the heap, but changes the symbolic representation of the program state.

## 4.3 Evaluation

In order to evaluate VeriFast, a case study was performed to see how practical it is to use VeriFast to annotate Java Card applets that are more than a toy project. It gives us an idea of how much the annotation overhead is, where we can improve the tool, and whether we can actually find bugs in existing code using this approach. A full description of the case study can be found in [68].

### 4.3.1 Annotation Overhead

The more information the developer gives in the annotations about how the applet should behave, the more VeriFast can prove about it. It is up to the developer to choose whether he wants to use VeriFast as a tool to only detect certain kinds of errors (unexpected exceptions and incorrect use of the API), or whether he wants to prove full functional correctness of the program. Both modi operandi are supported by the tool. For the case study, we used two large applets. For one applet, we used the annotations to prove that the applet does not contain transaction errors, performs no out of bounds operations on buffers, and never dereferences null pointers. For the other applet, the annotations supported the same guarantees, as well as a functional verification of the program. On average, about one line of annotations for each two lines of source code was required.

Another type of annotation overhead is the time it took to actually write the annotations. The verification of the first applet was performed by a senior software engineer without prior experience with the VeriFast tool, but with regular opportunities to consult VeriFast expert users during the verification effort. We did not keep detailed effort logs, but a rough estimate of the effort that was required is 20 man-days to annotate 1,500 lines of code. This includes time spent learning the VeriFast tool and the Java Card API specifications. The other applet was annotated by a VeriFast specialist and took about 5 man-days for 350 lines of code, excluding the time it took to add some new required features to the tool.

### 4.3.2 Bugs and Other Problems

A number of problems were discovered in the applets. The first applet contained 25 locations where the smart card state could become inconsistent, three potential null pointer dereferences, three potential variable casting problems, and seven potential out of bounds errors.

The second applet had already been verified using another verification technology, so it was not very surprising that no functional problems were found in the code. However, VeriFast *did* identify four locations in the code where transactions were not used properly. Transactional safety is a property that the other tool did not verify.

### 4.3.3 VeriFast Strengths

Compared to other program verifiers, VeriFast has two advantages: speed and soundness. That is, VeriFast usually reports in only a couple of seconds (usually less) whether the application is correct or whether it contains a potential bug. Secondly, if VeriFast deems a program to be correct, then that program is guaranteed to be free from unexpected exceptions, API usage and assertion violations.

## 4.4 Future Work

VeriFast is a work in progress. The work that will be performed in the context of the NESSoS NoE will build on the work that has been done in the previous years. Some future work topics include:

1. Reducing annotation overhead by automatically inferring more proof steps, while preserving responsiveness and diagnosability. This includes the investigation of integrating automatic approaches such as Abductor[24].
2. Simplifying the verification of Java Card transactions and transient arrays.
3. Simplifying the verification of fine-grained concurrent data structures by developing logics that allow more elegant proofs.
4. Improving the interactive annotation insertion experience, by improving the understandability of the symbolic trace and symbolic state representation.
5. Investigating the specification and verification of full functional correctness of entire programs (not just individual data structures).
6. Implementing full programming language support, both syntactically (parse rules) and semantically (e.g. fully axiomatizing the semantics of bitwise operations).

Clearly many ways of going forward have been identified. In the next two quarters, the consortium will identify priorities in light of the NESSoS specific objectives. For example, links to the verification of web technologies are being investigated.

# 5 Information-Flow for Secure Services

Information-flow policies impose constraints on how information entering a program can flow to outputs of the program. For instance, such policies can limit how public output can depend on confidential input, or how high integrity output can be influenced by low integrity input. Information-flow analysis is particularly relevant in the context of web applications that are composed of components from potentially different sources with mutual distrust. Initially, we have positioned this work in the service composition task, yet the ongoing research is more general and relates to the work package as a whole. We restrict ourselves to sketching initial results. The full papers [11, 6] are available in the appendix (section A).

## 5.1 Introduction

A baseline policy for information-flow security is non-interference: given a labeling of input and output channels as either confidential (high, or H) or public (low, or L), a (deterministic) program is non-interferent if there are no two executions with the same public inputs (but different confidential inputs) that lead to different public outputs. This definition of non-interference can be generalized from the two security levels H,L to an arbitrary partially ordered set of security levels.

Enforcing non-interference is a challenging problem that has received substantial attention from the research community for many decades. The most prominent class of enforcement techniques are based on static analysis. Static techniques include the use of security type systems [85, 46, 65], or program verification-based approaches [10]. While impressive progress has been made over the past decades - still - the static enforcement of non-interference is often overly restrictive in the sense that many intuitively secure programs are rejected. The reason for this is that (i) non-interference is too strong a property for many practical applications, and (ii) that the absence of run-time information forces sound security analyses to assume the (however rare) worst-case scenario. These limitations provide major obstacles for the wider use of information-flow analysis in future web-applications. In this project, we address these limitations along two lines. First, we propose novel techniques for dynamic information-flow analysis. The goal is to make information-flow analysis more permissive by leveraging the information available at run-time. Second, we propose novel techniques for quantitative information-flow analysis. Quantitative information-flow analysis is based on quantitative generalizations of non-interference; it enables one to certify the security of programs that leak only negligible amounts of information. This clearly is early work, i.e. work in progress in which IMDEA and KUL collaborate and share part of the load. This deliverable highlights the results we aim for, yet it is still in the making.

## 5.2 Dynamic Information-Flow Analysis

The last few years have seen a renewed interest in *dynamic* techniques for information-flow analysis. Several authors [43, 70, 5, 27] have developed run time monitors that can provably guarantee non-interference. At the price of some impact on performance, these monitors can be more permissive than static methods, and they require less annotation effort. Next to run time monitors, an alternative class of dynamic techniques is based on *secure multi-execution (SME)* [35, 25]. The core idea here is to guarantee non-interference by executing multiple copies of the program (one per security level), and to ensure that the copy at level $l$ only outputs to channels at level $l$, and that it only gets access to inputs from channels that are below or equal to $l$. Secure multi-execution based enforcement has been implemented at the level of the operating system [25], virtual machine [35], browser [20] or library [52]. At an even higher performance cost, it offers perfect permissiveness (no false positives), and no need for annotations of the code.

However, an important disadvantage of the existing implementation techniques for SME is that they all require modifications to the underlying computing infrastructure (OS, browser, virtual machine, trusted libraries), and this makes it hard to deploy these techniques in scenarios where that infrastructure is distributed and heterogeneous. In particular, this makes it hard to deploy SME on the web.

### 5.2.1 Secure multi-execution through static program transformation

In the course of this project, we are developing a new implementation technique for SME, based on static program transformation. We have proved the correctness of our technique for a model language that includes dynamic code

evaluation. Moreover, we intend to provide a prototype implementation for JavaScript.

**Example: JavaScript advertising** JavaScript code is used in web applications to make it possible to perform client-side computations. In many scenarios, the fact that scripts run with the same privileges as the website loading the script leads to security problems. One important example are advertisements; these are commonly implemented as scripts, and in the absence of security countermeasures (for example, WebJail, section 2.1) such scripts can leak all information present in the web page that they happen to be part of.

JavaScript advertisments are a challenging application area for information flow security, as they may need some access to the surrrounding web page (to be able to provide context-sensitive advertising), and as they are sometimes implemented using dynamic code evaluation (for instance by using JavaScript's `eval()` function).

The following code snippet shows a very simple context-sensitive advertisement in JavaScript.

```
var keywords = document.getElementById("keywords").textContent;
var img = document.getElementById("adimage");
img.src = 'http://adprovider.com/SelectAd.php?keywords='+keywords
```

Line 1 looks up some keywords in the surrounding web page; these keywords will be used by the ad provider to provide a personalized, context-sensitive advertisement. Line 2 locates the element in the document in which the advertisement should be loaded, and finally line 3 generates a request to the advertisement provider site to generate an advertisement (in the form of an image) related to the keywords sent in the request.

Obviously, a malicious advertisement can easily leak any information in the surrounding page to the ad provider or to any third party. Here is a simple malicious ad that leaks the contents of a password field to the ad provider:

```
// Malicious: steal a password instead of keywords
var password = document.getElementById("password").textContent;
var img = document.getElementById("adimage");
img.src = 'http://adprovider.com/SelectAd.php?keywords='+password
```

Information-flow security enforcement can mitigate this threat: if one labels the keywords as public information and the password as confidential information, then (treating network output as public output) enforcing non-interference will permit the non-malicious ad, but block the malicious one.

The example ad script above loads an image from a server at a third-party server. Instead of loading an image, it could also load a script from the server that can then render the ad and further interact with the user (e.g. make the advertisement react to mouse events). In the example below, we illustrate the essence of this technique using the `XMLHttpRequest` API and JavaScript `eval()`.

```
var keywords = document.getElementById("keywords").textContent;
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET', 'http://adprovider.com/getAd.php?keywords='+keywords, false);
xmlhttp.send(null);
eval( xmlhttp.responseText)
```

Lines 2-4 send the keywords to the ad provider, and expect a (personalized) script in response. Line 5 then evaluates the script that was received.

This possibility of dynamically generating or loading new code and evaluating it on the fly further complicates the enforcement of information flow security policies.

The enforcement mechanism we have developed in this project provides effective protection against these security problems of malicous scripts. We propose a program transformation that transforms any script into a script that (1) is guaranteed to be non-interferent, and (2) behaves identical to the original script if that script was non-interferent to start with.

## 5.3 Quantitative Information-Flow Analysis

Confidentiality is a property that captures that no secret information is exposed to unauthorized parties; it is one of the most fundamental security properties and an essential requirement for most security-critical applications.

Unfortunately, perfect confidentiality is often difficult or even impossible to achieve in practice. In some cases, perfect confidentiality is in conflict with the functional requirements of a system. For example, the result of a statistical query on a medical database necessarily reveals some information about the individual entries in the database. In other cases, perfect confidentiality is in conflict with non-functional requirements such as bounds on the resource-usage. For example, variations in the execution time of a program may reveal partial information about the program's input; however a (perfectly secure) implementation with constant execution time may have inacceptable performance.

Because such conflicting requirements are ubiquitous, there is a need for tools that enable formal reasoning about imperfect confidentiality. Quantitative approaches to confidentiality can provide such tools: first, quantitative notions of confidentiality can express a continuum of degrees of security, making them an ideal basis for reasoning about the trade-off between security and conflicting requirements such as utility [36] or performance [57]. Second, despite their flexibility, a number of quantitative notions of confidentiality are backed up by rigorous operational security guarantees such as lower bounds on the effort required for brute-forcing a secret.

### 5.3.1 Quantitative measures of confidentiality

While convergence has been achieved for definitions of perfect confidentiality (they are subsumed under the cover term *non-interference* and differ mainly in the underlying system and adversary models) this is not the case for their quantitative counterparts: there is a large number of proposals for quantitative confidentiality properties, and their relationships (e.g. in terms of the assumptions made and the guarantees provided) are often not well-understood.

In particular, there are two active and independent lines of research dealing with quantitative notions of confidentiality. The first line is motivated by the privacy-preserving publishing of data, with *differential privacy* [36] as the emerging consensus definition. The second line is motivated by tracking the information-flow in arbitrary programs, where most approaches quantify *leakage* as reduction in entropy about the program's input. In this paper, we focus on min-entropy as a measure of leakage because it is associated with strong operational security guarantees [76].

**Problem Statement** There have been efforts to understand the connections between the different notions of confidentiality proposed within each line of research (see [39] and [45, 76], respectively). The first studies of the relationship between differential privacy and quantitative notions of information-flow are emerging [28, 4], however, they do not directly compare leakage and differential privacy in terms of the security guarantees they deliver. Such a comparison could be highly useful, as it could enable one to transfer existing analysis techniques and enforcement mechanisms from one line of research to the other.

**Initial Results** In this project, we have addressed this open question. To begin with, we identify information-theoretic channels as a common model for casting differential privacy and leakage, where we assume that the input domain is fixed to $\{0,1\}^n$. Based on this model, we formally contrast the compositionality properties of differential privacy and leakage under sequential and parallel composition.

It is not difficult to see that there can be no general upper bound for differential privacy in terms of the leakage about the entire input.[1] However, it has been an open question whether it is possible to give upper bounds for the leakage in terms of differential privacy.

In this chapter, we will address this open question. To begin with, we identify information-theoretic channels as a common model for casting differential privacy and leakage, where we assume that the input domain is fixed to $\{0,1\}^n$. Based on this model, we formally contrast the compositionality properties of differential privacy and leakage under sequential and parallel composition.

We observe a difference in the behavior of leakage and differential privacy under parallel composition, and we exploit this difference to construct, for every $n$, a channel that is $\varepsilon$-differentially private and that leaks an amount of information that grows linearly with $n$. This result implies there can be no general (i.e. independent of the domain size) upper bound for the leakage of all $\varepsilon$-differentially private channels.

The situation changes, however, if we consider channels on input domains of bounded size. For such channels, we exhibit the following connections between leakage and differential privacy.

For the case $n = 1$, we give a complete characterization of leakage in terms of differential privacy. More precisely, we prove an upper bound for the leakage of every $\varepsilon$-differentially private channel. Moreover, we show

[1] Intuitively, the leakage of a single sensitive bit (e.g. from a medical record) can entirely violate an individual's privacy; on a technical level, no deterministic program satisfies differential privacy even if it leaks only a small amount of information, because differentially private programs are necessarily probabilistic.

that this bound is tight in the sense that, for every $\varepsilon$, there is an $\varepsilon$-differentially channel whose leakage matches the bound.

For the case $n > 1$, we prove upper bounds for the leakage of every $\varepsilon$-differentially private channel in terms of $n$ and $\varepsilon$. Technically, we achieve this by covering the channel's input domain by spheres of a fixed radius (with respect to the Hamming metric). The definition of $\varepsilon$-differential privacy ensures that the elements within each sphere produce similar output, where similarity is quantified in terms of $\varepsilon$ and the sphere radius. Based on this similarity and a recent characterization of the maximal leakage of channels [21, 58], we establish upper bounds for the information leaked about the elements of each sphere. By summing over all spheres, we obtain bounds for the information leaked about the entire input domain.

Our bounds are parametric in the number and the radius of the spheres used for covering the domain. We show how coding theory can be used for obtaining good instantiations of these parameters. In particular, we give examples where we derive bounds based on different classes of covering codes. We also exhibit limits for the bounds that can be obtained using our proof technique in terms of the sphere-packing bound. We perform an empirical evaluation that shows that the bounds we derived are close to this theoretical limit; moreover, we give an example channel whose leakage is only slightly below this limit, demonstrating the accuracy of our analysis.

Finally, although an explicit formula that precisely characterizes leakage in terms of privacy for finite input domains is still elusive, we show that such a characterization is in fact decidable. More precisely, we show that, for all $n$ and all rational functions $r$ (i.e. all quotients of polynomials with integer coefficients), one can decide whether the leakage of all $\varepsilon$-differentially channels is upper-bounded by $\log_2 r(\varepsilon)$.

In summary, our contribution is to prove formal connections between leakage and differential privacy, the two most influential quantitative notions of confidentiality to date. In particular, (i) we prove upper bounds for the leakage in terms of differential privacy for channels with bounded input domain, and (ii) we show that there can be no such bounds that hold for unbounded input domains.

### 5.3.2 Quantitative analysis of side-channels in web-applications

The program logic of web-applications is distributed between the browser- and the server-side. Both sides communicate over the web, exposing the application's internal state to eavesdroppers on the traffic. For many web-applications, this implies a serious threat to the privacy of their users. To counter this threat, privacy-aware web-applications typically hide their internal communication using encryption.

However, web-traffic leaks some information even if it is entirely encrypted. We mention three high-profile examples: First, the timing of encrypted packets of a login-procedure has been used to deduce the timing of individual keystrokes, which in turn has been used to speed up the brute-forcing of passwords [77]. Second, the length of encrypted VoIP packets has been used to deduce information about the spoken phonemes, which in turn has been used for recovering the spoken language [90], user identity [7], and even entire phrases [89] from encrypted traffic. Third, patterns in encrypted web-traffic have been successfully used for fingerprinting websites and breaking the privacy of their users [79].

Although these examples reveal subtle information leaks, the practical relevance of side-channel attacks against encrypted web-traffic was not always undisputed. The situation changed in 2010 when Chen et al. [26] exposed blatant information leaks in a number of high-profile web-applications.[2] In particular, they carried out attacks in which they (1) infer illnesses and medications of users from the HTTPS-traffic of a medical web-application, (2) deduce the annual income of users from the HTTPS-traffic of a web-application for preparing tax-returns, and (3) reconstruct search queries from WAP2-encrypted wireless traffic.

Technically, the attacks exploit that the data and control flow of the web-applications lead to observable patterns in the corresponding encrypted traffic. For example, AJAX GUI widgets, such as input boxes with auto-suggestion, generate traffic in response to individual keystrokes. The value of each individual keystroke is of low entropy, and is often almost entirely determined by the pattern of packet numbers and sizes of the triggered traffic. For longer input words, the fraction of meaningful character combinations decreases, which further simplifies the reconstruction of the input to the form. As another example, consider a web-application in which branching decisions depend on secret user inputs. These inputs can be revealed through differences in the patterns of sizes and numbers of packets that are sent in the different branches of the application. Such data and control-flow dependencies are ubiquitous in modern web applications, and hence also the corresponding information leaks.

[2] The authors only mention the Google, Yahoo, and Bing search engines; they use pseudonyms for the other targeted websites.

**Problem Statement** The initial proposals for mitigating side-channels in web-applications include the padding of packets (to hide true packet sizes), the introduction of noise packets (to hide true packet numbers), and the merging of application states (to increase the entropy of secrets). Chen et al. [26] perform a comparison of the performance and security of some of these countermeasures. While this comparison is an important first step, it is ad-hoc and specific to individual web-applications. Zhang et al. [94] make progress in terms of automating the quantitative analysis of side-channels in web-applications, but without the necessary semantic and information-theoretic foundations. It is an open problem to investigate these foundations and provide a principled method for the quantitative analysis of side-channels and their mitigation in web-applications.

Moreover, all countermeasures against side-channels in web-applications proposed to date focus on the provider-side of the application. For a user, this means to put a high amount of trust in the provider: She needs to trust that the provider takes the problem serious enough to apply (potentially expensive) countermeasures, and that they are implemented correctly. It would be desirable reduce this high amount of trust by enabling users to mitigate side-channels on the client-side, which however is still an open problem.

**Initial Results** We proposed the first formal tools for reasoning about leaks in web browsing traffic: models of websites, of attackers, and of countermeasures, as well as means to measure security in such a scenario. These tools allow formal arguments about the security guarantees of countermeasures applied to web browsing traffic. Furthermore, we proposed novel countermeasures which can be applied in a user's browser, unlike most previously known countermeasures.

# 6 Interactions

## 6.1 Current Interaction with Other WPs

WP8 has a number of links to the work done in WP9, in particular in Task 9.2. Task 9.2.1 deals with formal verification of applications, aiming for vulnerability detection that comes early in the life cycle. This relates directly with the work we presented in this deliverable about the VeriFast program verifier. Task 9.2.2 proposes new techniques to improve on testing applications. One of the action points here is the automatic generation of test data for web applications, much like the technique we applied while evaluating the contribution presented in Section 2.2. Finally, task 9.2.3 covers run-time monitoring of data flow and information flow, which corresponds directly with the three contributions in Section 2 and the contribution in Section 5.

## 6.2 New Initiatives

SIEMENS and LMU are planning cooperation on the topic of secure navigation paths. Although a web application is meant to be in general stateless, the web pages offer navigation links that are assumed to determine the usual navigation paths a user takes when interacting with the application. If a user does not follow the paths, an attack can be assumed, where for instance a user just clicks on a link he receives in an e-mail or on a malicious web page. Each part of a web page is referred to as navigational node that can change individually for interactive Web 2.0 applications. We assume that users do not leave predefined secure navigation paths, provided by the graphical user interface of the Web application. Consequently, the use of a monitor to restrict the paths to only secure ones would be a big step towards secure Web applications where session high-jacking, phishing and similar attacks are not possible. The envisaged goal is to come up with a sound Web engineering technique, which integrates the specification of secure navigation paths into the engineering process of secure Web applications. Therefore we plan to extend UWE's navigation states model, which is at the moment used to define further security features, as authentication and access control for navigational nodes [23].

# 7 Conclusion

The service creation means must be improved and extended to deal with security needs. Service creation means both aggregating and (1) composing services from pre-existing building blocks (services and more traditional components), as well as (2) programming new services from scratch using a state-of-the-art programming language. The service creation context will typically aim for techniques and technologies that support compile and build-time feedback. One could argue that security support for service creation must focus on and enable better static verification. Then (3) the service execution support must be enhanced to deal with hooks and building blocks that facilitate effective security enforcement at run-time.

This report gathers the results of new R&D that has been conducted in the first year of this NoE.

The collection is based on work that has occurred at CNR, IMDEA, INRIA, KUL, LMU, Siemens and UNITN. The contributions in chapter 2 focus on web application security. This work has mainly been done by KUL - though inspired and triggered by input from Siemens. Also this part covers some collaboration with SAP, one of the partners in the NESSoS IAB. The contributions in chapter 3 focus on service composition and service configuration. The work on service composition includes contributions from CNR and INRIA, who will collaborate on some of the related topics in the remaining time of this project. The work on service configuration is based on collaboration between KUL and UNITN. Chapter 4 addresses language extensions that can enable formal verification. This work has been delivered by KUL; the related prototypes are part of the NESSoS Workbench. Chapter 5 presents an effective collaboration towards achieving end-to-end properties in secure services. The focus has been on information flow and the work builds upon collaboration between IMDEA and KUL.

Clearly, this work package has been covering a broad range of topics to a certain extend in a bottom-up fashion. Published work has been described briefly; (unpublished) work in progress has been articulating the current progress, but remains inherently incomplete. To grasp all details that are highlighted in this deliverable, one should consult the publications that have been enumerated in appendix A and that are fully included in an extended version of this deliverable.

Looking forward to the second year of the NoE's research program, we want to highlight an improved and intensified collaboration: within task 8.1 and 8.2: at least Siemens/KUL, INRIA/CNR. Furthermore, we will aim to start collaboration between WP 7 and WP 8 via task 8.2; and between WP 9 and WP 8 via task 8.3. In the mid term, we intend to couple web service run time support to some of the application case studies.

# Bibliography


[1] Apache.org. https://blogs.apache.org/infra/entry/apache_org_04_09_2010.

[2] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[3] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. *Computer Security Foundations Symposium, IEEE*, 0:290–304, 2010.

[4] M. S. Alvim, K. Chatzikokolakis, P. Degano, and C. Palamidessi. Differential privacy versus quantitative information flow. *CoRR*, abs/1012.4250v1, 2010.

[5] T. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, 2010.

[6] M. Backes, M. Berg, and B. Köpf. Non-uniform distributions in quantitative information-flow. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 367–375. ACM, 2011.

[7] M. Backes, G. Doychev, M. Dürmuth, and B. Köpf. Speaker Recognition in Encrypted Voice Streams. In *Proc. 15th European Symposium on Research in Computer Security (ESORICS '10)*, LNCS 6345, pages 508–523. Springer, 2010.

[8] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.

[9] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52:83–91, June 2009.

[10] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114, 2004.

[11] G. Barthe and B. Köpf. Information-theoretic Bounds for Differentially Private Mechanisms. In *Proc. 24th IEEE Computer Security Foundations Symposium (CSF '11), to appear*, pages 191–204. IEEE, 2011.

[12] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, and R. Zunino. Securing java with local policies. *Journal of Object Technology*, 8(4):5–32, 2009.

[13] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK*, pages 316–332, 2005.

[14] M. Bartoletti, P. Degano, and G. L. Ferrari. Planning and verifying service composition. *Journal of Computer Security (JCS)*, 17(5):799–837, 2009. (Abridged version In *Proc. 18th Computer Security Foundations Workshop (CSFW)*, 2005).

[15] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Secure service orchestration. In *Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures*, pages 24–74, 2007.

[16] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal*, pages 32–47, 2007.

[17] M. Bartoletti, P. Degano, G.-L. Ferrari, and R. Zunino. Local policies for resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 31(6):1–43, 2009.

[18] M. Bartoletti and R. Zunino. Locust: a tool for model checking usage policies. Technical report, University of Pisa, 2008.

[19] F. Besson, T. P. Jensen, and D. L. Métayer. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[20] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *NSS*, 2011.

[21] C. Braun, K. Chatzikokolakis, and C. Palamidessi. Quantitative notions of leakage for one-try attacks. *Electr. Notes Theor. Comput. Sci.*, 249:75–91, 2009.

[22] J. Burns. Cross site reference forgery: An introduction to a common web application weakness. *f Security Partners, LLC*, 2005.

[23] M. Busch, A. Knapp, and N. Koch. Modeling Secure Navigation in Web Information Systems. In J. Grabis and M. Kirikova, editors, *10th International Conference on Business Perpectives in Informatics Research*, LNBIP. Springer Verlag, 2011.

[24] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *ACM SIGPLAN Notices*, 44(1):289–300, 2009.

[25] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla. Preventing information leaks through shadow executions. In *ACSAC*, 2008.

[26] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proc. 31st IEEE Symposium on Security and Privacy (S&P '10)*, pages 191–206. IEEE Computer Society, 2010.

[27] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *CSF*, pages 200–214, 2010.

[28] M. R. Clarkson and F. B. Schneider. Quantification of integrity. Cornell Computing and Information Science Technical Reports, 2011. http://hdl.handle.net/1813/22012.

[29] W. A. S. Consortium. Web Hacking Incident Database.

[30] G. Costa, P. Degano, and F. Martinelli. Modular plans for secure service composition. *Journal of Computer Security*, 2011. To Appear.

[31] G. Costa, P. Degano, and F. Martinelli. Secure service orchestration in open networks. *Journal of Systems Architecture - Embedded Systems Design*, 57(3):231–239, 2011.

[32] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Lecture Notes in Computer Science*, pages 18–34. Springer Berlin / Heidelberg, 2010.

[33] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. In *Lecture Notes in Computer Science*. Springer, September 2011.

[34] P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Automatic and precise client-side protection against csrf attacks - downloads. https://distrinet.cs.kuleuven.be/software/CsFire/esorics2011/, 2011.

[35] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[36] C. Dwork. Differential Privacy. In *Proc. 33rd Intl. Colloquium on Automata, Languages and Programming (ICALP '06)*, volume 4052 of *LNCS*, pages 1–12. Springer, 2006.

[37] D. Florencio and C. Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web (WWW '07)*, New York, NY, USA, 2007. ACM.

[38] K. Fullam, T. Klos, G. Muller, J. Sabater, A. Schlosser, Z. Topol, K. Barber, J. Rosenschein, L. Vercouter, and M. Voss. A specification of the agent reputation and trust (art) testbed: experimentation and competition for trust in agent societies. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 512–518. ACM, 2005.

[39] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4), 2010.

[40] G. Gheorghe, B. Crispo, R. Carbone, L. Desmet, and W. Joosen. Deploy, adjust and readjust: Supporting dynamic reconfiguration of policy enforcement. In *ACM/IFIP/USENIX 12th International Middleware Conference*, volume 7049. IFIP/Springer, December 2011.

[41] Google. Google Latitude. https://www.google.com/latitude/.

[42] T. Goovaerts, L. Desmet, and W. Joosen. Scalable authorization middleware for service oriented architectures. In *Engineering Secure Software and Systems*, February 2011.

[43] G. L. Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[44] S. Hachem, A. Toninelli, A. Pathak, and V. Issarny. Policy-based access control in mobile social ecosystems. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 57–64. IEEE, 2011.

[45] S. Hamadou, V. Sassone, and C. Palamidessi. Reconciling belief and vulnerability in information flow. In *Proc. 31st IEEE Symposium on Security and Privacy (S&P '10)*, pages 79–92. IEEE Computer Society, 2010.

[46] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[47] S. Hisao. Tiny HTTP Proxy in Python.

[48] Involver. Tweets To Pages. http://www.facebook.com/TweetsApp.

[49] B. Jacobs and F. Piessens. The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.

[50] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Programming Languages and Systems (APLAS 2010)*, pages 304–311. Springer-Verlag, November 2010.

[51] B. Jacobs, J. Smans, and F. Piessens. Verifast: Imperative programs as proofs. In *VSTTE workshop on Tools & Experiments*, August 2010.

[52] M. Jaskelioff and A. Russo. Secure multi-execution in haskell. In *PSI*, 2011.

[53] M. Johns and J. Winter. RequestRodeo: client side protection against session riding. *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5–17, 2006.

[54] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 1–10, 2006.

[55] S. Kaffille and G. Wirtz. Engineering autonomous trust-management requirements for software agents: Requirements and concepts. *Innovations and Advances in Computer Sciences and Engineering*, pages 483–489, 2010.

[56] D. E. Knuth. *The Art of Computer Programming, Volume 2*. Addison-Wesley Publishing Company, 1971.

[57] B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *Proc. IEEE Computer Security Foundations Symposium (CSF '09)*, pages 324–335. IEEE, 2009.

[58] B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *Proc 23rd. IEEE Computer Security Foundations Symposium (CSF '10)*, pages 44–56. IEEE, 2010.

[59] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[60] B. Livshits and L. Meyerovich. CONSCRIPT: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser, 2009.

[61] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Extensible Web Browser Security. In *4th International Conference on Detection of Intrusions, Malware, and Vulnerability Assesment (DIMVA)*, 2007.

[62] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. *Financial Cryptography and Data Security*, pages 238–255, 2009.

[63] G. Maone. Noscript 2.0.9.9. http://noscript.net/, 2011.

[64] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja Safe active content in sanitized JavaScript. *October*, 2008.

[65] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[66] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: Lightweight protection against session hijacking. In *Third International Symposium, ESSoS 2011*, volume 6542, pages 87–100. Springer, February 2011.

[67] OWASP Top 10 Web Application Security Risks.

[68] P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, and F. Piessens. The belgian electronic identity card: a verification case study. In *Proceedings of the International Workshop Automated Verification of Critical Systems (AVOCS'11)*, September 2011.

[69] R. Saadi, M. Rahaman, V. Issarny, and A. Toninelli. Composing trust models towards interoperable trust management. *Trust Management V*, pages 51–66, 2011.

[70] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*, pages 352–365, 2009.

[71] J. Samuel. Requestpolicy 0.5.20. http://www.requestpolicy.com, 2011.

[72] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[73] W. Security. XSS Worms: The impending threat and the best defense.

[74] A. Seyedi, R. Saadi, and V. Issarny. Proximity-based trust inference for mobile social networking. *Trust Management V*, pages 253–264, 2011.

[75] C. Skalka and S. F. Smith. History effects and verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan*, pages 107–128, 2004.

[76] G. Smith. On the foundations of quantitative information flow. In *Proc. Intl. Conf. of Foundations of Software Science and Computation Structures (FoSSaCS '09)*, LNCS 5504, pages 288–302. Springer, 2009.

[77] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proc. 10th USENIX Security Symposium*. USENIX Association, 2001.

[78] L. D. F. P. W. J. Steven Van Acker, Philippe De Ryck. Webjail: Least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[79] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proc. 23rd IEEE Symposium on Security and Privacy*, pages 19–30. IEEE Computer Society, 2002.

[80] G. Suryanarayana, J. Erenkrantz, S. Hendrickson, and R. Taylor. Pace: An architectural style for trust management in decentralized applications. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 221–230. IEEE, 2004.

[81] G. Suryanarayana and R. Taylor. Sift: A simulation framework for analyzing decentralized reputation-based trust models. 2007.

[82] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[83] Alexa: The Web information company.

[84] L. Vercouter, S. Casare, J. Sichman, and A. Brandao. An experience on reputation models interoperability based on a functional ontology. In *Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2007.

[85] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2/3):167–188, 1996.

[86] W3C. W3C Standards and drafts - Javascript APIs. http://www.w3.org/TR/#tr_Javascript_APIs.

[87] W. W. W. C. (W3C). The "Buy Something" scenario, 2009. http://www.w3.org/2001/03/WSWS-popa/paper51.

[88] Performance Benchmark - Monitor Page Load Time | Webmetrics.

[89] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In *Proc. 29th IEEE Symposium on Security and Privacy (S&P '08)*, pages 35–49. IEEE Computer Society, 2008.

[90] C. V. Wright, L. Ballard, F. Monrose, and G. M. Masson. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In *Proc. 16th USENIX Security Symposium*. USENIX Association, 2007.

[91] XSSed | Cross Site Scripting (XSS) attacks information and archive.

[92] M. Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Main, 2010.

[93] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.

[94] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *Proc. 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 595–606. ACM, 2010.

# A Appendix - Relevant Papers

The work described in this deliverable is based on the following papers:

1. *Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen* **WebJail: Least-privilege Integration of Third-party Components in Web Mashups** *ACSAC, Orlando, Florida, USA, 5-9 December 2011*

2. *Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens* **Automatic and Precise Client-Side Protection against CSRF Attacks** *European Symposium on Research in Computer Security (ESORICS 2011), Lecture Notes in Computer Science, volume 6879, pages 100-116, Leuven, Belgium, 12-14 September 2011*

3. *Rachid Saadi, Mohammad Ashiqur Rahaman, Valérie Issarny, and Alessandra Toninelli* **Composing Trust Models towards Interoperable Trust Management** *Proceedings of Trust Management V: 5th IFIP WG 11. 11 International Conference, IFIPTM 2011, Copenhagen, Denmark, June 29-July 1, 2011*

4. *Sara Hachem, Alessandra Toninelli, Animesh Pathak, and Valérie Issarny* **Policy-based Access Control in Mobile Social Ecosystems** *International Symposium on Policies for Distributed Systems and Networks (POLICY), 2011 IEEE*

5. *Amir Seyedi, Rachid Saadi, and Valérie Issarny* **Proximity-Based Trust Inference for Mobile Social Networking** *Proceedings of Trust Management V: 5th IFIP WG 11. 11 International Conference, IFIPTM 2011, Copenhagen, Denmark, June 29-July 1, 2011*

6. *Gabriela Gheorghe, Bruno Crispo, Roberto Carbone, Lieven Desmet, and Wouter Joosen* **Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement** *ACM/IFIP/USENIX 12th International Middleware Conference, 2011*

7. *Gilles Barthe and Boris Köpf* **Information-theoretic Bounds for Differentially Private Mechanisms**

8. *Michael Backes, Matthias Berg, and Boris Köpf* **Non-Uniform Distributions in Quantitative Information-Flow** *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011*

# WebJail: Least-privilege Integration of Third-party Components in Web Mashups

Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, Wouter Joosen
IBBT-Distrinet, Katholieke Universiteit Leuven, 3001 Leuven, Belgium
Steven.VanAcker@cs.kuleuven.be

## ABSTRACT

In the last decade, the Internet landscape has transformed from a mostly static world into Web 2.0, where the use of web applications and mashups has become a daily routine for many Internet users. Web mashups are web applications that combine data and functionality from several sources or components. Ideally, these components contain benign code from trusted sources. Unfortunately, the reality is very different. Web mashup components can misbehave and perform unwanted actions on behalf of the web mashup's user.

Current mashup integration techniques either impose no restrictions on the execution of a third-party component, or simply rely on the Same-Origin Policy. A least-privilege approach, in which a mashup integrator can restrict the functionality available to each component, can not be implemented using the current integration techniques, without ownership over the component's code.

We propose WebJail, a novel client-side security architecture to enable least-privilege integration of components into a web mashup, based on high-level policies that restrict the available functionality in each individual component. The policy language was synthesized from a study and categorization of sensitive operations in the upcoming HTML 5 JavaScript APIs, and full mediation is achieved via the use of deep aspects in the browser.

We have implemented a prototype of WebJail in Mozilla Firefox 4.0, and applied it successfully to mainstream platforms such as iGoogle and Facebook. In addition, micro-benchmarks registered a negligible performance penalty for page load-time (7ms), and the execution overhead in case of sensitive operations (0.1ms).

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; H.3.5 [**Information Storage and Retrieval**]: Web-based services


*ACSAC* '11 Dec. 5-9, 2011, Orlando, Florida USA


## Keywords

Web Application Security, Web Mashups, Sandbox, Least-privilege integration

## 1. INTRODUCTION

The Internet has seen an explosion of dynamic websites in the last decade, not in the least because of the power of JavaScript. With JavaScript, web developers gain the ability to execute code on the client-side, providing for a richer and more interactive web experience. The popularity of JavaScript has increased even more since the advent of Web 2.0.

Web mashups are a prime example of Web 2.0. In a web mashup, data and functionality from multiple stakeholders are combined into a new flexible and lightweight client-side application. By doing so, a mashup generates added value, which is one of the most important incentives behind building mashups. Web mashups depend on collaboration and interaction between the different mashup components, but the trustworthiness of the service providers delivering components may strongly vary.

The two most wide-spread techniques to integrate third-party components into a mashup are via script inclusion and via (sandboxed) iframe integration, as will be discussed in more detail in Section 2. The script inclusion technique implies that the third-party component executes with the same rights as the integrator, whereas the latter technique restricts the execution of the third-party component according to the Same-Origin Policy. More fine-grained techniques (such as Caja [23] or FBJS [31]) require (some form of) ownership over the code to transform or restrict the component to a known safe subset before delivery to the browser. This makes these techniques less applicable to integrate third-party components directly from their service providers.

To enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components, web mashups should integrate components according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. Unfortunately, least-privilege integration of third-party mashup components can not be achieved with the current script-inclusion and frame-integration techniques. Moreover, the need for least-privilege integration becomes highly relevant, especially because of the augmented capabilities of the upcoming HTML5 JavaScript APIs [32] (such as access to local storage, geolocation, media capture and cross-domain communication).

In this paper, we propose WebJail, a novel client-side se-

curity architecture to enable the least-privilege integration of third-party components in web mashups. The security restrictions in place are configurable via a high-level composition policy under control of the mashup integrator, and allow the use of legacy mashup components, directly served by multiple service providers.

In summary, the contributions of this paper are:

1. a novel client-side security architecture, WebJail, that supports least-privilege composition of legacy third-party mashup-components

2. the design of a policy language for WebJail that is tuned to support the effective use of WebJail to limit access to the powerful upcoming HTML5 APIs

3. the implementation of WebJail and its policy language in Firefox, and evaluation and discussion of performance and usability

The rest of this paper is structured as follows. Section 2 sketches the necessary background, and Section 3 further elaborates the problem statement. In Section 4, the WebJail least-privilege integration architecture is presented and its three layers are discussed in more detail. Next, the prototype implementation in Firefox is described in Section 5, followed by an experimental evaluation in Section 6 and discussion in Section 7. Finally, Section 8 discusses related work, and Section 9 summarizes the contributions.

# 2. BACKGROUND

This section briefly summarizes the Same-Origin Policy. Next, Section 2.2 discusses how mashups are constructed and gives some insights in the state-of-practice on how third-party mashup components get integrated.

## 2.1 Same-Origin Policy

Currently, mashup security is based on the de facto security policy of the web: the Same-Origin Policy (SOP) [34]. An origin is a domain name-protocol-port triple, and the SOP states that scripts from one origin should not be able to access content from other origins. This prevents scripts from stealing data, cookies or login credentials from other sites. In addition to the SOP, browsers also apply a frame navigation policy, which restricts the navigation of frames to its descendants [1].

Among others, the Same-Origin Policy allows a per-origin separation of JavaScript execution contexts. Contexts are separated based on the origin of the window's document, possibly relaxed via the `document.domain` property to a right-hand, fully-qualified fragment of its current hostname. Within an execution context, the SOP does not impose any additional security restriction.

## 2.2 Integration of mashup components

The idea behind a web mashup is to integrate several web applications (components) and mash up their code, data and results. The result is a new web application that is more useful than the sum of its parts. Several publicly available web applications [25] provide APIs that allow them to be used as third-party components for web mashups.

To build a client-side mashup, an integrator selects the relevant in-house and third-party components, and provides the necessary glue code on an integrating web page to retrieve the third-party components from their respective service providers and let them interact and collaborate with each other.

As stated before, the two most-widespread techniques to integrate third-party components into a web mashup are through script inclusion or via (sandboxed) iframe-integration [4, 18].

**Script inclusion.** HTML script tags are used to execute JavaScript while a webpage is loading. This JavaScript code can be located on a different server than the webpage it is executing in. When executing, the browser will treat the code as if it originated from the same origin as the webpage itself, without any restrictions of the Same-Origin Policy.

The included code executes in the same JavaScript context, has access to the code of the integrating webpage and all of its datastructures. All sensitive JavaScript operations available to the integrating webpage are also available to the integrated component.

**(Sandboxed) iframe integration.** HTML iframe tags allow a web developer to include one document inside another. The integrated document is loaded in its own environment almost as if it were loaded in a separate browser window. The advantage of using an iframe in a mashup is that the integrated component from another origin is isolated from the integrating webpage via the Same-Origin Policy. However, the code running inside of the iframe still has access to all of the same sensitive JavaScript operations as the integrating webpage, albeit limited to its own execution context (i.e. origin). For instance, a third-party component can use local storage APIs, but only has access to the local storage of its own origin.

HTML 5 adds the "sandbox" attribute to the iframe element, allowing an integrator to disable all security-sensitive features through its "allow-scripts" keyword. Obviously, this very coarse-grained control has only a very limited applicability in a web mashup context.

# 3. PROBLEM STATEMENT

In this section, the attacker model is specified, as well as two typical attack vectors. Next, the increasing impact of insecure mashup composition is discussed in the context of the upcoming set of HTML5 specifications. Finally, the security assessment is concluded by identifying the requirements for secure mashup composition, namely the least-privilege integration of third-party mashup components.

## 3.1 Attacker model

Our attacker model is inspired by the definition of a *gadget attacker* in Barth *et al.* [1]. The term gadget in their definition should, in the context of this paper, be read as "third-party mashup component".

We describe the attacker in scope as follows:

**Malicious third-party component provider** The attacker is a malicious principal owning one or more machines on the network. The attacker is able to trick the integrator in embedding a third-party component under control of the attacker.

We assume a mashup that consists of multiple third-party components from several service providers, and an honest mashup consumer (i.e. end-user). A malicious third-party

component provider attempts to steal sensitive data outside its trust boundary (e.g. reading from origin-specific client-side storage), impersonate other third-party components or the integrator (e.g. requesting access to geolocation data on behalf of the integrator) or falsely operate on behalf of the end-user towards the integrator or other service providers (e.g. requesting cross-application content with XMLHttpRequest).

We have identified two possible ways in which an attacker could present himself as a malicious third-party component provider: he could offer a malicious third-party component towards mashup integrators (e.g. via a malicious advertisement, or via a malicious clone of a popular component), or he could hack into an existing third-party component of a service provider and abuse the prior existing trust relationship between the integrator and the service provider.

In this paper, we consider the mashup integrator as trusted by the mashup consumer (i.e. end-user), and an attacker has no control over the integrator, except for the attacker's ability to embed a third-party components of his choice. In addition, we assume that the attacker has no special network abilities (such as sniffing the network traffic between client and servers), browser abilities (e.g. extension under control of the attacker or client-side malware) and is constrained in the browser by the Same-Origin Policy.

## 3.2 Security-sensitive JavaScript operations

The impact of running arbitrary JavaScript code in an insecure mashup composition is equivalent to acquiring XSS capabilities, either in the context of the component's origin, or in the context of the integrator. For instance, a malicious third-party component provider can invoke typical security-sensitive operations such as the retrieval of cookies, navigation of the browser to another page, launch of external requests or access and updates to the Document Object Model (DOM).

However, with the emerging HTML5 specification and APIs, the impact of injecting and executing arbitrary JavaScript has massively increased. Recently, JavaScript APIs have been proposed to access geolocation information and system information (such as CPU load and ambient sensors), to capture audio and video, to store and retrieve data from a client-side datastore, to communicate between windows as well as with remote servers.

As a result, executing arbitrary JavaScript becomes much more attractive to attackers, even if the JavaScript execution is restricted to the origin of the component, or a unique origin in case of a sandbox.

## 3.3 Least-privilege integration

Taking into account the attack vectors present in current mashup composition, and the increasing impact of such attacks due to newly-added browser features, there is clearly a need to limit the power of third-party mashup components under control of the attacker.

Optimally, mashup components should be integrated according to the least-privilege principle. This means that each of the components is only granted access to data or functionality necessary to perform its core function. This would enable the necessary collaboration and interaction while restricting the capabilities of untrusted third-party components.

Unfortunately, a least-privilege integration of third-party mashup components can not be achieved with the current script-inclusion and iframe-integration techniques. These techniques are too coarse-grained: either no restrictions (or only the Same-Origin Policy) are imposed on the execution of a third-party component, implicitly inviting abuse, or JavaScript is fully disabled, preventing any potential abuse but also fully killing desired functionality.

To make sure that attackers described in Section 3.1 do not exploit the insecure composition attack vectors and multiply their impact by using the security sensitive HTML5 APIs described in Section 3.2, the web platform needs a security architecture that supports least-privilege integration of web components. Since client-side mashups are composed in the browser, this architecture must necessarily be implemented in the browser. It should satisfy the following requirements:

**R1 Full mediation.** The security-sensitive operations need to be fully mediated. The attacker can not circumvent the security mechanisms in place.

**R2 Remote component delivery.** The security mechanism must allow the use of legacy third-party components and the direct delivery of components from the service provider to the browser environment.

**R3 Secure composition policy.** The secure composition policy must be configurable (and manageable) by the mashup integrator. The policy must allow fine-grained control over a single third-party component, with respect to the security-sensitive operations in the HTML5 APIs.

**R4 Performance** The security mechanism should only introduce a minimal performance penalty, unnoticeable to the end-user.

Existing technologies like e.g. Caja [23] and FBJS [31] require pre-processing of mashup components, while ConScript [21] does not work in a mashup context because it depends on the mashup component to load and enforce its own policy. A more thorough discussion of related work can be found in Section 8.

# 4. WEBJAIL ARCHITECTURE

To enable least-privilege integration of third-party mashup components, we propose WebJail, a novel client-side security architecture. WebJail allows a mashup integrator to apply the least-privilege principle on the individual components of the mashup, by letting the integrator express a secure composition policy and enforce the policy within the browser by building on top of the deep advice approach of ConScript [21].

The secure composition policy defines the set of security-sensitive operations that the component is allowed to invoke. Each particular operation can be allowed, disallowed, or restricted to a self-defined whitelist. Once loaded, the deep aspect layer will ensure that the policy is enforced on every accesspath to the security-sensitive operations, and that the policy can not be tampered with.

The WebJail architecture consists of three abstraction layers as shown in Figure 1. The upper layer, the *policy layer*, associates the secure composition policy with a mashup component, and triggers the underlying layers to enforce the

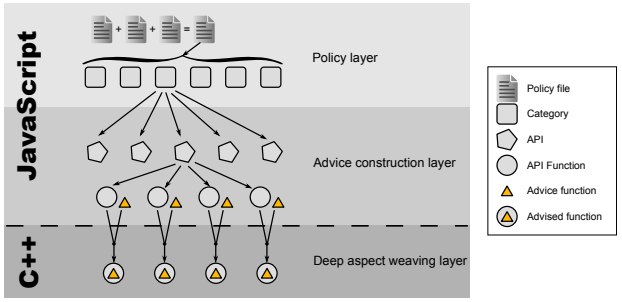



Figure 1: The WebJail architecture consists of three layers: The policy layer, the advice construction layer and the deep aspect weaving layer.

policy for the given component. The lower layer, the *deep aspect weaving layer*, enables the deep aspect support with the browser's JavaScript engine. The *advice construction layer* in between takes care of mapping the higher-level policy blocks onto the low-level security-sensitive operations via a 2-step policy refinement process.

In this section, the three layers of the WebJail will be described in more detail. Next, Section 5 will discuss a prototype implementation of this architecture in Mozilla Firefox.

## 4.1 Policy layer

The policy layer associates the secure composition policy with the respective mashup component. In this section, an analysis of security-sensitive operations in the HTML5 APIs is reported and discussed, as well as the secure composition policy itself.

### 4.1.1 Security-sensitive JavaScript operations

As part of this research, we have analyzed the emerging specifications and browser implementations, and have identified 86 security-sensitive operations, accessible via JavaScript APIs. We have synthesized the newly-added features of these specifications in Figure 2, and we will briefly summarize each of the components in the next paragraphs. Most of these features rely on (some form of) user-consent and/or have origin-restrictions in place.

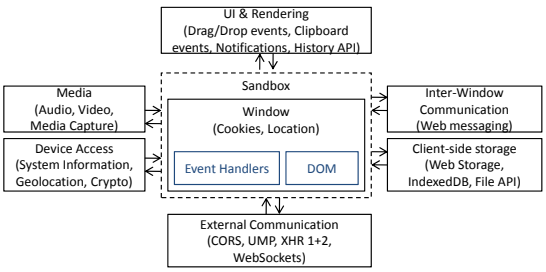



Figure 2: Synthesized model of the emerging HTML5 APIs

Central in the model is the *window* concept, containing the document. The window manifest itself as a browser window, a tab, a popup or a frame, and provides access to the location and history, event handlers, the document and its associated DOM tree. Event handlers allow to register for a specific event (e.g. being notified of mouse clicks), and access to the DOM enables a script to read or modify the document's structure on the fly. Additionally, a *sandbox* can impose coarse-grained restrictions on an iframe, as mentioned in Section 2.2.

*Inter-frame communication* allows sending messages between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (postMessage).

*Client-side storage* enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

*External communication* features such as CORS, UMP, XMLHttpRequest level 1 and 2, and websockets allow an application to communicate with remote websites, even in cross-origin settings.

*Device access* allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information and ambient sensors.

*Media* features enable a web application to play audio and video fragments, as well as capture audio and video via a microphone or webcam.

*The UI and rendering* features allow subscription to clipboard and drag-and-drop events, issuing desktop notifications and populating the history via the History API.

For a more thorough analysis of the HTML5 APIs, we would like to refer to an extensive security analysis we have carried out, commissioned by the European Network and Information Security Agency (ENISA) [7].

### 4.1.2 Secure composition policy

The policy layer associates the secure composition policy with a mashup component, and deploys the necessary security controls via the underlying layers. As composition granularity, we have chosen the iframe level; i.e. mashup components are each loaded in their separate iframe.

In particular, within WebJail the secure composition policy is expressed by the mashup integrator, and attached to a particular component via a newly-introduced *policy* attribute of the iframe element of the component to be loaded.

```
1 <iframe src="http://untrusted.com/compX/"
      policy="https://integrator.com/compX.policy"/>
```

We have grouped the identified security-sensitive operations in the HTML5 APIs in nine disjoint categories, based on their functionality: DOM access, Cookies, External communication, Inter-frame communication, Client-side storage, UI & Rendering, Media, Geolocation and Device access.

For a third-party component, each category can be fully disabled, fully enabled, or enabled only for a self-defined whitelist. The whitelists contain category-specific entries. For example, a whitelist for the category "DOM Access" contains the ids of the elements that might be read from or updated in the DOM. The nine security-sensitive categories are listed in Table 1, together with their underlying APIs, the amount of security-sensitive functions in each API, and their WebJail whitelist types.

The secure composition policy expresses the restrictions for each of the security-sensitive categories, and an example policy is shown below. Unspecified categories are disallowed by default, making the last line in the example policy obsolete.

```
1 { "framecomm" : "yes",
2   "extcomm" : [ "google.com", "youtube.com" ],
3   "device" : "no" }
```

| Categories and APIs (# op.) | Whitelist |
|---|---|
| **DOM Access** | ElemReadSet, ElemWriteSet |
| DOM Core (17) | |
| **Cookies** | KeyReadSet, KeyWriteSet |
| cookies (2) | |
| **External Communication** | DestinationDomainSet |
| XHR, CORS, UMP (4) | |
| WebSockets (5) | |
| Server-sent events (2) | |
| **Inter-frame Communication** | DestinationDomainSet |
| Web Messaging (3) | |
| **Client-side Storage** | KeyReadSet, KeyWriteSet |
| Web Storage (5) | |
| IndexedDB (16) | |
| File API (4) | |
| File API: Dir. and Syst. (11) | |
| File API: Writer (3) | |
| **UI and Rendering** | |
| History API (4) | |
| Drag/Drop events (3) | |
| **Media** | |
| Media Capture API (3) | |
| **Geolocation** | |
| Geolocation API (2) | |
| **Device Access** | SensorReadSet |
| System Information API (2) | |
| **Total number of security-sensitive operations: 86** | |

Table 1: Overview of the sensitive JavaScript operations from the HTML 5 APIs, divided in categories.

It is important to note that WebJails or regular frames can be used inside WebJails. In such a case, the functionality in the inner frame is determined by the policies imposed on enclosing frames, in addition to its own policy (if it has one, as is the case with a WebJail frame). Allowing sensible cascading of policies implies that "deeper" policies can only make the total policy more strict. If this were not the case, a WebJail with a less strict policy could be used to "break out" of the WebJail restrictions.

The semantics of a policy entry for a specific category can be thought of as a set. Let $\mathcal{V}$ be the set of all possible values that can be listed in a whitelist. The "allow all" policy would then be represented by the set $\mathcal{V}$ itself, a whitelist would be represented by a subset $w \subseteq \mathcal{V}$ and the "allow none" policy by the empty set $\phi$. The relationship "$x$ is at least as strict as $y$" can be represented as $x \subseteq y$. Using this notation, the combined policy $p$ of 2 policies $a$ and $b$ is the intersection $p = a \cap b$, since $p \subseteq a$ and $p \subseteq b$.

After loading, parsing and combining all the policies applicable to the WebJail protected iframe, the policy is enforced via the underlying layers.

## 4.2 Advice construction layer

The task of the advice construction layer is to build advice functions based on the high-level policy received from the policy layer, and apply these advice functions on the low-level security-sensitive operations via deep aspect technology in the deep advice weaving layer.

To do so, the advice construction layer applies a 2-step refinement process. For each category of the secure composition policy, the set of relevant APIs is selected. Next for each API, the individual security-sensitive operations are processed. Consider for instance that a whitelist of type "KeyReadSet"[1] is specified for the client-side storage in the composition policy. This is first mapped to the various storage APIs in place (such as Web Storage and File API), and then advice is constructed for the security-sensitive operations in the API (e.g. for accessing the *localStorage* object).

The advice function decides, based on the policy, whether or not the associated API function will be called: if the policy for the API function is "allow all", or "allow some" and the whitelist matches, then the advice function allows the call. Otherwise, the call is blocked.

On successful completion of its job, the advice construction layer has advice functions for all the security-sensitive operations across the nine categories relevant for the specific policy. Next, the advices are applied on the original operations via the deep advice weaving layer.

## 4.3 Deep aspect weaving layer

The *(advice, operation)* pairs received from the advice construction layer are registered into the JavaScript engine as deep advice. The result of this weaving is that the original API function is replaced with the advice function, and that all accesspaths to the API function now go through the advice function. The advice function itself is the only place where a reference to the original API function exists, allowing it to make use of the original functionality when desired.

# 5. PROTOTYPE IMPLEMENTATION

To show the feasibility and test the effectiveness of WebJail, we implemented a prototype by modifying Mozilla Firefox 4.0b10pre.The modifications to the Mozilla code are localized and consist of ±800 lines of new code (±300 JavaScript, ±500 C++), spread over 3 main files. The prototype currently supports the security-sensitive categories external and inter-frame communication, client-side storage, UI and rendering (except for drag/drop events) and geolocation.

Each of the three layers of the implementation will be discussed now in more detail.

## 5.1 Policy layer

The processing of the secure composition policy via the *policy* attribute happens in the frame loader, which handles construction of and loading content into frames. The specified policy URL is registered as the policy URL for the frame to be loaded, and any content loaded into this frame will be subject to that WebJail policy, even if that content issues a refresh, submits a form or navigates to another URL.

When an iframe is enclosed in another iframe, and both specify a policy, the combinatory rules defined in Section 4 are applied on a per-category basis. To ease up parsing of a policy file, we have chosen to use the JavaScript Object Notation (JSON).

Once the combined policy for each category has been calculated, the list of APIs in that category is passed to the advice construction layer, along with the combined policy.

## 5.2 Advice construction layer

The advice construction layer builds advice functions for individual API functions. For each API, the advice construction layer knows what functions are essential to enforce the policy and builds a specific advice function that enforces it.

The advice function is a function that will be called instead of the real function. It will determine whether or not the real function will be called based on the policy and the arguments passed in the function call. Advice functions in WebJail are written in JavaScript and should expect 3 ar-

[1]Such a whitelist contains a set of keys that may be read

guments: a function object that can be used to access the original function, the object on which the function was invoked (i.e. the `this` object) and a list with the arguments passed to the function.

```
1  function makeAdvice(whitelist) {
2    var myWhitelist = whitelist;
3
4    return function(origf, obj, vp) {
5      if(myWhitelist.ROindexOf(vp[0])>=0) {
6        return origf.ROapply(obj, vp);
7      } else {
8        return false;
9      }
10   };
11 }
12
13 myAdvice = makeAdvice(['foo', 'bar']);
14 registerAdvice(myFunction, myAdvice);
15 disableAdviceRegistration();
```

Figure 3: Example advice function construction and weaving

The construction of a rather generic example advice function is shown in Figure 3. The listing shows a function `makeAdvice`, which returns an advice function as a closure containing the whitelist. Whenever the advice function is called for a function to which the first argument (`vp[0]`) is either 'foo' or 'bar', then the original function is executed. Otherwise, the advice function returns false.

Note that in the example, `ROindexOf` and `ROapply` are used. These functions were introduced to prevent prototype poisoning attacks against the WebJail infrastructure. They provide the same functionality as `indexOf` and `apply`, except that they have the `JSPROP_READONLY` and `JSPROP_PERMANENT` attributes set so they can not be modified or deleted.

Next, each *(advice, operation)* pair is passed on to the deep aspect weaving layer to achieve the deep aspect weaving.

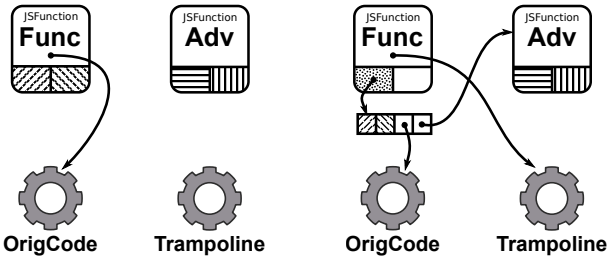
## 5.3 Deep aspect weaving layer

The deep aspect weaving layer makes sure that all codepaths to an advised function pass through its advice function. Although the code from WebJail is the first code to run in a WebJail iframe, we consider the scenario that there can be code or objects in place that already reference the function to be advised. It is necessary to maintain the existing references to a function, if they exist, so that advice weaving does not break code unintentionally.

The implementation of the deep aspect weaving layer is inspired by ConScript. To register deep advice, we introduce a new function called `registerAdvice`, which takes 2 arguments: the function to advise (also referred to as the 'original' function) and its advice function. Line 14 of Figure 3 illustrates the usage of the `registerAdvice` function.

In Spidermonkey, Mozilla's JavaScript engine, all JavaScript functions are represented by `JSFunction` objects. A `JSFunction` object can represent both a native function, as well as a JIT compiled JavaScript function. Because WebJail enforces policies on JavaScript APIs and all of these are implemented with native functions, our implementation only considers `JSFunction` objects which point to native code[2].

The process of registering advice for a function is schematically illustrated in Figure 4. Consider a native function `Func` and its advice function `Adv`. Before deep aspect weaving, the `JSFunction` object of `Func` contains a reference to a native C++ function `OrigCode`.




(a) Before weaving (b) After weaving

Figure 4: Schematic view of deep aspect weaving.

At weaving time, the value of the function pointer in `Func` (which points to `OrigCode`) and a reference to `Adv` are backed up inside the `Func` object. The function pointer inside `Func` is then directed towards the `Trampoline` function, which is an internal native C++ function provided by WebJail.

At function invocation time, the `Trampoline` function will be called as if it were the original function (`OrigCode`). This function can retrieve the values backed up in the weaving phase. From the backed up function pointer pointing to `OrigCode`, a new anonymous `JSFunction` object is created. This anonymous function, together with the current `this` object and the arguments to the `Trampoline` function are passed to the advice function `Adv`. Finally, the result from the advice function is returned to the calling code.

In reality, the `registerAdvice` function is slightly more complicated. In each `JSFunction` object, SpiderMonkey allocates 2 private values, known as "reserved slots", which can be used by Firefox to store opaque data. As shown in Figure 4, the reserved slots of `Func` (hatched diagonally) are backed up in the weaving phase together with the other values. During invocation time, these reserved slots are then restored into the anonymous function mentioned earlier.

Note that all code that referenced `Func` still works, although calls to this function will now pass through the advice function `Adv` first. Also note that no reference to the original code `OrigCode` is available. The only way to call this code is by making use of the advice function.

To prevent any other JavaScript code from having access to the `registerAdvice` function, it is disabled after all advice from the policy has been applied. For this purpose, WebJail provides the `disableAdviceRegistration` function, which disables the use of the `registerAdvice` function in the current JavaScript context.

# 6. EVALUATION

## 6.1 Performance

We performed micro-benchmarks on WebJail to evaluate its performance overhead with regard to page load-time and function execution. The prototype implementation is built on Mozilla Firefox 4.0b10pre, and compiled with the GNU C++ compiler v4.4.4-14ubuntu5. The benchmarks were performed on an Apple MacBook Pro 4.1, with an Intel Core 2 Duo T8300 CPU running at 2.40GHz and 4GB of memory, running Ubuntu 10.10 with Linux kernel version `2.6.35-28-generic`.

[2]Although WebJail could be implemented for non-native functions as well.

### 6.1.1 Page load-time overhead

To measure the page load-time overhead, we created a local webpage (`main.html`) that embeds another local page (`inner.html`) in an iframe with and without a local policy file. `inner.html` records a timestamp (`new Date().getTime()`) when the page starts and stops loading (using the body `onload` event). WebJail was modified to record the starttime before anything else executes, so that policy retrieval, loading and application is taken into account. After the results are submitted, `main.html` reloads.

We averaged the results of 1000 page reloads. Without WebJail, the average load-time was 16.22ms ($\sigma$ = 3.74ms). With WebJail, the average is 23.11ms ($\sigma$ = 2.76ms).

### 6.1.2 Function execution overhead

Similarly, we used 2 local pages (`main.html` and `inner.html`) to measure function execution overhead. `inner.html` measures how long it takes for 10000 iterations of a piece of code to execute. We measured 2 scenarios: a typical XMLHttpRequest invocation (constructor, `open` and `send` functions) and a localStorage set and get (`setItem` and `getItem`). Besides measuring a baseline without WebJail policy, we measured each scenario when restricted by 3 different policies: "allow all", "allow none" and a whitelist with 5 values. The averages are summarized in Table 2.

| | XMLHttpRequest | localStorage |
|---|---|---|
| Baseline | 1.25 ms | 0.37 ms |
| "Allow all" | 1.25 ms (+ 0%) | 0.37 ms (+ 0%) |
| "Allow none" | 0.07 ms (- 94.4%) | 0.04 ms (- 89.2 %) |
| Whitelist | 1.33 ms (+ 6.4%) | 0.47 ms (+ 27%) |

Table 2: Function execution overhead

To conclude, we have registered a negligible performance penalty for our WebJail prototype: a page load-time of 7ms, and an execution overhead in case of sensitive operations about 0.1ms.

## 6.2 Security

As discussed in Subsection 5.3, the `registerAdvice` function disconnects an available function and makes it available only to the advice function. Because of the use of deep aspects, we can ensure that no other references to the original function are available in the JavaScript environment, even if such references already existed before `registerAdvice` was called. We have successfully verified this full mediation of the deep aspects using our prototype implementation.

Because advice functions are written in JavaScript and the advice function has the only reference to the original function, it would be tempting for an attacker to attack the WebJail infrastructure. The retrieval and application of a WebJail policy happens before any other code is executed in the JavaScript context. In addition, the `registerAdvice` function is disabled once the policy has been applied. The only remaining attack surface is the advice function during its execution. The advice functions constructed by the advice construction layer are functionally equivalent to the example advice function created in Figure 3. We know of 3 attack vectors: prototype poisoning of `Array.prototype.indexOf` and `Function.prototype.apply`, and `toString` redefinition on `vp[0]` (the first argument to the example advice function in Figure 3). By introducing the readonly copies `ROindexOf` and `ROapply` (See Subsection 5.2), we prevent an attacker from exploiting the first 2 attack vectors. The third vector, `toString` redefinition, was verified in our prototype implementation and is not an issue because `toString` is never called on the argument `vp[0]`.

## 6.3 Applicability

To test the applicability of the WebJail architecture, we have applied our prototype implementation to mainstream mashup platforms, including iGoogle and Facebook. As part of the setup, we have instrumented responses from these platforms to include secure composition policies, by automatically injecting a *policy* attribute in selected iframes. Next, we have applied both permissive composition policies as well as restricted composition policies and verified that security-sensitive operations for the third-party components were executed as usual in the first case, and blocked in the latter case. For instance, as part of the applicability tests, we applied WebJail to control Geolocation functionality in the Google Latitude[11] component integrated into iGoogle, as well as external communication functionality of the third-party Facebook application "Tweets To Pages"[14] integrated into our Facebook page.

# 7. DISCUSSION AND FUTURE WORK

In the previous sections, we have showed the feasibility of the WebJail architecture via a prototype implementation in Firefox, and evaluated the performance, security and applicability. By applying micro-benchmarks, we measured a negligible overhead, we discussed how the WebJail architecture achieves full mediation via deep aspect weaving, and we briefly illustrated the applicability of WebJail in mainstream mashup platforms.

In this section, we will discuss some points of attention in realizing least-privilege integration in web mashups and some opportunities for further improvements.

First, the granularity chosen for the secure composition policies for WebJail is primarily driven by the ease of configuration for the mashup integrator. We strongly believe that the category level of granularity increases the adoption potential by integrators and browsers, for instance compared to semantically rich and expressive security policies as is currently the case in wrapper approaches or ConScript. In fact, we chose to introduce this policy abstraction to let the integrator focus on the "what" rather than the "how". A next step could be to define policy templates per mashup component type (e.g. advertisement and geotagging components).

Nevertheless, more fine-grained policies could also be applied to achieve least-privilege integration, but one should be aware of the potential risk of creating an inverse sandbox. The goal of a least-privilege integration architecture, such as WebJail, is to limit the functionality available to a (possibly) malicious component. In case the policy language is too expressive, an attacker could use this technology to achieve the inverse. An attacker could integrate a legitimate component into his website and impose a malicious policy on it. The result is effectively a hardcoded XSS attack in the browser. For instance, the attacker could introduce an advice that leaks all sensitive information out of a legitimate component as part of its least-privilege composition policy without being stopped by the Same-Origin Policy.

One particular area where we see opportunities for more fine-grained enforcement are cross-domain interactions. Ongoing research on Cross-Site Request Forgery (CSRF) [5,

6, 28, 20] already differentiates between benign and potentially malicious cross-domain requests, and restricts the latter class as part of a browser extension. This line of research could be seen as complementary to the presented approach, and a combination of both would allow a more fine-grained enforcement for cross-domain interactions.

Second, a possible technique to escape a modified JavaScript execution context in an iframe, would be to open a new window and execute JavaScript in there. We have anticipated this attack by hardcoding policies for e.g. the `window.open` function. This is however not the best approach. The upcoming HTML 5 specs include the sandbox attribute for iframes. This specification states that a sandbox should prevent content from creating new auxiliary browsing contexts. Mozilla Firefox does not support the sandbox attribute yet. The hardcoded policy for `window.open` is a quick fix while we are working on our own full implementation of the sandbox attribute in Mozilla Firefox.

Another way to escape WebJail is to access the window object of the parent or a sibling frame and make use of the functions in that JavaScript context (e.g. `parent.navigator.geolocation.getCurrentPosition`). In such a scenario, accessing another JavaScript context falls under the Same-Origin Policy and will only be possible if both the caller and callee are in the same origin. To avoid this attack, the WebJail implementation must restrict access to sensitive operations in other execution contexts under the Same-Origin Policy.

Thirdly, the categories in the policy files of WebJail are a result of a study of the sensitive JavaScript operations in the new HTML5 APIs. Most of the HTML5 APIs are working drafts and might change in the future. The category list in WebJail is therefore an up-to-date snapshot, but might be subject to change in the future. Even after the specifications for HTML5 are officially released, the functionality in browsers might keep changing. To cope with this evolving landscape, WebJail can easily be extended to support additional categories and APIs as well.

Finally, the WebJail architecture is tailored to support least-privilege integration in mashups that are built via iframe-integration. An interesting future track is to investigate how to enable browsers to support least-privilege script-inclusion integration as well. Since in such a scenario, one can not build on the fact that a separate execution context is created, we expect this to be a challenging trajectory.

# 8. RELATED WORK

There is a broad set of related work that focuses on the integration of untrusted JavaScript code in web applications.

### *JavaScript subsets.*

A common technique to prevent undesired behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed by the integrator.

ADSafe[3] and FBJS[31] requires third-party components to be written in a JavaScript subset that is known to be safe. The ADSafe subset removes several unsafe features from JavaScript (e.g. global variables, eval, ...) and provides safe alternatives through the `ADSAFE` object. Caja[23], Jacaranda[15] and Live Labs' Websandbox[22] take a different approach. Instead of heavily restricting the developer's language, they transform the JavaScript code into a safe version. The transformation process is based on both static analysis and rewriting to integrate runtime checks.

These techniques effectively support client-side least-privilege integration of mashup components. The main disadvantage is the tight coupling of the security features with the third-party component code. This requires control over the code, either at development or deployment time, which conflicts with legacy components and remote component delivery (R2), and reduces the applicability to mashup scenarios where the integrator delivers the components to the browser.

### *JavaScript instrumentation and access mediation.*

Instead of restricting a third-party component to a JavaScript subset, access to specific security-sensitive operations can be mediated. Mediation can consist of blocking the call, or letting a policy decide whether or not to allow it.

BrowserShield[26] is a server-side rewriting technique, that rewrites certain JavaScript functions to use safe equivalents. These safe equivalents are implemented in the "bshield" object that is introduced through the BrowserShield JavaScript libraries that are injected into each page. BrowserShield makes use of a proxy to inject its code into a webpage.

Self-protecting JavaScript[24, 19] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring any browser modifications. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. The advice is *non-deep* advice, meaning that by protecting one operation, different access paths to the same operation are not automatically protected. The main challenge of this approach is to ensure full mediation (R1) without breaking the component's legitimate functionality (e.g. via removal of prototypes), since both policy and third-party component code live in the same JavaScript context.

Browser-Enforced Embedded Policies (BEEP)[16] injects a policy script at the server-side. The browser will call this policy script before loading another script, giving the policy the opportunity to vet the script about to be loaded. The loading process will only continue after the approval of the policy. This approach offers control over which scripts are loaded, but is too coarse grained to assign privileges to specific components.

ConScript[21] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript, except that ConScript uses *deep* advice, thus protects all access paths to a function. The price for using deep advice is the need for client-side support in the JavaScript engine. A limitation of ConScript is that policies are not composition policies: the policies are provided by and applied to the same webpage, which conflicts with remote component delivery (R2) and secure composition policy configurable by the integrator (R3).

In contrast to the techniques described above, WebJail offers the integrator the possibility to define a policy that restricts the behavior of a third-party component in an isolated way. Additionally, all of the techniques above use JavaScript as a policy language. This amount of freedom complicates the writing of secure policies: protection against all the emerging HTML5 APIs is fully up to policy writer and can be error-prone, a problem that the WebJail policy language is not susceptible to.

### *Web application code and data analysis.*

A common protection technique against XSS vulnerabilities or attacks is server-side code or data analysis. Even though these techniques can only be used to check if a component matches certain security requirements and do not enforce a policy, we still discuss them here, since they are a server-side way to ensure that a component meets certain least-privilege integration requirements *out-of-the-box*.

Gatekeeper[12] is a mostly static [sic] enforcement mechanism designed to defend against possibly malicious JavaScript widgets on a hosting page. Gatekeeper analyzes the complete JavaScript code together with the hosting page. In addition, Gatekeeper uses runtime enforcement to disable dynamic JavaScript features.

XSS-Guard[2] aims to detect and remove scripts that are not intended to be present in a web application's output, thus effectively mitigating XSS attacks. XSS-Guard dynamically learns what set of scripts is used for an HTTP request. Using this knowledge, subsequent requests can be protected.

Recently, Mozilla proposed the Content Security Policy (CSP) [29], which allows the integrator to insert a security policy via response headers or meta tags. Unfortunately, CSP only supports restrictions on a subset of the security-sensitive operations discussed in this paper, namely operations potentially leading to content injection (e.g. script inclusion and XHR).

### *Information flow control.*

Information flow control techniques can be used to detect unauthorized information sharing or leaking between origins or external parties. This is extremely useful for applications that are allowed to use sensitive data, such as a location, but are not allowed to share that data.

Both Magazinius et al.[18] and Li et al.[17] have proposed an information flow control technique that prevents unauthorized sharing of data. Additionally, both techniques support authorized sharing by means of declassification, where a certain piece of data is no longer considered sensitive.

Secure multi-execution[9] detects information leakage by simultaneously running the code for each security level. This approach is a robust way to detect information leakage, but does not support declassification.

Information flow control techniques themselves are not suited for enforcing least-privilege integration. Likewise, WebJail is not suited to enforce information flow control, since it would be difficult to cover all possible leaks. Both techniques are complementary and can be used together to ensure least-privilege integration without unauthorized information leaking.

### *Isolating content using specialized HTML.*

Another approach to least-privilege integration is the isolation of untrusted content. By explicitly separating the untrusted code, it becomes easier to restrict its behavior, for example by preventing script execution.

The "untrusted" attribute[10] on a div element aims to allow the browser to make the difference between trusted and untrusted code. The idea is to enclose any untrusted content with such a div construct. This technique fails to defend against injecting closing tags, which would trivially circumvent the countermeasure.

The new "sandbox" attribute of the iframe element in HTML 5[13] provides a safer alternative, but is very coarse-grained. It only supports limited restrictions, and as far as JavaScript APIs are concerned, it only supports to completely enable or disable JavaScript.

ADJail[30] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to. Changes to the shadow page are replicated to the hosting page if those changes conform to the specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. ADJail limits DOM access and UI interaction with the component, but does not restrict the use of all other sensitive operations like WebJail can.

### *User-provided policies.*

Mozilla offers *Configurable Security Policies*[27], a user-configurable policy that is part of the browser. The policy allows the user to explicitly enable or disable certain capabilities for specific internet sites. An example is the option to disallow a certain site to open a popup window. Some parts of this idea have also been implemented in the Security zones of Internet Explorer.

The policies and enforcement mechanism offered by this technique resemble WebJail. The major difference is that these policies are user-configurable, and thus not under control of the integrator. Additionally, the policies do not support a different set of rules for the same included content, in two different scenarios, whereas WebJail does.

## 9. CONCLUSION

In this paper we have presented WebJail, a novel client-side security architecture to enables least-privilege integration of third-party components in web mashups. The WebJail security architecture is compatible with legacy mashup components, and allows the direct delivery of components from the service providers to the browser.

We have designed a secure composition language for WebJail, based on a study of security-sensitive operations in HTML5 APIs, and achieved full mediation by applying deep aspect weaving within the browser.

We have implemented a prototype of WebJail in Mozilla Firefox 4.0, and applied it successfully to mainstream platforms such as iGoogle and Facebook. In addition, we have evaluated the performance of the WebJail implementation using micro-benchmarks, showing that both the page load-time overhead (±7ms) and the execution overhead of a function advised with a whitelist policy (±0.1ms) are negligible.

## 10. ACKNOWLEDGMENTS

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT, the Research Fund K.U.Leuven and the EU-funded FP7-projects WebSand and NESSoS.

The authors would also like to thank Maarten Decat and Willem De Groef for their contribution to early proof-of-concept implementations [8, 33] to test the feasibility of the presented research.

## 11. REFERENCES

[1] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*,

52:83–91, June 2009.

[2] P. Bisht and V. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *5th GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assesment*, July 2008.

[3] D. Crockford. ADsafe – making JavaScript safe for advertising. `http://adsafe.org/`.

[4] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *15th Nordic Conference in Secure IT Systems (NordSec 2010)*. Springer, 2011.

[5] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Lecture Notes in Computer Science*, volume 5965, pages 18–34. Springer Berlin / Heidelberg, February 2010.

[6] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. In V. Atluri and C. Diaz, editors, *Computer Security - ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 100–116. Springer Berlin / Heidelberg, 2011.

[7] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A security analysis of next generation web standards. Technical report, G. Hogben and M. Dekker (Eds.), European Network and Information Security Agency (ENISA), July 2011.

[8] M. Decat. Ondersteuning voor veilige Web Mashups. Master's thesis, Katholieke Universiteit Leuven, 2010.

[9] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. *2010 IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[10] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[11] Google. Google Latitude. `https://www.google.com/latitude/`.

[12] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[13] I. Hickson and D. Hyatt. HTML 5 Working Draft - The sandbox Attribute. `http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox`, June 2010.

[14] Involver. Tweets To Pages. `http://www.facebook.com/TweetsApp`.

[15] Jacaranda. Jacaranda. `http://jacaranda.org`.

[16] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[17] Z. Li, K. Zhang, and X. Wang. Mash-if: Practical information-flow control within client-side mashups. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 251 –260, 28 2010-july 1 2010.

[18] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 15–23, New York, NY, USA, 2010. ACM.

[19] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *The 15th Nordic Conf. in Secure IT Systems. Springer Verlag*, 2010.

[20] G. Maone. Noscript 2.0.9.9. `http://noscript.net/`, 2011.

[21] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[22] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`.

[23] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[24] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[25] Programmable Web. Keeping you up to date with APIs, mashups and the Web as platform. `http://www.programmableweb.com/`.

[26] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[27] J. Ruderman. Configurable Security Policies. `http://www.mozilla.org/projects/security/components/ConfigPolicy.html`.

[28] J. Samuel. Requestpolicy 0.5.20. `http://www.requestpolicy.com`, 2011.

[29] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[30] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium*, Aug. 2010.

[31] The FaceBook Team. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`.

[32] W3C. W3C Standards and drafts - Javascript APIs. `http://www.w3.org/TR/#tr_Javascript_APIs`.

[33] Willem De Groef. ConScript For Firefox. `http://cqrit.be/conscript/`.

[34] M. Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Main`, 2010.

# Automatic and Precise Client-Side Protection against CSRF Attacks

Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens

IBBT-DistriNet
Katholieke Universiteit Leuven
3001 Leuven, Belgium
{philippe.deryck,lieven.desmet}@cs.kuleuven.be

**Abstract** A common client-side countermeasure against Cross Site Request Forgery (CSRF) is to strip session and authentication information from malicious requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures are typically too strict, thus breaking many existing websites that rely on authenticated cross-origin requests, such as sites that use third-party payment or single sign-on solutions.
The contribution of this paper is the design, implementation and evaluation of a request filtering algorithm that automatically and precisely identifies *expected* cross-origin requests, based on whether they are preceded by certain indicators of collaboration between sites. We formally show through bounded-scope model checking that our algorithm protects against CSRF attacks under one specific assumption about the way in which good sites collaborate cross-origin. We provide experimental evidence that this assumption is realistic: in a data set of 4.7 million HTTP requests involving over 20.000 origins, we only found 10 origins that violate the assumption. Hence, the remaining attack surface for CSRF attacks is very small. In addition, we show that our filtering does not break typical non-malicious cross-origin collaboration scenarios such as payment and single sign-on.

**keywords:** CSRF, web security, browser security.

## 1 Introduction

From a security perspective, web browsers are a key component of today's software infrastructure. A browser user might have a session with a trusted site A (e.g. a bank, or a webmail provider) open in one tab, and a session with a potentially dangerous site B (e.g. a site offering cracks for games) open in another tab. Hence, the browser enforces some form of isolation between these two origins A and B through a heterogeneous collection of security controls collectively known as the *same-origin-policy* [18]. An *origin* is a (protocol, domain name, port) triple, and restrictions are imposed on the way in which code and data from different origins can interact. This includes for instance restrictions that prevent scripts from origin B to access content from origin A.

An important known vulnerability in this isolation is the fact that content from origin B can initiate requests to origin A, and that the browser will treat these requests as being part of the ongoing session with A. In particular, if the session with A was authenticated, the injected requests will appear to A as part of this authenticated session. This enables an attack known as *Cross Site Request Forgery (CSRF)*: B can initiate effectful requests to A (e.g. a bank transaction, or manipulations of the victim's mailbox or address book) without the user being involved.

CSRF has been recognized since several years as one of the most important web vulnerabilities [3], and many countermeasures have been proposed. Several authors have proposed server-side countermeasures [3,4,10]. However, an important disadvantage of server-side countermeasures is that they require modifications of server-side programs, have a direct operational impact (e.g. on performance or maintenance), and it will take many years before a substantial fraction of the web has been updated.

Alternatively, countermeasures can be applied on the client-side, as browser extensions. The basic idea is simple: the browser can strip session and authentication information from malicious requests, or it can block such requests. The difficulty however is in determining when a request is malicious. Existing client-side countermeasures [9,5,12,13,16,19] are typically too strict: they block or strip all cross-origin requests of a specific type (e.g. GET, POST, any). This effectively protects against CSRF attacks, but it unfortunately also breaks many existing websites that rely on authenticated cross-origin requests. Two important examples are sites that use third-party payment (such as PayPal) or single sign-on solutions (such as OpenID). Hence, these existing client-side countermeasures require extensive help from the user, for instance by asking the user to define white-lists of trusted sites or by popping up user confirmation dialogs. This is suboptimal, as it is well-known that the average web user can not be expected to make accurate security decisions.

This paper proposes a novel client-side CSRF countermeasure, that includes an automatic and precise filtering algorithm for cross-origin requests. It is *automatic* in the sense that no user interaction or configuration is required. It is *precise* in the sense that it distinguishes well between malicious and non-malicious requests. More specifically, through a systematic analysis of logs of web traffic, we identify a characteristic of non-malicious cross-origin requests that we call the *trusted-delegation assumption*: a request from B to A can be considered non-malicious if, earlier in the session, A explicitly delegated control to B in some specific ways. Our filtering algorithm relies on this assumption: it will strip session and authentication information from cross-origin requests, unless it can determine that such explicit delegation has happened.

We validate our proposed countermeasure in several ways. First, we formalize the algorithm and the trusted-delegation assumption in Alloy, building on the formal model of the web proposed by [1], and we show through bounded-scope model checking that our algorithm protects against CSRF attacks under this assumption. Next, we provide experimental evidence that this assumption is

realistic: through a detailed analysis of logs of web traffic, we quantify how often the trusted-delegation assumption holds, and show that the remaining attack surface for CSRF attacks is very small. Finally, we have implemented our filtering algorithm as an extension of an existing client-side CSRF protection mechanism, and we show that our filtering does not break typical non-malicious cross-origin collaboration scenarios such as payment and single sign-on.

In summary, the contributions of this paper are:

- The design of a novel client-side CSRF protection mechanism based on request filtering.
- A formalization of the algorithm, and formal evidence of the security of the algorithm under one specific assumption, the trusted-delegation assumption.
- An implementation of the countermeasure, and a validation of its compatibility with important web scenarios broken by other state-of-the-art countermeasures.
- An experimental evaluation of the validity of the trusted-delegation assumption.

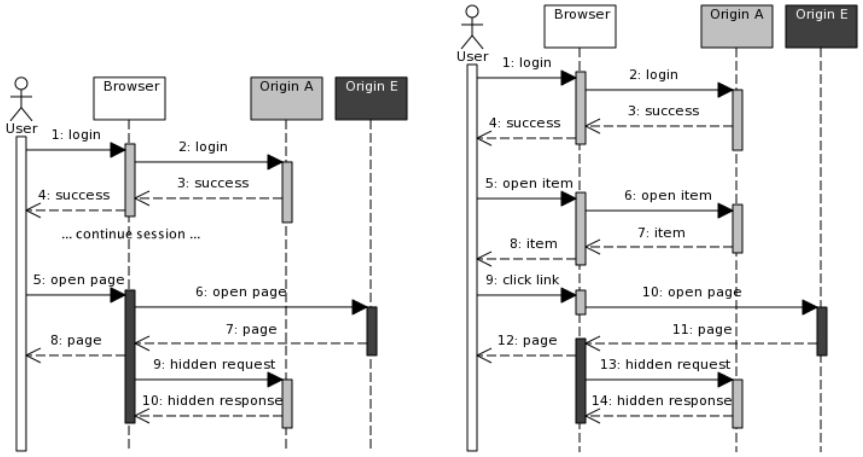
The remainder of the paper is structured as follows. Section 2 explains the problem using both malicious and non-malicious scenarios. Section 3 discusses our request filtering mechanism. Section 4 introduces the formalization and results, followed by the implementation in Section 5. Section 6 experimentally evaluates the trusted-delegation assumption. Finally, Section 7 extensively discusses related work, followed by a brief conclusion (Section 8).

## 2 Cross-origin HTTP Requests

The key challenge for a client-side CSRF prevention mechanism is to distinguish malicious from non-malicious cross-origin requests. This section illustrates the difficulty of this distinction by describing some attack scenarios and some important non-malicious scenarios that intrinsically rely on cross-origin requests.

### 2.1 Attack Scenarios

*A1. Classic CSRF* Figure 1(a) shows a classic CSRF attack. In steps 1–4, the user establishes an authenticated session with site A, and later (steps 5–8) the user opens the malicious site E in another tab of the browser. The malicious page from E triggers a request to A (step 9), the browser considers this request to be part of the ongoing session with A and automatically adds the necessary authentication and session information. The browser internally maintains different *browsing contexts* for each origin it is interacting with. The shade of the browser-lifeline in the figure indicates the origin associated with the browsing context from which the outgoing request originates (also known as *the referrer*). Since the attack request originates from an E browsing context and goes to origin A, it is *cross-origin*.




(a) Classic CSRF (b) Link injection

**Figure 1.** CSRF attack scenarios

For the attack to be successful, an authenticated session with A must exist when the user surfs to the malicious site E. The likelihood of success can be increased by making E content-related to A, for instance to attack a banking site, the attacker poses as a site offering financial advice.

*A2. Link Injection* To further increase the likelihood of success, the attacker can inject links to E into the site A. Many sites, for instance social networking sites, allow users to generate content which is displayed to other users. For such a site A, the attacker creates a content item which contains a link to E. Figure 1(b) shows the resulting CSRF scenario, where A is a social networking site and E is the malicious site. The user logs into A (steps 1–4), opens the attacker injected content (steps 5–8), and clicks on the link to E (step 9) which launches the CSRF attack (step 13). Again, the attack request is cross-origin.

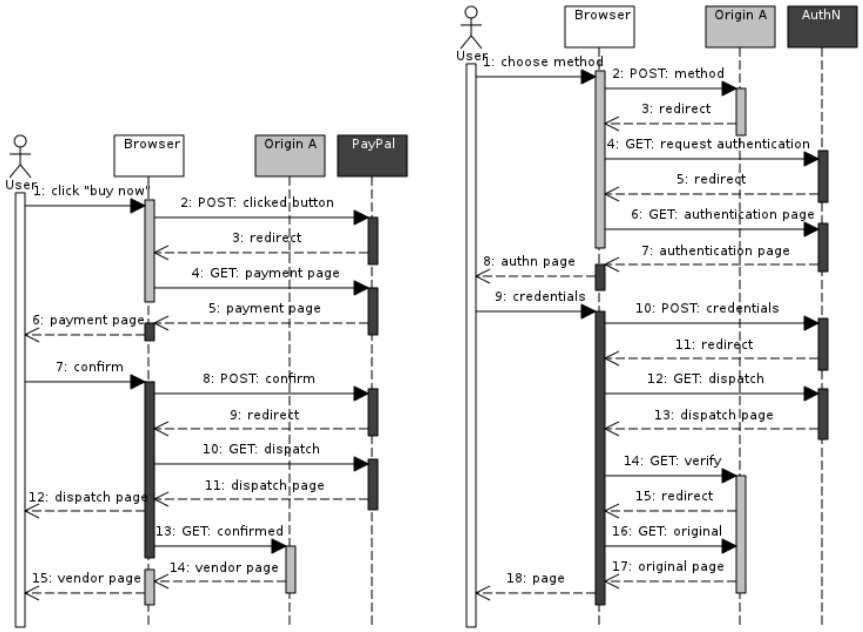
### 2.2 Non-malicious Cross-Origin Scenarios

CSRF attack requests are cross-origin requests in an authenticated session. Hence, forbidding such requests is a secure countermeasure. Unfortunately, this also breaks many useful non-malicious scenarios. We illustrate two important ones.

*F1. Payment Provider* Third-party payment providers such as PayPal or Visa 3D-secure offer payment services to a variety of sites on the web.

Figure 2(a) shows the scenario for PayPal's *Buy Now* button. When a user clicks on this button, the browser sends a request to PayPal (step 2), that redirects the user to the payment page (step 4). When the user accepts the

payment (step 7), the processing page redirects the browser to the dispatch page (step 10), that loads the landing page of the site that requested the payment (step 13).




(a) Payment scenario (b) Central authentication scenario

**Figure 2.** Non-malicious cross-origin scenarios

Note that step 13 is a cross-origin request from PayPal to A in an authenticated session, for instance a shopping session in web shop A.

*F2. Central Authentication* The majority of interactive websites require some form of authentication. As an alternative to each site using its own authentication mechanism, a single sign-on service (such as OpenID or Windows Live ID) provides a central point of authentication.

An example scenario for OpenID authentication using MyOpenID is shown in Figure 2(b). The user chooses the authentication method (step 1), followed by a redirect from the site to the authentication provider (step 4). The authentication provider redirects the user to the login form (step 6). The user enters the required credentials, which are processed by the provider (step 10). After verification, the provider redirects the browser to the dispatching page (step 12), that redirects to the processing page on the original site (step 14). After processing the authentication result, a redirect loads the requested page on the original site (step 16).

Again, note that step 16 is a cross-origin request in an authenticated session.

These two scenarios illustrate that mitigating CSRF attacks by preventing cross-origin requests in authenticated sessions breaks important and useful web scenarios. Existing client-side countermeasures against CSRF attacks [5,13,16] either are incompatible with such scenarios or require user interaction for these cases.

## 3 Automatic and Precise Request Stripping

The core idea of our new countermeasure is the following: client-side state (i.e. session cookie headers and authentication headers) is stripped from all cross-origin requests, except for *expected* requests. A cross-origin request from origin A to B is *expected* if B previously (earlier in the browsing session) *delegated* to A. We say that B *delegates* to A if B either issues a POST request to A, or if B redirects to A using a URI that contains parameters.

The rationale behind this core idea is that (1) non-malicious collaboration scenarios follow this pattern, and (2) it is hard for an attacker to trick A into delegating to a site of the attacker: forcing A to do a POST or parametrized redirect to an evil site E requires the attacker to either identify a cross-site scripting (XSS) vulnerability in A, or to break into A's webserver. In both these cases, A has more serious problems than CSRF.

Obviously, a GET request from A to B is not considered a delegation, as it is very common for sites to issue GET requests to other sites, and as it is easy for an attacker to trick A into issuing such a GET request (see for instance attack scenario A2 in Section 2).

Unfortunately, the elaboration of this simple core idea is complicated somewhat by the existence of HTTP redirects. A web server can respond to a request with a *redirect* response, indicating to the browser that it should resend the request elsewhere, for instance because the requested resource was moved. The browser will follow the redirect automatically, without user intervention. Redirects are used widely and for a variety of purposes, so we cannot ignore them. For instance, both non-malicious scenarios in Section 2 heavily depend on the use of redirects. In addition, attacker-controlled websites can also use redirects in an attempt to bypass client-side CSRF protection. Akhawe et al. [1] discuss several examples of how attackers can use redirects to attack web applications, including an attack against a CSRF countermeasure. Hence, correctly dealing with redirects is a key requirement for security.

The flowgraph in Figure 3 summarizes our filtering algorithm. For a given request, it determines what session state (cookies and authentication headers) the browser should attach to the request. The algorithm differentiates between simple requests and requests that are the result of a redirect.

*Simple Requests* Simple requests that are not cross-origin, as well as expected cross-origin requests are handled as unprotected browsers handle them today.

The browser automatically attaches the last known client-side state associated with the destination origin (point 1). The browser does not attach any state to non-expected cross-origin requests (point 3).

*Redirect Requests* If a request is the consequence of a redirect response, then the algorithm determines if the redirect points to the origin where the response came from. If this is the case, the client-side state for the new request is limited to the client-side state known to the previous request (i.e. the request that triggered this redirect) (point 2). If the redirect points to another origin, then, depending on whether this cross-origin request is expected or not, it either gets session-state automatically attached (point 1) or not (point 3).

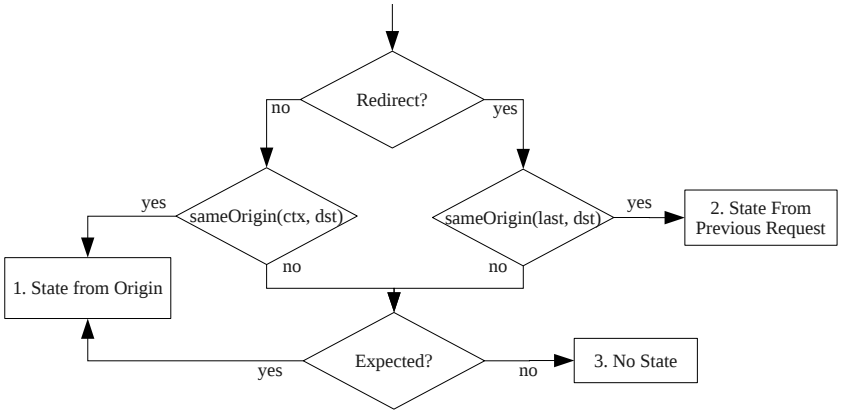



**Figure 3.** The request filtering algorithm

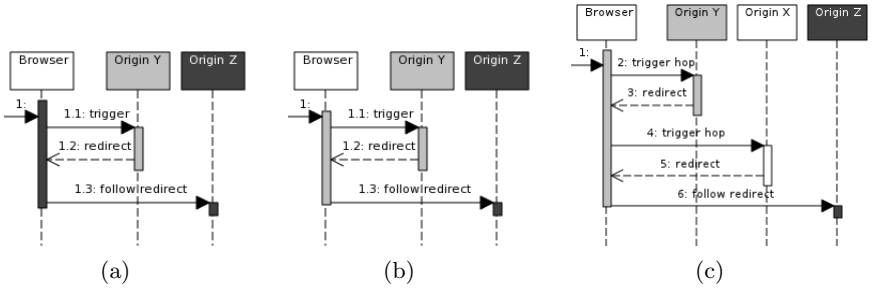
*When Is a Request Expected?* A key element of the algorithm is determining whether a request is *expected* or not. As discussed above, the intuition is: a cross-origin request from B to A is expected if and only if A first delegated to B by issuing a POST request to B, or by a parametrized redirect to B. Our algorithm stores such trusted delegations, and an assumption that we rely on (and that we refer to as the *trusted-delegation assumption*) is that sites will only perform such delegations to sites that they trust. In other words, a site A remains vulnerable to CSRF attacks from origins to which it delegates. Section 6 provides experimental evidence for the validity of this assumption.

The algorithm to decide whether a request is expected goes as follows.

For a simple cross-origin request from site B to site A, a trusted delegation from site A to B needs to be present in the delegation store.

For a redirect request that redirects a request to origin Y (light gray) to another origin Z (dark gray) in a browsing context associated with some origin $\alpha$, the following rules apply.

1. First, if the destination (Z) equals the source (i.e. $\alpha = Z$) (Figure 4(a)), then the request is expected if there is a trusted delegation from Z to Y in the delegation store. Indeed, Y is effectively doing a cross-origin request to Z by redirecting to Z. Since the browsing context has the same origin as the destination, it can be expected not to manipulate redirect requests to misrepresent source origins of redirects (cfr next case).
2. Alternatively, if the destination (Z) is not equal to the source (i.e. $\alpha \neq Z$) (Figure 4(b)), then the request is expected if there is a trusted delegation from Z to Y in the delegation store, since Y is effectively doing a cross-origin request to Z. Now, the browsing context might misrepresent source origins of redirects by including additional redirect hops (origin X (white) in Figure 4(c)). Hence, our decision to classify the request does not involve X.




(a) (b) (c)

**Figure 4.** Complex cross-origin redirect scenarios

Finally, our algorithm imposes that expected cross-origin requests can only use the GET method and that only two origins can be involved in the request chain. These restrictions limit the potential power an attacker might have, even if the attacker successfully deceives the trusted-delegation mechanism.

*Mapping to Scenarios* The reader can easily check that the algorithm blocks the attack scenarios from Section 2, and supports the non-malicious scenarios from that section. We discuss two of them in more detail.

In the PayPal scenario (Figure 2(a)), step 13 needs to re-use the state already established in step 2, which means that according to the algorithm, the request from PayPal to A should be expected. A trusted delegation happens in step 2, where a cross-origin POST is sent from origin A to PayPal. Hence the GET request in step 13 is considered expected and can use the state associated with origin A. Also note how the algorithm maintains the established session with PayPal throughout the scenario. The session is first established in step 3. Step 4 can use this session, because the redirect is an internal redirect on the PayPal origin. Step 8 can use the last known state for the PayPal origin and step 10 is yet another internal redirect.

In the link injection attack (Figure 1(b)), the attack happens in step 13 and is launched from origin E to site A. In this scenario, an explicit link between A and E exists because of the link injected by the attacker. This link is however not a POST or parametrized redirect, so it is not a trusted delegation. This means that the request in step 10 is not considered to be expected, so it can not access the previously established client-side state, and the attack is mitigated.

## 4 Formal Modeling and Checking

The design of web security mechanisms is complex: the behaviour of (same-origin and cross-origin) browser requests, server responses and redirects, cookie and session management, as well as the often implicit threat models of web security can lead to subtle security bugs in new features or countermeasures. In order to evaluate proposals for new web mechanisms more rigorously, Akhawe et al. [1] have proposed a model of the Web infrastructure, formalized in Alloy.

The base model is some 2000 lines of Alloy source code, describing (1) the essential characteristics of browsers, web servers, cookie management and the HTTP protocol, and (2) a collection of relevant threat models for the web. The Alloy Analyzer – a bounded-scope model checker – can then produce counter-examples that violate intended security properties if they exist in a specified finite scope.

In this section, we briefly introduce Akhawe's model and present our extensions to the model. We also discuss how the model was used to verify the absence of attack scenarios and the presence of functional scenarios.

### 4.1 Modeling our countermeasure

The model of Akhawe et al. defines different principals, of which `GOOD` and `WEBATTACKER` are most relevant. `GOOD` represents an honest principal, who follows the rules imposed by the technical specifications. A `WEBATTACKER` is a malicious user who can control malicious web servers, but has no extended networking capabilities.

The concept of `Origin` is used to differentiate between origins, which correspond to domains in the real world. An origin is linked with a server on the web, that can be controlled by a principal. The browsing context, modeled as a `ScriptContext`, is also associated with an `origin`, that represents the origin of the currently loaded page, also known as the referrer.

A `ScriptContext` can be the source of a set of `HTTPTransaction` objects, which are a pair of an `HTTPRequest` and `HTTPResponse`. An HTTP request and response are also associated with their remote destination origin. Both an HTTP request and response can have headers, where respectively the `CookieHeader` and `SetCookieHeader` are the most relevant ones. An HTTP request also has a `method`, such as `GET` or `POST`, and a `queryString`, representing URI parameters. An HTTP response has a `statusCode`, such as `c200` for a content result or `c302` for a redirect. Finally, an HTTP transaction has a cause, which can be none,

such as the user opening a new page, a `RequestAPI`, such as a scripting API, or another `HTTPTransaction`, in case of a redirect.

To model our approach, we need to extend the model of Akhawe et al. to include (a) the accessible client-side state at a certain point in time, (b) the trusted delegation assumption and (c) our filtering algorithm. We discuss (a) and (b) in detail, but due to space constraints we omit the code for the filtering algorithm (c), which is simply a literal implementation of the algorithm discussed in Section 3.

*Client-Side State* We introduced a new signature `CSState` that represents a client-side state (Listing 1.1). Such a state is associated with an `Origin` and contains a set of `Cookie` objects. To associate a client-side state with a given request or response and a given point in time, we have opted to extend the `HTTPTransaction` from the original model into a `CSStateHTTPTransaction`. Such an extended transaction includes a `beforeState` and `afterState`, respectively representing the accessible client-side state at the time of sending the request and the updated client-side state after having received the response. The `afterState` is equal to the `beforeState`, with the potential addition of new cookies, set in the response.

```
sig CSState {
  dst: Origin,
  cookies: set Cookie
}

sig CSStateHTTPTransaction extends HTTPTransaction {
  beforeState : CSState,
  afterState : CSState
} {
  //The after state of a transaction is equal to the before state + any additional cookies set in the
       response
  beforeState.dst = afterState.dst
  afterState.cookies = beforeState.cookies + (resp.headers & SetCookieHeader).thecookie

  // The destination origin of the state must correspond to the transaction destination origin
  beforeState.dst = req.host
}
```

**Listing 1.1.** Signatures representing our data in the model

*Trusted-delegation Assumption* We model the trusted-delegation assumption as a fact, that honest servers do not send a POST or parametrized redirect to web attackers ((Listing 1.2).

```
fact TrustedDelegation {
  all r : HTTPRequest | {
    (r.method = POST || some (req.r).cause & CSStateHTTPTransaction)
      &&
    ((some (req.r).cause & CSStateHTTPTransaction && getPrincipalFromOrigin[(req.r).cause.req.
        host] in GOOD) || getPrincipalFromOrigin[transactions.(req.r).owner] in GOOD)
    implies
      getPrincipalFromOrigin[r.host] not in WEBATTACKER
  }
}
```

**Listing 1.2.** The fact modeling the trusted-delegation assumption

### 4.2 Using Model Checking for Security and Functionality

We formally define a CSRF attack as the possibility for a web attacker (defined in the base model) to inject a request with at least one existing cookie attached to it (this cookie models the session/authentication information attached to requests) in a session between a user and an honest server (Listing 1.3).

```
pred CSRF[r : HTTPRequest] {
    //Ensure that the request goes to an honest server
    some getPrincipalFromOrigin[r.host]
    getPrincipalFromOrigin[r.host] in GOOD

    //Ensure that an attacker is involved in the request
    some (WEBATTACKER.servers & involvedServers[req.r]) || getPrincipalFromOrigin[(
         transactions.(req.r)).owner] in WEBATTACKER

    // Make sure that at least one cookie is present
    some c : (r.headers & CookieHeader).thecookie | {
      //Ensure that the cookie value is fresh (i.e. that it is not a renewed value in a redirect chain
           )
      not c in ((req.r).*cause.resp.headers & SetCookieHeader).thecookie
    }
}
```

**Listing 1.3.** The predicate modeling a CSRF attack

We provided the Alloy Analyzer with a universe of at most 9 HTTP events and where an attacker can control up to 3 origins and servers (a similar size as used in [1]). In such a universe, no examples of an attacker injecting a request through the user's browser were found. This gives strong assurance that the countermeasure does indeed protect against CSRF under the trusted delegation assumption.

We also modeled the non-malicious scenarios from Section 2, and the Alloy Analyzer reports that these scenarios are indeed permitted. From this, we can also conclude that our extension of the base model is consistent.

Space limitations do not permit us to discuss the detailed scenarios present in our model, but the interested reader can find the complete model available for download at [6].

## 5 Implementation

We have implemented our request filtering algorithm in a proof-of-concept add-on for the Firefox browser, and used this implementation to conduct an extensive practical evaluation. First, we have created simulations for both the common attack scenarios as well as the two functional scenarios discussed in the paper (third party payment and centralized authentication), and verified that they behaved as expected.

Second, in addition to these simulated scenarios, we have verified that the prototype supports actual instances of these scenarios, such as for example the use of MyOpenID authentication on sourceforge.net.

| | Test scenarios | Result |
|---|---|---|
| HTML | 29 cross-origin test scenarios | ✓ |
| CSS | 12 cross-origin test scenarios | ✓ |
| ECMAScript | 9 cross-origin test scenarios | ✓ |
| Redirects | 20 cross-origin redirect scenarios | ✓ |

**Table 1.** CSRF benchmark

Third, we have constructed and performed a CSRF benchmark[1], consisting of 70 CSRF attack scenarios to evaluate the effectiveness of our CSRF prevention technique (see Table 1). These scenarios are the result of a CSRF-specific study of the HTTP protocol, the HTML specification and the CSS markup language to examine their cross-origin traffic capabilities, and include complex redirect scenarios as well. Our implementation has been evaluated against each of these scenarios, and our prototype passed all tests successfully.

The prototype, the scenario simulations and the CSRF benchmark suite are available for download [6].

## 6 Evaluating the Trusted-Delegation Assumption

Our countermeasure drastically reduces the attack surface for CSRF attacks. Without CSRF countermeasures in place, an origin can be attacked by any other origin on the web. With our countermeasure, an origin can only be attacked by another origin to which it has delegated control explicitly by means of a cross-origin POST or redirect. We have already argued in Section 3 that it is difficult for an attacker to cause unintended delegations. In this section, we measure the remaining attack surface experimentally.

We conducted an extensive traffic analysis using a real-life data set of 4.729.217 HTTP requests, collected from 50 unique users over a period of 10 weeks. The analysis revealed that 1.17% of the 4.7 million requests are treated as delegations in our approach. We manually analyzed all these 55.300 requests, and classified them in the interaction categories summarized in Table 2.

For each of the categories, we discuss the resulting attack surface:

**Third party service mashups.** This category consists of various third party services that can be integrated in other websites. Except for the single sign-on services, this is typically done by script inclusion, after which the included script can launch a sequence of cross-origin GET and/or POST requests towards offered AJAX APIs. In addition, the service providers themselves often use cross-origin redirects for further delegation towards content delivery networks.

As a consequence, the origin A including the third-party service S becomes vulnerable to CSRF attacks from S. This attack surface is unimportant, as in these scenarios, S can already attack A through script inclusion, a more powerful attack than CSRF.

[1] The benchmark can be applied to other client-side solutions as well, and is downloadable at [6].

| | # requests | | POST | redir. |
|---|---|---|---|---|
| Third party service mashups | 29.282 | (52,95%) | 5.321 | 23.961 |
| *Advertisement services* | *22.343* | *(40,40%)* | *1.987* | *20.356* |
| *Gadget provider services (appspot, mochibot, gmodules, ...)* | *2.879* | *(5,21%)* | *2.757* | *122* |
| *Tracking services (metriweb, sitestat, uts.amazon, ...)* | *2.864* | *(5,18%)* | *411* | *2.453* |
| *Single Sign-On services (Shibboleth, Live ID, OpenId, ...)* | *1.156* | *(2,09%)* | *137* | *1.019* |
| *3rd party payment services (Paypal, Ogone)* | *27* | *(0,05%)* | *19* | *8* |
| *Content sharing services (addtoany, sharethis, ...)* | *13* | *(0,02%)* | *10* | *3* |
| Multi-origin websites | 13.973 | (25,27%) | 198 | 13.775 |
| Content aggregators | 8.276 | (14,97%) | 0 | 8.276 |
| *Feeds (RSS feeds, News aggregators, mozilla fxfeeds, ...)* | *4.857* | *(8,78%)* | *0* | *4.857* |
| *Redirecting search engines (Google, Comicranks, Ohnorobot)* | *3.344* | *(6,05%)* | *0* | *3.344* |
| *Document repositories (ACM digital library, dx.doi.org, ...)* | *75* | *(0,14%)* | *0* | *75* |
| False positives (wireless network access gateways) | 1.215 | (2,20%) | 12 | 1.203 |
| URL shorteners (gravatar, bit.ly, tinyurl, ...) | 759 | (1,37%) | 0 | 759 |
| Others (unclassified) | 1.795 | (3,24%) | 302 | 1.493 |
| **Total number of 3rd party delegation initiators** | **55.300** | **(100%)** | **5.833** | **49.467** |

**Table 2.** Analysis of the trusted-delegation assumption in a real-life data set of 4.729.217 HTTP requests

In addition, advertisement service providers P that further redirect to content delivery services D are vulnerable to CSRF attacks from D whenever a user clicks an advertisement. Again, this attack surface is unimportant: the delegation from P to D correctly reflects a level of trust that P has in D, and P and D will typically have a legal contract or SLA in place.

**Multi-origin websites.** Quite a number of larger companies and organizations have websites spanning multiple origins (such as *live.com* - *microsoft.com* and *google.be* - *google.com*). Cross-origin POST requests and redirects between these origins make it possible for such origins to attack each other. For instance, *google.be* could attack *google.com*. Again, this attack surface is unimportant, as all origins of such a multi-origin website belong to a single organization.

**Content aggregators.** Content aggregators collect searchable content and redirect end-users towards a specific content provider. For news feeds and document repositories (such as the ACM digital library), the set of content providers is typically stable and trusted by the content aggregator, and therefore again a negligible attack vector.

Redirecting search engines register the fact that a web user is following a link, before redirecting the web user to the landing page (e.g. as Google does for logged in users). Since the entries in the search repository come from all over the web, our CSRF countermeasure provides little protection for such search engines. Our analysis identified 4 such origins in the data set: *google.be*, *google.com*, *comicrank.com*, and *ohnorobot.com*.

**False positives.** Some fraction of the cross-origin requests are caused by network access gateways (e.g. on public Wifi) that intercept and reroute requests towards a payment gateway. Since such devices have man-in-the-middle capabilities, and hence more attack power than CSRF attacks, the resulting attack surface is again negligible.

**URL shorteners.** To ease URL sharing, URL shorteners transform a shortened URL into a preconfigured URL via a redirect. Since such URL shortening

services are open, an attacker can easily control a new redirect target. The effect is similar to the redirecting search engines; URL shorteners are essentially left unprotected by our countermeasure. Our analysis identified 6 such services in the data set: *bit.ly*, *gravatar.com*, *post.ly*, *tiny.cc*, *tinyurl.com*, and *twitpic.com*.

**Others(unclassified)** For some of the requests in our data set, the origins involved in the request were no longer online, or the (partially anonymized) data did not contain sufficient information to reconstruct what was happening, and we were unable to classify or further investigate these requests.

In summary, our experimental analysis shows that the trusted delegation assumption is realistic. Only 10 out of 23.592 origins (i.e. 0.0042% of the examined origins) – the redirecting search engines and the URL shorteners – perform delegations to arbitrary other origins. They are left unprotected by our countermeasure. But the overwhelming majority of origins delegates (in our precise technical sense, i.e. using cross-origin POST or redirect) only to other origins with whom they have a trust relationship.

## 7 Related Work

The most straightforward protection technique against CSRF attacks is server-side mitigation via validation tokens [4,10]. In this approach, web forms are augmented with a server-generated, unique validation token (e.g. embedded as a hidden field in the form), and at form submission the server checks the validity of the token before executing the requested action. At the client-side, validation tokens are protected from cross-origin attackers by the same-origin-policy, distinguishing them from session cookies or authentication credentials that are automatically attached to any outgoing request. Such a token based approach can be offered as part of the web application framework [14,7], as a server-side library or filter [15], or as a server-side proxy [10].

Recently, the `Origin` header has been proposed as a new server-side countermeasure [3,2]. With the `Origin` header, clients unambiguously inform the server about the origin of the request (or the absence of it) in a more privacy-friendly way than the `Referer` header. Based on this origin information, the server can safely decide whether or not to accept the request. In follow-up work, the `Origin` header has been improved, after a formal evaluation revealed a previously unknown vulnerability [1]. The Alloy model used in this evaluation also formed the basis for the formal validation of our presented technique in Section 4.

Unfortunately, the adoption rate of these server-side protection mechanisms is slow, giving momentum to client-side mitigation techniques as important (but hopefully transitional) solutions. In the next paragraphs, we will discuss the client-side proxy RequestRodeo, as well as 4 mature and popular browser addons (CsFire, NoScript ABE, RequestPolicy, and CSD[2]). In addition, we will evaluate

[2] Since the client-side detection technique described in [17] is not available for download, the evaluation is done based on the description in the paper.

how well the browser addons enable the functional scenarios and protect against the attack scenarios discussed in this paper (see Table 3).

RequestRodeo [9] is a client-side proxy proposed by Johns and Winter. The proxy applies a client-side token-based approach to tie requests to the correct source origin. In case a valid token is lacking for an outgoing request, the request is considered suspicious and gets stripped of cookies and HTTP `authorization` headers. RequestRodeo lies at the basis of most of the client-side CSRF solutions [5,12,13,16,19], but because of the choice of a proxy, RequestRodeo often lacks context information, and the rewriting technique on raw responses does not scale well in a web 2.0 world.

| | Functional scenarios | | Attack scenarios | |
|---|---|---|---|---|
| | F1. Payment Provider | F2. Central Authentication | A1. Classic CSRF | A2. Link Injection |
| CsFire [5] | × | × | ✓ | ✓ |
| NoScript ABE [13] [a] | × | × | ✓ | ✓ |
| RequestPolicy [16] | ⋈ [b] | ⋈ [b] | ✓[c] | ✓[c] |
| Client-Side Detection [17] | × | × | ✓ | ✓ |
| **Our Approach** | ✓ | ✓ | ✓ | ✓ |

[a] ABE configured as described in [8]
[b] Requires interactive feedback from end-user to make the decision
[c] Requests are blocked instead of stripped, impacting the end-user experience

**Table 3.** Evaluation of browser-addons

CsFire [5] extends the work of Maes *et al.* [11], and strips cookies and HTTP `authorization` headers from a cross-origin request. The advantage of stripping is that there are no side-effects for cross-origin requests that do not require credentials in the first place. CsFire operates autonomously by using a default client policy which is extended by centrally distributed policy rules. Additionally, CsFire supports users creating custom policy rules, which can be used to blacklist or whitelist certain traffic patterns. Without a central or user-supplied whitelist, CsFire does not support the payment and central authentication scenario.

To this extent, we plan to integrate the approach presented in this paper to the CsFire Mozilla Add-On distribution in the near future.

NoScript ABE [13], or Application Boundary Enforcer, restricts an application within its origin, which effectively strips credentials from cross-origin requests, unless specified otherwise. The default ABE policy only prevents CSRF attacks from the internet to an intranet page. The user can add specific policies, such as a CsFire-alike stripping policy [8], or a site-specific blacklist or whitelist. If configured with [8], ABE successfully blocks the three attack scenarios, but disables the payment and central authentication scenario.

RequestPolicy [16] protects against CSRF by blocking all cross-origin requests. In contrast to stripping credentials, blocking a request can have a very noticeable effect on the user experience. When detecting a cross-origin redirect, RequestPolicy injects an intermediate page where the user can explicitly allow the redirect. RequestPolicy also includes a predefined whitelist of hosts that are allowed to send cross-origin requests to each other. Users can add exceptions to

the policy using a whitelist. RequestPolicy successfully blocks the three attack scenarios (by blocking instead of stripping all cross-origin requests) and requires interactive end-user feedback to enable the payment and central authentication scenario.

Finally, in contrast to the CSRF *prevention* techniques discussed in this paper, Shahriar and Zulkernine proposes a client-side *detection* technique for CSRF [17]. In their approach, malicious and benign cross-origin requests are distinguished from each other based on the existence and visibility of the submitted form or link in the originating page, as well as the visibility of the target. In addition, the expected content type of the response is taken into account to detect false negatives during execution. Although the visibility check closely approximates the end-user intent, their technique fails to support the script inclusions of third party service mashups as discussed in Section 6. Moreover, without taking into account the delegation requests, expected redirect requests (as defined in Section 3) will be falsely detected as CSRF attacks, although these requests are crucial enablers for the payment and central authentication scenario.

## 8 Conclusion

We have proposed a novel technique for protecting at client-side against CSRF attacks. The main novelty with respect to existing client-side countermeasures is the good trade-off between security and compatibility: existing countermeasures break important web scenarios such as third-party payment and single-sign-on, whereas our countermeasure can permit them.

**Acknowledgements** This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, IBBT, IWT, the Research Fund K.U. Leuven and the EU-funded FP7-projects WebSand and NESSoS.

## References

1. Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. *Computer Security Foundations Symposium, IEEE*, 0:290–304, 2010.
2. Adam Barth, Collin Jackson, and Ian Hickson. The web origin concept. `http://tools.ietf.org/html/draft-abarth-origin-09`, November 2010.
3. Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.
4. Jesse Burns. Cross site reference forgery: An introduction to a common web application weakness. *f Security Partners, LLC*, 2005.

5. Philippe De Ryck, Lieven Desmet, Thomas Heyman, Frank Piessens, and Wouter Joosen. Csfire: Transparent client-side mitigation of malicious cross-domain requests. In *Lecture Notes in Computer Science*, pages 18–34. Springer Berlin / Heidelberg, 2010.
6. Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Automatic and precise client-side protection against csrf attacks - downloads. `https://distrinet.cs.kuleuven.be/software/CsFire/esorics2011/`, 2011.
7. Django. Cross site request forgery protection. `http://docs.djangoproject.com/en/dev/ref/contrib/csrf/`, 2011.
8. Informaction Forums. Which is the best way to configure ABE? `http://forums.informaction.com/viewtopic.php?f=23\&t=4752`, July 2010.
9. Martin Johns and Justus Winter. RequestRodeo: client side protection against session riding. *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5–17, 2006.
10. Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 1–10, 2006.
11. Wim Maes, Thomas Heyman, Lieven Desmet, and Wouter Joosen. Browser protection against cross-site request forgery. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 3–10. ACM, November 2009.
12. Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. *Financial Cryptography and Data Security*, pages 238–255, 2009.
13. Giorgio Maone. Noscript 2.0.9.9. `http://noscript.net/`, 2011.
14. Ruby on Rails. Actioncontroller::requestforgeryprotection. `http://api.rubyonrails.org/classes/ActionController/RequestForgeryProtection.html`, 2011.
15. Owasp. Csrf guard. `http://www.owasp.org/index.php/CSRF_Guard`, October 2008.
16. Justin Samuel. Requestpolicy 0.5.20. `http://www.requestpolicy.com`, 2011.
17. Hossain Shahriar and Mohammad Zulkernine. Client-side detection of cross-site request forgery attacks. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 358 –367, November 2010.
18. Michal Zalewski. Browser security handbook. `http://code.google.com/p/browsersec/wiki/Main`, 2010.
19. William Zeller and Edward W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.

# Composing Trust Models towards Interoperable Trust Management *

Rachid Saadi, Mohammad Ashiqur Rahaman,
Valerie Issarny, and Alessandra Toninelli

ARLES Project-Team
INRIA CRI Paris-Rocquencourt, France
{name.surname}@inria.fr

**Abstract.** Computational trust is a central paradigm in today's Internet as our modern society is increasingly relying upon online transactions and social networks. This is indeed leading to the introduction of various trust management systems and associated trust models, which are customized according to their target applications. However, the heterogeneity of trust models prevents exploiting the trust knowledge acquired in one context in another context although this would be beneficial for the digital, ever-connected environment. This is such an issue that this paper addresses by introducing an approach to achieve interoperability between heterogeneous trust management systems. Specifically, we define a trust meta-model that allows the rigorous specification of trust models as well as their composition. The resulting composite trust models enable heterogeneous trust management systems to interoperate transparently through mediators.

## 1 Introduction

With people getting increasingly connected virtually, trust management is becoming a central element of today's open distributed digital environment. However, existing trust management systems are customized according to specific application domains, hence implementing different trust models. As a result, it is nearly impossible to exploit established trust relations across systems. While a trust relation holding in one system does not systematically translate into a similar relation in another system, it is still a valuable knowledge, especially if the systems relate to the same application domains (e.g., e-commerce, social network). This is such an issue that we are addressing in this paper.

To the best of our knowledge, little work investigates interoperability between heterogeneous trust models. The closest to our concern is the work of [19], which describes a trust management architecture that enables dealing with a variety of trust metrics and mapping between them. However, the architecture deals with the composition at the level of trust values and do not account for the variety of trust models. In particular, one may want to differentiate between direct trust values and reputation-based ones when composing them. In general, what is needed is a way to formalize heterogeneous trust models and their composition. Such a concern is in particular addressed

* Work supported by EU-funded project FP7-231167 CONNECT and by EU-funded project FP7-256980 NESSOS.

in [9,21], which introduce trust meta-models based on state of the art trust management systems. Nevertheless, little detail is given and the paper does not describe how to exploit the meta-model for composing heterogeneous trust models and this achieve interoperability. Dealing with the heterogeneity of trust models is also investigated in [4,20]. However, the study is for the sake of comparison and further concentrates on reputation-based models. Summarizing, while the literature is increasingly rich of trust models, dealing with their composition remains a challenge.

Towards overcoming the interoperability challenge faced by trust management systems, this paper introduces a comprehensive approach based on the definition of a reference trust meta-model. Specifically, based on the state of the art (Section 2), the trust meta-model formalizes the core entities of trust management systems, i.e., trust roles, metrics, relations and operations (Section 3). The trust meta-model then serves specifying the composition of trust models in terms of mapping rules between roles, from which trust mediators are synthesized (Section 4). Trust mediators transparently implement mapping between respective trust relations and operations of the composed models. While this paper introduces the composition approach from a theoretical perspective, we are currently implementing it as part of the CONNECT project[1] on next generation middleware for interoperability in complex systems of systems (Section 5).

## 2 Trust Model Definition

As in particular defined in [5]: *i.e.*, *A trustor trusts a trustee with regard to its ability to perform a specific action or to provide a specific service*. Hence, any trust model may basically be defined in terms of the three following elements:

1. *Trust roles* abstract the representative behaviors of stakeholders from the standpoint of trust management, in a way similar to role-based access control model [3].
2. *Trust relations* serve specifying trust relationships holding among stakeholders, and
3. *Trust assessment* define how to compute the trustworthiness of stakeholders.

We further define trust relations and assessment below.

### 2.1 Trust relations

We identify two types of trust relationships, i.e., *direct* and *indirect*, depending on the number of stakeholders that are involved to build the trust relationship:

*Direct trust:* A direct trust relationship represents a trust assertion of a subject (i.e., trustor) about another subject (i.e., trustee). It is thus a one-to-one trust relation (denoted *1:1)*) since it defines a direct link from a trustor (**1**) to a trustee (**1**). One-to-one trust relations are maintained locally by trustors and represent the trustors' personal opinion regarding their trustees [10]. For example, a one-to-one relation may represent a belonging relationship (e.g., employees trust their company), a social relationship (e.g., trust among friends), or a profit-driven relationship (e.g., a person trusts a trader for managing its portfolio).

[1] http://connect-forever.eu/

*Recommendation-based trust:* As opposed to a direct trust relationship, a recommendation-based relationship represents a subject's trustworthiness based on a third party's opinion. This can be either (i) transitive-based or (ii) reputation-based.

*Transitive-based trust relations* are one-to-many (denoted *1:N*). Such a relation enables a trustor (**1**) to indirectly assess the trustworthiness of an unknown trustee through the recommendations of a group of trustees(**N**). Hence, the computation of 1:N relations results from the concatenation and/or aggregation of many 1:1 trust relations. The concatenation of 1:1 trust relations usually represents a transitive trust path, where each entity can trust unknown entities based on the recommendation of its trustees. Thus, this relationship is built by composing personal trust relations [1,18]. Furthermore, in the case where there exist several trust paths that link the trustor to the recommended trustee, the aggregation can be used to aggregate all given trust recommendations [7].

*Reputation-based trust relations* are many-to-one (denoted *N:1*) and result from the aggregation of many personal trust relationships having the same trustee. Hence, the N:1 trust relation allows the definition of the reputation of each trustee within the system. Reputation systems may then be divided into two categories depending on whether they are (i) *Centralized* or (ii) *Distributed*. With the former, the reputation of each participant is collected and made publicly available at a centralized server (e.g., eBay, Amazon, Google, [14]). With the latter, reputation is spread throughout the network and each networked entity is responsible to manage the reputation of other entities (e.g., [7,23]).

### 2.2 Trust Assessment

Trust assessment, i.e., assigning values to trust relationships, relies on the definition of: (i) trust metrics characterizing how trust is measured and (ii) operations for composing trust values.

*Trust metrics:* Different metrics have been defined to measure trust. This is due to the fact that one trust metric may be more or less suitable to a certain context. Thus, there is no widely recognized way to assign trust values. Some systems assume only binary values. In [24], trust is quantified by qualitative labels (e.g., high trust, low trust etc.). Other solutions represent trust by a numerical range. For instance, this range can be defined by the interval [-1..1] (e.g., [12]), [0..n] (e.g., [1,18]) or [0..1] (e.g., [7]). A trust value can also be described in many dimensions, such as: (Belief, Disbelief, Uncertainty) [7].

In addition, several definitions exist about the semantics of trust metrics. This is for instance illustrated by the meaning of zero and negative values. For example, zero may indicate lack of trust (but not distrust), lack of information, or deep distrust. Negative values, if allowed, usually indicate distrust, but there is a doubt whether distrust is simply trust with a negative sign, or a phenomenon of its own.

*Trust operations:* We define four main operations for the computation of trust values associated with the trust relations given in Section 2.1 (see table 1): *bootstrapping*, *refreshing*, *aggregation*, and *concatenation*.

The *bootstrapping* operation initializes the *a priori* values of 1:1 and N:1 trust relations. Trust bootstrapping consists of deciding how to initialize trust relations in order to efficiently start the system and also allow newcomers to join the running system

| | Bootstrapping | Aggregation | Concatenation | Refreshing |
|---|---|---|---|---|
| One-to-One (1:1) | X | | | X |
| One-to-Many (1:N) | | X | X | |
| Many-to-One (N:1) | X | X | | X |

Table 1: Trust assessment operations

[16]. Most existing solutions simply initialize trust relation with a fixed value (e.g., 0.5 [6], a uniform Beta probabilistic distribution [8]). Other approaches include among others: initializing existing trust relations according to given peers recommendations [17]; applying a sorting mechanism instead of assigning fixed values [18]; and assessing trustees into different contexts (e.g., fixing a car, babysitting, etc.) and then inferring unknown trust values from known ones of similar or correlate contexts [16,2].

All the solutions dealing with 1:N trust assessment mainly define the *concatenation* and the *aggregation* operations, in order to concatenate and to aggregate trust recommendations by computing the average [18], the minimum or the product [1] of all the intermediary trust values. In the case of Web service composition, some approaches (e.g., [15]) evaluate the recommendation for each service by evaluating its provider, whereas other approaches (e.g., [11]) evaluate the service itself in terms of its previous invocations, performance, reliability, etc. Then, trust is composed and/or aggregated according to the service composition flow (sequence, concurrent, conditional and loop).

Aggregation operations such as Bayesian probability (e.g., [13]) are often used for the assessment of N:1 (reputation-based) trust relations. Trust values are then represented by a beta Probability Density Function [8], which takes binary ratings as inputs (i.e., positive or negative) from all trustors. Thus, the reputation score is refreshed from the previous reputation score and the new rating [14]. The advantage of Bayesian systems is that they provide a theoretically sound basis for computing reputation scores and can also be used to predict future behavior.

Finally, refreshing operations are mainly trigged by trustors to refresh 1:1 and N:1 trust relations, after receiving stakeholders' feedback.

## 3 Trust Meta-Model

Following the above, we formally define the trust meta-model as: $TM =< \mathbb{R}, \mathbb{L}, \mathbb{M}, \mathbb{O} >$, where $\mathbb{R}$, $\mathbb{L}$, $\mathbb{M}$ and $\mathbb{O}$ are the finite sets of trust roles, relations, metrics and operations.

### 3.1 Trust Meta-Model Formalization

As detailed below, each set of $TM$ consists of $elements$ where an element can have a $simple\ value$ (e.g., string) or a complex value. A complex value of an element is either an exclusive combination of values (only one of the values) $\vee v$ (e.g., $v_1 \vee v_2 \vee v_3$) or an inclusive combination of values (one or more elements) $\diamond v$ (e.g., $v_1 \wedge v_2 \wedge (v_3 \vee v_4)$) of elements.

**Role set $\mathbb{R}$:** The role set contains all the roles $r$ played by the stakeholders of the trust model. A role $r$ of $\mathbb{R}$ is simply denoted by its name:

$$r =< \text{name:string} > \tag{1}$$

where: the attribute $name$ of type $string$ represents the name or the identifier of the role[2]. In our meta-model, a stakeholder is represented as a $Subject\ s$, playing a number of roles, $r_1$, $r_2$...and $r_n$, which is denoted as $s \triangleright r_1, r_2 ... r_n$.

**Metric set $\mathbb{M}$:** The metric set describes all the trust metrics that can be manipulated by the trust model. A metric is formally denoted as a pair:

$$m =< \text{name:string, type:string} > \tag{2}$$

where: $name$ and $type$ are strings and respectively define the name and the type. The $type$ can be a simple type (e.g., probability([0..1]), label(good, bad), etc.) or a composition of simples ones (e.g., tuple (believe([0..1]), uncertainty([0..1])).

**Relation set $\mathbb{L}$:** A relation set $\mathbb{L}$ contains all the trust relations that are specified by the trust model. We specifically denote a trust relation as a tuple:

$$\begin{aligned} l =< \text{name:string, ctx:string, type:string, trustor:}\vee r_i\text{, trustee:}\vee r_j\text{, value:}m_k > \\ \text{with } r_i, r_j \in \mathbb{R} \text{ and } m_k \in \mathbb{M} \end{aligned} \tag{3}$$

where: (i) $name$ identifies the relation; (ii) $ctx$ describes the context of the relationship in terms of the application domain (e.g., selling); (iii) $type$ represents the cardinality of the relation and is denoted by one of the following arities: 1:1, 1:N or N:1; (iv) $trustor$ and $trustee$ are roles where a trust relation relates a $trustor$ role with a $trustee$ role; (v) $value$ is an element from the metric set and thus reflects the trust measure given by the trustor to the trustee through this relation. In the above, note that different trustors can establish the same type of relationship with different trustees. Thus, as a trust relation is binary and between a $trustor$ role and a $trustee$, the exclusive combination of roles (e.g., $r_1 \vee r_2 \vee r_3$) is used to describe these elements

**Operation set $\mathbb{O}$:** The operation set specifies the operations that can be performed over relations by a subject, either to assess the trustworthiness of another subject or to communicate (i.e., request/response) trust values associated with desired subjects (see Figure 1). As defined in Section 2, trust assessment relies on the bootstrapping, aggregation, concatenation and refreshing operations, whereas, the communication of a trust value relies on the request and response operations. An operation is formally denoted as:

$$\begin{aligned} o =< \text{name:string, host:}\vee r_i\text{, type:string, input:}\diamond l_j\text{, output:}\diamond l_k\text{, via:}\diamond l_n\text{, call:}\diamond o > \\ \text{Where } r_i \in \mathbb{R}, l_j, l_k, l_n, \in \mathbb{L}, \text{ and } o \in \mathbb{O} \end{aligned} \tag{4}$$

[2] Note that the name can in particular be specified by an ontological concept that formally describes this role into a given trust ontology although this is not longer discussed in this paper.

where: (i) $name$ identifies uniquely an operation; (ii) $host$ instantiates the role(s) that hosts and executes the operation; (iii) $type$ defines the operation (i.e., request, response, bootstrapping, aggregation, concatenation, and refreshing); (iv) $input$ gives the trust relations that are required to perform an assessment operation or are received by a communication operation; (v) $output$ gives the trust relations that are provided, as the result of either an assessment operation or a communication; (vi) $via$ specifies the trust relationship that should hold with the role with which the communication happens, while its value is *self* in the case of assessment; and (vii) $call$ denotes a continuation (see Figure 1). Note that $input$ and $output$ are complex values, i.e., logical conjunction of one or more relations.

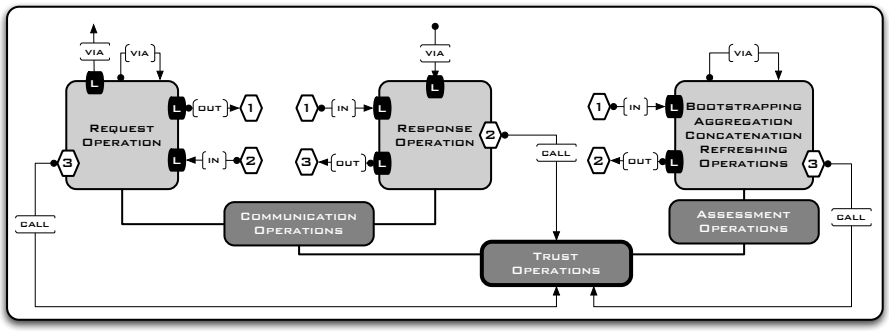



Fig. 1: Operation continuation

**Trust graph *TG*:** We associate the definition of a *trust graph* with any trust model $TM$ for the sake of graphical representation. Specifically, the trust graph $TG(\mathbb{R}, \mathbb{E})$ associated with a given $TM$ is a directed graph with the vertices representing the set of roles $\mathbb{R}$ of $TM$, and the set of edges $\mathbb{E}$ representing the relationship between roles according to $\mathbb{L}$. Hence, each edge is labeled by the referenced relation $l$ from the set of relations $\mathbb{L}$ and the type of that relation, i.e., 1:1, 1:N or N:1.

### 3.2 Example

We illustrate the expressiveness of our trust meta-model by considering the specification of representative trust models associated with two selling transaction scenarios. Precisely, we introduce the specification of an eBay like centralized trust model (see Table 2) and of a fully distributed one (see Table 3). Both trust models aim at assessing transaction behaviors of sellers.

Figure 2 depicts the trust graphs of both models; the centralized trust model, i.e., $TM_C$ (on the left in the figure), is defined with three roles, i.e., $r_S$=$Seller$, $r_B$=$Buyer$, and $r_M$=$Manager$, whereas the distributed trust model, i.e., $TM_D$ (on the right in the figure), is defined with the unique role $r_C$=$Customer$, which can be either a seller or a buyer.

Focusing on the specification of $TM_C$ in Table 2 , the roles $Buyer$ and $Seller$ have a direct trust relationship (i.e., $l_0$) with the $Manager$ that manages the sellers' reputation (i.e., $l_3$). Thus, any $Buyer$ can: (i) query the $Manager$ about the reputation

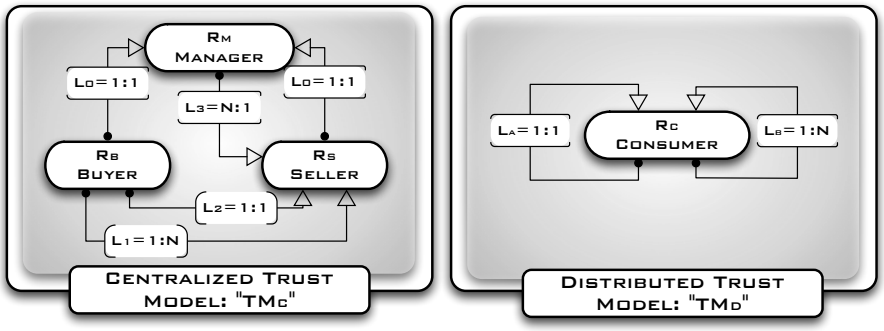



Fig. 2: Trust graphs of the centralized ($TM_C$) and the distributed ($TM_D$) trust models.

of a $Seller$ (i.e., $l_1$), and (ii) provide the $Manager$ with its feedback (i.e., $l_2$) after a selling transaction. Hence, a $Buyer$ has to perform a request operation (i.e., $o_4$) to get the reputation of the seller, so that it can compute locally the trustworthiness of the seller (i.e., $o_1$). After a transaction is completed, a $Buyer$ can provide its feedback to the $Manager$ by triggering a request operation (i.e., $o_8$). The $Manager$ in turn processes (i.e., $o_9$) this feedback request to compute and refresh the reputation of the concerned $Seller$ (i.e., $o_3$).

| Role set $\mathbb{R}$ |
|---|
| $r_S$ = <name="Buyer"> |
| $r_B$ = <name="Seller"> |
| $r_M$ = <name="Manager"> |
| **Metric set $\mathbb{M}$** |
| $m_0$ = <name="Reputation", type="Probability"> |
| $m_1$ = <name="Recommendation", type="Probability"> |
| $m_2$ = <name="Rate", type= "Five Semantic labels"> |
| **Relation set $\mathbb{L}$** |
| $l_0$ = < name="ServerRecommendation", ctx= "Selling", type=1:1, trustor=($r_S \vee r_B$), trustee=$r_M$, metric=$m_1$ > |
| $l_1$ = < name= "SellerTrustworthiness", ctx= "Selling", type=1:N, trustor=$r_S$, trustee=$r_B$, metric=$m_1$ > |
| $l_2$ = < name="BuyerFeedback", ctx= "Selling", type=1:1, trustor=$r_S$, trustee=$r_B$, metric=$m_2$ > |
| $l_3$ = < name="SellerReputation", ctx= "Selling", type=N:1, trustor=$r_B$, trustee=$r_M$, metric=$m_0$ > |
| **Operation set $\mathbb{O}$** |
| $o_0$ = <name="getManagerTrustworthiness", host=($r_S \vee r_B$), type=request, in=$l_0$, out=$l_0$ > |
| $o_1$ = <name="assessSellerTrustworthiness", host=$r_S$, type=concatenation, in=($l_0 \wedge l_3$) , out=$l_1$ > |
| $o_2$ = <name= "assessBuyerFeedback", host=$r_S$, type=update, in=$l_2$, out=$l_2$, call=$o_8$ > |
| $o_3$ = <name="setSellerReputation", host=$r_M$, type=aggregation, in=$l_2$, out=$l_3$ > |
| $o_4$ = <name="getSellerTrustworthiness", host=$r_S$,type=request, via=$l_0$, out=$l_1$, in=$l_3$, call=$o_1$ > |
| $o_5$ = <name="getSellerReputation", host=$r_M$,type=response, in=$l_3$, out=$l_3$ > |
| $o_6$ = <name="sendSellerReputation", host=$r_M$,type=response, via=$l_0$, in=$l_1$, out=$l_3$, call=$o_5$ > |
| $o_7$ = <name="getBuyerFeedback", host=$r_S$,type=request, in=$l_2$, out=$l_2$ > |
| $o_8$ = <name="sendBuyerFeedback", host=$r_S$,type=request, via=$l_0$, out=$l_2$ > |
| $o_9$ = <name="updateSellerReputation", host=$r_M$,type=response, via=$l_0$, in=$l_2$, call= $o_3$ > |

Table 2: Centralized Trust model: $TM_C$.

Regarding the distributed model $TM_D$ specified in Table 3, the role $Customer$ of the distributed model can maintain a direct trust relationship with other $Customers$

(i.e., $l_a$) and can then ask trustee $Customers$ to get their recommendation about unknown $Customers$ that are sellers (i.e., $l_b$). Hence, a $Customer$ can perform a request operation (i.e., $o_d$) to get a recommendation of an unknown $Customer$ seller, so that the requester $Customer$ can compute locally the trustworthiness of the $Seller$ (i.e., $o_b$ and $o_c$). After the transaction is completed, the requester $Customer$ can provide its feedback to other $Customers$ by triggering a request operation (i.e., $o_f$). The recipient $Customer$ can process (i.e., $o_h$) this feedback to refresh its relationship with the concerned $Seller$ (i.e., $o_g$) and can also in turn propagate this feedback by calling the $o_f$.

| Role set $\mathbb{R}$ |
|---|
| $r_C$ = <name="Customer "> |
| **Metric set $\mathbb{M}$** |
| $m_a$ = <name="Recommendation", type="Probability"> |
| **Relation set $\mathbb{L}$** |
| $l_a$ = < name="DirectCustomer Trustworthiness", ctx= "auction", type=1:1, trustor=$r_C$, trustee=$r_C$, metric=$m_a$ > |
| $l_b$ = < name="TransitiveCustomer Trustworthiness", ctx= "auction", type=1:N, trustor=$r_C$, trustee=$r_C$, metric=$m_a$ > |
| **Operation set $\mathbb{O}$** |
| $o_a$ = <name="getLocalCustomerTrustworthiness", host=$r_C$, type=request, in=$l_a$, out=$l_a$ > |
| $o_b$ = <name="assessCustomerTrustworthiness1", host=$r_C$, type=concatenation, in=$(l_a \wedge (l_a \vee l_b))$ , out=$l_b$, call=$o_c$ > |
| $o_c$ = <name="assessCustomerTrustworthiness2", host=$r_C$, type=aggregation, in=$l_b$ , out=$l_b$ > |
| $o_d$ = <name="getRemoteCustomerTrustworthiness", host=$r_C$, type=request, via=$l_a$, out=$l_b$, in=$(l_a \vee l_b)$, call=$o_b$ > |
| $o_e$ = <name="sendCustomerTrustworthiness", host=$r_C$,type=response, via=$l_a$, in=$l_b$, out=$(l_a \vee l_b)$, call=$(o_a \vee o_d$ > |
| $o_f$ = <name="sendCustomerFeedback", host=$r_C$,type=request, via=$l_a$, out=$l_a$ > |
| $o_g$ = <name="setCustomerTrustworthiness", host=$r_C$,type=update, in=$l_a$, out=$l_a$, call=$o_f$ > |
| $o_h$ = <name="updateCustomerTrustworthiness", host=$r_C$,type=response, via=$l_a$, in=$l_a$, call= $o_g$ > |

Table 3: Distributed Trust model: $TM_D$.

## 4 Composing Trust Models

Given the specification of trust models, their composition relies on mapping their respective roles so that: (i) the trustworthiness of the various roles can be assessed, (ii) existing trust relations can be queried, and (iii) trust feedbacks can be propagated transparently from one trust model to another. Further, the existing trust relations and operations are extended to relate roles from the composed models, and new assessment operations are required to map trust relations from one model to another. Finally, the resulting mapping and extensions are implemented through mediation [22] so as to make composition transparent to existing systems, which leads us to introduce the corresponding *mediator role*.

Formally, the composition, denoted $\bigoplus$, of two trust models $TM_x$ and $TM_y$, which introduces the trust model $TM_{xy}$, is defined as follows:

$$
\begin{aligned}
TM_{xy} &= TM_x \bigoplus_{\Psi^{xy}} TM_y \\
&= < \mathbb{R}_x, \mathbb{M}_x, \mathbb{L}_x, \mathbb{O}_x > \bigoplus_{\Psi^{xy}} < \mathbb{R}_y, \mathbb{M}_y, \mathbb{L}_y, \mathbb{O}_y > \\
&= \left\langle \begin{array}{l} \mathbb{R}_{xy} = \mathbb{R}_x \cup \mathbb{R}_y \cup \mu\mathbb{R}_{xy} \\ \mathbb{M}_{xy} = \mathbb{M}_x \cup \mathbb{M}_y \\ \mathbb{L}_{xy} = \mathbb{L}_x^+ \cup \mathbb{L}_y^+ \\ \mathbb{O}_{xy} = \mathbb{O}_x^+ \cup \mathbb{O}_y^+ \cup \mu\mathbb{O}_{xy} \end{array} \right\rangle
\end{aligned}
\tag{5}
$$

where:

- $\Psi^{xy}$ is the set of mapping rules over roles that enables the composition of $TM_x$ and $TM_y$;
- $\mu\mathbb{R}_{xy}$ and $\mu\mathbb{O}_{xy}$ are the new sets of mediator roles and mediation operations, respectively;
- ($\mathbb{L}_x^+$ and $\mathbb{L}_y^+$) and ($\mathbb{O}_x^+$ and $\mathbb{O}_y^+$) are the extended relations and operations, respectively.

In the following, we elaborate on the mediation process to generate the sets of mediator roles, and mediation operations (i.e., $\mu\mathbb{R}_{xy}$, and $\mu\mathbb{O}_{xy}$) and extended relations and operations (i.e., $\mathbb{L}_x^+$, $\mathbb{O}_x^+$ $\mathbb{L}_y^+$, $\mathbb{O}_y^+$).

**Algorithm 1**: Trust_Models_Composition($TM_x, TM_y, \Psi^{xy}$)

**Input(s)** : Trust models $TM_x$ and $TM_y$
The set of Mapping rules $\Psi^{xy}$
**Output(s)**: The trust model composition $TM_{xy} =< \mathbb{R}_{xy}, \mathbb{M}_{xy}, \mathbb{L}_{xy}, \mathbb{O}_{xy} >$

**begin**
  // Initialize trust models sets for composition
  $\mathbb{L}_x^+ = \mathbb{L}_x$ ; $\mathbb{L}_y^+ = \mathbb{L}_y$
  $\mathbb{O}_x^+ = \mathbb{O}_x$ ; $\mathbb{O}_y^+ = \mathbb{O}_y$
  **foreach** $(\psi_k^{xy} = (\psi_k^{xy} = (r_i : TM_{m=\{x,y\}}) \odot (r_j : TM_{n=\{x,y\},m\neq n})) \in \Psi^{xy})$ **do**
    $Relation\_Mediation(r_i, \mathbb{L}_m^+, r_j, \mathbb{L}_n^+, \odot)$
    **if** $(\odot == " \underset{\mu r_k}{\bowtie} ")$ **then**
      **if** $\mu r_k \notin \mu\mathbb{R}_{xy}$ **then**
        $\mu\mathbb{R}_{xy} = \mu\mathbb{R}_{xy} \cup \{\mu r_k\}$
      $Operation\_Mediation(r_i, \mathbb{L}_m^+, \mathbb{O}_m^+, r_j, \mathbb{L}_n^+, \mathbb{O}_n^+, \mu r_k)$
  $\mathbb{R}_{xy} = \mathbb{R}_x \cup \mathbb{R}_y \cup \mu\mathbb{R}_{xy}$
  $\mathbb{M}_{xy} = \mathbb{M}_x \cup \mathbb{M}_y$
  $\mathbb{L}_{xy} = \mathbb{L}_x^+ \cup \mathbb{L}_y^+$
  $\mathbb{O}_{xy} = \mathbb{O}_x^+ \cup \mathbb{O}_y^+ \cup \mu\mathbb{O}_{xy}$
**end**

### 4.1 Role Mapping

The mapping of roles from 2 distinct models is explicitly defined through a set of mapping rules defined as follows:

$$\psi_k^{st} = (r_s : TM_s) \odot (r_d : TM_t) \tag{6}$$

where, $\odot$ is asymmetric and maps the source role $r_s$ of $TM_s$ to the target role $r_t$ of $TM_t$. We further refine $\odot$ into two mapping operators:

- *The See operator*, noted "$\succ$", simply associates a source role with a target role so as to define that the role $r_t$ of $TM_t$ is seen as $r_s$ in $TM_t$. For instance, in the

selling transaction scenarios, $(r_B : TM_C) \succ (r_C : TM_D)$ means that $Buyers$ (i.e., $r_B : TM_C$) of the centralized trust model are seen by the distributed trust model ($TM_D$) as $Customers$ ($r_C : TM_D$).

– *The $Mimic$ operator*, noted "$\underset{\mu r}{\bowtie}$", specifies that $r_s$ should be able to request trust values of $TM_t$ as if it was $r_t$. This is practically achieved through the mediator role $\mu r$ that translates $r_s$ requests into $r_t$ requests. For instance, the rule $(r_C : TM_D) \underset{\mu r}{\bowtie} (r_S : TM_A)$ means that any customer is able to request trust values as if it was a buyer in the centralized trust management system, thanks to the mediation achieved by $\mu r$.

The computation of the composition of trust models $TM_x$ and $TM_y$ is detailed in Algorithm 1. The algorithm iterates on mapping rules for each of which it invokes $Relation\_Mediation$ (see line 5) so as to extend relation sets, namely: $\mathbb{L}_x^+$ and $\mathbb{L}_y^+$, (see Section 4.2). Then, according to the definition of $Mimic$ rules, mediator roles (i.e., $\mu r$) are added to the set of mediator roles (see lines 7-8), and $Operation\_Mediation$ is invoked so as to perform mediation over the communication operations (see line 9) of the composed trust models (see Section 4.3).

### 4.2 Relation Mediation

**Algorithm 2**: Relation_Mediation$(r_s, \mathbb{L}_s^+, r_t, \mathbb{L}_t^+, \odot)$

**Input(s)** : Roles $r_s$ and $r_t$ ; Relation sets $\mathbb{L}_s^+$ and $\mathbb{L}_t^+$ ; Mapping operation $\odot$

**Output(s)**: The source and the target relation sets: $\mathbb{L}_s^+$ and $\mathbb{L}_t^+$

```
1  begin
2      if ⊙ = " ≻ " then        /* Ψ^xy is defined with the "See" Operator */
3          foreach (l_i ∈ L_t^+) do      /* Find relations with the trustee r_s */
4              if l_i.trustee ⊐ r_t then
5                  l_i.trustee ←(r_t)— (r_t ∨ r_s)        /* Add r_s as a trustee */
6      if ⊙ = " ⋈ (μr) " then    /* Ψ^xy is defined with the "Mimic" Operator */
7          foreach (l_i ∈ L_s^+) do      /* Find relations with the trustee r_s */
8              if l_i.trustor ⊐ r_s then
9                  l_i.trustee ←(l_i.trustee)— (l_i.trustee ∨ μr)        /* Add μr as a
                       trustee */
10         foreach (l_i ∈ L_t^+) do      /* Find relations with the trustor r_t */
11             if l_i.trustor ⊐ r_t then
12                 l_i.trustor ←(r_t)— (r_t ∨ μr)        /* Add μr as a trustee */
13 end
```

The aim of relation mediation is to extend the trust relations of the original models to roles of the other. More precisely, for any trust relation:
$l =<$ name:string, ctx:string, type:string, trustor:$\vee r_i$, trustee:$\vee r_j$, metric:$m_k >$ of $\mathbb{L}_x$ and $\mathbb{L}_y$ of the composed models $TM_x$ and $TM_y$, its *trustee* and *trustor* elements are possibly extended to account for mapping between roles.

Algorithm 2 details the corresponding extension where: (i) function $e \sqsupset v$ returns true if $v$ is in $e$, and (ii) $e \overset{v_i}{\longleftarrow} v_j$ replaces the value $v_i$ in $e$ with the value $v_j$. As shown in the algorithm, the extension of trust relations depends on the type of the mapping operator. The $See$ operator defines which local trustee (target role $r_t$) corresponds to the source role ($r_s$). Therefore, all the relations $l_i$ (from the source trust model) that consider the source role as a trustee ($l_i.trustee \sqsupset r_t$) are extended with the target role (see lines 2-5). The $Mimic$ operator introduces a new mediator role that plays trustees of the source role as a trustee in the source trust model, and plays the target role as a trustor in the target trust model. This leads to the corresponding extension of the trust models relations of $\mathbb{L}_x$ (see lines 7-9) and $\mathbb{L}_y$ (see lines 10-12).

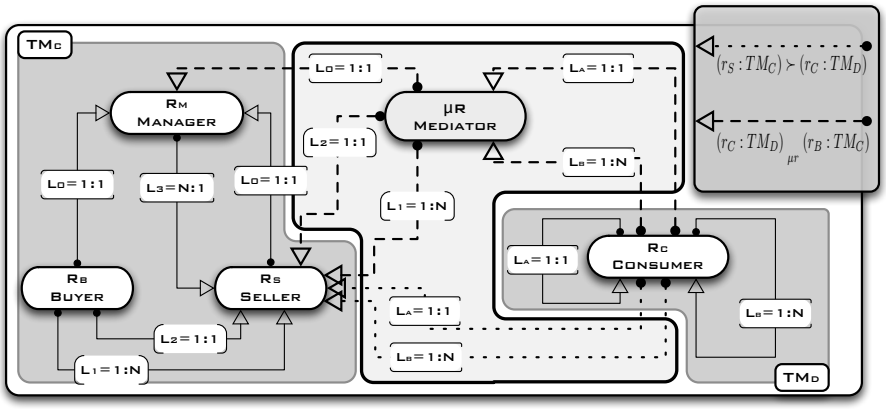



Fig. 3: Trust graph $TG_{CD}$

Figure 3 depicts the trust graph $TG_{CD}$ resulting from the composition of $TM_C$ and $TM_D$, while Table 4 details the associated trust roles, metric and relations where new mediator role and extended relations are highlighted in grey. The composition relies on two mapping rules that allow a $Customer$ of $TM_D$ to assess a seller of $TM_C$. The rule using the $See$ operator represents how sellers are perceived in $TM_D$, while the second rule using the $Mimic$ operator introduces a mediator role that enables $Costumers$ to request $TM_C$ as $Buyers$. Thus, "$r_B : TM_C \succ r_C : TM_D$" leads to extend the trustee element of $l_a$ and $l_b$ by replacing $r_C$ with $(r_C \vee r_B)$. The mapping rule "$r_C : TM_D \underset{\mu r}{\bowtie} r_S : TM_C$" extends the relations that sink into the role $Customer$ (i.e., $l_a$ and $l_b$) with the mediator role $\mu r$. In addition, all the relations that originate from the role $Buyer$ (i.e., $l_0$, $l_1$ and $l_2$) also originate from the mediator role $\mu r$.

| Roles set $\mathbb{R}$ | |
|---|---|
| $r_S$ | = <name="Buyer"> |
| $r_B$ | = <name="Seller"> |
| $r_M$ | = <name="Manager"> |
| $r_C$ | = <name="Customer "> |
| $\mu r$ | = <name="Customer Mediator"> |

| Metric set $\mathbb{M}$ | |
|---|---|
| $m_0$ | = <name="Reputation", type="Probability"> |
| $m_1$ | = <name="Recommendation", type="Probability"> |
| $m_2$ | = <name="Rate", type= "Five Semantic labels"> |
| $m_a$ | = <name="Recommendation", type="Probability"> |

| Relation set $\mathbb{L}$ | |
|---|---|
| $l_0$ | = < name="ServerRecommendation", ctx= "Selling", type=1:1, trustor=$((r_S \vee \mu r) \vee r_B)$, trustee=$r_M$, metric=$m_1$> |
| $l_1$ | = < name= "SellerTrustworthiness", ctx= "Selling", type=1:N, trustor=$(r_S \vee \mu r)$, trustee=$r_B$, metric=$m_1$ > |
| $l_2$ | = < name="BuyerFeedback", ctx= "Selling", type=1:1, trustor=$(r_S \vee \mu r)$, trustee=$r_B$, metric=$m_2$ > |
| $l_3$ | = < name="SellerReputation", ctx= "Selling", type=N:1, trustor=$r_B$, trustee=$r_M$, metric=$m_0$ > |
| $l_a$ | = < name="DirectCustomer Trustworthiness", ctx= "auction", type=1:1, trustor=$r_C$, trustee=$((r_C \vee r_B) \vee \mu r)$, metric=$m_a$ > |
| $l_b$ | = < name="TransitiveCustomer Trustworthiness", ctx= "auction", type=1:N, trustor=$r_C$, trustee=$((r_C \vee r_B) \vee \mu r)$, metric=$m_a$ > |

Table 4: $TM_C$ and $TM_D$ Composition: Role, Metric, and Relation sets

### 4.3 Operation Mediation

Operation mediation serves translating request operations from one model into requests in the other model, according to the mappings between roles defined using the *Mimic* operator. More precisely, consider a request operation by $r_s$ for a relation:

<name="l", ctx="c", type="t", trustor="$r_s$", trustee="tee", metric="v"> of $TM_s$

where $l \in \mathbb{L}_s$, $tor \in \mathbb{R}_s$, while $tee \in \mathbb{R}_t$ and $r_s$:$TM_s \underset{\mu r}{\bowtie} r_t$:$TM_t$. Then, operation mediation first identifies the matching relations:

<name: string, ctx="c", type: string, trustor="$r_t$", trustee="tee", metric: m> of $TM_t$

that should be requested in the target model using a request operation of $\mathbb{O}_t$. Replies are finally normalized using the mediation operation given by $\mu\mathbb{O}_{xy}$ for use in the source trust model. Operation mediation is practically implemented in a transparent way by the mediator that intercepts and then translates $r_s$ requests, as given in Algorithm 3. In the algorithm, the mediator interacts with $r_s$ (see lines 2-4) and $r_t$ (see lines 5-7). Then, the mediator computes the matching relation for each output relation (see lines 11-18) of the reply, where we assume that there is only one such relation (see lines 12-13) and requests its value using the appropriate request operation (see lines 16-18). We further consider that the mediator ($\mu r$) embeds a library of mediation functions that translate and normalize heterogeneous trust metrics, which are invoked by mediation operations $\mu o$ (see lines 12-14). Finally, for each update (i.e., bootstrapping and refreshing) triggered by the response, as specified in the corresponding $call$ element (see lines 19-20), the matching relations is sought in $\mathbb{L}_t$ (see line 23) and its value requested (see lines 25-28).

Figure4 depicts the basic mediation process (left hand side) and its extension with update (right hand side), as performed by the mediator. First, the mediator receives the request $in$ (step 1). Then, it invokes the corresponding request in the target model

**Algorithm 3**: Operation_Mediation($r_i, \mathbb{L}_m^+, \mathbb{O}_m^+, r_j, \mathbb{L}_n^+, \mathbb{O}_n^+, \mu r_k$)

**Input(s)** : Source role $r_s$, relation $\mathbb{L}_s^+$ and operation set $\mathbb{O}_s^+$
Target role $r_t$, relation $\mathbb{L}_t^+$ and operation set $\mathbb{O}_t^+$
The mediator role $\mu r$

**Output(s)**: The source, the target and the mediation operation sets: $\mathbb{O}_s^+$, $\mathbb{O}_t^+$ and $\mu\mathbb{O}_{st}$

1 **begin**
2 **foreach** $(o_i \in \mathbb{O}_s^+)$ **do** /* Find operation with the host $r_s$ */
3 **if** $o_i.type = "response" \wedge o_i.via.trustor \sqsupset r_s$ **then**
4 $o_i.host \xleftarrow{o_i.host} (o_i.host \vee \mu r)$ /* Add $\mu r$ as a host */
5 **foreach** $(o_i \in \mathbb{O}_t^+)$ **do** /* Find relations with the host $r_t$ */
6 **if** $o_i.type \neq "response" \wedge o_i.host \sqsupset r_t$ **then**
7 $o_i.host \xleftarrow{r_t} (r_t \vee \mu r)$ /* Add $\mu r$ as a host */
8 **foreach** $(o_i \in \mathbb{O}_s^+)$ **do** /* Find operation with the host $r_s$ */
// Request mediation
9 **if** $o_i.type = "response" \wedge o_i.host \sqsupset \mu r$ **then**
10 **if** $(o_i.out \neq null)$ **then**
11 **foreach** $l_k \sqsupset o_i.out$ **do**
// Create a new mediated operation $\mu o$
12 $\mu o.host$=$\mu r$ ; $\mu o.type$="mediation"
// Find a similar output relation into $\mathbb{L}_t$
13 $l^* = findSimilarRelation(l_k, \mathbb{L}_t^+)$
14 $\mu o.in = l^*$ ; $\mu o.out = l_k$
15 $\mu\mathbb{O}_{st} = \mathbb{O}_{st} \cup \{\mu o\}$
// The relation $l^*$ need to be requested
16 $o^* = findOperation(type = "request", l^*, \mathbb{O}_t^+)$
17 $o^*.call \xleftarrow{o^*.call} (o^*.call) \vee \mu o$
18 $o_i.call \xleftarrow{o_i.call} (o_i.call) \vee o^*$
// Update mediation
19 **foreach** $o_k \sqsupset o_i.call$ **do**
20 **if** $o_k.type = "refresh" \vee o_k.type = "booststrap"$ **then**
21 **foreach** $l_p \sqsupset o_k.in$ **do**
22 $\mu o.host$=$\mu r$ ; $\mu o.type$="mediation"
23 $l^* = findSimilarRelation(l_p, \mathbb{L}_t^+)$
24 $\mu o.in = l_p$ ; $\mu o.out = l^*$
25 $o^* = findOperation(type = o_k.type, l^*, \mathbb{O}_t^+)$
26 $\mu o.call = o^*$
27 $\mu\mathbb{O}_{st} = \mu\mathbb{O}_{st} \cup \{\mu o\}$
28 $o_i.call \xleftarrow{o_i.call} (o_i.call) \vee \mu o$
29 **end**

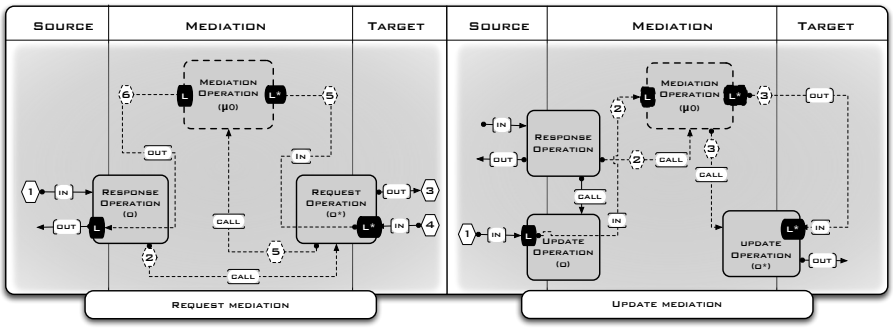



Fig. 4: Operation mediation process

(steps 2 to 4) and upon receipt of the result, it normalizes the value using the mediation operation $\mu\mathbb{O}_{ts}$ (steps 5-6). Finally, the reply *out* is returned. In the case of update (on the figure right hand side), the relation matching the one given as input is sought in the target model using the mediation operation $\mu\mathbb{O}_{st}$ (step 2), leading to invoke the corresponding update operation of the target model (step 3).

As an example, Table 5 gives the operation set $\mathbb{O}_{1,2}$ resulting from the composition of $TM_C$ and $TM_D$.

| Operation set $\mathbb{O}$ |
|---|
| $o_0$ = <name="getManagerTrustworthiness", host=$((r_S \vee \mu r) \vee r_B)$, type=request, in=$l_0$, out=$l_0$ > |
| $o_1$ = <name="assessSellerTrustworthiness", host=$(r_S \vee \mu r)$, type=concatenation, type="product", in=$(l_0 \wedge l_3)$ , out=$l_1$ > |
| $o_2$ = <name= "assessBuyerFeedback", host=$(r_S \vee \mu r)$, type=update, type="rating", in=$l_2$, out=$l_2$, call=$o_8$ > |
| $o_3$ = <name="setSellerReputation", host=$r_M$, type=aggregation, in=$l_2$, out=$l_3$ > |
| $o_4$ = <name="getSellerTrustworthiness", host=$r_S$,type=request, via=$l_0$, out=$l_1$, in=$l_3$, call=$o_1 \vee \mu o_1$ > |
| $o_5$ = <name="getSellerReputation", host=$r_M$,type=response, in=$l_3$, out=$l_3$ > |
| $o_6$ = <name="sendSellerReputation", host=$r_M$,type=response, via=$l_0$, in=$l_1$, out=$l_3$, call=$o_5$ > |
| $o_7$ = <name="getBuyerFeedback", host=$(r_S \vee \mu r)$,type=request, in=$l_2$, out=$l_2$, call=$\mu o_2$ > |
| $o_8$ = <name="sendBuyerFeedback", host=$(r_S \vee \mu r)$,type=request, via=$l_0$, out=$l_2$ > |
| $o_9$ = <name="updateSellerReputation", host=$r_M$,type=response, via=$l_0$, in=$l_2$, call= $o_3$ > |
| $o_a$ = <name="getLocalCustomerTrustworthiness", host=$r_C$, type=request, in=$l_a$, out=$l_a$ > |
| $o_b$ = <name="assessCustomerTrustworthiness1", host=$(r_C \vee \mu r)$, type=concatenation, in=$(l_a \wedge (l_a \vee l_b))$ , out=$l_b$, call=$o_c$ > |
| $o_c$ = <name="assessCustomerTrustworthiness2", host=$(r_C \vee \mu r)$, type=aggregation, , in=$l_b$ , out=$l_b$ > |
| $o_d$ = <name="getRemoteCustomerTrustworthiness", host=$r_C$,type=request, via=$l_a$, out=$l_b$, in=$(l_a \vee l_b)$, call=$o_b$ > |
| $o_e$ = <name="sendCustomerTrustworthiness", host=$(r_C \vee \mu r)$,type=response, via=$l_a$, in=$l_b$, out=$(l_a \vee l_b)$, call=$(o_a \vee o_d) \vee o_4 \vee o_7$ > |
| $o_f$ = <name="sendCustomerFeedback", host=$r_C$,type=request, via=$l_a$, out=$l_a$ > |
| $o_g$ = <name="setCustomerTrustworthiness", host=$(r_C \vee \mu r)$,type=update, in=$l_a$, out=$l_a$, call=$o_f$ > |
| $o_h$ = <name="updateCustomerTrustworthiness", host=$(r_C \vee \mu r)$,type=response, via=$l_a$, in=$l_a$, call= $o_g \vee \mu o_3$ > |
| $\mu o_1$ = <name="Translate$l_1 l_b$", host=$\mu r$,type=mediation, in=$l_1$, out=$l_b$ > |
| $\mu o_2$ = <name="Translate$l_2 l_a$", host=$\mu r$,type=mediation, in=$l_2$, out=$l_a$ > |

Table 5: $TM_C$ and $TM_D$ Composition: Operation set

The response operation $o_e$ should be able to assess $Sellers$ of $TM_C$ since its outputs (i.e., $l_a$ and $l_b$) contain relations that sink into the $Seller$ role (see Table 4). To

do so, $o_e$ is extended (see lines 9-18) to enable the mediator role $\mu r$ (when it performs this operation) to retrieve similar $o_e$ output relations in $TM_C$, i.e., the relations $l_a$ and $l_b$ that are respectively similar to $l_1$ and $l_2$. The operation $o_e$ can hence call $o_4$ or $o_7$ to search for $l_1$ or $l_2$. Then, as for $o_e$, the called operations are extended as well, by calling the mediation operations $\mu o_1$ and $\mu o_2$ to translate respectively $l_1$ and $l_2$ into $l_b$ and $l_a$. Thus, $o_e$ is able to reply the appropriate trust relationships which are interpretable by $Customers$. Moreover, Algorithm 3 (see lines 19-28) enables $Customers$ feedback to be propagated to the $Manager$ of the target model $TM_C$, so that the reputation of $Sellers$ can be refreshed with the source model feedback. According to the resulting operation set (see Table 5), when the mediator role $\mu r$ performs the response operation $o_h$, it calls $\mu o_3$ to translate the feedback denoted by the relation $l_a$ into $Buyer$ feedback, I.e., $l_2$. Then, $\mu o_3$ is able to call $o_2$ with the $l_2$ to advertise its feedback to $TM_C$ $Manager$.

## 5 Conclusion

In this paper, we have introduced a trust meta-model as the basis to express and to compose a wide range of trust models. The composition of trust models enables assessing the trustworthiness of stakeholders across heterogeneous trust management systems. Such a composition is specified in terms of mapping rules between roles. Rules are then processed by a set of mediation algorithms to overcome the heterogeneity between the trust metrics, relations and operations associated with the composed trust models. We are currently implementing our approach as part of the *Connect* project[3] where we have defined an XML-based description of the trust meta-model, which we call TMDL (i.e., Trust Model Description Language). Thus, mediators are synthesized on-the-fly given the TMDL description of Trust models.

As future work, we are also considering the implementation of a simulator to a priori assess the behavior of trust composition of given trust models and thus allows fine tuning of the mapping rules. We are also investigating the use of ontologies to specify the semantics of trust model elements and thus possibly infer the mapping rules as well as infer the similarity of trust relations from the semantics.

## References


1. A. Abdul-Rahman and S. Hailes. A distributed trust model. In *NSPW: New Security Paradigms Workshop*, pages 48–60, New York, USA, 1997. ACM Press.
2. S. Ahamed, M. Monjur, and M. Islam. CCTB: Context correlation for trust bootstrapping in pervasive environment. In *2008 IET 4th International Conference on Intelligent Environments*, pages 1–8, 2008.
3. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.
4. K. Fullam, T. Klos, G. Muller, J. Sabater, A. Schlosser, Z. Topol, K. Barber, J. Rosenschein, L. Vercouter, and M. Voss. A specification of the Agent Reputation and Trust (ART) testbed. In *Conference on Autonomous agents and multiagent systems*, pages 512–518. ACM, 2005.
5. T. Grandison and M. Sloman. A survey of trust in internet applications. *Communications Surveys & Tutorials, IEEE*, 3(4):2–16, 2009.


[3] http://connect-forever.eu/

6. M. Haque and S. Ahamed. An omnipresent formal trust model (FTM) for pervasive computing environment. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, 2007.
7. A. Jøsang and S. Pope. Semantic constraints for trust transitivity. In *APCCM: 2nd Asia-Pacific conference on Conceptual modelling*, pages 59–68, Newcastle, New South Wales, Australia, 2005. Australian Computer Society, Inc.
8. A. Jsang and R. Ismail. The beta reputation system. In *Proceedings of the 15th Bled Electronic Commerce Conference*, pages 17–19, 2002.
9. S. Kaffille and G. Wirtz. Engineering Autonomous Trust-Management Requirements for Software Agents: Requirements and Concepts. *Innovations and Advances in Computer Sciences and Engineering*, pages 483–489, 2010.
10. H. Kautz, B. Selman, and M. Shah. Referral Web: combining social networks and collaborative filtering. *Communications of the ACM*, 40(3):63–65, 1997.
11. Y. Kim and K. Doh. Trust Type based Semantic Web Services Assessment and Selection. *Proceedings of ICACT, IEEE Computer*, pages 2048–2053, 2008.
12. S. Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, University of Stirling, Scotland, 1994.
13. L. Mui, M. Mohtashemi, C. Ang, P. Szolovits, and A. Halberstadt. Ratings in distributed systems: A bayesian approach. In *Proceedings of the Workshop on Information Technologies and Systems (WITS)*, pages 1–7. Citeseer, 2001.
14. P. Nurmi. A bayesian framework for online reputation systems. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 121–121, Feb. 2006.
15. S. Paradesi, P. Doshi, and S. Swaika. Integrating Behavioral Trust in Web Service Compositions. In *Proceedings of the 2009 IEEE International Conference on Web Services*, pages 453–460. IEEE Computer Society, 2009.
16. D. Quercia, S. Hailes, and L. Capra. TRULLO-local trust bootstrapping for ubiquitous devices. *Proc. of IEEE Mobiquitous*, 2007.
17. A. Rahman and S. Hailes. Supporting trust in virtual communities. *IEEE Hawaii International Conference on System Sciences*, page 6007, 2000.
18. R. Saadi, J. M. Pierson, and L. Brunie. Establishing trust beliefs based on a uniform disposition to trust. In *ACM SAC: Trust, Reputation, Evidence and other Collaboration Know-how track*. ACM Press, 2010.
19. G. Suryanarayana, J. Erenkrantz, S. Hendrickson, and R. Taylor. PACE: an architectural style for trust management in decentralized applications. In *Software Architecture, 2004. WICSA 2004.*, pages 221–230. IEEE, 2004.
20. G. Suryanarayana and R. Taylor. SIFT: A Simulation Framework for Analyzing Decentralized Reputation-based Trust Models. *Technical Report UCI-ISR-07-5*, 2007.
21. L. Vercouter, S. Casare, J. Sichman, and A. Brandao. An experience on reputation models interoperability based on a functional ontology. In *Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2007.
22. G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 2002.
23. R. Zhou, K. Hwang, and M. Cai. Gossiptrust for fast reputation aggregation in peer-to-peer networks. *IEEE Transactions on Knowledge and Data Engineering*, pages 1282–1295, 2008.
24. P. R. Zimmermann. *The official PGP user's guide*. MIT Press, Cambridge, MA, USA, 1995.

# Policy-based Access Control in Mobile Social Ecosystems

Sara Hachem, Alessandra Toninelli, Animesh Pathak, and Valérie Issarny
{sara.hachem, alessandra.toninelli, animesh.pathak, valerie.issarny}@inria.fr
INRIA Paris-Rocquencourt, France

***Abstract*—The novel scenarios enabled by emerging mobile social applications raise serious concerns regarding access control of users' contextual and social data. Given the variety of existing and upcoming social applications, it is important to provide (i) generic yet flexible policy models that combine expressivity with personalization, (ii) actual running infrastructures to enforce policy-based access control on heterogenous devices with minimal development/deployment effort, and (iii) user-interfaces to allow the easy specification of policies without dealing with the complexity of the underlying policy and data models. Toward this goal, in this paper we make three contributions. First, we present a novel policy framework for controlling access to social data in mobile applications. The framework allows the representation of expressive policies based on users' social interactions, which can be easily extended with new domain data models, while keeping policy model compatibility intact. Secondly, we demonstrate how we integrated the policy framework as part of Yarta, a middleware for managing mobile users' social ecosystems, implemented and deployed on laptops and smart phones. Third, we show the graphical policy editor provided with the policy framework to allow non-technology savvy users to easily specify and manage their access control policies.**

## I. Introduction

The popularity of social applications has been steadily increasing on the Web over the past few years. Advances in wireless network technologies and the widespread diffusion of smart phones equipped with sensing capabilities offer promising chances to enhance social applications and make them truly pervasive in everyday life. In addition, the formation of ad hoc networks enables social encounters between proximate users with common interests, anywhere and anytime [1], [2].

The novel scenarios enabled by these emerging *mobile social applications*, however, raise serious concerns regarding privacy and access control of users' data. The most critical aspect is that mobile social applications manage contextual data, such as personal contents, user interests and activities, as well as human relationships, which are sensitive per se and can be further used to infer sensitive information. It is therefore crucial to ensure an adequate level of control on information describing users' social environments and interactions.

Given the variety of existing and upcoming social applications, it is important to design a policy framework to be as generic and flexible as possible, so that several different applications can make use of it. This raises several requirements: on one hand, it calls for (i) generic yet flexible policy models that combine expressivity with personalization. On the other hand, it raises the need for (ii) actual running infrastructures to enforce policy-based access control on heterogenous devices with a minimal development and deployment effort, as well as (iii) user-interfaces to allow the easy specification of policies without dealing with the complexity of the underlying policy and data models.

Most policy models, even those designed for ubiquitous applications, are not able to represent access control choices within the complex dynamics of social interactions. Some recent efforts propose policy-based approaches to control access to shared resources in social networking applications [3]–[5]. Nevertheless, all these solutions rely on a rather simple modeling of social networks, which only includes the base "friendship" relationship, often considered as bi-directional (which is not the case in most real life social relationships, e.g., the "know of" relationship). In addition, current access control policy-based frameworks for social applications are generally designed to protect specific resources, such as media contents [3], [5] or medical information [4], and do not provide a generic model for any type of social information as an accessible resource.

This lack of generality, both in modeling access conditions and accessed resource, hinders the reuse of existing policy frameworks and their adaptation to new social application scenarios. Policies, based on various social constraints, should be designed to allow the specification of access control directives on different resources. Furthermore, they should be defined by relying on an interoperable data model to allow policy reuse across different user applications and policy exchange between different users of the same application. At the same time, policies should be customized for a specific application and its domain data model, to avoid imprecise and/or incorrect security directives. For example, for a user wishing to share different types of content (such as pictures and notes) via a social application, the generic concept for "content" would not be suitable as he would need to specify distinct policies for pictures and notes, respectively.

As far as the need for running infrastructures is concerned, not only are actual components needed to enforce access control, but those components should also be able to execute on a variety of mobile devices (e.g., smart phones) with minimal configuration effort. To the best of our knowledge, only a few solutions have been actually implemented as running systems [3], [5] but they may be difficult to reuse out of their proof-of-concept scenario, whereas no complete

The work is this article is partially supported by EC FP7 ICT NOE NESSOS project.

policy infrastructure exists for access control in mobile social applications. Note that the availability of fully implemented policy architectures has been crucial in the past towards the adoption of policies in real application scenarios for distributed systems, such as QoS and network management [6] or autonomous agent coordination [7].

A further requirement for the policy architecture is the availability of a user-interface to allow both application developers and end-users to specify their own security policies without dealing with the complexity of the underlying policy model. At present, policy frameworks for social applications often extend the interface of the target application [8], which are by definition application-specific, or they present policies according to their internal representation, thus requiring significant technical expertise to deal with the interface [3]. This is not suitable for social scenarios, where mobile users become the security administrator of their devices and applications, managing their own data and applications, rather than simply relying on external centralized or outsourced security management services [9].

Based on the above discussion, we claim the need for a flexible policy architecture to enforce access control in mobile social applications. Toward this goal, we make the following contributions. First, we have designed a novel policy framework for controlling access to social data in mobile applications. The framework is based on the semantic policy model presented in [10] and allows the representation of expressive policies based on users' *mobile social ecosystem* (MSE), that is, the rich set of social interactions occuring between people in mobile environments, according to various social relationships (e.g., friends, family or co-workers), different activities performed on content (commenting, tagging, etc.), as well as formation of groups and organization of events. The adoption of semantic technologies makes the extension of the policy model with new domain data models straightforward, while keeping policy model compatibility intact. Secondly, we have integrated the policy framework as part of Yarta, a middleware for managing mobile users' social ecosystems [11]. Third, we have provided a graphical policy editor to allow non-technology savvy users to easily specify and manage their access control policies.

The paper is structured as follows. Section II presents our policy framework, and in Section III we describe how we integrated it within the Yarta middleware architecture. Then, we provide details about the graphical policy editor we implemented (Section IV). To assess our contribution, in Section V we provide extensive evaluation of the current prototype implementation for smart phones. Finally, in Section VI we discuss related work. Conclusions and future work follow.

## II. A Policy Framework for Mobile Social Ecosystems

Our policy model allows the specification of access control policies to protect users' social data, based on social data themselves. Each policy defines under which conditions (*i.e., the social context)* a given *resource* is accessible via a certain *action*. The accessible resource is any information within the user's mobile social ecosystem. The action can either be `read`, `add`, or `remove` (a set of) triple(s) from/to the user's MSE data. The social context represents any socially meaningful information that constrains access to a resource. Note that the social context itself is expressed in terms of the user's MSE. For example, one may define the following policy: *Any member of INRIA can read the list of the friends of Alice who are also members of INRIA* Figure 1. In this case, the list of Alice's friends is the accessed resource (all nodes linked to alice via the *knows* relationship and are members of INRIA), while the social context is people who are members INRIA. Thus, the model supports the definition of access control policies based on users' social relationships and activities.

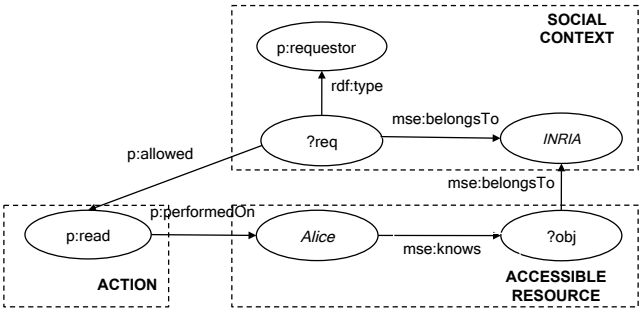



Fig. 1. Example Policy Graph

### A. Policy Representation Details

We use the representational model presented in [10] to specify policies. In particular, a policy is represented as a set of attributes with predetermined values, either constant or variable with constraints over the range of values. The current knowledge describing the mobile social ecosystem where the access request takes place (and which is the target of the request as well) is also modeled in terms of attribute/value pairs. For a policy to be "in effect" the attribute values that define the MSE knowledge have to match the definition of the policy attributes with constrained values (i.e., policy constraints).

Both the base MSE model and the policy model are represented using the Resource Description Framework (RDF)[1], a base Semantic Web standard, and the RDF query language SPARQL[2]. In particular, the MSE is represented as a graph of RDF triples, i.e., subject-predicate-object triples, with each statement describing an attribute and its value. A policy is a set of RDF statements or SPARQL triple patterns. The set of RDF statements and SPARQL triple patterns defining a policy is linked as a graph of nodes and arcs. Figure 1 shows the graph-based representation of the example policy introduced above. The prefixes `p:` and `mse:` represent the namespace of the policy model and the MSE model, respectively, while terms preceded by question mark represent variables. For the sake of simplicity we omit the namespace for instances.

By relying on a graph representation, our policy model allows the definition of any type of logical relation between the

[1]http://www.w3.org/TR/rdf-primer/
[2]http://www.w3.org/TR/rdf-sparql-query/

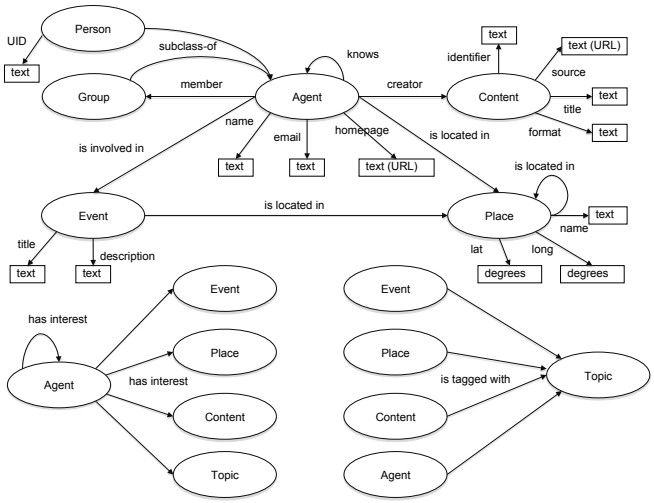



Fig. 2. MSE Data Model

base elements defined above, e.g., between the data requestor and the data owner, the requestor and the resource, the requestor and the environment, or any other relation the user might wish to specify. All these constraints can be specified by simply drawing new arcs between the policy nodes. Therefore, the model is able to represent existing policy models, such as role-based, identity-based or attribute-based, with enhanced expressive capabilities.

Finally, accessible resources can be defined in a flexible way by simply building appropriate graph patterns. For example, instead of controlling access to Alice's friends, we might control access to any information about her only by replacing the triple `(Alice, knows, ?obj)` with the less constrained `(Alice, ?pred, ?obj)`. Similarly, several different policies may be defined, possibly using multi-hop graph patterns.

We provide a rich representation of MSEs and the interactions possible in them based on the expressive and extensible model defined in [11]. For the sake of clarity we recall in Figure 2 the graph of first-class entities and relationships. Beside the availability of a base MSE model, access control policies for a specific mobile social application should be defined in terms of the domain knowledge characterizing that application. Towards this goal, our base MSE model is designed to be easily augmentable thanks to the extensibility features of RDF: this allows the easy specification of access control policies for different mobile social application scenarios.

On the other hand, RDF allows the association of a formal semantics to policy and MSE data models, and this supports simple reasoning over them. Because all mobile social applications share a common MSE data and policy model, they rely on a shared foundation of common meaning, which they can further extend based on specific requirements. By means of automated reasoning, classes and properties defined in application-specific extensions are put in clear semantic relation with base classes, thus enabling the reuse of existing policies in different applications. In summary, RDF reasoning capabilities, based on explicit semantics, and extensibility features are the key enablers for the reuse and exchange of policies between mobile social applications.

### B. Policy Evaluation

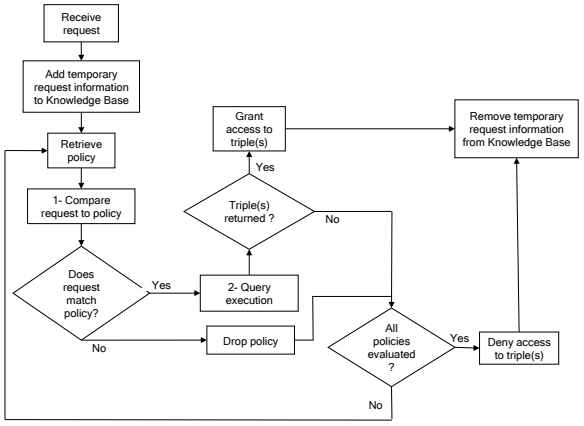



Fig. 3. Policy evaluation process

We now show how a policy is evaluated to answer a request for accessing data, given an MSE knowledge. Whenever a request is received, each policy currently enforced by the user is evaluated to determine whether the requested resource is accessible. In the following we describe the matching process for a single policy. The same process can be repeated for all policies. Note that, since the default behavior is to prohibit access unless differently stated, evaluation is performed until a policy granting access to the requested resource is found.

In general, access will take place within certain circumstances, i.e., a specific requestor asking to perform a given action on a resource. The resource can be known or unknown, depending on the action: in case of add/remove, the resource is known, while in case of a read action, the resource is in general unknown (because the requestor is typically asking for a type of information, but does not exactly know which information will be returned).

In particular, each access request on MSE data is represented as a SPARQL query, which always includes a set of RDF statements describing the requestor and the action. To avoid useless query execution, we perform an initial matching between the SPARQL queries representing the access request and the policy, respectively. Policy evaluation resolves then to the execution of two main steps as shown in Figure 3:

**1. Matching the access request against the policy:** This step is needed to verify if their graph patterns are compatible. If this is the case, the two queries are merged to create a single query, which combines the policy with information describing the current request (i.e., who is the requestor, what is the action, which resource is currently requested). If not, the policy does not apply to the current request and the request for access fails (for the considered policy).

**2. Executing the resulting query over the knowledge base:** Temporary information about the requestor and the action is added to the knowledge base, which also contains user's MSE data. Then, the combined query, obtained at the previous step,

is executed over the knowledge base; if any result is returned (i.e., at least one RDF triple), access is granted to that result. If not, the request for access fails (again, for the considered policy).

Note that when the requested resource is a specific (set of) RDF triple(s), access is granted if and only if each triple is accessible. This case applies to `add`/`remove` actions only, where the requestor knows exactly which triples to access. On the other hand, if the requested resource is a SPARQL pattern (i.e., a `read` action), access is granted to only those triples that are allowed by current policies, while others are simply filtered out during the evaluation process.

### C. An Example of Policy Specification & Evaluation

Let us recall the example policy represented in Figure 1: *Any member of INRIA can read the list of the friends of Alice who are also members of INRIA*. This policy can be formalized as the following SPARQL query:

```
CONSTRUCT {Alice mse:knows ?obj. }
where {    ?req rdf:type p:requestor
           ?req mse:belongsTo INRIA
           ?req p:allowed p:read.
           p:read p:performedOn Alice.
           Alice mse:knows ?obj.
           ?obj mse:belongsTo INRIA.}
```

Let us now consider the following access request: *Bob would like to see Alice's friends*. The corresponding SPARQL formalization reads as follows:

```
CONSTRUCT {Alice mse:knows ?obj. }
where {    Bob rdf:type p:requestor.
           Bob p:allowed p:read.
           p:read p:performedOn Alice.
           Alice mse:knows ?obj.
}
```

**Step 1:** The following elements are checked, to ensure that the element defined in the policy matches with the one defined in the request.

- Action. In this case, the action is read, as shown by the pattern `(p:read, p:performedOn, ?obj)`.
- Requestor. In this case, since the policy does not refer to a specific requestor's identity, the variable `?req` matches with `Bob`.
- Resource. In this case, the resource triple `(Alice, mse:knows, ?obj)` matches for the policy and the request, while the triple `(?obj, mse:belongsTo, INRIA)` needs to checked at the next step.

**Step 2:** The output of step 1 is the following SPARQL query:

```
CONSTRUCT {Alice mse:knows ?obj. }
where {    Bob rdf:type p:requestor          (1)
           Bob mse:belongsTo INRIA           (2)
           Bob p:allowed p:read.             (3)
           p:read p:performedOn Alice.       (4)
           Alice mse:knows ?obj.             (5)
           ?obj mse:belongsTo INRIA.         (6)
}
```

Temporary data, i.e., RDF statements #1, #3 and #4, are added to the Knowledge Base. The query is executed over the knowledge base, which contains both MSE data and temporary data. From the example we can see that the above query will succeed only if the MSE knowledge includes triple #2: `(Bob, mse:belongsTo, INRIA)`, and it will return all RDF triples (if any) matching the pattern constrained by statements #5 and #6. In other words, only if Bob is a member of INRIA, he will be returned the list of her friends who are also members of INRIA.

## III. Integrating the Policy Framework within Yarta Middleware Architecture

We have integrated our policy framework within Yarta [11], a middleware architecture providing mobile social application developers with a set of functionalities that allow them to easily create, manage and securely exchange MSE data. The middleware consists of two layers, as shown in Figure 4: the *MSE Management Middleware* layer, which is responsible for storing, managing and allowing access to MSE data, and the *Mobile Middleware* layer, which takes care of low level communication/coordination issues.

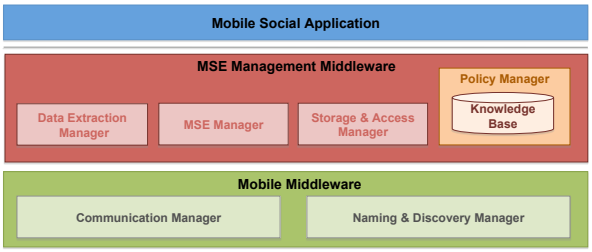



Fig. 4. Yarta Middleware Architecture

In this paper we only focus on components that, by integrating with the Yarta middleware, implement the policy model described above, namely the Knowledge Base and the Policy Manager. More details about the middleware architecture and its components can be found in [11].

Each user's social graph is managed by the *Knowledge Base* (KB) middleware component. The KB offers a set of interfaces describing the base concepts needed to manipulate RDF graphs, i.e., nodes, triples and graphs, as well as rich application programming interfaces to allow the retrieval, insertion and removal of data to/from the KB. The KB is also able to handle the merging of MSE graphs coming from different users.

To ensure access control enforcement according to the policies defined in the system, the KB is wrapped by a *Policy Manager* (PM). The Policy Manager intercepts any tentative access action on the KB, and performs reasoning on defined policies and the access request's context to determine whether the action is permitted. In particular, the PM evaluates all applicable policies with respect to current access conditions and, if no valid policy to allow access is found, it denies access to the requested resource (negative default).

In more detail, the PM middleware component, shown in Figure 5, consists of different subcomponents: the *Policy*

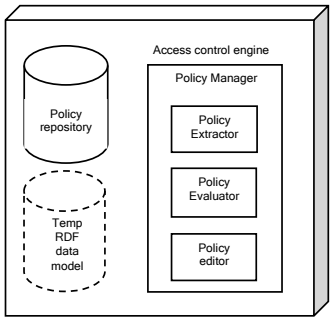



Fig. 5. Policy Manager components

*Extractor*, the *Policy Evaluator*, the *Policy Repository* and the *Policy Editor*. The Policy Extractor is in charge of extracting all user policies from the Policy Repository when the application is launched or when policies are edited. Extracted policies will then be ready for evaluation whenever a request for access is received. User policies are stored in the Policy Repository as a set of SPARQL queries, with each query representing a policy. This repository is accessible to the policy manager only.

The Policy Evaluator is in charge of performing policy evaluation upon access request. In particular, when a request is received, the Policy Evaluator receives information including the requested resource, the requestors identity, the requested action and a local copy of the user's MSE knowledge base. It adds temporary information about the request to its local knowledge base and combines it with extracted policies, as described in Section II. By executing the so obtained query on the local knowledge base, the Policy Evaluator builds a filtered data model, used to answer the access request.

In particular, the Policy Evaluator provides two main filtering functionalities. In case the input is a specific triple or set of triples (to be read, added or deleted from the graph), it returns a positive or negative response that allows or deny, respectively, access to that triple. In particular, for a set of triples, access is permitted only if all triples are accessible based on access control policies. In case the input is a query on the user's graph (only for read actions), the PM retrieves query results and filters them out based on current policies, thus returning to the KB only those triples that satisfy both the access request and the policies.

## IV. Policy Editing

A powerful policy framework such as ours may be of little use unless the (non-technical) end-user is able to employ it to easily specify access control policies over his data. As seen in Section II-C, the specification of even a simple policy may need several lines of SPARQL, a language that itself can be quite intimidating to the end-user. Also, the end-user might not be familiar with the RDF representation of the MSE data model used by the social application in question. Instead, one would like to use a more natural interface, such as those used to set filters for incoming messages in email clients. For this reason, we have designed and implemented a policy editor that hides the complexity of low level SPARQL queries and RDF graphs. Users can add new policies, view, edit or delete existing ones, all via an intuitive graphical interface. The various categories of resources (e.g., Person, Group, etc.) are automatically made available in the editor, as are the instances currently in the knowledge base.

We illustrate the use of the policy editor by using the policy introduced earlier in the paper. As shown in Figure 6, Alice, the owner of the data, can specify this policy using our policy editor to identify the following 4 properties of the policy:

**1. Requestor:** This can be specified either intensionally – i.e., by selecting a specific person as the eligible requestor, or extensionally – i.e., in terms of specific sets that group people together based on common properties or criteria, such as members of a group (e.g., "INRIA") or friends of a person. The latter method allows the owner to grant access to a set of users using a single policy.

**2. Action:** This can be `read`, `add`, or `remove`.

**3. Resource to be Accessed:** The editor allows the owners to grant access either to all information in the KB, or very specific information (e.g., "the list of friends of a person") by selecting one of the resource categories in the policy editor.

**4. Additional Conditions:** Additionally, the owner can specify resource-related conditions that the result of the query should satisfy (e.g. "returned set of users must also be members of INRIA").

Although the policy editor makes the specification of policies easy, its content/options differ from application to application due to the different MSE data models used by them. Implementing the code for it for each application manually can therefore be tedious and error-prone. To alleviate the above problem, the options in the UI of the policy editor are generated dynamically, based on the MSE data model used by the application, as well as the current state of the owner's KB. In particular, we use the RDF description of the former to auto-generate the categories in the policy editor, while the individual members of the categories are populated dynamically by querying the KB at policy creation time. Finally, to allow users to define more complex policies, the editor also provides them with the raw SPARQL file which they can edit before it is loaded by the policy manager.

## V. Implementation and Evaluation

### A. Prototype Implementation

The Yarta middleware prototype is written in Java 2 SE and has been deployed both on laptops running Windows/Mac OS, and on smart phones running Android. Similarly to Section III, we mainly focus on Knowledge Base and Policy Manager components here.

In the laptop prototype, both the Knowledge Base and the Policy Manager rely on capabilities offered by the Jena Semantic Web Framework [12], which is at present the most comprehensive framework to manage RDF and Web Ontology Language (OWL)[3] data in Java applications. Jena provides a set of native APIs for data manipulation and also supports the

[3]http://www.w3.org/TR/owl-features/

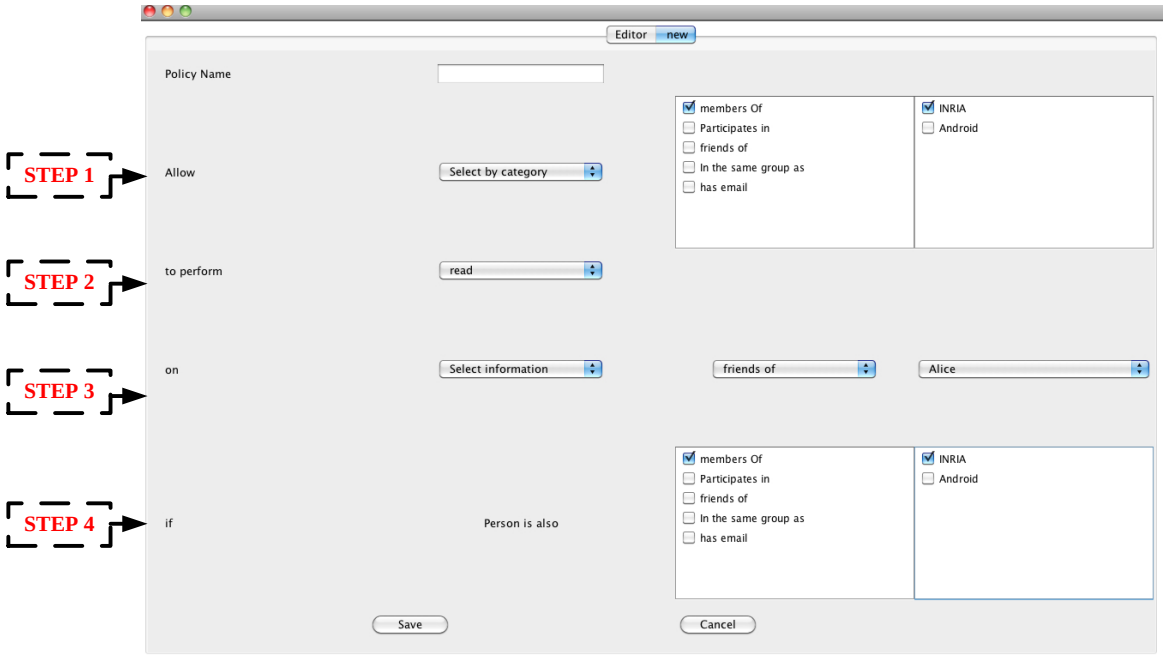



Fig. 6. Policy editor, with annotations for each step in the policy specification process.

SPARQL query language to retrieve data according to graph patterns. We also rely on Jena features to handle duplicates due to the merging of multiple graphs. The KB currently uses the filesystem as a backing store. For the Android prototype we exploit Androjena[4], an Android-compatible port of the Jena framework that has been recently released. This allows us to transparently run Yarta on the laptop and on the mobile phone by replacing Jena libraries with Androjena ones.

The Policy Editor was written in J2SE using the Swing graphical framework. We use the RDF description of the data model to auto-generate the categories in the policy editor using the Apache Velocity template engine[5] and RDFReactor[6].

### B. Experimental Results on Smart Phones

Ideally, the policy management subsystem should be responsive and scale well with KB size, and not depend on the type of the policy being evaluated. To evaluate the performance and scalability of the Policy Manager component, we executed several tests based on the following four parameters.

- size of KB: We increased the size of the knowledge base for social networks ranging from 400 to 2500 people.
- type of policy: We created three different policy sets. Each set presents a policy category - *Set 1* for requestor-related policies, *Set 2* for resource-related policies, and *Set 3* for policies with cross-related conditions, i.e., constraints binding different elements of policies (e.g., the requestor and the resource).
- type of response : Both `grant access` and `deny access` responses were generated
- type of action: We performed tests for `read`, `add`, and `remove` operations.

**Testbed Setup:** The tests were run on a Google Nexus One mobile phone running Android 2.2 OS, equipped with a 1 GHz processor and 512 MB of RAM. To provide test cases, we exploited an anonymized data set that was gathered from Facebook [13]. We selected 10 sizes ranging from 400 to 2500 people, and for each size value we randomly extracted 10 sub-graphs from the data set using the snowball sampling algorithm [14], which has been shown to preserve the topological structure of graphs. For each extracted sub-graph, we created a KB (in RDF) containing nodes of type `Person` and `mse:knows` relationships between them. For testing purposes we also added `mse:email` and `mse:homepage` attributes for each person. Then, for each RDF file and each policy category defined above, we evaluated the following requests:

1) `add email`
2) `remove email`
3) `read email`
4) `read friends`
5) `read graph` with nodes about a person

and calculated the average evaluation time for all graphs per request for each policy set.

**Experimental Results:**The performance results of the Policy Manager are summarized in Figures 7 and 8. Due to lack of space, not all results are shown. The time needed to evaluate requests increases linearly with the size of the knowledge base. All types of action and policy were seen to show similar evaluation times (at most 600 milliseconds), except for the `read` requests evaluated with policies with cross-related elements, which took up to 3000 milliseconds for 2500 people, as shown in Figure 7(a). This is understandable given that

[4]http://code.google.com/p/androjena/
[5]http://velocity.apache.org/
[6]http://semanticweb.org/wiki/RDFReactor

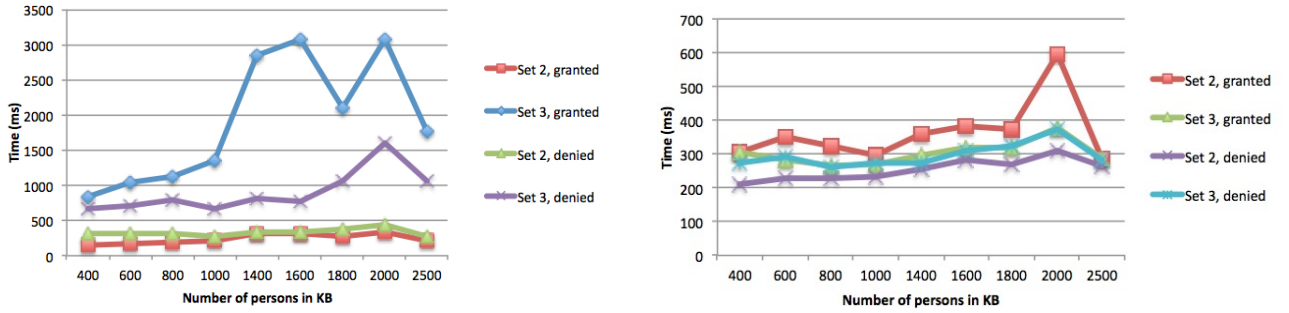



Fig. 7. Time taken by the Policy Manager to a) `read graph` b) `remove triple`

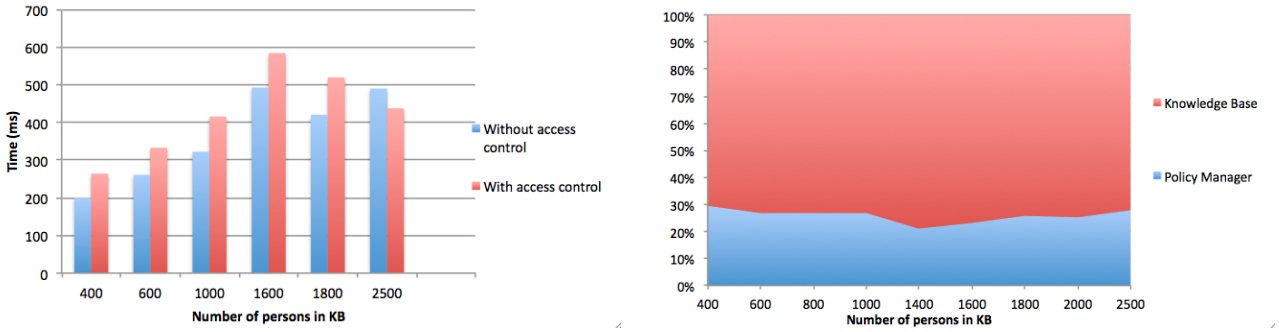



Fig. 8. a) `total time` and b) `time ratio` for add action and policy Set 3

cross-related conditions are more complex than strictly subject or resource-related ones.

Additionally, the response time was seen to be independent of the outcome of the evaluation, again with the exception of the `read graph` requests evaluated with policies with cross-related elements. Also, as shown in Figure 7(a), denying read access to a graph requires less time than allowing to do so.

Regarding the overhead introduced by the PM, we tested the evaluation times needed to execute requests with and without policy enforcement in place. The outcome showed that the overhead due to access control depends on the requested operation. As shown in Figure 8(a), enforcing access control on `add` requests results in a minor overhead unlike `read` requests results. This is mainly due to performance limitations of currently available implementation tools, which in case of a `read` have a major impact.

As for the share of work done by the PM, our experiments showed different behaviors for `read`/`remove` requests and `add` requests. In the former case, the PM requires more than 90% of the total time needed to execute a request issued to the KB, while in the latter it takes no more than 30% of the time. The reason is that, in the current implementation of `read`/`remove` operations, the PM performs time-consuming filtering operations on the knowledge base, while the KB only returns or remove triples. On the contrary, to answer `add` requests, the Knowledge Base performs more time consuming operations, such as range and domain checking. Note that `add` operations are generally faster than `read`/`remove` as they do not require retrieval of triples from the knowledge base.

## VI. Related Work

With the increasing popularity of social applications, different solutions have been proposed in recent years to control access to personal data and content shared via social networking applications. In this section we discuss access control models and frameworks that are specifically targeted to social applications, possibly mobile.

A first category of solutions extends popular social applications (e.g., Facebook) to provide enhanced access control over personal data or added functionalities. For example, [4] proposes fine-grained access policies to medical data, based on the purpose of the requested access. The policy model is SecPAL [15], which however does not provide support for MSE data modeling nor semantic inference. Authors of [3] complement Facebook-like applications with support for collaborative policy editing, which can be useful when multiple users have interest in restricting access to a resource (e.g., a picture). To automatically infer user policies based on user behavior, authors of [5] propose machine learning techniques. These are all interesting efforts towards the provisioning of new functionalities for policy-based access control in social applications. However, they do not focus on providing expressive policy models nor enforcement infrastructures like Yarta.

Semantic technologies have also emerged as a promising choice to represent and reason about policies. In particular, recent work proposed to adopt semantic policies to control access to resources in social networking applications, such as Facebook [16] [17]. Both works propose, with some differ-

ences, to represent policies as Semantic Web Rule Language (SWRL) rules[7]: by reasoning on the social knowledge base, represented in OWL, they derive the set of active permissions. Similarly to us, these approaches provide rich descriptions of social interactions via semantic modeling. In [16] authors also introduce the concept of trust in a relationship, which we do not currently handle. However, since reasoning is performed on the whole knowledge base to infer the current list of permissions, this rule-based approach has two main limitations: first, forward reasoning requires the whole social knowledge to be stored in the same node in order to obtain all valid permissions, thus making the system inherently centralized. Furthermore, whenever a change in the social knowledge occurs, all permissions must be re-computed, which may raise efficiency concerns. Also, existing literature on rule-based systems reports of extremely critical situations, where the rule-base size made rule management such a complicated task to require a team of expert administrators [18].

Authors of [19] propose a relation-based access control model to support data sharing among large groups of users. The main idea is to represent permissions as relations between users and data, thus decoupling them from roles as in role-based access control. Authors provide a formal representation of their model, which includes E-R diagrams and their mapping to Description Logic, to allow reasoning. They do not, however, provide any specific social model as this is not their primary focus. In addition, the framework adopts hierarchies whose semantics may not always be clearly defined, which might make the model not easily manageable.

To the best of our knowledge, none of the systems presented above provide a full policy infrastructure like Yarta does. Some systems do provide proof-of-concept implementations, which however are designed for a specific application or resource, and we are not aware of their availability for reuse. On the contrary, our policy framework is generic and expressive enough to be used for any mobile social application, and it is integrated in a reusable middleware architecture whose components allow the enforcement of access control. Finally, only few solutions provide user-interfaces. In particular, [3] provides a user-interface, which however requires to deal with concepts like strong/weak conditions that might not be intuitive to define in practice, while [17] requires to deal directly with a SWRL editor. Our GUI allows non technical users to specify policies without dealing with the underlying SPARQL model.

## VII. Conclusion

The novel scenarios enabled by emerging mobile social applications raise serious concerns regarding access control of users' contextual and social data. In this paper we presented a novel semantic policy framework for controlling access to social data in mobile applications. The framework allows the representation of expressive policies based on users' (MSE) and makes the extension of the policy model with new domain data models straightforward, while keeping policy model compatibility intact. We have integrated the policy framework as part of the Yarta middleware architecture and evaluated it via extensive testing. Finally, we provide a graphical policy editor to allow easy specification and management of users' access control policies. Our evaluation shows that our initial implementation of this novel policy framework scales well, although the response times have room for improvement in some cases.

Our current and future work include the validation of our policy framework in emerging application scenarios, such as social gatherings and smart city applications. We are also working on the optimization of the policy evaluation process and we are planning to test the usability of our interface via field testing with non-technical users.

[7] http://www.w3.org/Submission/SWRL/

## References


[1] E. F. Churchill and C. A. Halverson, "Guest editors' introduction: Social networks and social networking," *IEEE Internet Computing*, 2005.

[2] Q. Jones and S. A. Grandhi, "P3 systems: Putting the place back into social networks," *IEEE Internet Computing*, vol. 9, no. 5, 2005.

[3] R. Wishart, D. Corapi, S. Marinovic, and M. Sloman, "Collaborative privacy policy authoring in a social networking context," 2010.

[4] P. Kodeswaran and E. Viegas, "A policy based infrastructure for social data access with privacy guarantees," *Policies for Distributed Systems and Networks (POLICY), 2010 IEEE International Symposium on*, 2010.

[5] M. Shehab, G. Cheek, H. Touati, A. C. Squicciarini, and P.-C. Cheng, "User centric policy management in online social networks," *Policies for Distributed Systems and Networks, 2010 IEEE Intl. Symp. on*, 2010.

[6] L. Lymberopoulos, E. Lupu, and M. Sloman, "An adaptive policy-based framework for network services management," *Journal of Network and Systems Management*, vol. 11, no. 3, 2003.

[7] A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, "Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement," *POLICY*, 2003.

[8] A. Besmer, H. R. Lipford, M. Shehab, and G. Cheek, "Social applications: exploring a more secure framework," *5th Symposium on Usable Privacy and Security*, ser. SOUPS '09.

[9] D. Smetters and R. E. Grinter, "Moving from the design of usable security technologies to the design of useful secure applications," *New Security Paradigms Workshop*, 2002.

[10] A. Toninelli, R. Montanari, O. Lassila, and D. Khushraj, "What's on users' minds? toward a usable smart phone security model," *Pervasive Computing, IEEE*, vol. 8, no. 2, pp. 32–39, April-June 2009.

[11] A. Toninelli, A. Pathak, and V. Issarny, "Yarta: A middleware for managing mobile social ecosystems " *International Conference on Grid and Pervasive Computing (GPC 2011)*, 2011.

[12] "Jena," last visited: May 2010, http://jena.sourceforge.net/.

[13] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," *4th ACM European conference on Computer systems*, 2009.

[14] J. Illenberger, "Estimating properties of snowball-sampled (social) networks," "http://matsim.org/uploads/Seminar2008_Illenberger_SnowballSampling.pdf", 2008.

[15] A. D. G. Moritz Y. Becker, Cédric Fournet, "Secpal: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, pp. 619–665, june 2010.

[16] B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham, "A semantic web based framework for social network access control," *14th ACM symposium on Access control models and technologies*, ser. SACMAT '09., 2009.

[17] N. Elahi, M. Chowdhury, and J. Noll, "Semantic access control in web based communities," *Computing in the Global Information Technology*, 2008. ICCGI '08.

[18] J. Bachant and J. McDermott, "Readings from the ai magazine," R. Engelmore, Ed. Menlo Park, CA, USA: American Association for A.I., 1988, ch. R1 Revisited: four years in the trenches.

[19] F. Giunchiglia, R. Zhang, and B. Crispo, "RelBac: Relation based access control," *Semantics, Knowledge and Grid, Intl. Conf. on*, 2008.

# Proximity-Based Trust Inference for Mobile Social Networking *

Amir Seyedi, Rachid Saadi, and Valérie Issarny

ARLES Project-Team
INRIA CRI Paris-Rocquencourt, France
{name.surname}@inria.fr

**Abstract.** The growing trend to social networking and increased prevalence of new mobile devices lead to the emergence of mobile social networking applications where users are able to share experience in an impromptu way as they move. However, this is at risk for mobile users since they may not have any knowledge about the users they socially connect with. Trust management then appears as a promising decision support for mobile users in establishing social links. However, while the literature is rich of trust models, most approaches lack appropriate trust bootstrapping, i.e., the initialization of trust values. This paper addresses this challenge by introducing proximity-based trust initialization based on the users' behavioral data available from their mobile devices or other types of social interactions. The proposed approach is further assessed in the context of mobile social networking using users behavioral data collected by the MIT reality mining project. Results show that the inferred trust values correlate with the self-report survey of users relationships.

**Key words:** Trust bootstrapping, mobile social network, small worlds

## 1 Introduction

Portable devices have gained wide popularity and people are spending a considerable portion of their daily life using their mobile devices. This situation together with the success of social networking lead to the emergence of mobile social networking. However, anytime and anywhere interactions have a built-in risk factor. Development of trust-based collaborations is then the solution to reduce the vulnerability to risk and to fully exploit the potential of spontaneous social networking [5]. In our work, we aim at developing a trust management method for mobile social networking. Then, the challenge we are addressing here is how to initiate trust values and how to evaluate unknown mobile users using initiated trust values, to enable impromptu social networking.

Computational trust brings the human concept of trust into the digital world, which leads to a new kind of open social ecosystem [13]. In general, the notion

* Work supported by EU-funded project FP7-231167 CONNECT and by EU-funded project FP7-256980 NESSOS.

of trust can be represented by a relation that links trustors to trustees. The literature [18] includes two main categories of relations to set trust values for trustees, namely: (i) direct-based and (ii) recommendation-based relation.

Most existing trust models focus on assessing recommendation-based relationships [19] and lack the bootstrapping stage, which is how to initialize direct trust in order to efficiently start the trust model operation. This is very problematic and challenging, since recommendation-based relationships are built upon bootstrapped direct-based relationships. Indeed, most solutions that address trust assessment make one of the following assumptions:

- Trust initialization is not a problem of the model; it is the responsibility of the actors of the system [8]. However, this task remains challenging, especially when it comes to evaluating trustees numerically (e.g., 0.1, 0.2, 0.15, etc.).
- The trust model initially evaluates trust relationships with a fixed value (e.g., 0.5 [9], a uniform Beta probabilistic distribution [10], etc.) or according to the trust disposition of the trustor [15] (i.e., pessimistic, optimistic, or undecided). In [17], trust is initialized by asking trustors to sort their trustees rather than assigning fixed trust values. There are other bootstrapping solutions [1, 2, 16] that assess trustees into different contexts (e.g., fixing a car, babysitting, etc.) and then automatically infer unknown trust values from known ones of similar or correlative contexts. However, if no prior related context exists, these solutions lack initialization of trust.

We have developed our trust model based on the hypothesis that it is possible to measure and bootstrap trust from human social behavior. Therefore, in this paper, we investigate a formal approach that quantifies human proximity from which possible trust relationships are transparently and automatically inferred and assessed on behalf of the trustor. We choose proximity between people as an effective measure for trust. Because, proximity between people is not only a matter of trust, but it increases trust affinity as well [4]. In other words, people spend more time with those whom they trust and, at the same time, if they start spending time with new people, it is likely that trust relationships will arise and evolve.

In order to better understand the contribution and evolution of proximity in the human society, consider the fact that a society is initiated by people who live in the same territory. Fukuyama [7], describing the role of trust in a society, mentions that people can build efficient economy and social organization, if they have wide and efficient trust networks. It shows clearly how trust and proximity of people are tied together to initiate a successful society. As a result, today we have different cultures and societies in the world simply because of their founders being at different location and proximity. Building on this social knowledge, this paper introduces a method for bootstrapping trust values in mobile environments, based on the *proximity* of people. However, in today's virtual world expanding the physical one, proximity is not just about the physical distance between people. Practically, while people who are physically close

maybe detected using technologies such as Bluetooth [6], other types of proximity like phone calls, emails, social network interactions, etc. can be detected by the implementation of virtual sensors. We classify the range of proximity-based trust values semantically for further judgments based on these values. Then, the initiated trust values can be used to calculate similarity between people from the standpoint of trust. Similar people can make good recommendations to each other. Hence, they can evaluate not-directly-known users on each others behalf. So, when mobile users are about to interact with unknown users, they may acquire the trust knowledge through known similar users. The process should be feasible in a limited number of hops because of the small world phenomenon [14].

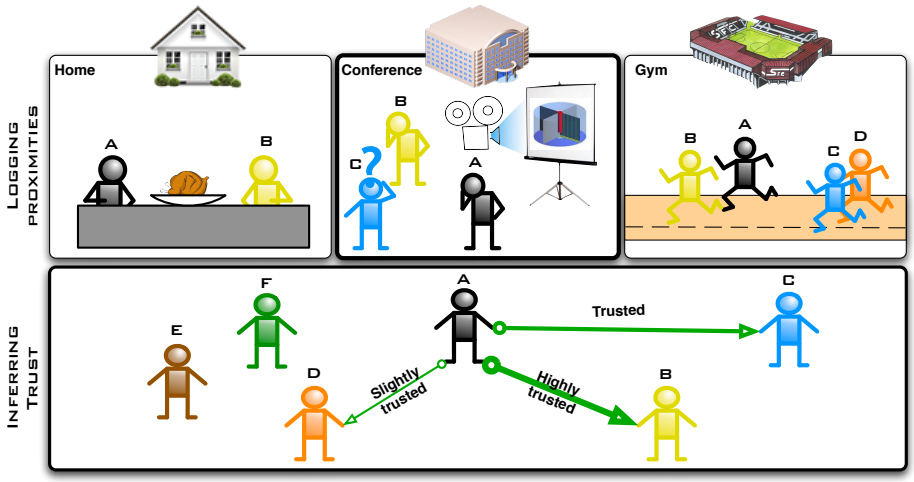



**Fig. 1.** User A evaluates others based on the observed proximity with others

The next section characterizes proximity towards trust assessment and is then followed by proposed proximity-based trust initialization in Section 3. Then, we evaluate the effectiveness of the proposed approach using the MIT reality mining dataset. Finally, we conclude in Section 5, summarizing our contribution and sketching our future work.

## 2 Trust and Proximity

We make the assumption that there is a strong correlation between proximity observations and real social relationship [6]. Proximity itself can be considered as a consequence of trust relationship, while at the same time the longer users are in proximity, the higher the probability of their friendship increases. Moreover, as noted by [4], proximity is a measure of trust as well as a cause to trust between users.

Thus, we argue that *proximity* is the *nearness* of any two persons in space or time. Let $P$ be the proximity between two persons. Two persons are in *physical proximity* if the nearness happens in the same space and time. Two persons are in *virtual proximity* if the nearness happens only in the time dimension. Physical proximity can be detected by various technologies (e.g., blue-tooth, Wi-Fi, etc.) and likewise virtual proximity through monitoring of social activities (e.g., chat, SMS, voice call, liking a content on facebook etc.). The collected proximity-related data provides information such as when, where, how frequently, and for how long people were in the proximity of each other.

In general, the definition of proximity takes several forms (from physical to virtual) and differs according to context (work, home, etc.) as well as it can be quantified by duration or frequency. From a social point of view, from the context in which proximity happens, we may identify a quality difference between the observed proximities. For instance, if the proximity happens at home, it is more intimate than a proximity in a professional meeting. Hence, in order to be able to aggregate of different types of proximity and consider the value difference between them, we characterize a proximity data type, namely $\eta$, as a tuple: $\eta =< p, l, t, d_s, d_d, s, m >$ where,

- Proximity type ($p$): Proximity type has two modes, virtual and physical. For instance, this helps distinguishing between a face-to-face interaction and virtual proximity.
- Location ($l$): Location is the position in physical space, in which the proximity happens. Location meaning can be expanded semantically, by looking to social semantic aspects of different definitions for location. For instance, *home* is a location in which trust is included by definition. Location has an effect on intimacy, e.g., the difference between outdoor and indoor proximity.
- Time ($t$): The time context is the temporal measurement of an instance in which the proximity happens. However time definition can be expanded semantically. For instance, weekend or working time has different social values. We take into account the quality difference of proximity as time changes, e.g., during weekend, being in the proximity of friends is more likely and therefore is a more valuable proximity.
- Source device type ($d_s$): Device type helps to includes the nature of device in terms of mobility etc. e.g., mobile device like smart-phone versus laptop). The observed proximity from a mobile device is more reliable as people are more likely to have their mobile device always with them. Then, $d_s$ is the source device that belongs to the observer user.
- Destination device type ($d_d$): $d_d$ is the destination device of a user who has been observed.
- Sensing method ($s$) (e.g., physical sensors, virtual sensors): Sensing methods take into account the technology effect on measurement method. or example bluetooth detects people in a shorter range than wifi does. So the detected proximity by bluetooth is more reliable as it catches the closer users.
- Measurement type ($m$): Measurement type indicates the difference between duration and frequency of a proximity. Hence, we introduce a proximity coefficient, which is necessary for combining different types of proximity.

Proximity data types enable us to consider value difference between proximities. We in particular assign a coefficient for each proximity data. This can be done using techniques such as fuzzy logic; logic can decide the weight of specific proximity data types in terms of trust. Thus, several sources of proximity can be combined by a weighted average using their coefficients. Let $k_{\eta_i}$ be the coefficient for an observed proximity of type $\eta_i$. $k_{\eta_i}$ is calculated for any given $\eta_i$ by logical aggregation of proximity data type parameters. $K = \{k_{\eta_1}, k_{\eta_2}, k_{\eta_3}, ...\}$ is the set of coefficients for different types of proximity, coefficients are bounded to the range of $[0, 1]$ . Accordingly, $K_B^A$ is the set of all proximity coefficients between users $A$ and $B$. An example of three different proximity data types is shown in Table 1.

| $\eta$ | $p$ | $l$ | $t$ | $d_s$ | $d_d$ | $s$ | $m$ |
|---|---|---|---|---|---|---|---|
| $\eta_1$ | physical | anywhere | anytime | mobile | mobile | bluetooth | duration |
| $\eta_2$ | physical | office | working time | mobile | laptop | WiFi | duration |
| $\eta_3$ | virtual | anywhere | night | mobile | mobile | SMS | frequency |

**Table 1.** Proximity Data Types

Given the above types of proximity we define proximity records as:

`ProximityRecord=<UserID,`$\eta$`, Value>`

where the `Proximity` tuple is composed by the `UserID`, which is the unique identifier of the observed user, $\eta$ is the data type of the observed proximity; and `value` is the observed proximity, which is duration or frequency based on the data type. Hence, each user's device is assigned with a set of `Proximity` tuples called *observed set*, as exemplified in Table 2.

| UserID | $\eta$ | Value |
|---|---|---|
| B | $\eta_1$ | 200h |
| C | $\eta_1$ | 20h |
| D | $\eta_1$ | 80h |
| E | $\eta_1$ | 30h |
| F | $\eta_1$ | 0,5h |

**Table 2.** An example of proximity duration data provided by user $A$ device

## 3 Proximity-based Trust Initialization

Proximity-based trust initiates trust values between nodes with a one-to-one trust relationship. Using an appropriate conversion method for various proximity data is the most challenging part of trust calculation. As a matter of fact, it is

a difficult task to make a conversion from varied types of observed proximity to a range of trust values that are meaningful. For the conversion of proximity records to *Proximity-based trust* values, we use *standard score* formula. Standard score has a normalization effect on the amount of observed proximities according to the observer(user's) average activity.

### 3.1 Definitions

Normalization has several positive outcomes. First, social interaction quantity varies a lot according to user personality in a social network [12]; as a result the amount of proximity varies substantially for different nodes [6]. Second, there are multiple types of proximity; normalized scoring eases the process of combining trust values according to different proximity data types.

**Definition 1 (Standard Score).** *A* standard score *[3] indicates how many standard deviations ($\sigma$), an observation or datum($x$) is above or below the mean($\mu$):*

$$Z = \frac{datum - mean}{standard\ deviation} = \frac{x-\mu}{\sigma} \tag{1}$$

As a result of using standard score, each peer normalizes trust values based on their average proximity duration with anyone. Therefore, each trust value is unbiased and bounded into a determined range, which hides the effect of variation of proximity duration due to peers having various behavior. Hence, by using standard score formula, we process the observed proximity as follows.

**Definition 2 (Observed Set).** *The Observed Set (OS) of a user includes all the users that have been detected in proximity of a given user.*

We use a time finite subset of observed proximities for trust evaluation. Therefore, we define the proximity window function as:

**Definition 3 (Proximity Window Function).** *The proximity window function $P^w_{\eta_i}(A,B)$ accumulates the proximity of user $B$ monitored by user $A$ during the time window $\underline{w}$ and with proximity type $\eta_i$.*

**Definition 4 (Proximity-based Trust Function).** *Proximity-based trust is basically calculated using* standard score formula*. The proximity-based trust function is denoted by $T^t_{\eta_i}$ and is formally defined as:*

$$\begin{array}{ll} T^t_{\eta_i} : \mathbb{U} \times \mathbb{U} \to \mathbb{R} & \mathbb{U} : \textit{set of Users} \\ (A,B) \to T^t_{\eta_i}(A,B) & \mathbb{R} : \textit{set of real numbers} \end{array}$$

$$T^t_{\eta_i}(A,B) = \begin{cases} -\infty & if(B \notin OS_A) \\ \frac{P^w_{\eta_i}(A,B) - \mu^w_{\eta_i}}{\sigma^w_{\eta_i}} & if(B \in OS_A \wedge t \leq w) \\ (1-\alpha) * T^p_{\eta_i}(A,B) + \alpha * \frac{P^w_{\eta_i}(A,B) - \mu^w_{\eta_i}}{\sigma^w_{\eta_i}} & if(B \in OS_A \wedge t > w) \end{cases} \tag{2}$$

*where:*

- $T^t_{\eta_i}(A,B)$ *is the proximity-based trust value given by user* $A$ *for user* $B$ *at the instant* $t$ *with proximity data type of* $\eta_i$.
- $T^p_{\eta_i}(A,B)$ *is the past acquired proximity-based trust value by user* $A$ *for user* $B$ *at the instant* $p = t - t\%w$ *with proximity data type* $\eta_i$.
- $P^w_{\eta_i}(A,B)$ *is the cumulative proximity of* $B$ *in the given period of time* $w$ *with proximity data type* $\eta_i$.
- $\alpha$ *is a coefficient which is in range of* $]0,1[$ *and defines how significant is the impact of new observed proximities on the last calculation of proximity based trust value.*
- $\mu^w_{\eta_i}$ *and* $\sigma^w_{\eta_i}$ *are, respectively, the observed period average and the standard deviation during the time window* $w$ *with proximity data type* $\eta_i$.

**Definition 5 (Proximity-based Trust Aggregation Function).** *The proximity-based trust aggregation function is for combining trust values, which is inferred from different proximity data types. It is formally defined as:*

$$\begin{aligned} T^t &: \mathbb{U} \times \mathbb{U} \to \mathbb{R} \qquad \mathbb{U} : \textit{set of Users} \\ (A,B) &\to T^t(A,B) \quad \mathbb{R} : \textit{set of real numbers} \end{aligned}$$

$$T^t(A,B) = \begin{cases} -\infty & if(B \notin OS_A) \\ \dfrac{\sum_{i=1}^{|K_B^A|} k_{\eta_i} * T^t_{\eta_i}(A,B)}{\sum_{i=1}^{|K_B^A|} k_{\eta_i}} & if(B \in OS_A) \end{cases} \tag{3}$$

where $T^t_{\eta_i}$ and $k_{\eta_i}$ are the trust value and the coefficient for proximity type $\eta_i$, respectively $K_B^A$ is the set of coefficients for all the observed proximity types between users $A$ and $B$. Thus, by using the equation 2, we consider only the proximity that occurs during time window $w$. Then, for a new time window, the latest assessed trust value ($T^p_{\eta_i}$) is used to serve as an input for new trust assessment.

### 3.2 Semantical Trust Inference

Given $T^t$, we are able to infer trust relationships between users. For the moment we do so by splitting the trust scale equally into four sections with respect to the normal distribution probability density in each **area** and according to the experiments of trust calculation we did using real proximity data from [6] (Illustrated in Figure 2), we define four trust levels and each level includes 25 percent of the observed peers, namely: *Unknown Trust*, *Slightly trusted*, *Moderately trusted* and *Highly trusted*. Unknown Trust represents all the persons whose trust ($T^t$) is assessed into interval $]-\infty, -0.67[$. For this category, we consider that the system has insufficient information to infer trust. All the other that are assessed over $-0.67$ are considered as trusted entities and they are classified into three

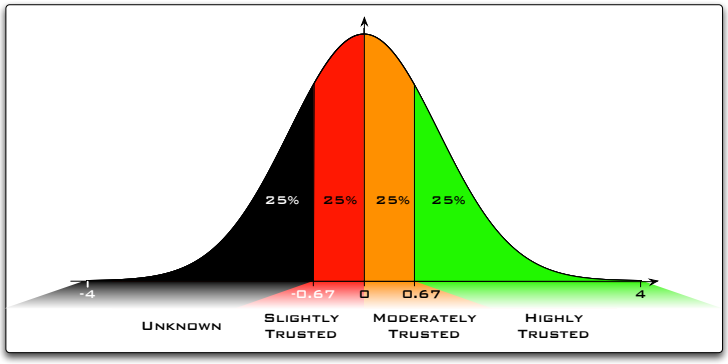



**Fig. 2.** Trust Scale for normalized Trust values

other categories: (i) people who may be slightly trusted (i.e., people assessed in range of $[-0.67, 0[$), (ii) people who may be moderately trusted (i.e., ones assessed into the range $[0, 0.67[$) and (iii) people who may be highly trusted (i.e., ones assessed into the range $[0.67, +\infty[$).

Then, according to our classification, which is based on density of peers in each score interval and exprimental results, we consider that a proximity-based trust relationship is established if an inferred proximity-based trust value is higher than $-0.67$. We define this relation as follows:

**Definition 6 (Proximity-based Trust relationship).**
*Let $A$ and $B$ be two users. If $T(A, B) \geq 0.67$, we say that the Trust relation is verified between $A$ and $B$. Formally:*

$$if\ T(A, B) \geq -0.67\ then\ A \xrightarrow{T} B$$

From the defined relationship, we introduce two trust concepts, namely, the trustee set (Definition **??**) and the trustor set (Definition **??**). Those sets are defined for each user and gather respectively his trustees and his trustors.

***Example:*** To explain our approach, we use the following example. User $A$ has a mobile device with a proximity logging application. The observed set of user $A$ is composed by: $OS_A = \{B, C, D, E, F\}$. Table 2 introduced in Section 2 shows an example of cumulative proximity duration that can be provided by user $A$ device with $\eta_1$ proximity data type. For instance: $P^w_{\eta_1}(A, B) = 200h$. Considering an unbounded time window (i.e., $w = +\infty$), the proximity duration average is: $\mu^w_{\eta_1} = 66.1$. For calculating the standard deviation, we first compute the difference of each data point from the mean, and then square the result: $(200 - 66.1)^2 = 17929.21$, $(80 - 66.1)^2 = 193.21$. We repeat the same process for the other values. Then, we calculate the standard deviation by dividing the sum of these values by the number of values and take the square root:

$\sigma^w_{\eta 1} = \sqrt{\frac{17929.21+2125.21+193.21+1303.21+4303.36}{5}} = 71.90$

The proximity-based trust value for the user B with 200 hours of proximity is then calculated using Formula: 3, $T^t_{\eta_i}(A,B) = \frac{200-66.1}{71.90} = 1.86$

Therefore, a one-to-one trust relation is established between users A and B (i.e., $A \xrightarrow{T} B$), which means that $B \in Tee(A)$ and $A \in Tor(B)$. Moreover, $A$ may highly trust $B$ because $T^t(A,B) \geq 0.67$.

In next section we evaluate our approach experimentally using the MIT real mobility data.

## 4 Experimental Evaluation

A full-fledged evaluation of our approach needs a large-scale proximity dataset with different data types, in order to have possibilities for combining trust values from different types of observed proximity. Also, for a multi-hop trust estimation, large number of users is needed. That aside, a survey of trust and/or other social facts such as friendship between the observed users is needed to make a comparison between inferred trust values and real social facts. While to the best of our knowledge there is no such kind of vast proximity dataset publicly available, we have used the reality mining dataset[1] [6] as the only existing public dataset of mobile phone proximity records with self-report survey data. The reality mining project was developed by the MIT Media Lab during years 2004-2005 by using 100 Nokia 6600s with Context logging software. They have gathered 330,000 hours of continuous behavioral data logged by the mobile phones of the subjects. They also did a survey in which they asked users about friendship and whom are they going to meet.

To illustrate the capability of our approach, we answer the following question: To what extent the bootstrapped trust values are in correlation with real social facts(e.g. friendship)?

We run the evaluation with the following steps. First, we calculate the proximity-based trust values between users. Then, by comparing the calculated proximity-based trust values of each user to the answers he provided in the survey, we verify if the inferred trust values are coherent with friendship.

From the reality mining dataset, we can calculate the proximity duration between two persons which has been detected by bluetooth. We apply the proximity-based trust function (Equation 2) to the proximity durations in order to obtain proximity-based trust values, $T^t(A,B)$, of each user. From the survey, each person predicts his possible future proximity with a friend, or if they are going to meet any other person inside or outside the lab. From this survey, we may tell that mentioned persons are either friends or they are important from

[1] http://reality.media.mit.edu/

| Group | Average of minimum trust | Average of trust values |
|---|---|---|
| Friends | 1.4070 | 2.0209 |
| inLab | -0.3079 | 0.7696 |
| outLab | 0.0068 | 1.0460 |

**Table 3.** Average $T^t(A, B)$ value for reported people in survey

the user point of view. We can make the judgment that it is probable that trust relationships exist between the reported users. For these groups (Friends, inLab, outLab), average trust value is shown in Table 3. To find out the relevance of the proximity-based trust values, as we can perceive, highest average of trust values are assigned to friends. Based on the given definitions (Figure 2), friends are assigned with highly trusted notion. For inLab group, which is the people that users meet inside the MIT Lab, the values are overall located in slightly trusted and moderately trusted classification. For outLab group, which usually consist of friends, family and friends of friends, that a person meets outside of working area, the values are around the barriers of highly trusted group. This experiment shows that trust values are related to the social strength of a relationship. For instance, highest values belong to friends. Additionally, we calculated similarity between users, which is used for the trust transitivity calculation. Table 4 shows that similarity values are behaving very similar to the proximity-based trust values, and they change with the characteristics of relationship. Knowing that similarity is a measure of trust, this arrangement evidences the social fact that friends are similar in their relationships and they are favorite recommender to each other.

The average of minimums is for showing the minimum value that is inferred by a user for each group.

| Group | Average of minimum similarities | Average of similarities |
|---|---|---|
| Friends | 0.2913 | 0.4828 |
| inLab | -0.0858 | 0.2520 |
| outLab | 0.0089 | 0.3372 |

**Table 4.** Average of similarity for reported people in survey

Figure 3 shows the percentage of trust values in each semantical classification of trust, for the trust that is inferred for users in different groups of friends and known people inside and outside MIT Lab. Friends are removed from both inLab and outLab groups. As we see, friends have the highest percentage of nodes assessed as highly trusted peers(36%), while inLab peers (e.g. colleagues) are often moderately trusted(46%). For the outLab group, the highly trusted nodes are more than inLab group, but the slightly trusted nodes are increasing. This can be commented by the fact that users meet more random people out of their

working place and at the same time more intimate persons are around them than work.

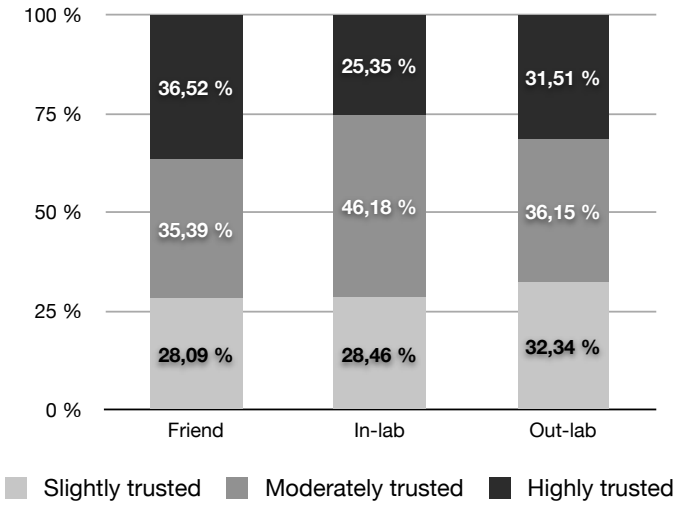



**Fig. 3.** Trust value distribution in different context of proximity

## 5 Conclusion and Future Work

In this paper, we proposed a novel method to bootstrap trust values from proximity between the people which can also be used for trust inference for unknown users. This approach is suitable for mobile social networking applications. We formalized different types of proximity and introduced proximity data types. The evaluation using real proximity data shows that inferred values are correlated with real social facts. For future work, we aim at creation and evaluation of large dataset of different types of proximity. At the same time, user opinions of trust should be surveyed and included in such kind of dataset. We aim at using fuzzy logic for aggregation of different proximity, according to the level of contribution they can provide to trust value. Also, a large body of work exists in the domain of estimation and recommendation, they may be adapted and evaluated for trust recommendation and transitivity within this approach. Hence, for further evaluation of our approach, and in order to better investigate other available possibilities in trust assessment, we are looking into the deployment of this approach as part of the yarta middle ware framework[2]. Yarta middle ware support the development of mobile social applications.

[2] https://gforge.inria.fr/projects/yarta/

## References


1. S. Ahamed, E. Hoque, F. Rahman, and M. Zulkernine. Towards Secure Trust Bootstrapping in Pervasive Computing Environment. In *11th IEEE High Assurance Systems Engineering Symposium, 2008. HASE 2008*, pages 89–96, 2008.
2. S. Ahamed, M. Monjur, and M. Islam. CCTB: Context correlation for trust bootstrapping in pervasive environment. In *2008 IET 4th International Conference on Intelligent Environments*, pages 1–8, 2008.
3. A. Aron. *Statistics for the behavioral and social sciences*. Prentice Hall, 1997.
4. J. Bruneel, A. Spithoven, and A. Maesen. Building Trust: A Matter of Proximity? *Babson College Entrepreneurship Research Conference (BCERC) 2007*.
5. L. Capra. Engineering human trust in mobile system collaborations. *SIGSOFT Softw. Eng. Notes*, 29(6):107–116, November 2004.
6. N. Eagle, A. S. Pentland, and D. Lazer. Inferring friendship network structure by using mobile phone data. *Proceedings of the National Academy of Sciences*, 106(36):15274–15278, September 2009.
7. F. Fukuyama. *Trust : the social virtues and the creation of prosperity*. Free Press, 1995.
8. J. Golbeck and J. Hendler. Filmtrust: movie recommendations using trust in web-based social networks. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 1, pages 282–286, 2006.
9. M. Haque and S. Ahamed. An Omnipresent Formal Trust Model (FTM) for Pervasive Computing Environment. In *Proceedings of the 31st Annual International Computer Software and Applications Conference-Volume 01*, pages 49–56. IEEE Computer Society, 2007.
10. A. Jøsang and S. Pope. Semantic constraints for trust transitivity. In *APCCM '05: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, pages 59–68, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
11. R. Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*, pages 1137–1145, 1995.
12. R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 611–617, New York, NY, USA, 2006. ACM.
13. S. P. Marsh. Formalising trust as a computational concept, 1994.
14. M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. In *Proceedings of the National Academy of Science, USA*, volume 99, pages 2566–2572, 2002.
15. F. Perich, J. Undercoffer, L. Kagal, A. Joshi, T. Finin, and Y. Yesha. In reputation we believe: Query processing in mobile ad-hoc networks. *Mobile and Ubiquitous Systems, Annual International Conference on*, 0:326–334, 2004.
16. D. Quercia, S. Hailes, and L. Capra. TRULLO-local trust bootstrapping for ubiquitous devices. *Proc. of IEEE Mobiquitous*, 2007.
17. R. Saadi, J. Pierson, and L. Brunie. T2D: A Peer to Peer trust management system based on Disposition to Trust. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1472–1478. ACM, 2010.
18. J. Sabater and C. Sierra. Review on Computational Trust and Reputation Models. *Artificial Intelligence Review*, 24(1):33–60–60, September 2005.
19. F. E. Walter, S. Battiston, and F. Schweitzer. A model of a trust-based recommendation system on a social network. *Autonomous Agents and Multi-Agent Systems*, 16:57–74, February 2008.

# Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement

Gabriela Gheorghe[1], Bruno Crispo[1], Roberto Carbone[2],

Lieven Desmet[3], and Wouter Joosen[3]

[1] DISI, Università degli Studi di Trento, Italy, First.Last@disi.unitn.it
[2] Security and Trust Unit, FBK, Trento, Italy, carbone@fbk.eu
[3] IBBT-Distrinet, K.U.Leuven, 3001 Leuven, Belgium, First.Last@cs.kuleuven.be

**Abstract.** For large distributed applications, security and performance are two requirements often difficult to satisfy together. Addressing them separately leads more often to fast systems with security holes, rather than secure systems with poor performance. For instance, caching data needed for security decisions can lead to security violations when the data changes faster than the cache can refresh it. Retrieving such fresh data without caching it impacts performance. In this paper, we analyze a subproblem: how to dynamically configure a distributed authorization system when both security and performance requirements change. We examine data caching, retrieval and correlation, and propose a runtime management tool that, with external input, finds and enacts the customizations that satisfy both security and performance needs. Preliminary results show it takes around two seconds to find customization solutions in a setting with over one thousand authorization components.

**Key words:** configuration, policy, enforcement, middleware, cache

## 1 Introduction

Systems like Facebook, with hundreds of millions of users that add 100 million new photos every day, have data centres of at least thousands of servers [17, 20]. Similarly, eBay uses over 10000 Java application servers [23]. Such large applications need an infrastructure that can adapt to its usage constraints, since performance is a must for business. Performance usually refers to the number of user requests (e.g., clicked links on a web page) serviced per time unit. For Facebook, for instance, it has been stated that if website latency is reduced by 600ms, the user click-rate improves by more than 5%[24]. To achieve performance, Twitter, Google and eBay are using caching (page caching for user pages, local caching for scripts), replication, and data partitioning.

Application providers want security and privacy requirements to be satisfied. Application policies focus on user authentication and authorization, and data privacy. The problem of satisfying all these needs, i.e., *policy enforcement*, becomes complex in a distributed system with multiple constraints and multiple

users. When user data, profiles and action histories are spread across several domains, it is difficult to make this data available safely and consistently across the system. Maintaining this data is essential for correct security decisions, but is complicated by different domains introducing specific data access patterns.

Caching can impair policy enforcement. Commonly, user certificates, authorization decisions or user sessions are cached so that they are quickly retrieved. But when this data changes faster than it is cached, the decision of authenticating or authorizing may no longer be correct, in which case revenue or human lives can be at stake. For instance, on eBay when a buyer wants to bid on an item that is advertised by a seller, there can be a policy on the seller's side that would reject receiving bids when the buyer has a history of bad payment. If at one moment in time the customer's history is clean, by the time the bid request gets to the seller's side it will be accepted since it relies on a cached value of the payment history that is clean; this decision might not be true, in fact, if there is one bad payment reported before the cache is refreshed.

Attribute retrieval can also affect policy enforcement. Attributes refer to all data needed for authorization decisions (e.g. user profile, history, system state, user state). Retrieval can be done directly or using a mediator, in which case it runs the risk of staleness. Also, if attributes are semantically related, their retrieval should be the same way (we call it *correlation*). Attributes are not always retrieved directly, for two reasons: (1) it is costly to retrieve them every time, and (2) the data is private. For instance, eBay has multiple security domains (i.e., groups of providers with similar policies); to authorize a buyer operation in domain A, the eBay system needs to check the buyer history of bad payments with any seller for the current month, and buyer history is maintained in domain B. If domain B gives the freshest buyer history to domain A just for computing the bad payment events, all non-payment data would be disclosed.

When security enforcement and performance collide, such vulnerabilities occur in different systems tuned for performance. The Common Vulnerabilities and Exposures database [21] reveals numerous entries related to cache and configuration management. For instance, OpenSSL suffers from race conditions when caching is enabled or when access rights for cache modification are wrong (CVE-2010-4180, CVE-2010-3864, CVE-2008-7270). Similar problems of access restrictions to cached content are with IBM's DB2 or IBM's FileNet Content Manager (CVE-2010-3475, CVE-2009-1953), as well as ISC's BIND (CVE-2010-0290, CVE-2010-0218). Exploiting such issues is reported to lead to buffer overflows, downgrade to unwanted ciphers, cache poisoning, data leakages, or even bypassing authentication or authorization. Protocol implementations like OpenSSL and BIND, or servers like IBM's and RSA Access Manager, are examples of technologies that need to scale fast, but cannot scale *fast and safely*.

We argue that the techniques to improve a distributed application's performance must be tailored to the application's security needs. In our view, caching, retrieval and correlation of enforcement attributes require a management layer that intersects both the application and its deployment environment. To our knowledge, these aspects have not yet been approached in security enforcement.

To bridge this gap, we suggest a method and framework for adjusting at runtime the security subsystem *configurations*, i.e., the ways to connect components and tune connection parameters. This adjustment requires to find ways to connect security components so that their connections satisfy a set of constraints, specified by domain experts or administrators (e.g., eBay security architects or security domain administrators) and included as annotations in the XACML security policy. The system configuration that allows for both security and performance needs is *dynamic*. Constraints can change because tolerance to performance overheads or inaccurate enforcement decisions can vary; security domains can vary in dimension; network topology can change. Since varying runtime constraints imply re-evaluation rounds, we want to automate the reconfiguration of the authorization subsystem. Thus, our contributions are:

1. We show the need to consider security and performance restrictions together, rather than separately. We focus on attribute retrieval, caching, and correlation. To our knowledge, we are the first to look at caching from the perspective of the impact of stale attributes over the authorization verdict in the policy enforcement process.
2. We present a method to dynamically compute the correct configurations of policy enforcement services, by transforming system constraints into a logic formula solved with a constraint solver.
3. We suggest the first middleware tool to perform adaptive system reconfiguration on security constraints. Having split computation in two phases, preliminary results show that the heavier computation phase takes 2.5 seconds to compute a solution for over 1000 authorization components.

The paper is structured as follows. After an overview of the standard enforcement model (Section 2.1) and an illustrative example (Section 2.2), Section 3 overviews properties of enforcement attributes. Section 4 describes our approach and proposed architecture, while Sections 5 and 6 describe our prototype in more details. Related work is presented in Section 7 and Section 8 concludes.

## 2 Background and example

To be able to evaluate the configuration of the authorization system and how it can be adjusted, this section first describes the reference model in policy enforcement, and then presents an example that we find illustrative for our case.

### 2.1 The Reference Enforcement Model

The eXtensible Access Control Markup Language (XACML)[4] is the de facto reference in authorization languages and architectures. XACML has been widely adopted in industry, examples ranging from IBM's Tivoli Security Manager [13], to Oracle, JBoss and Axiomatics [3]. XACML has been an OASIS standard

[4] `http://www.oasis-open.org/committees/xacml/`

since 2003, and apart from the XML-based language specification, it proposes an authorization processing model based on the terminology of IETF's RFC3198[5]. Fig. 1 shows the three main components (greyed out) that we consider:

**Policy Decision Point** (PDP) is the entity that makes the authorization decision, based on the evaluation of the applicable policies;
**Policy Enforcement Point** (PEP) performs two functions: makes decision requests, and enforces (implements) the authorization decision of the PDP;
**Policy Information Point** (PIP) is the entity that acts like a source of attribute values to the PDP that makes the decisions.

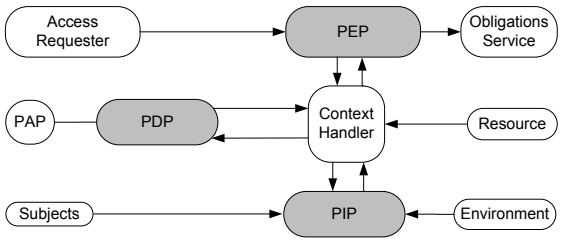



**Fig. 1.** The XACML reference architecture.

There are two variations of the XACML model, shown in Fig.3: the *attribute push model*, where the PEP collects the attributes offered by the PIPs and sends them to the PDP, and the *attribute pull model*, when the PDP collects all relevant attributes in order to make a decision. The pull model has a minimal load on the client and PEP and is designed for cases when the authorization process is entirely performed on the PDP; the push model, conversely, allows for more flexibility to the client and the PEP needs to ensure all required data reaches the PDP. In practice (e.g., PERMIS [5]), a *hybrid model* is used, whereby some attributes are pushed and some are pulled. To our knowledge, no distinction has been made as to when to pull and when to push an attribute.

There have been efforts to augment the XACML reference architecture: Higgins[6] proposes a new authorization architecture for managing online identity. A similar identity management variation was proposed by Djordjevic et al. [7], suggesting a common governance layer that incorporates the PEP but also manages interactions with an identity broker. Overall, such meta-models still keep the original XACML architecture. Thus, using the XACML reference as our model suffices to support a claim of general applicability.

### 2.2 Illustrative example

A *security subsystem* is the ensemble of components that perform authentication or authorization enforcement – a set of PIPs, PEPs, and PDPs. We follow

[5] `http://tools.ietf.org/html/rfc3198`
[6] `http://www.eclipse.org/higgins/`

some architectural points from eBay [16], whereby the security subsystem is divided into security domains (i.e., groups of components under similar security policies). User credentials, profile and state are spread across different domains; access patterns to such data can vary, and domains need to authenticate to other domains when user data is retrieved. We target policies whose evaluation require

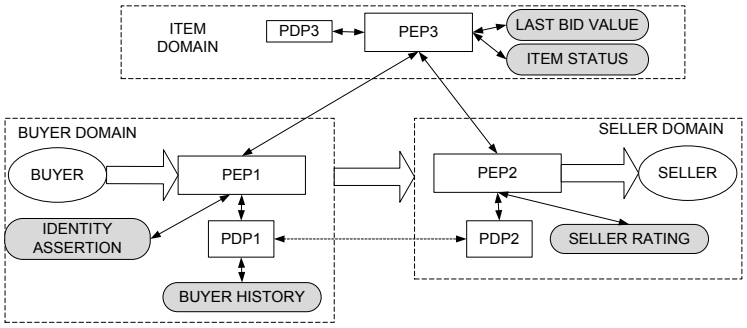



**Fig. 2.** Cross-domain authorization for a Buyer in BuyerDomain to bid in SellerDomain, with item information from ItemDomain. The greyed boxes are the bid authorization attributes: identity assertion, history, seller rating, item status and bid value.

scattered security data, e.g. the policy (**P1**)"*an authenticated buyer is allowed to make a bid on available items from a seller only if the buyer's history shows no delayed payment for the last month*". The buyer history for the current month, seller's available items on sale, identity tokens, and sellers' ratings – this data is kept at various locations where it may change at runtime, with an impact over allowing or disallowing further user actions (e.g., bids). If buyer attributes are not updated correctly or if fresh values are not retrieved in time, buyers will be blocked from certain sellers, or buyers will buy items that they should not normally access. Fig. 2 shows a multi-domain system where in order for a buyer in domain Buyer Domain to be allowed to bid to a Seller in the Seller Domain, its request must pass through the security subsystem of Buyer Domain, gather some data from Item Domain, and pass through the security subsystem of Seller Domain, which in its turn can also require item data from Item Domain.

For us, the problem of attribute management as shown above can be solved by finding a tradeoff between security correctness and performance requirements. This observation is confirmed by Randy Shoup, distinguished architect at eBay, who admits that an efficient caching system aims to "maximise cache hit ratio within storage constraints, requirements for availability, and tolerance for staleness. It turns out that this balance can be surprisingly difficult to strike. Once struck, our experience has shown that it is also quite likely to change over time" [22]. In this paper, we propose a framework to compute and recompute such balance at runtime and adapt the security subsystem accordingly: we do not want to change the entities of the security subsystem, but the connections between them that involve retrieval and caching of security attributes.

## 3 Attribute Configuration

Attributes are application-level assets, and have value in the enforcement process (that is aware of application semantics). In eBay, buyer history, item status and seller rating are attributes; in Shibboleth [14], the LDAP class *eduPerson* defines name, surname, affiliation, principal name, targetId as usual identity attributes.

How to obtain attributes is specified by *configuration (meta)data*. For instance in Shibboleth, identity and service providers publish information about themselves: descriptors of provision, authentication authority and attribute authority, etc. Such metadata influences the enforcement process, and can cover:

- visibility (e.g., is the assertion server reachable? is the last bid value public?),
- location or origin restrictions (e.g., for EU buyers use EU server data),
- access pattern of security subsystem to contact third-party (e.g., if the item data can be encrypted, can be backed up or logged),
- connection parameters (e.g., how often should buyer history data be cached or refreshed? who should retrieve it?).

Our point is that this configuration level, including attribute metadata, complements the enforcement process and can influence it. We examine three aspects: (1) push/pull models for attribute retrieval, (2) caching of attributes, and (3) attributes to be handled in the same way (coined 'correlation').

### 3.1 Attribute Retrieval

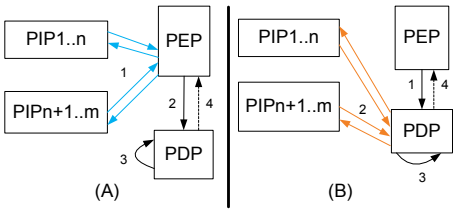
In our eBay example in Fig. 2, *attribute push* from PEP1 to PDP2 means that some attributes – 'identity assertion' , 'buyer history' and item information from Item Domain – are pushed to the Seller Domain and then to PDP2, which will make the final access decision. The 'buyer history', 'last bid value' and 'item status' can also be *pulled* by PDP2 at the moment when it needs such data.

From the point of view of performance, both attribute push and attribute pull can be problematic. In the attribute pull case, an excessive load on PDP2 can make it less responsive for other buyer requests. Performance is linked with security: low PDP2 performance can be made into a Denial of Service attack by saturating PDP2 with requests that require intensive background work. The case of all attributes being retrieved by PEP1 and given to PDP2 is also delicate: for example, if PEP1 knows that PDP2 requires the number of delayed payments of the buyer, it means that PEP1 will do the computation instead of PDP2; this scheme can put too much load on PEP1, that can have a negative effect on PEP1's fast interception of further events.

From the point of view of trust and intrusiveness, the push scheme is problematic. In Fig. 2, PDP2 must trust what PEP1 computed as the highest customer rating this month, and this potentially sensitive data would need to travel between PEP1 and PDP2. This decision bears several risks, of which: PEP1 might compute the wrong rating value; the decision might be delayed; if the rule changes from "any delayed payments", to "under five", then PEP1's logic should

be updated. The security subsystem should decide if exporting the computation of the delayed payment count to PEP1 costs more than the privacy of such local data to Buyer Domain. In either case, PEP1 and the communication channel should be trusted not to tamper with the data, and the policy should be fixed.

To determine when to use the push or the pull scheme, we have analysed a number of possible policy enforcement configurations and several existing scenarios and their solutions – PERMIS [4, 5], Shibboleth [14], and Axiomatics [3]. In particular, PERMIS looks at role-based authorization in multiple Grid domains. There are two kinds of regulations on subject credentials in PERMIS: policies on what credentials can be attached to the authorization request leaving from Buyer Domain (policies enforced by PDP1 in Fig. 2) and can reach Seller Domain; and policies on what credentials can be trusted by Seller Domain as the destination of an authorization request from Buyer Domain. Validating a buyer credential in Seller Domain – done by PDP2 – involves a chain of trust among issuers that have different rights to issue attributes for different sets of subjects. On a similar note, Shibboleth [14] enforces attribute release policies whereby once a user has successfully authenticated to a remote website, certain user attributes can be provided on demand via back-channels by requesting Web sites while other attributes cannot. These examples confirm that user or system attributes are sensitive, and should not always be pushed to the PDP.




| Push scheme | Pull scheme |
|---|---|
| stateless attributes | state attributes |
| environment attributes | behaviour history |
| not sensitive user data | sensitive user data |
| request details | obligations |
| destination details | domain decisions |
| stateless context | stateful context |

**Fig. 3.** Diagrams A and B show the classic attribute push and pull models. The table to the right shows how different attributes fit the push or pull scheme better.

Based on the security considerations above, we separate the applicability of the push and pull schemes based on types of attributes (see Fig. 3, right). The push scheme is more appropriate for those attributes that can be safely collected by the PEP or can only be retrieved by the PEP rather than the PDP (e.g., message annotations whose semantics is known by the PEP: IP address of a buyer and payment server, description of a token service, location and country of a certificate server). A similar treatment may be for attributes unlikely to change frequently: user identity, group membership, security clearances or roles, authentication tokens, or any constant data about system resources or the environment that the PEP can retrieve easily. Conversely, the PDP directly interacts with PIPs when it needs specific application logic and related state (e.g.,

customer rating, payment obligations), history of a user (e.g., bidding history of a user), or data that only the PDP is allowed to access (e.g., highest bid).

### 3.2 Attribute Caching

Caching in policy enforcement can be of three types: of (1) attributes, (2) decision, and (3) policy [25]. The PDP may cache policies that change little in terms of content, but caching attributes with different change rates is difficult. For instance, bid history, while expected to change a lot, can be constant if the user does not bid; conversely, feedback rating, while expected to change slowly, can change fast for a seller who does not ship items to buyers over a period of time. For similar reasons, the PDP or the PEP might maintain a cache of the decisions already made. Unlike attributes, that tend to change values the most frequently, decisions and policies are more static. We hereafter focus on attribute caching, but our framework can apply to decision and policy caches too.

**Table 1.** Different caching concerns in enforcement.

| *Scheme* | **What to cache** | **Where to cache** | **How to invalidate** |
|---|---|---|---|
| Attribute caching | attributes | PEP or PDP | time limit |
| Decision caching | policy decisions | PDP | explicit |
| Policy caching | entire policy | PEP | explicit |

Since cached attributes are attributes too, the push and pull scheme applies to caches as well as to attribute retrieving. New values of the attributes needed to make a policy decision can be either (1) pushed to the entity that needs them, be it the PEP or the PDP, or (2) pulled by the same entity that needs them. There are two cases that combine both caching and attribute retrieving issues:

1. *push to PEP cache, push to PDP*: a PEP pushes attributes to the PDP, and whenever an attribute is updated, the PEP cache is notified and updated. The scheme relies on an external entity to notify the PEP of attribute changes. Inaccurate notifications can compromise decision correctness.
2. *pull to PEP cache, push to PDP*: a PEP that pushes attributes to the PDP, and periodically queries attributes for fresh values (pull). This case does not use a third-party but puts more load on the PEP. Also, the PEP should have poll times that depend on the change rate of the attributes to be cached.

The cases when the PDP pulls attributes by itself and stores them are similar in that cached values need to be refreshed at a rate decided either by a (trusted) third-party or at the rate at which the PDP can query the data sources. From this last point of view – the polling rate of the PDP – cache management has the notion of cache invalidation policy, whereby the cached values have a validity time; when they become invalid, the manager will retrieve fresh copies. Table 1 shows how to invalidate caches depending on what security aspect is cached.

### 3.3 Attribute Correlation

In an interview with Gunnar Peterson (IEEE Security & Privacy Journal on Building Security), Gerry Gebel (president of Axiomatics Americas) pointed out that there can be different freshness values for related attributes [10]. The example given is that of a user with multiple roles in an organisation; whenever the user requests access to a system resource, the PDP needs to retrieve the current role of the user; some of the user's roles may have been cached (in the PDP, PEP or PIP) hence there might be different freshness values to be maintained for several role attributes. In our eBay scenario in Fig. 2, it can be that more than one item attribute changes from the moment a buyer reads it, and before the last bid. In order for the security subsystem to allow the buyer to bid, it must gather buyer information, item restrictions, *and* item information. If at this stage, the system can access the freshest bid value, but not the freshest item status flag, then the buyer could still be allowed to bid on a sold item.

We generalise such examples to the idea that correlated attribute need a common refresh rate for all the attributes in the group (e.g., all role attributes from an LDAP server, username and password attributes, source IP and port attributes, role and mutually exclusive role list, etc). Some attribute correlations are application-dependent (e.g., mutually exclusive roles, like bidder and item provider for the same item), others are application-independent (e.g., username and password for single sign-on). Hence, bundling attributes that should be used together is a must when enforcement decisions need to be accurate. With PEPs or PDPs likely to use overlapping attribute sets to enforce a cross-domain policy, synchronization over common attributes is essential for attribute consistency.

Having a control over the attributes used in policy enforcement implies the need for a management layer that we call 'configuration layer'. We continue by describing our solution to the issues above.

## 4 Approach and Proposed Architecture

We consider the setting of a distributed application on top of which there is a security subsystem in charge of enacting several security policies. This subsystem consists of multiple PEPs, PIPs and PDPs. In the state of the art so far, the connections between these components are fixed once a policy is deployed. In our view, they should change at runtime, since this way the system can *adapt* to varying security or performance constraints. How often these changes are incorporated into the security subsystem depends on which of security or performance is paramount, and on the disruption incurred when actually changing the connections among security components.

We will use the term *wiring* for enabling a configuration on the concrete infrastructure (realizing the connections among PIPs, PEPs and PDPs in order to support the attribute management features explained above). We want to rewire the different authorization components with hints from the application domain and knowledge of the enforced policies. We see such hints supplied with the

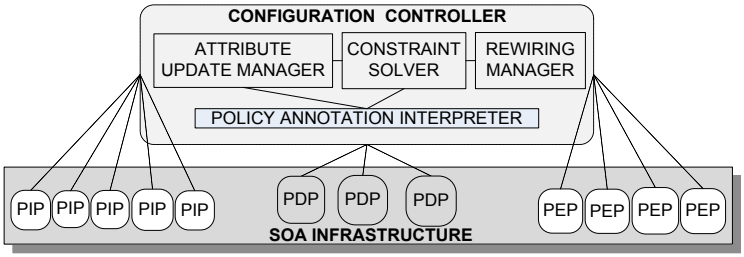



**Fig. 4.** The architecture of our solution.

XACML policy, since it is likely that the policy writer, aware of some application domain details, can formulate requirements for the treatment of attributes. We also assume that security components have standard features: a cache storage space (hence also cache size and cache refresh policies), permissions over enforcement attributes, or other properties (domain, location, etc).

The architecture of our solution is presented in Fig. 4. Instead of having a set of hardwired connections between the PEPs, PIPs and PDP, we suggest a configuration layer on top of the SOA infrastructure. This layer enables the dynamic rewiring according to varying runtime conditions of the application. Responsible of this task is a Configuration Controller (CC) consisting of four components: a Policy Annotation Interpreter (PAI), an Attribute Update Manager (AUM), a Constraint Solver (CS) and a Rewiring Manager (RM).

The **Policy Annotation Interpreter (PAI)** extracts the policy annotations relevant for the configuration of the security subsystem.

The **Attribute Update Manager (AUM)** monitors the value changes of security attributes and notifies or directly propagates the new values to PEPs and PDPs or to their respective caches. This component also manages attribute synchronization and consistency. It should be connected to all PIPs needed to enforce a policy. The purpose of the AUM is to propagate attribute changes to the security subsystem. The environment, user properties or security relevant state might change at runtime, so the security subsystem needs to be notified.

The **Constraint Solver (CS)** is the component that finds solutions to satisfy the connection constraints of PEPs, PIPs and PDPs. Such constraints cover attribute retrieval, caching and correlation, and are specified in the security policy. The CS processes the policy, searches the solution space, and selects the configurations of the security subsystem that do not violate the constraints.

The **Rewiring Manager (RM)** enacts the solutions found by the CS. The RM resides at the middleware layer and is dependent on the communication infrastructure (in this case, a SOA infrastructure). How this infrastructure is rewired is not within the scope of this paper, and was addressed before [12].

With this approach in mind, we continue by examining types of runtime constraints, how they can be specified, and how the CS can handle them. There are several assumptions for our running system: first, all PIPs are considered to provide fresh information to the other security components. Then, PEPs can cache PIP data, and PDPs can cache enforcement decisions and policies.

### 4.1 Annotating XACML policies

For the eBay authorization policy that requires attributes 'user token' and 'user feedback rating', our approach is qualitative: if there are multiple providers of user tokens, or if the rating attribute has to be fresh, our aim is to ensure such quality considerations are respected. This additional data (where an attribute can be retrieved from, if it can be stale or pushable) belongs in the authorization policy. The reason is that the policy writer usually has the correct idea over attribute usage and invalidation with respect to the correctness of the policy decision; the policy writer knows if the user token does not change a lot so that it can be pushed to the PDP, or if the buyer/seller feedback rating is critical and volatile so should not be cached. The XACML 3.0 syntax does not natively

```
<xs:element name="AttrProps" type="att-xacml:AttrPropsType"/>
<xs:complexType name="AttrPropsType">
   <xs:attribute name="AttrId" type="xs:anyURI" use="required">
   <xs:sequence>
     <xs:element ref="att-xacml:Providers" minOccurs="1" maxOccurs="unbounded">
     <xs:element name="AttrNotCached" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrCacheable" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrPushable" minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="att-xacml:CorrelAttr" minOccurs="0" maxOccurs="unbounded">
   </xs:sequence>
</xs:complexType>
```

**Fig. 5.** Attribute meta-data elements in the `att-xacml` schema.

support attributes about subject, environment or resource attributes. We suggest to enrich the default XACML syntax with annotations specific to the following aspects: (1) what PIP/PEP provides an attribute, (2,3) if an attribute can be cached and where, (4) if an attribute is pushable or pullable and where, (5) the correlation among different attributes. Until a XACML profile bridges this gap, a solution to this problem is to specify attribute metadata as an element in an enhanced schema we call `att-xacml` and part of which is shown in Fig. 5. The `AttrPropsType` type consists of the properties of attributes that we are interested in: `attribute-id`, and a series of elements to indicate on what PEPs or PDPs they can be pushed, cached or not cached, what PIPs provide them, and correlated attributes. We assume that these features, attached to each attribute in a policy, are interpreted by an extension of the XACML engine. The PAI processes the semantics of these attributes for the CS.

### 4.2 Satisfying configuration constraints

The restrictions on attributes that were specified in a XACML-friendly syntax in Section 4.1 will reach the Constraint Solver (CS). The CS has to match these constraints against its runtime view over the authorization subsystem, which we

call *runtime topology*. The result is a number of solutions that satisfy all constraints; we call these solutions *configuration solutions*, or *configurations*. If the CS finds no solution, then the constraints are not satisfiable and the security subsystem remains unchanged. The CS is shown in Fig. 6. Satisfying configura-

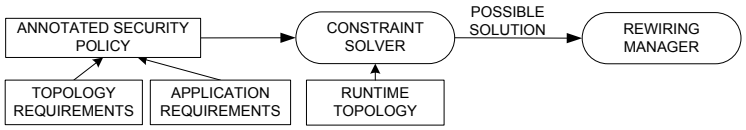



**Fig. 6.** Model for configuration and rewiring in our approach.

tion constraints is complicated by the existence of different and interdependent constraint types. It is not in the scope of this paper to analyse the impact of each of these constraints over the resulting configurations. Yet, for now, we acknowledge that reconfiguration depends on how often and what triggers the CS to re-evaluate its previously found configurations and issue new ones. Here we see two possibilities: the re-evaluation can be per policy, or per policy subpart. In the first case, the trigger of the reconfiguration is a new policy version, and the changes required for the new configuration might be scarce and far apart in the topology. In the second case, the trigger of a reconfiguration is a change in the runtime topology that relates to an attribute provider or consumer referred to by a security policy; here, the reconfiguration changes are likely to be closer together in the topology. It is hard to say which one happens more often, since this is application dependent. A tradeoff must be made between a system that reconfigures itself either too frequently, or never.

### 4.3 (Re)Wiring

Rewiring of PEPs, PDPs, and PIPs refers to changing the connections among these components. These connections are established at the SOA *middleware* level, that enables component intercommunication. In our previous work [11], we envisioned the Enterprise Service Bus (ESB) as the manager of the security subsystem. The main reason is that, by design, the ESB controls the deployed security components and their connections; when runtime conditions change (e.g., components appear or disappear, security policies are updated), the ESB can modify message routing on the fly (e.g., validate credentials before they reach an authorization server), trigger attribute queries (e.g., check the feedback rating attribute every minute and not every hour) or change attribute propagation to security domains (e.g., what entities are entitled to receive bid product updates). In particular, the dynamic authorization middleware in [12] applies the dynamic dispatch system of an ESB to enable run-time rewiring of authorization components across services. Each authorization component (PEP, PDP, PIP and PAP) is enriched with an authorization composition contract, and a single administration point allows the wiring and rewiring of authorization components. Using lifecycle and dependency management, the architecture guarantees that

authorization wirings are consistent (i.e., all required and provided contract interfaces match), and that they remain consistent as the rewiring happens. Since rewiring was addressed before, here we focus on the constraint solving problem.

## 5 Configuration Prototype

From the components shown in Fig. 6, we concentrated on the constraint solver component. While aspects about the RM and the AUM have been approached in previous work, runtime constraint solving is the most challenging aspect of our solution. We assume the CS receives runtime data about the components of the security subsystem, and attribute constraints from the deployed policies (Section 3), and produces configuration solutions that satisfy the constraints. We have prototyped the constraint solving in Java with the SAT4J[7] SAT solver.

### 5.1 Constraint Solving with a SAT solver

We used a SAT solver because we wanted to obtain configuration solutions for a wide range and number of constraints (Section 5.2 explains why it is difficult to do this without a SAT solver). Given a propositional formula in conjunctive normal form (CNF) describing both the problem and its constraints, a SAT solver is designed to find an assignment of the propositional variables that satisfy the formula. The solver can also address partial maximum (PMAX) SAT problems: given a set of clauses S (a 'maximization set'), find assignments that maximise the number of satisfied clauses in S. This approach naturally maps to the configuration problem that administrators are faced with. For a security-aware application, satisfying all security constraints is paramount, while performance constraints should be maximized if possible. The reciprocal is treated similarly.

Fig. 7 shows two ways to use a SAT solver in our problem: in the first case, an encoded input is fed to the solver to find a valid configuration. If some part of the input changes at runtime, the whole input will be encoded again, and the solver will recompute all solutions from scratch. This method requires a massive effort to re-encode the entire problem, even if only a small part of it has changed. We employed an alternative incremental approach. The SAT solver receives an initial set of clauses, and if a subset of these clauses change at runtime, its internal mechanisms recompute only the subpart of the problem that has changed. A maximisation set can be provided in both cases. Next, we describe the encoding, offline and runtime input in our approach.

*Offline Input.* The administrator sets up the general settings. The set $PIPs$ of all possible PIPs, the set $PEPs$ of PEPs, and the set $PDPs$ of PDPs, say which attributes can be provided by each component. Specifically, for each $pip \in PIPs$, $pep \in PEPs$, and $pdp \in PDPs$, the following data is provided:

– `provide`$(pip, a)$ iff PIP $pip$ can provide attribute $a$;

[7] `http://www.sat4j.org/`

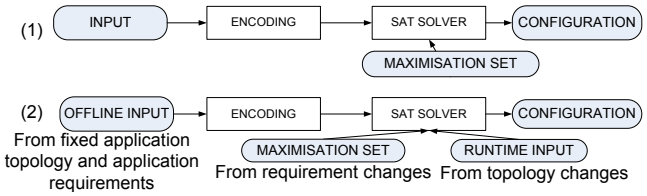



**Fig. 7.** Two SAT solver methods to solve constraints (inputs and outputs shown darker)

- `provide`$(pep, a)$ iff PEP $pep$ can provide attribute $a$;
- `needs`$(pdp, a)$ iff PDP $pdp$ needs attribute $a$;
- `pull`$(a, pdp)$ iff PDP $pdp$ needs attribute $a$ directly from some PIP;
- `correlation`$(pdp)$ iff PDP $pdp$ either pulls or pushes all attributes;
- `not_cacheable`$(a, pdp)$ iff PDP $pdp$ needs freshness for attribute $a$;
- `cached`$(a, pep)$ iff PEP $pep$ can tolerate a stale attribute $a$.

These predicates become propositional variables for the solver, along with:

- `arch`$(pdp, pip, a)$ means that $a$ is pulled directly by $pdp$ from $pip$.
- `arch`$(pdp, pep, a)$ means that $a$ is pushed from $pep$ to $pdp$.
- `arch`$(pep, pip, a)$ means that $a$ is pushed from $pip$ to $pep$.

The solver assigns the truth values to the `arch`() variables; these give the active connections of each component, and each set of assignments is a configuration.

We also use the notation $PIPs_a \subset PIPs$ (and $PEPs_a \subset PEPs$) as the set of PIPs (PEPs) such that `provide`$(pip, a)$ (`provide`$(pep, a)$, resp.). Similarly, $PDPs_a \subset PIPs$ is the set of PDPs such that `needs`$(pdp, a)$ is true. In the attribute retrieval model, we consider the following formulae:

$$\texttt{pull}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pip, a) | pip \in PIPs_a\}$$
$$\texttt{push}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pep, a) | pep \in PEPs_a\}$$

saying that an attribute $a$ is pulled by (pushed to) the PDP $pdp$ if and only if there is an arch between $pdp$ and a PIP $pip$ (PEP $pep$, resp.). Thus, if the $pdp$ asks for $a$, and `pull`$(a, pdp)$ is true, then $a$ must be retrieved directly from a $pip$.

We use the notation $\oplus\{v_1, v_2, \dots\}$ meaning that exactly one of $v_1, v_2, \dots$ is true, and $\ominus\{v_1, v_2, \dots\}$ meaning that at most one of $v_1, v_2, \dots$ is true. Encoding the possible paths and constraints means a conjunction of the following formulae:

- For each $a$ required by $pdp$, exactly one connection must exist between $pdp$ and either a PIP providing $a$ or a PEP providing $a$. This is expressed by the following formula, for each `needs`$(pdp, a)$ in the Offline Input:

$$\texttt{needs}(pdp, a) \supset \oplus\big\{\texttt{arch}(pdp, x, a) | x \in PIPs_a \cup PEPs_a\big\} \quad (1)$$

  Moreover, in case there is an arch between $pdp$ and $pep$, providing $a$, then there must be also a connection between $pep$ and a $pip$ providing $a$ (formula (2)). Otherwise (formula (3)) no arch between $pep$ and $pip$ providing $a$

is necessary. For each $\texttt{needs}(pdp, a)$ and $pep \in PEPs_a$:

$$\texttt{arch}(pdp, pep, a) \supset \oplus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \quad (2)$$

$$\neg\,\texttt{arch}(pdp, pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \quad (3)$$

– If a *pip* does not provide $a$ then no connection providing $a$ can exist between *pip* and other components. For each $\texttt{provide}(pip, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\}$$

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pep \in PEPs_a\}$$

Similarly for the PEPs: for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

– To show the advantage of this approach, we consider another constraint: we suppose that each component can provide (e.g. for performance reasons) an attribute only to a finite number of components (in this case, one). To model that, for each $\texttt{provide}(pip, a)$ in the Offline Input, we consider the following:

$$\texttt{provide}(pip, a) \supset \ominus\{(\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\} \quad (4)$$

$$\texttt{provide}(pip, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pep \in PEPs_a\} \quad (5)$$

Formula (4) (same for (5)) says that if *pip* provides $a$ then there must be at most one connection between *pip* and the PDPs (PEPs, resp.) requesting $a$. It is similar for the PEPs, so for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

– Without data showing that PDPs cache attributes, we considered only PEPs to do that. This choice does not limit generality. We had to encode the constraint: if *pdp* asks for $a$, and $\texttt{not\_cacheable}(a, pdp)$ is true, then $a$ cannot be retrieved from a *pep* that cached $a$. So for each $\texttt{not\_cacheable}(a, pdp)$ in the Offline Input, we have the conjunction:

$$\texttt{not\_cacheable}(a, pdp) \supset \bigwedge\{\neg(\texttt{cached}(a, pep) \wedge \texttt{arch}(pdp, pep, a)) | pep \in PEPs_a\}$$

– The attribute correlation constraint can be described as follows: if *pdp* has correlated attributes, then all its attributes must be either pulled or pushed. For each $\texttt{needs}(pdp, a)$ in the Offline Input, this can be modeled as follows:

$$\texttt{correlation}(pdp) \supset (\texttt{pull}(pdp) \vee \texttt{push}(pdp))$$

where: $\texttt{pull}(pdp) \equiv \bigwedge\{\texttt{pull}(a, pdp)\}$, and $\texttt{push}(pdp) \equiv \bigwedge\{\texttt{push}(a, pdp)\}$

Notice that by defining PMAX-SAT problems it is also possible to impose constraints like: if possible, preference should be given to attribute provisioning via indirect paths, i.e., from PIP to PEP to PDP, over direct paths, i.e., from PIP to PDP. This can be obtained by maximising the number of truth values for the variables corresponding to the arches from PEPs to PDPs.

In satisfiability problems, coding the at-most-one operator $\ominus$ is often problematic. If there are $n > 1$ variables, an improved SAT encoding of $\ominus\{v_1, \ldots, v_n\}$ is the logarithmic bitwise encoding in [9]. This operator's CNF encoding requires $\lceil \log_2(n-1) \rceil$ auxiliary variables and $n\lceil \log_2(n-1) \rceil$ clauses. With this encoding, the CNF formula corresponding to (1) has a complexity (in terms of number of clauses) of $O(|PDPs| \cdot \mathrm{N_{a\_pdp}} \cdot (\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}}) \cdot \log_2(\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}})$, where $\mathrm{N_{a\_pdp}}$ is the average number of attributes requested by each PDP, and $\mathrm{N_{pips\_a}}$ ($\mathrm{N_{peps\_a}}$) is the average number of PIPs (PEPs, resp.) providing a specific attribute. These constraints generate a large number of clauses. This is why the encoding is computed offline and fed to the solver "on stand-by mode".

*Runtime Input.* The offline input abstracts a global view of the scenario. The runtime input specifies a current scenario, indicating the security components currently being used, their current set of attributes provided/needed, and any other current constraints. The runtime input is an instantiation of the offline input. For instance, if the Offline Input file contains `provide(pip1, a1)` and `provide(pip2, a2)`, but in the current scenario `pip1` is not available, then the Runtime Input will contain the following assignment: `provide(pip1, a1) = false`.

### 5.2 Constraint Solving without a SAT solver

For independent constraints over components, an algorithm to choose configuration solutions may be faster than the SAT solver. Such an algorithm can perform a graph traversal to select the first PIP or PEP that satisfies the requirements of a PDP and some boolean constraints. Graph traversal works best in scenarios where existing connections have no impact on choosing further connections; yet it may become problematic when the graph is dynamic and local connections change global state, that in turn changes other local connections. This can happen, for example, when bandwidth restrictions limit the number of possible connections of a component; or when a user cannot re-rate a seller until the next time they win the auction. Designing such an algorithm is not easy, since it is tightly bound to the constraints: whenever constraints appear or disappear, the algorithm needs to be rewritten. With a SAT solver, the set of constraints can vary without influencing the process of producing a solution, but performance might suffer. Still, it is worth to design such a generic algorithm as future work.

## 6 Evaluation and Discussion

The performance of our prototype depends on several aspects: the size and semantics of the offline input file, the size and semantics of the runtime input, and

the internal performance of the SAT solver. Since each SAT solver uses a different solving technique, the performance of our prototype depends on the choice of solver. Apart from this dependency, we considered that the most relevant aspect to test is the encoding of the offline input, that need be recomputed several times along the lifetime of an application. This recomputing can be triggered by components that appear or disappear, or by changes in what components provide or require. The runtime input, on the other hand, is only a (moderately small) subset of the maximal offline one, and its impact over the solution finding time depends on the SAT technique used by each individual solver.

Therefore, we wanted to measure the offline CNF generation and the constraint solving time for a number of configuration topologies, while keeping the runtime input minimal. To obtain different offline inputs, there were several parameters to vary: the number of authorization components, the number of attributes and constraints on attributes, the distribution of attributes per authorization component. In choosing which of these parameters to vary and which to keep constant, we considered that the number of attributes is fixed in each scenario, since the administrator should know beforehand the attributes required for all security decisions in a particular application. Hence, we generated over 100 offline input files to describe topologies with varying numbers of PIPs, PDPs, and PEPs, with a fixed number of attributes provided/needed by each. In order to assign attributes to each component, we implemented an algorithm for unbiased random selection as described by Knuth [18, Sec.3.4.2].

To test our Java encoding against the SAT4J solver with the aims described above, we configured the JVM to run with 500 to 1000MB of memory, and we used JRE1.5.0. We ran each of the input files against a minimal runtime configuration file of one constraint, with the following parameter changes: 25, 50, 75 PDPs; 50, 100, 250, 500, 750 PIPs; 25, 50, 75, 100, 150, 200, 250 PEPs. The number of attributes was constant to 100, with 3 attributes per component in all cases. We measured the time (in milliseconds as per the Java call $System.currentTimeMillis()$) for the static CNF generation, as well as the time it takes SAT4J to compute the first solution from the already generated encoding (the solve time). The results for 50 PDPs can be seen in Fig. 8; those for 25 and 75 PDPs are similar. The figures show a linear increase in the offline CNF generation in the number of authorization components (Fig. 8, right) with a similar linear increase in solve time (Fig. 8, left).

These preliminary results are very encouraging in that the time taken to compute the constraints from the offline input is very short for a moderate application size (e.g., about 2.5 seconds for 750 PIPs, 250 PEPs and 50 or 75 PDPs), and also that they grow linearly with the number of components, for a fixed number of attributes. Even though satisfiability is generally known to be an NP-complete problem (and on a general case, it can be expected to obtain an exponential growth in problem complexity and hence performance), the linear relation that we have obtained is justified in that we are using a generic tool to solve a particular instance of a satisfiability problem.

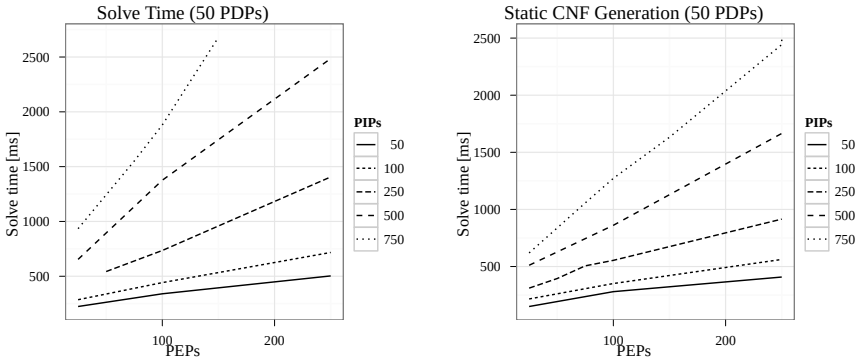



**Fig. 8.** The time to solve our constraint problem with 50PDPs, and varying numbers of PEPs and PIPs (left) and the time to generate the offline CNF clauses (right)

## 7 Related Work

Some policy configuration features come with commercial products: Tivoli Access Manager [13], Axiomatics Policy Server [3], Layer 7 Policy Manager [19], Shibboleth [14] and Globus Toolkit 4 [1]. Tivoli provides central management and enforcement of data security policies across multiple PEPs, with partial policy replication and decision caching. Axiomatics offers central administration of policies, and can manage multiple PDP instances. Shibboleth focuses on federated identity management, allows for session caching and attribute backup, but does not consider attribute freshness. Globus considers resource and policy replication. All these products address different policy management aspects differently, and never consider together cache consistency, synchronisation and security guarantees. The users of these products cannot change such features from a single console, and hence cannot understand how one impacts another.

Two previous works are particularly relevant to our approach. First, Colombo et al. [6] observe that attribute updates may invalidate PDP's policy decisions. The work only focuses on XACML's expressive limitations, does not consider system reconfigurations and is not supported by any implementation. Second, the work of Goovaerts et al. [12] focuses on matching of provisions in the authorization infrastructure. The aim is to make the system that enforces a policy seamless and scalable when those components that provide security attributes change or disappear. They do not discuss how attribute provisioning impacts the correctness of the enforcement process.

Several other works are related to ours. Policy management is also tackled in Ponder [8] but aspects such as caching consistency and synchronisation are not considered. PERMIS [5] proposes an enforcement model for the Grid that uses centralised context data stores and PDP hierarchies with visibility restrictions over attributes provided across domains. Like us, Ioannidis [15] separates policy enforcement from its management, suggests a multi-layer architecture to enforce locally some part of a global policy, but does not discuss consistency of policy

data at different locations and propagation of updates. The thesis of Wei [25] is the first to look at PDP latency in large distributed applications for role based access control. He does authorization recycling by caching previous decisions, but does not consider cache staleness. Atluri and Gal [2] offer a formal framework for access control based on changeable data, but the main difference is that their work is on changing the authorization process rather than configuring security components to manage such data appropriately.

## 8 Conclusions and future work

Our work is the first to consider the impact of properties about authorization components and their connections, over the correctness of the enforcement process. We believe that performance tuning uncovers security flaws in distributed applications and we concentrate on security attribute management. Our management solution can help administrators in two ways: it can generate system configurations where a set of security constraints need always be satisfied (along with maximising performance constraints), or can check an existing configuration of the system against a given set of (security or performance) constraints. The configuration solutions can be recomputed at runtime and preliminary results show an overhead of a few seconds for a system with one thousand components. To our knowledge, this is the first tool to perform a fully verified authorization system reconfiguration for a setting whose security constraints would otherwise be impossible to verify manually.

**Acknowledgements.** This work was partially supported by EU-FP7 NESSoS, the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, the Research Fund K.U.Leuven, and by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento within "team 2009-Incoming" FP7-COFUND. We also thank Wouter De Borger and Stephan Neuhaus for technical comments.

## References

1. Globus Alliance: Globus Toolkit 4 API. `http://www.globus.org/toolkit/docs/4.2/4.2.1/security/` (Nov 2010)
2. Atluri, V., Gal, A.: An authorization model for temporal and derived data: securing information portals. ACM Trans. Inf. Syst. Secur. 5, 62–94 (February 2002)
3. Axiomatics: Axiomatics Policy Server 4.0. `http://www.axiomatics.com/products/axiomatics-policy-server.html` (Nov 2010)
4. Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: a modular authorization infrastructure. Concurr. Comput. : Pract. Exper. 20, 1341–1357 (2008)
5. Chadwick, D.W., Su, L., Laborde, R.: Coordinating access control in grid services. Concurrency and Computation: Practice and Experience 20(9), 1071–1094 (2008)

6. Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A Proposal on Enhancing XACML with Continuous Usage Control Features. In: Grids, P2P and Services Computing. pp. 133–146. Springer US (2010)
7. Djordjevic, I., Dimitrakos, T.: A note on the anatomy of federation. BT Technology Journal 23, 89–106 (October 2005)
8. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the ponder language. In: Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on. pp. 529 –543 (2001)
9. Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. J. Autom. Reason. 35, 143–179 (October 2005)
10. Gebel, G., Peterson, G.: Authentication and TOCTOU. http://analyzingidentity.com/2011/03/18/ (2011)
11. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: IFIPTM. vol. 321, pp. 63–78 (2010)
12. Goovaerts, T., Desmet, L., Joosen, W.: Scalable authorization middleware for service oriented architectures. In: ESSoS, Madrid, Spain (Feb 2011)
13. IBM: IBM Tivoli Access Manager. http://www-01.ibm.com/software/tivoli/products/access-mgr-e-bus/ (Nov 2010)
14. Internet2MiddlewareInitiative/MACE: Shibboleth 2. https://wiki.shibboleth.net/confluence/display/SHIB2/Home (2011)
15. Ioannidis, S., Bellovin, S.M., Ioannidis, J., Keromytis, A.D., Anagnostakis, K.G., Smith, J.M.: Virtual private services: Coordinated policy enforcement for distributed applications. I. J. Network Security 4(1), 69–80 (2007)
16. Kassaei, F.: eBay identity assertion framework. http://www.slideshare.net/farhangkassaei/ebay-identity-assertion-framework-iaf (May 2010)
17. Kerner, S.M.: Inside Facebook's Open Source Infrastructure. http://www.developer.com/open/article.php/3894566/ (July 2010)
18. Knuth, D.E.: The art of computer programming, vol. 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
19. Layer 7 Technologies: Policy Manager for XML Gateways. http://www.layer7tech.com/products/policy-manager-for-xml-gatewaysl (Nov 2010)
20. Miller, R.: The Facebook Data Center FAQ. http://www.datacenterknowledge.com/the-facebook-data-center-faq/ (September 2010)
21. Mitre: Common Vulnerabilities and Exposures. http://cve.mitre.org/ (2011)
22. Shoup, R.: Scalability best practices - Lessons from eBay. InfoQ http://www.infoq.com/articles/ebay-scalability-best-practices (May 2008)
23. Shoup, R.: More Best Practices from Large Scale Websites - Lessons from eBay. Talk at QCon San Francisco 2010, http://qconsf.com/sf2010 (November 2010)
24. Wei, D., Jiang, C.: Frontend Performance Engineering in Facebook. O'Reilly Velocity, Web Performance and Operations Conference (June 2009)
25. Wei, Q.: Towards Improving the Availability and Performance of Enterprise Authorization Systems. Ph.D. thesis, University of British Columbia (2009)

# Information-theoretic Bounds for Differentially Private Mechanisms

Gilles Barthe
*IMDEA Software Institute*
*gilles.barthe@imdea.org*

Boris Köpf
*IMDEA Software Institute*
*boris.koepf@imdea.org*

***Abstract*—There are two active and independent lines of research that aim at quantifying the amount of information that is disclosed by computing on confidential data. Each line of research has developed its own notion of confidentiality: on the one hand, differential privacy is the emerging consensus guarantee used for privacy-preserving data analysis. On the other hand, information-theoretic notions of leakage are used for characterizing the confidentiality properties of programs in language-based settings.**

**The purpose of this article is to establish formal connections between both notions of confidentiality, and to compare them in terms of the security guarantees they deliver. We obtain the following results. First, we establish upper bounds for the leakage of every $\epsilon$-differentially private mechanism in terms of $\epsilon$ and the size of the mechanism's input domain. We achieve this by identifying and leveraging a connection to coding theory.**

**Second, we construct a class of $\epsilon$-differentially private channels whose leakage grows with the size of their input domains. Using these channels, we show that there cannot be domain-size-independent bounds for the leakage of all $\epsilon$-differentially private mechanisms. Moreover, we perform an empirical evaluation that shows that the leakage of these channels almost matches our theoretical upper bounds, demonstrating the accuracy of these bounds.**

**Finally, we show that the question of providing optimal upper bounds for the leakage of $\epsilon$-differentially private mechanisms in terms of rational functions of $\epsilon$ is in fact decidable.**

***Keywords*-Quantitative Information-Flow, Differential Privacy, Information Theory.**

## I. Introduction

Confidentiality is a property that captures that no secret information is exposed to unauthorized parties; it is one of the most fundamental security properties and an essential requirement for most security-critical applications.

Unfortunately, perfect confidentiality is often difficult or even impossible to achieve in practice. In some cases, perfect confidentiality is in conflict with the functional requirements of a system. For example, the result of a statistical query on a medical database necessarily reveals some information about the individual entries in the database. In other cases, perfect confidentiality is in conflict with non-functional requirements such as bounds on the resource-usage. For example, variations in the execution time of a program may reveal partial information about the program's input; however a (perfectly secure) implementation with constant execution time may have inacceptable performance.

Because such conflicting requirements are ubiquitous, there is a need for tools that enable formal reasoning about imperfect confidentiality. Quantitative approaches to confidentiality can provide such tools: first, quantitative notions of confidentiality can express a continuum of degrees of security, making them an ideal basis for reasoning about the trade-off between security and conflicting requirements such as utility [11] or performance [21]. Second, despite their flexibility, a number of quantitative notions of confidentiality are backed up by rigorous operational security guarantees such as lower bounds on the effort required for brute-forcing a secret.

While convergence has been achieved for definitions of perfect confidentiality (they are subsumed under the cover term *noninterference* and differ mainly in the underlying system and adversary models) this is not the case for their quantitative counterparts: there is a large number of proposals for quantitative confidentiality properties, and their relationships (e.g. in terms of the assumptions made and the guarantees provided) are often not well-understood.

In particular, there are two active and independent lines of research dealing with quantitative notions of confidentiality. The first line is motivated by the privacy-preserving publishing of data, with *differential privacy* [11] as the emerging consensus definition. The second line is motivated by tracking the information-flow in arbitrary programs, where most approaches quantify *leakage* as reduction in entropy about the program's input. In this paper, we focus on min-entropy as a measure of leakage because it is associated with strong operational security guarantees, see [34].

There have been efforts to understand the connections between the different notions of confidentiality proposed within each line of research (see [13] and [16], [34], respectively). The first studies of the relationship between differential privacy and quantitative notions of information-flow are emerging [2], [8], however, they do not directly compare leakage and differential privacy in terms of the security guarantees they deliver (see the section on related work). Such a comparison could be highly useful, as it could enable one to transfer existing analysis techniques and enforcement mechanisms from one line of research to the other.

It is not difficult to see that there can be no general upper bound for differential privacy in terms of the leakage about

the entire input.[1] However, it has been an open question whether it is possible to give upper bounds for the leakage in terms of differential privacy.

In this paper, we will address this open question. To begin with, we identify information-theoretic channels as a common model for casting differential privacy and leakage, where we assume that the input domain is fixed to $\{0,1\}^n$. Based on this model, we formally contrast the compositionality properties of differential privacy and leakage under sequential and parallel composition.

We observe a difference in the behavior of leakage and differential privacy under parallel composition, and we exploit this difference to construct, for every $n$, a channel that is $\epsilon$-differentially private and that leaks an amount of information that grows linearly with $n$. This result implies there can be no general (i.e. independent of the domain size) upper bound for the leakage of all $\epsilon$-differentially private channels.

The situation changes, however, if we consider channels on input domains of bounded size. For such channels, we exhibit the following connections between leakage and differential privacy.

For the case $n = 1$, we give a complete characterization of leakage in terms of differential privacy. More precisely, we prove an upper bound for the leakage of every $\epsilon$-differentially private channel. Moreover, we show that this bound is tight in the sense that, for every $\epsilon$, there is an $\epsilon$-differentially channel whose leakage matches the bound.

For the case $n > 1$, we prove upper bounds for the leakage of every $\epsilon$-differentially private channel in terms of $n$ and $\epsilon$. Technically, we achieve this by covering the channel's input domain by spheres of a fixed radius (with respect to the Hamming metric). The definition of $\epsilon$-differential privacy ensures that the elements within each sphere produce similar output, where similarity is quantified in terms of $\epsilon$ and the sphere radius. Based on this similarity and a recent characterization of the maximal leakage of channels [4], [23], we establish upper bounds for the information leaked about the elements of each sphere. By summing over all spheres, we obtain bounds for the information leaked about the entire input domain.

Our bounds are parametric in the number and the radius of the spheres used for covering the domain. We show how coding theory can be used for obtaining good instantiations of these parameters. In particular, we give examples where we derive bounds based on different classes of covering codes. We also exhibit limits for the bounds that can be obtained using our proof technique in terms of the sphere-packing bound. We perform an empirical evaluation that shows that the bounds we derived are close to this theoretical limit; moreover, we give an example channel whose leakage is only slightly below this limit, demonstrating the accuracy of our analysis.

Finally, although an explicit formula that precisely characterizes leakage in terms of privacy for finite input domains is still elusive, we show that such a characterization is in fact decidable. More precisely, we show that, for all $n$ and all rational functions $r$ (i.e. all quotients of polynomials with integer coefficients), one can decide whether the leakage of all $\epsilon$-differentially channels is upper-bounded by $\log_2 r(\epsilon)$.

In summary, our main contribution is to prove formal connections between leakage and differential privacy, the two most influential quantitative notions of confidentiality to date. In particular, (i) we prove upper bounds for the leakage in terms of differential privacy for channels with bounded input domain, and (ii) we show that there can be no such bounds that hold for unbounded input domains. Finally, (iii) we show that the question of a precise characterization of leakage in terms of differential privacy is in fact decidable.

The remainder of this paper is structured as follows. In Section II we introduce differential privacy and leakage. We cast both properties in a common model and compare their compositionality in Section III. We prove bounds for the leakage in terms of differential privacy in Sections IV and V, and we show their decidability in Section VI. We present related work in Section VII before we conclude in Section VIII.

## II. Preliminaries

In this section we review the definitions of min-entropy leakage and differential privacy, the two confidentiality properties of interest for this paper. For completeness, we also briefly review the most important analysis tools and enforcement mechanisms for each property.

### A. Min-entropy Leakage

In quantitative information-flow, one typically characterizes the security of a program in terms of the difficulty of guessing the input to the program when only the output is known. The difficulty of guessing can be captured in terms of information-theoretic entropy, where different notions of entropy correspond to different kinds of guessing [5]. In this paper, we focus on min-entropy as a measure, because it is associated with strong security guarantees, see [34]. However, instead of characterizing the security of a program in terms of the remaining entropy, we characterize the amount of leaked information in terms of the reduction in entropy about the program's input when the output is observed. Both viewpoints are informally related by the equation

*initial entropy = leaked information + remaining entropy.*

Formally, we model the input to a probabilistic program as a random variable $X$ and the output as a random variable $Y$.

[1] Intuitively, the leakage of a single sensitive bit (e.g. from a medical record) can entirely violate an individual's privacy; on a technical level, no deterministic program satisfies differential privacy even if it leaks only a small amount of information, because differentially private programs are necessarily probabilistic.

The dependency between $X$ and $Y$ is formalized as a conditional probability distribution $P_{Y|X}$ and is determined by the program's semantics. Such a conditional probability distribution $P_{Y|X}$ forms an information-theoretic *channel* from $X$ to $Y$. As is standard in the literature on quantitative information-flow, we will use the notion of a channel as the basis for our analysis.

We consider an adversary that receives the outcomes $Y$ of a channel $P_{Y|X}$ and wants to determine the corresponding value of $X$, where we assume that $X$ is distributed according to $P_X$. The initial uncertainty about the chosen element of $X$ is given by the *min-entropy* [31]

$$H_\infty(X) = -\log_2 \max_x P_X(x)$$

of $X$, which captures the probability of correctly guessing the outcome of $X$ in one shot.

The *conditional min-entropy* $H_\infty(X|Y)$ of $X$ given $Y$ is defined by

$$H_\infty(X|Y) = -\log_2 \sum_y P_Y(y) \max_x P_{X|Y}(x, y)$$

and captures the probability of guessing the value of $X$ in one shot when the outcome of $Y$ is known.

The *(min-entropy) leakage* $L$ of a channel $P_{Y|X}$ with respect to the input distribution $P_X$ characterizes the reduction in uncertainty about $X$ when $Y$ is observed,

$$L = H_\infty(X) - H_\infty(X|Y) \ ,$$

and is the logarithm of the factor by which the probability of guessing the value of $X$ is reduced by observing $Y$. Note that $L$ is not a property of the channel $P_{Y|X}$ alone as it also depends on $P_X$. We eliminate this dependency by considering the maximal leakage over all input distributions.

**Definition 1** (Maximal Leakage)**.** *The* maximal leakage *ML of a channel $P_{Y|X}$ is the maximal reduction in uncertainty about $X$ when $Y$ is observed*

$$ML(P_{Y|X}) = \max_{P_X}(H_\infty(X) - H_\infty(X|Y)) \ , \qquad (1)$$

*where the maximum is taken over all possible input distributions.*

The following result appears in [4], [23] and shows how the maximal leakage can be computed from the channel $P_{Y|X}$. For completeness, we include its proof in the appendix.

**Theorem 1.** *The maximal leakage of a channel $P_{Y|X}$ can be computed by*

$$ML(P_{Y|X}) = \log_2 \sum_y \max_x P_{Y|X}(y, x) \ ,$$

*where the maximum is assumed (e.g.) for uniformly distributed input.*

If the channel is deterministic, i.e. if for every $x$ there is a $y$ such that $P_{Y|X}(y, x) = 1$, we obtain $ML(P_{Y|X}) = \log_2 |Range(Y)|$. A direct consequence of this observation is that, for deterministic programs, any algorithm for computing the size of $Range(Y)$ (which corresponds to the size of the set of reachable final states of the program) can be used for computing $ML$, see also [22].

### B. Differential Privacy

In research on privacy-preserving data publishing, differential privacy by Dwork et al. [11] is the most popular definition of privacy to date; it quantifies the influence of individual records of an input dataset on the output of a publishing algorithm.

Informally, a probabilistic algorithm satisfies $\epsilon$-differential privacy if its output is robust (i.e. bounded in terms of $\epsilon$) with respect small changes in the input. This robustness ensures privacy because if two datasets differ only in one individual record, the algorithm's output on both datasets will be almost indistinguishable. An adversary will not be able to deduce from the output the value (or the presence) of any individual record in the dataset.

Formally, we consider algorithms $M$ that take as input subsets of a set $D$ of elements without further substructure. The distance between two input sets $D_1, D_2 \subseteq D$ is the size of their symmetric difference $D_1 \ominus D_2$ defined as

$$D_1 \ominus D_2 = (D_1 \setminus D_2) \cup (D_2 \setminus D_1) \ .$$

The following definition is a variant from [27] of the definition by Dwork et al. [12].

**Definition 2** (Differential Privacy)**.** *A randomized algorithm $M$ satisfies* $\epsilon$-differential privacy *if, for all input sets $D_1, D_2 \subseteq D$ and for all $S \subseteq Range(M)$*

$$P[M(D_1) \in S] \leq e^{\epsilon |D_1 \ominus D_2|} P[M(D_2) \in S] \ . \qquad (2)$$

Notice that in the literature, differential privacy is sometimes defined using alternative metrics on $D$, leading to different security guarantees. The metric used in Definition 2 protects the presence (resp. absence) of individual elements in a dataset; other metrics protect (non-binary) values of the individual records in a database [12], a setting which we do not consider in this paper.

Observe that that Definition 2 requires that the secret data is released by a randomized algorithm: a deterministic algorithm with non-constant output will not satisfy $\epsilon$-differential privacy for any finite $\epsilon$. Dwork et al. [12] have proposed a method for making deterministic programs differentially private by adding a certain amount of noise to the algorithm's output. The amount of noise that is required for achieving $\epsilon$-differential privacy depends on $\epsilon$ and the degree of sensitivity of the program, which is a form of Lipschitz-continuity.

**Definition 3** (Sensitivity)**.** *For a function $F$ that maps subsets of $D$ to real numbers, one defines the* sensitivity *$S(F)$ of $F$ by*

$$\max_{D_1, D_2 \subseteq D} \frac{|f(D_1) - f(D_2)|}{|D_1 \ominus D_2|} .$$

The so-called *Laplacian mechanism* turns a deterministic algorithm $F$ into an $\epsilon$-differentially private probabilistic algorithm. This is achieved by adding symmetric, exponentially distributed noise to the output of $F$. The center of the noise is the output of $F$ and the standard deviation of the noise is calibrated to $S(F)$ and $\epsilon$. One obtains the following theorem, which is fundamental for most applications of differential privacy to date.

**Theorem 2** (Laplacian Mechanism [12])**.** *For a function $F$ mapping subsets of $D$ to real numbers and a random variable $N$ distributed according to $N \sim Lap(S(f)/\epsilon)$, the probabilistic algorithm*

$$M = F + N$$

*defined by $M(D) = F(D) + N$ satisfies $\epsilon$-differential privacy.*

While the Laplacian mechanism is the most influential enforcement mechanism to date, there are several variants and alternative mechanisms emerging, see e.g. [24], [30], [32], [36]. In this paper, we analyze the leakage of differentially private algorithms without making any assumptions about the mechanism used to achieve differential privacy. In particular, this implies that our results apply to all existing and future differentially private mechanisms.

## III. Contrasting Leakage and Differential Privacy

In this section, we cast leakage and differential privacy in a common model. Based on this model, we will formally compare the basic assumptions made, and guarantees delivered by, both notions. Moreover, the model enables us to compare both notions in terms of their behavior under sequential and parallel composition. This comparison serves two purposes. First, it systematizes and completes existing knowledge. Second, the exposed differences form the basis for our results in Section V.

### A. A Common Model for Leakage and Differential Privacy

Differential privacy and leakage are properties of objects of different types. Differential privacy (Definition 2) is a property of algorithms that take as input sets of (a possibly infinite number of) basis elements, whereas leakage (Definition 1) is a property of programs that take as input the elements of a fixed finite set.

We cast both properties in a common model by assuming that the base set $D$ of the data-publishing algorithm has finite size $n$, i.e. $D = \{d_1, \ldots, d_n\}$. With this assumption, we can describe every subset $D' \subseteq D$ by a vector $x' \in \{0,1\}^n$, where the $k$th component $\pi_k(x')$ is set to 1 if and only if $d_k \in D'$. Notice that for two sets $D', D'' \subseteq D$ with vector representations $x', x'' \in \{0,1\}^n$, the set distance $|D' \ominus D''|$ corresponds to the number of positions in which $x'$ and $x''$ differ, i.e. to their Hamming distance.

Furthermore, we will assume that the range of the algorithm is also finite. Although in theory, differentially private algorithms typically map to a continuum of real numbers, the active range of most practical implementations will be discrete, bounded, and finite.

For a given algorithm $M$ according to definition 2, we hence define a channel $P_{Y|X}$ with $Range(X) = \{0,1\}^n$ by $P_{Y|X}(y, x) = P[M(D') = y]$, where $x$ is the characteristic vector of $D' \subseteq D$. In the remainder of this paper we will always assume that $Range(X) = \{0,1\}^n$, that $Range(Y)$ is finite, and that the program is given in terms of a channel $P_{Y|X}$.

### B. Security Guarantees

Definitions 1 and 2 do not bear much structural resemblance at first sight. We will recast both definitions in a form that better exhibits their differences and similarities.

We first introduce additional notation. For $x \in \{0,1\}^n$, $i \in \{1, \ldots, n\}$, and $\dot{x} \in \{0,1\}$, let $x[i/\dot{x}]$ denote the bit vector that is equal to $x$ except that the $i$th component $\pi_i(x)$ is replaced by $\dot{x}$, i.e. $\pi_i(x[i/\dot{x}]) = \dot{x}$ and $\pi_j(x[i/\dot{x}]) = \pi_j(x)$ for $i \neq j$. Let $\dot{X}$ denote a random variable that models the choice of one bit, i.e. $Range(\dot{X}) = \{0,1\}$. Then $x[i/\dot{X}]$ denotes the random variable with range $\{x[i/0], x[i/1]\}$ that is distributed according to $\dot{X}$. The condition $X = x[i/\dot{X}]$ denotes that the value assumed by $X$ corresponds to that of $x$, except for bit $i$ which is distributed according to $\dot{X}$.

The following lemma states that a differentially private mechanism protects every individual bit of any input vector, in the sense that even if the input is completely known except for a single bit, the result of the mechanism does not significantly influence the probability of that bit being set to 1. The result and the proof are inspired by the one sketched in the appendix of [12].

**Lemma 1.** *A channel $P_{Y|X}$ guarantees $\epsilon$-differential privacy if and only if, for all $x \in \{0,1\}^n$, all $i \in \{1, \ldots, n\}$, all distributions $P_{\dot{X}}$ of $\dot{X}$, and all $y \in Range(Y)$,*

$$P[\dot{X} = 1] \leq e^{\epsilon} P[\dot{X} = 1 \mid Y = y \wedge X = x[i/\dot{X}]] . \quad (3)$$

A proof of Lemma 1 can be found in the appendix. The following lemma is obtained by a direct reformulation of Definition 1; it exhibits the structural similarity behind Definition 1 and the statement in Lemma 1.

**Lemma 2.** *The leakage of channel $P_{Y|X}$ is upper-bounded by $ML(P_{Y|X}) \leq l$ if and only if, for all distributions $P_X$ we have that*

$$E_y[\max_x P[X = x | Y = y]] \leq 2^l \max_x P[X = x] , \quad (4)$$

*where $E_y$ denotes the expected value over $y$.*

The comparison of the statements in Lemmas 1 and 2 exhibits the differences in the guarantees provided by leakage and differential privacy.

Security guarantees based on leakage (Lemma 2) protect entire bit-vectors from being guessed, under the assumption that they are hard to guess a priori, i.e. before observing the system's output. Leakage-based guarantees are hence the adequate tool for protecting high-entropy secrets, such as cryptographic keys, passwords, or biometric data. However, leakage-based guarantees do not make immediate assertions about the difficulty of guessing individual bits, which is fundamental for the protection of privacy.

In contrast, differential privacy protects each individual bit in a bit-vector, even if all other bits are known (Lemma 1). This guarantee is adequate for protecting secrets in contexts where one cannot make reasonable assumptions about an adversary's background knowledge, such as the value of other bits or the distribution from which the secret bit is drawn.

### C. Compositionality

Well-behavedness under composition is an essential prerequisite for any meaningful notion of security. Without good composition properties, two secure systems may become insecure when combined.We next compare the compositionality properties of leakage and differential privacy.

*1) Sequential Composition:* We define sequential composition as the subsequent application of two queries to the same dataset, where the second query may also take the output of the first query into account. Formally, we model this by requiring that the input domain of the second channel is the cartesian product of the range of the first channel and the dataset. It was already known that the differential privacy of two sequentially composed channels is upper-bounded by the sum of the differential privacy of the individual channels [27]. We next show that the leakage shows the same additive behavior under this notion of sequential composition.

We begin by defining the sequential composition of channels.

**Definition 4.** *The* sequential composition $C_1 + C_2$ *of channels* $C_1 = P_{Y_1|X}$ *and* $C_2 = P_{Y_2|Y_1 \times X}$ *is defined as* $C_1 + C_2 = P_{Y_1 \times Y_2|X}$*, where*

$$P_{Y_1 \times Y_2|X}((y_1, y_2), x) = P_{Y_1|X}(y_1, x) P_{Y_2|Y_1 \times X}(y_2, (y_1, x)) \ .$$

We obtain the following properties of leakage and privacy of the sequential composition of two channels.

**Lemma 3.** *Let* $C_i$ *be channels that are* $\epsilon_i$*-differentially private* ($i = 1, 2$)*. Then*

1) $C_1 + C_2$ *is* $\epsilon_1 + \epsilon_2$*-differentially private, and*
2) $ML(C_1 + C_2) \leq ML(C_1) + ML(C_2)$*.*

*Proof:* For the original proof of Assertion 1), refer to [27]. For a proof on basis of our model, see the appendix. For Assertion 2), consider

$$\begin{aligned}
&ML(C_1 + C_2) \\
&\overset{(*)}{=} \log \sum_{(y_1, y_2)} \max_x P_{Y_1 \times Y_2|X}((y_1, y_2), x) \\
&\overset{(**)}{=} \log \sum_{(y_1, y_2)} \max_x P_{Y_1|X}(y_1, x) P_{Y_2|Y_1 \times X}(y_2, (y_1, x)) \\
&\leq \log \sum_{y_1} \sum_{y_2} \max_x P_{Y_1|X}(y_1, x) \max_{x, y_1} P_{Y_2|Y_1 \times X}(y_2, (y_1, x)) \\
&= \log \left( \sum_{y_1} \max_x P_{Y_1|X}(y_1, x) \right) \left( \sum_{y_2} \max_{y_1, x} P_{Y_2|Y_1 \times X}(y_2, (y_1, x)) \right) \\
&= ML(C_1) + ML(C_2),
\end{aligned}$$

where $x_1$, $x_2$, $y_1$, and $y_2$ range over $X_1$, $X_2$, $Y_1$, and $Y_2$ respectively. Note that (*) follows from Theorem 1 and (**) follows from Definition 4. ■

*2) Parallel Composition:* We define the parallel composition of channels as their application to disjoint subsets of the same dataset. It was already known that the maximum of the differential privacy bounds of the individual channels is also a bound for the differential privacy of their parallel composition [27]. As we will show next, the situation is different for leakage, which adds up under parallel composition.

**Definition 5.** *The* parallel composition $C_1 \times C_2$ *of channels* $C_1 = P_{Y_1|X_1}$ *and* $C_2 = P_{Y_2|X_2}$ *is defined as* $C_1 \times C_2 = P_{Y_1 \times Y_2|X_1 \times X_2}$*, where*

$$P_{Y_1 \times Y_2|X_1 \times X_2}((y_1, y_2), (x_1, x_2)) = P_{Y_1|X_1}(y_1, x_1) P_{Y_2|X_2}(y_2, x_2) \ .$$

Notice that the cartesian product of characteristic bit-vectors corresponds to the disjoint union of the underlying sets, as discussed in Section III-A. That is, Definition 5 indeed captures the application of channels to disjoint subsets of a dataset.

We can prove the following properties about leakage and privacy of the parallel composition of two channels.

**Lemma 4.** *Let* $C_i$ *be channels that are* $\epsilon_i$*-differentially private* ($i = 1, 2$)*. Then*

1) $C_1 \times C_2$ *is* $\max\{\epsilon_1, \epsilon_2\}$*-differentially private, and*
2) $ML(C_1 \times C_2) = ML(C_1) + ML(C_2)$*.*

*Proof:* For the original proof of Assertion 1), see [27]. For a proof in our channel-based model, see the appendix.

For Assertion 2) consider

$$\begin{aligned} ML(C_1 \times C_2) &\overset{(*)}{=} \log \sum_{(y_1,y_2)} \max_{(x_1,x_2)} P_{Y_1\times Y_2|X_1\times X_2}((y_1,y_2),(x_1,x_2)) \\ &= \log \sum_{y_1}\sum_{y_2} \max_{x_1}\max_{x_2} P_{Y_1|X_1}(y_1,x_1) P_{Y_2|X_2}(y_2,x_2) \\ &= \log \sum_{y_1} \max_{x_1} P_{Y_1|X_1}(y_1,x_1) \sum_{y_2} \max_{x_2} P_{Y_2|X_2}(y_2,x_2) \\ &= ML(C_1) + ML(C_2) \ , \end{aligned}$$

where $x_1$, $x_2$, $y_1$ and $y_2$ range over $X_1$, $X_2$, $Y_1$, and $Y_2$ respectively. Note that (*) follows from Theorem 1. ∎

Lemma 4 exhibits a difference in the behavior under parallel composition of differential privacy and leakage. In Section V, we will exploit this difference for proving the impossibility of general bounds for the leakage in terms of differential privacy.

## IV. Characterizing the Leakage of Differentially Private Mechanisms – The Case $n = 1$

In this section, we provide a characterization of the leakage of differentially private channels that take only a single bit of input. In particular, we prove upper bounds in terms of $\epsilon$ for the leakage of any $\epsilon$-differentially private channel. Moreover, we show that there is a channel whose leakage matches this bound. In Section V we consider the general case.

### A. Channels with 1-bit range

We first consider the case of channels whose range is also a single bit, i.e. $Range(X) = Range(Y) = \{0,1\}$. For simplicity of presentation, we will assume that all channels $P_{Y|X}$ are given in one of the two canonical forms $C_1(\epsilon, p)$ and $C_2(\epsilon, p)$ defined by the matrix in Figure 1. As the following Lemma shows, this assumption is not a restriction.

**Lemma 5.** *Let $P_{Y|X}$ be a channel. Then there are $p \in [0,1]$, $\epsilon > 0$, and $i \in \{1,2\}$ such that*

$$P_{Y|X} = C_i(\epsilon, p) \ .$$

*Proof:* Choose $i = 1, 2$ depending on which of the probabilities $P_{Y|X}(0,0)$ and $P_{Y|X}(0,1)$ is larger. There is always an $\epsilon > 0$ such that the ratio between the entries in the first column is given by $e^\epsilon$. The statement follows from the observation that the entries of each row must sum to 1. ∎

For what follows it is irrelevant whether we are dealing with the canonical representation $C_1$ or $C_2$ of a channel. We will hence drop the index and refer to the canonical representation as $C(\epsilon, p)$.

We obtain the following privacy guarantees for the canonical representation $C(\epsilon, p)$.

| $C_1(\epsilon,p) = P_{Y\|X}$ | $Y=0$ | $Y=1$ |
|---|---|---|
| $X=0$ | $p$ | $1-p$ |
| $X=1$ | $pe^\epsilon$ | $1-pe^\epsilon$ |

| $C_2(\epsilon,p) = P_{Y\|X}$ | $Y=0$ | $Y=1$ |
|---|---|---|
| $X=0$ | $pe^\epsilon$ | $1-pe^\epsilon$ |
| $X=1$ | $p$ | $1-p$ |

where $0 \le p \le 1$ and $pe^\epsilon \le 1$.

Figure 1. The canonical representations $C_1(\epsilon, p)$ and $C_2(\epsilon, p)$ of channels $P_{Y|X}$.

**Lemma 6.** *The channel $C(\epsilon, p)$ is $\epsilon$-differentially private if and only if*

$$p \le \frac{1}{e^\epsilon + 1} \ .$$

*Proof:* It suffices to show that $P_{Y|X}(y,x) \le e^\epsilon P_{Y|X}(y,x')$ for all $x, x', y \in \{0,1\}$. A direct calculation shows that this condition is satisfied if and only if $p \le \frac{e^\epsilon - 1}{e^{2\epsilon}-1} = \frac{1}{e^\epsilon+1}$. ∎

The following theorem characterizes the leakage of $\epsilon$-differentially private channels with 1-bit domain and range: It gives an upper bound for the leakage of every channel, and it shows that this bound can be matched.

**Theorem 3.** *Let $Range(X) = Range(Y) = \{0,1\}$.*

1) *If a channel $P_{Y|X}$ is $\epsilon$-differentially private, then*

$$ML(P_{Y|X}) \le \log_2 \frac{2e^\epsilon}{e^\epsilon + 1} \ .$$

2) *The channel $P_{Y|X} = C(\epsilon, \frac{1}{e^\epsilon+1})$ is $\epsilon$-differentially private and*

$$ML(P_{Y|X}) = \log_2 \frac{2e^\epsilon}{e^\epsilon + 1} \ .$$

*Proof:* For the proof of Assertion 1), consider an $\epsilon$-differentially private channel $P_{Y|X}$. From Lemma 6 it follows that there is a $p \le \frac{1}{e^\epsilon+1}$ such that $P_{Y|X} = C(\epsilon, p)$. We apply Theorem 1 to compute the maximal leakage of $C(\epsilon, p)$ as the sum of the column maximums of the matrix in Figure 1. We obtain

$$ML(C(\epsilon, p)) = \log_2(1 + p\ (e^\epsilon - 1)) \ .$$

Since $p \le \frac{1}{e^\epsilon+1}$, we conclude

$$ML(C(\epsilon, p)) \le \log_2 \frac{2e^\epsilon}{e^\epsilon + 1} \ .$$

For the proof of Assertion 2), we compute the maximal leakage of $C(\epsilon, \frac{1}{e^\epsilon+1})$ along the same lines as in the proof of Assertion 1. The $\epsilon$-differential privacy of the channel follows from Lemma 6. ∎

### B. Channels of arbitrary range

We now show that the characterization of the leakage of channels with binary range extends to channels with arbitrary (but finite) range.

Formally, we let $X = \{0, 1\}$, $Y = \{y_1 \dots y_k\}$ and let $P_{Y|X}$ be a channel. Then $P_{Y|X}$ can be represented by a matrix

| $P_{Y\|X}$ | $Y = y_1$ | $\dots$ | $Y = y_k$ |
|---|---|---|---|
| $X = 0$ | $p_1$ | $\dots$ | $p_k$ |
| $X = 1$ | $q_1$ | $\dots$ | $q_k$ |

where $p_1 + \ldots + p_k = q_1 + \ldots + q_k = 1$.

The following corollary states that the upper bounds for the leakage given in Theorem 3 also hold for channels of arbitrary range.

**Corollary 1.** *If $P_{Y|X}$ is $\epsilon$-differentially private, then the maximal leakage ML of $P_{Y|X}$ verifies*

$$ML(P_{Y|X}) \leq \log_2 \frac{2e^\epsilon}{e^\epsilon + 1} \ .$$

*Proof:* We construct an $\epsilon$-differentially private channel $P_{\bar{Y}|X}$ with $Range(\bar{Y}) = \{0, 1\}$ such that the leakage of $P_{\bar{Y}|X}$ matches that of $P_{Y|X}$. Technically, let $I = \{i \mid p_i \leq q_i\}$, $\bar{p} = \sum_{i \notin I} p_i$ and $\bar{q} = \sum_{i \in I} q_i$. Then we define $P_{\bar{Y}|X}$ as:

| $P_{\bar{Y}\|X}$ | $Y = 0$ | $Y = 1$ |
|---|---|---|
| $X = 0$ | $\bar{p}$ | $1 - \bar{p}$ |
| $X = 1$ | $1 - \bar{q}$ | $\bar{q}$ |

Note that $1 - \bar{p} = \sum_{i \in I} p_i$, $1 - \bar{q} = \sum_{i \notin I} q_i$, hence we have $\bar{p} \geq 1 - \bar{q}$ and $\bar{q} \geq 1 - \bar{p}$. Applying Theorem 1 to $P_{Y|X}$, we obtain

$$\begin{aligned} ML(P_{Y|X}) &= \log_2 \sum_y \max_x P_{Y|X}(y, x) \\ &= \log_2 (\sum_{i \notin I} p_i + \sum_{i \in I} q_i) \\ &= \log_2 (\bar{p} + \bar{q}) \\ &= ML(P_{\bar{Y}|X}) \end{aligned}$$

Moreover, as $P_{Y|X}$ is $\epsilon$-differentially private, we know that, for every $i \in I$, $q_i \leq e^\epsilon \ p_i$, and for every $i \notin I$, $p_i \leq e^\epsilon q_i$. We obtain $\sum_{i \in I} q_i \leq e^\epsilon \ \sum_{i \in I} p_i$ and $\sum_{i \notin I} p_i \leq e^\epsilon \ \sum_{i \notin I} q_i$. Hence $P_{\bar{Y}|X}$ is also differentially private, which concludes this proof. ∎

Tightness of the bound, as expressed in Theorem 3.2, immediately extends to channels with arbitrary output size, since one can view any channel with output of size 2 as a channel with output of size $k$.

## V. Bounds on the Leakage of Differentially Private Mechanisms on Arbitrary Input Domains

In Section IV, we have characterized the maximal leakage of differentially private channels with input domains of size 2. In this section, we consider the case of differentially private channels with input domains of arbitrary size. It turns out that a complete characterization of the leakage of such channels is a challenging problem. We take the following two steps towards such a characterization: first, we establish upper bounds for the leakage of any $\epsilon$-differentially private channel with an $n$-bit input domain. We obtain this result by exhibiting a connection to coding theory, through which we derive both concrete bounds for the leakage, and limits for the bounds that can be achieved using this connection.

Second, we construct an $\epsilon$-differentially channel that takes inputs of $n$ bits and leaks at least $n \log_2 \frac{2e^\epsilon}{e^\epsilon + 1}$ of those bits. By increasing $n$, one can hence leak an arbitrary amount of information, while retaining the privacy guarantee. This result implies that there can be no general (i.e. independent of the size of the input domain) upper bound for the leakage of $\epsilon$-differentially private channels.

### A. Bounds for Input Domains of Fixed Size

To obtain our bounds, we first cover the channel's input domain by spheres of a fixed radius with respect to the Hamming metric $\Delta$. Using Theorem 1 we obtain upper bounds for the information leaked about the elements of each sphere. By summing over all spheres, this yields bounds for the information leaked about the entire input domain. These bounds are parametric in the number of spheres and their radius. We observe that each sphere covering corresponds to a covering code, and we show how a given code can be used to instantiate the parameters and obtain a concrete bound for the leaked information; moreover, we show how the Hamming bound leads to limits for the bounds that can be obtained in this way.

Formally, a subset $\{x_1, \dots, x_m\} \subseteq \{0, 1\}^n$ is a *$d$-covering code of length $n$ and size $m$* if, for every $x \in \{0, 1\}^n$ there is an $i \in \{1, \dots, m\}$ such that $\Delta(x, x_i) \leq d$. This definition corresponds to the requirements that the spheres $U_d(x_i)$ with radius $d$ and center $x_i$, defined by

$$U_d(x_i) = \{x \in \{0, 1\}^n \mid \Delta(x, x_i) \leq d\} \ ,$$

cover the whole space. For an overview of research on covering codes, see [9].

For a given $d$-covering code, we obtain the following bounds for the leakage of differentially private mechanisms.

**Theorem 4.** *Let $P_{Y|X}$ be a channel with $Range(X) = \{0, 1\}^n$ and let $\{x_1, \dots, x_m\}$ be a $d$-covering code of length $n$. If $P_{Y|X}$ satisfies $\epsilon$-differential privacy, then its maximal leakage is upper-bounded by*

$$ML(P_{Y|X}) \ \leq \ d\, \epsilon \log_2 e + \log_2 m \ .$$

*Proof of Theorem 4.:*

$$\begin{aligned} ML(P_{Y|X}) &\overset{(*)}{=} \log_2 \sum_y \max_x P_{Y|X}(y,x) \\ &\leq \log_2 \sum_y \sum_{i=1}^{m} \max_{x \in U_d(x_i)} P_{Y|X}(y,x) \\ &\overset{(**)}{\leq} \log_2 \sum_y \sum_{i=1}^{m} P_{Y|X}(y,x_i) e^{\epsilon d} \\ &= \log_2 e^{\epsilon d} \sum_{i=1}^{m} \sum_y P_{Y|X}(y,x_i) \\ &= d\,\epsilon \log_2 e + \log_2 m \ , \end{aligned}$$

where $(*)$ follows from Theorem 1 and $(**)$ follows from the definition of differential privacy (Definition 2). ■

Note that the bounds obtained using Theorem 4 are of interest only for instantiations with $\epsilon d \log_2 e + \log_2 m \leq n$ since, trivially, $ML(P_{Y|X}) \leq n$ for a channel with inputs of $n$ bits. Moreover, note that a covering code is required for obtaining concrete bounds for $ML(P_{Y|X})$ in terms of $n$ and $\epsilon$. We next give two example instantiations with simple covering codes.

For the first instantiation, we consider a trivial $n$-covering code of size 1. With this code, we obtain the following corollary of Theorem 4.

**Corollary 2.** *Let $P_{Y|X}$ be a channel with Range$(X) = \{0,1\}^n$. If $P_{Y|X}$ satisfies $\epsilon$-differential privacy, then*

$$ML(P_{Y|X}) \leq n\,\epsilon \log_2 e \ .$$

For the second instantiation, we consider an $n$-ary repetition code of size 2. More precisely, we consider the code $\{0^n, 1^n\} \subseteq \{0,1\}^n$. Observe that for this code $d = \left\lfloor \frac{n}{2} \right\rfloor$, because each $x \in \{0,1\}^n$ has a Hamming distance of at most $\left\lfloor \frac{n}{2} \right\rfloor$ to either one of the codewords. Using this code, we obtain the following corollary of Theorem 4.

**Corollary 3.** *Let $P_{Y|X}$ be a channel with Range$(X) = \{0,1\}^n$. If $P_{Y|X}$ satisfies $\epsilon$-differential privacy, then*

$$ML(P_{Y|X}) \leq \left\lfloor \frac{n}{2} \right\rfloor \epsilon \log_2 e + 1 \ .$$

For a fixed $n$, Figures 2 and 3 depict the bounds obtained by trivial covering codes (Corollary 2) and repetition codes (Corollary 3) as functions of $\epsilon$. The corresponding curves cross, which illustrates that each of the codes gives tighter (i.e. lower) upper bounds for the leakage for different ranges of $\epsilon$.

While there is a large number of codes with which Theorem 4 can be instantiated and with which the bounds from Corollaries 2 and 3 could potentially be improved [9], the so-called Hamming-bound puts a theoretical limit on what can be achieved using our proof technique.

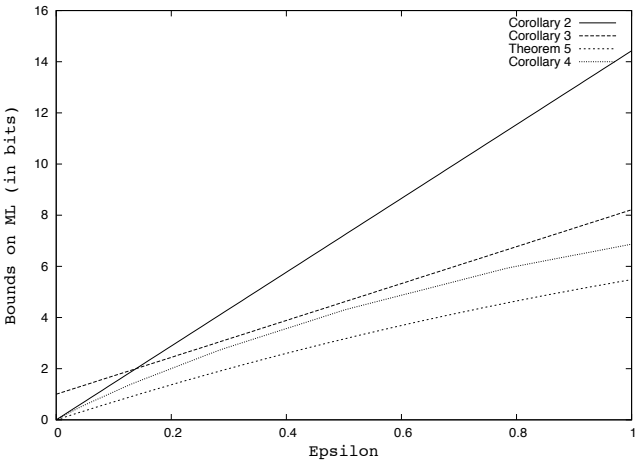



Figure 2. For a fixed input size of $n = 10$ bits, this figure depicts different upper and lower bounds for the maximal leakage of $\epsilon$-differentially private channels as functions of $\epsilon \in [0,1]$ (horizontal axis). The upper curve depicts the upper bounds obtained using trivial covering codes in Corollary 2. The second curve depicts the upper bounds obtained using repetition codes in Corollary 3. The intersection of both curves shows that each code leads to better bounds for different ranges of $\epsilon$. The third curve depicts the limit of what we can achieve using our proof technique, as stated in Corollary 4. Finally, the lower curve depicts the leakage of the channel constructed in Theorem 5. The small gap between the second curve and the third curve illustrates the good quality of the bounds obtained using repetition codes; the small gap between the second curve and the lower curve illustrates that, for small $n$, the leakage of the channel constructed in Theorem 5 is not far from optimal.

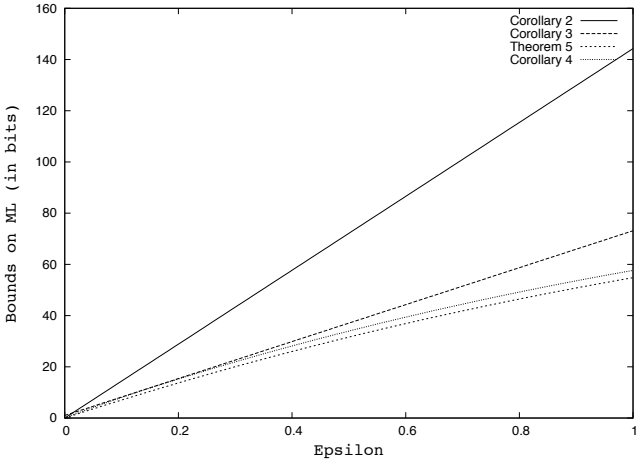



Figure 3. The curves in this figure depict the bounds as described in Figure 2, but for the case of $n = 100$ bits.

**Corollary 4.** *For given $n$ and $\epsilon > 0$, any bound $B(\epsilon, n)$ for the maximal leakage obtained by Theorem 4 will satisfy*

$$B(\epsilon, n) \geq \min_{d=0}^{n} \left( \epsilon d \log_2 e + n - \log_2 \sum_{i=0}^{d} \binom{n}{i} \right) .$$

*Proof:* Formally the Hamming bound

$$m \sum_{i=0}^{d} \binom{n}{i} \geq 2^n$$

is obtained by observing that a necessary requirement for any $d$-covering code of length $n$ and size $m$ is that the number of elements in the corresponding spheres sums to $2^n$. Inserting this bound into Theorem 4 yields the assertion. ∎

Figures 2 and 3 depict the limit given by Corollary 4 as a function of $\epsilon$. They also illustrate that, for $n = 10$, $n = 100$, and $\epsilon \in [0, 1]$, the bounds obtained by repetition codes are close to this limit.

### *B. Impossibility of General Bounds for the Leakage of Differentially Private Mechanisms*

Theorem 4 and Corollaries 2 and 3 give bounds for the leakage in terms of $n$ and $\epsilon$. In this section, we show that there can be no such bounds that are independent of $n$.

To this end we construct, for every $\epsilon > 0$ and every $n \in \mathbb{N}$, a channel that is $\epsilon$-differentially private and that leaks $n \log_2 \frac{2e^\epsilon}{e^\epsilon + 1}$ bits. Technically, we achieve this by $n$-fold parallel composition of the 1-bit channel from Theorem 3.2. The composition results developed in Section III-C then show that differential privacy remains invariant under composition, but that the leakage increases by a factor of $n$. We obtain the following theorem.

**Theorem 5.** *There is an $\epsilon$-differentially private channel $P_{Y|X}$ with Range$(X) = \{0, 1\}^n$ whose maximal leakage satisfies*

$$ML(P_{Y|X}) = n \log_2 \frac{2e^\epsilon}{e^\epsilon + 1} .$$

*Proof:* Theorem 3.2 shows that there is a 1-bit channel $C$ that leaks $\log_2 \frac{2e^\epsilon}{e^\epsilon + 1}$ bits. Consider the $n$-fold parallel composition $C^n = C \times \cdots \times C$ of $C$. Lemma 4.1 shows that $C^n$ is still $\epsilon$-differentially private. Lemma 4.2 shows that $C^n$ leaks $n \log_2 \frac{2e^\epsilon}{e^\epsilon + 1}$ bits, which concludes this proof. ∎

In Figures 2 and 3, we depict the leakage of the channel constructed in Theorem 5 as a function of $\epsilon$. It should be noted that for $\epsilon \to \infty$, $n \log_2 \frac{2e^\epsilon}{e^\epsilon + 1}$ converges from below to $n \log_2 2 = n$, which is the maximal leakage of a channel with input set $\{0, 1\}^n$.

An important consequence of Theorem 5 is that there cannot be domain-size-independent bounds for the leakage in terms of differential privacy.

**Corollary 5.** *For every $\epsilon > 0$ and $l > 0$, there exists an $\epsilon$-differentially private channel $C$ such that $ML(C) > l$.*

It is sufficient to consider the channel $C^n$ with a sufficiently large value for $n$, since the channel is $\epsilon$-differentially private for all $n$ and $ML(C^n) \to \infty$ as $n \to \infty$.

## VI. Decidability of Bounds for the Leakage of Differentially Private Channels

In contrast to the case $n = 1$ (see Theorem 3), our upper bounds for the leakage of differentially private mechanisms are not tight for the case $n > 1$. In particular, there is a gap between the upper bounds derived in Theorem 4 and the leakage of the channel constructed in Theorem 5, see Figures 2 and 3. Although we do not give a closed expression characterizing the leakage for $n > 1$, we next show that it is decidable whether a given rational function constitutes such a characterization. Our proof proceeds by reducing the problem to a system of polynomial inequalities over the reals.

### *A. Background on decidability of real closed fields*

For completeness, we provide a brief overview of the decidability results that are needed for our reduction; we refer the interested reader to [17] for further details.

A real closed field is an ordered field $\mathbb{F}$ such that every positive element of $\mathbb{F}$ is a square, and every polynomial $P$ with coefficients in $\mathbb{F}$ that is of odd degree has at least one root. Formally, the theory of ordered fields is obtained from the theory of closed fields by adding a binary relation $\leq$ that satisfies the axioms of partial orders: reflexivity, transitivity, and anti-symmetry, plus the axiom of totality

$$\forall a\ b \in \mathbb{F}.\ a \leq b \vee b \leq a$$

and compatibility axioms for addition and multiplication:

$$\forall a\ b\ c \in \mathbb{F}.\ a \leq b \Rightarrow a + c \leq b + c$$
$$\forall a\ b \in \mathbb{F}.\ (0 \leq a \wedge 0 \leq b) \Rightarrow 0 \leq a\ b$$

Then, the theory of real closed fields is obtained from the theory of ordered fields by adding an axiom for the existence of square roots

$$\forall x \in \mathbb{F}.\ x > 0 \Longrightarrow \exists y \in \mathbb{F}.\ x = y^2$$

and an axiom scheme for the existence of a root for polynomials $Q$ of odd order

$$\exists x \in \mathbb{F}.\ Q(x) = 0 .$$

Any sentence of the theory of real closed fields is valid if and only if it is valid for the real numbers. Moreover, Tarski [35] proved that the theory admits elimination of quantifiers, i.e. for every sentence $\phi$ of the theory, there exists an equivalent sentence $\psi$ that is quantifier-free, and then concluded that the theory of real closed fields is decidable. However, the decision procedure that can be extracted from Tarski's result is highly inefficient. More efficient algorithms have been devised subsequently, such as the Cylindrical Algebraic Decomposition method [10], or Hörmander's method [19].

It should be noted that many useful constructions can be expressed in the theory of real closed fields. For instance, one can use 1 and addition to encode any natural number, and hence any polynomial expression with integer coefficients. Moreover, one can encode the maximum of two expressions $e$ and $e'$ by introducing a fresh variable $x$ and requiring that

$$e \leq x \wedge e' \leq x \wedge (x = e \vee x = e') \ .$$

One could use such an encoding to express that the leakage of a channel is bounded by the logarithm of an expression that depends on $\epsilon$. For the sake of readability, we prefer to give a more direct formalization.

### B. Reduction

We prove the decidability of the existence of (optimal) rational upper bounds by characterizing them in the theory of real closed fields. In the following, we assume channels $P_{Y|X}$ with $Range(X) = \{0,1\}^n$ and $Range(Y) = \{y_1, \ldots, y_m\}$ arbitrary but finite. The following theorem formalizes our results.

**Theorem 6** (Decidability of Rational Bounds)**.** *Let $r$ and $s$ be polynomials with coefficients in $\mathbb{Z}$, such that $r$ and $s$ are strictly positive over $[1, \infty)$. Then the following questions are decidable.*

1) *For all $\epsilon > 0$ and for all $\epsilon$-differentially private channels $P_{Y|X}$ it holds that*

$$ML(P_{Y|X}) \leq \log_2 \frac{r(e^\epsilon)}{s(e^\epsilon)}$$

2) *For all $\epsilon > 0$ there exists an $\epsilon$-differentially private channel $P_{Y|X}$ such that*

$$ML(P_{Y|X}) = \log_2 \frac{r(e^\epsilon)}{s(e^\epsilon)} \ .$$

*Proof:* It is sufficient to find formulae of the theory of reals expressing that $\log_2 r/s$ is an upper bound (resp. a tight upper bound) of the leakage of every $\epsilon$-differentially private channel. As a first step, it is convenient to rephrase the problems by considering $\lambda = e^\epsilon$ instead of $\epsilon$. Formally, Assertion 1) of Theorem 6 is equivalent to the following assertion: for all $\lambda > 1$ and for all $\ln \lambda$-differentially private channels $P_{Y|X}$ it holds that

$$ML(P_{Y|X}) \leq \log_2 \frac{r(\lambda)}{s(\lambda)} \ .$$

As a further step, for all $x \in \{0,1\}^n$ and $y \in \{y_1, \ldots, y_m\}$ we introduce variables $p_{x,y}$ for representing the entries of a matrix representation of channels $P_{Y|X}$. The requirement that a matrix represents a valid channel can be expressed by the following formula

$$\mathsf{VC} \equiv \left( \bigwedge_{x,y} 0 \leq p_{x,y} \right) \wedge \left( \bigwedge_{x} \sum_{y} p_{x,y} = 1 \right) \ ,$$

where variables $x, y$ range over $Range(X)$ and $Range(Y)$, respectively. The requirement that a channel is $\log_2 \lambda$-differentially private can be expressed by the following formula

$$\mathsf{DP} \equiv \bigwedge_{\{x,x',y \,|\, \Delta(x,x')=1\}} p_{x',y} \leq \lambda p_{x,y}$$

where $\Delta$ denotes the Hamming distance.

Finally, we can express the requirement that the sum of the maximal elements of each column of a channel is upper-bounded by $r(\lambda)/s(\lambda)$, or equivalently that the maximal leakage of this channel is upper-bounded by $\log_2 (r(e^\epsilon)/s(e^\epsilon))$, for $\epsilon = \ln \lambda$, using the following formula

$$\mathsf{UB} \equiv \bigwedge_{0 \leq x_1, \ldots, x_m \leq 2^n - 1} \left( \sum_{1 \leq i \leq m} p_{x_i, y_i} \right) \ s(\lambda) \leq r(\lambda) \ .$$

Notice that UB bounds the sum over all possible combinations of column entries. In particular, it bounds the sum over the largest entries in each column, whose logarithm corresponds to the maximal leakage (see Theorem 1).

It follows immediately that the following formula is equivalent to Assertion 1) of Theorem 6

$$\forall \lambda \ \forall (p_{x,y}) : \ \lambda \geq 1 \Rightarrow \mathsf{VC} \Rightarrow \mathsf{DP} \Rightarrow \mathsf{UB}$$

In other words, the validity of a rational upper bound on the exponential of the leakage can be expressed in terms of polynomial inequalities, which concludes the proof for Assertion 1).

By a similar reasoning, one can establish that the following formula is equivalent to Assertion 2) of Theorem 6

$$\forall \lambda : \ \lambda \geq 1 \Rightarrow \exists (p_{x,y}) : \ \mathsf{VC} \wedge \mathsf{DP} \wedge \mathsf{EB}$$

where the formula EB indicates that the rational function is reached for some values.

$$\mathsf{EB} \equiv \bigvee_{0 \leq x_1, \ldots, x_m \leq 2^n - 1} \left( \sum_{1 \leq i \leq m} p_{x_i, y_i} \right) \ s(\lambda) = r(\lambda) \ .$$

■

A particular consequence of Theorem 6 is that, for every fixed $n$, it is decidable whether for every $\epsilon \geq 0$, $n \ \log_2 \frac{2e^\epsilon}{e^\epsilon + 1}$ is an upper bound of the leakage for $\epsilon$-differentially private channels.

## VII. Related work

We studied the relationship between two quantitative notions of confidentiality that have proposed in the context of two independent lines of research. The first line focuses on privacy-preserving data publishing; the second line focuses on information-flow analysis.

Privacy-preserving data publishing is a long-standing and well-studied problem, see e.g. the survey by Fung et al. [13]. Differential privacy by Dwork et al. [11] is the emerging consensus definition of privacy in the field. This is due to the fact that differential privacy enjoys desirable

properties such as independence of adversary knowledge, well-behavedness under composition of queries, and that it comes with enforcement mechanisms such as the Laplacian mechanism [12], [24], [30]. Despite the strong security guarantees it provides, differential privacy has proven to be useful in several applications [27], [33].

Information-theoretic notions of confidentiality have emerged from research on information-flow security and were initially targeted towards the analysis of covert channels [15], [25], [28]. More recently, they have also been applied for the analysis of side-channels in cryptographic algorithms [20], [21], [34], and for the quantitative analysis of anonymity protocols [6]. Moreover, a number of automatic analysis techniques have recently been proposed [3], [7], [18], [26], [29].

We are aware of two independent and concurrent studies that aim at understanding the connections between differential privacy and information-theoretic notions of confidentiality. Alvim et al. [2] perform an information-theoretic analysis of differential privacy mechanisms based on output pertubation, such as the Laplacian mechanism (see Section II-B). In particular, they give upper bounds for the mutual (min-)information between the outputs of the original query and the outputs of the perturbed query. Our analysis is more general in that it does not focus on a particular mechanism for achieving differential privacy; rather, it yields an information-theoretic characterization of the end-to-end confidentiality guarantees provided by any differentially private query. In more recent work, Alvim et al. [1] extend the model presented in the paper at hand to capture differentially private mechanisms with different metrics on the input domain. They also consider the question of bounds for the min-entropy leakage; moreover, they study the trade-off between leakage and utility.

Clarkson and Schneider [8] consider differential privacy in their study of quantitative integrity. In particular, they characterize differential privacy in terms of the mutual information between the output of a query and the input of the query, conditioned on knowledge of all but one individual in the input. This characterization bears resemblance to our analysis of the case $n = 1$ in Section IV; the main difference lies in the security guarantees provided and the notions of entropy used: Clarkson and Schneider give bounds on the transmission of information using Shannon entropy, whereas our approach gives bounds on the probability of guessing the input using min-entropy. Our approach goes beyond the leakage about one individual in that we give upper and lower bounds, together with decidability results, for the leakage of differentially private queries on arbitrary input domains.

## VIII. Conclusion

We performed an information-theoretic analysis of differential privacy. In particular, we established the first upper bounds for the min-entropy leakage of differentially private mechanisms and provided empirical evidence of their accuracy. On a practical level, our contributions pave the way for applying tools from differential privacy for bounding the leakage of programs. On a conceptual level, we unveiled a connection to coding theory, and we demonstrated how this connection can be leveraged for deriving precise bounds from covering codes.


## Acknowledgments

The authors would like to thank Miguel Andrés, Catuscia Palamidessi, and the anonymous reviewers for their helpful feedback.

This research was supported by FP7-ICT Project NESSoS (256980), by FP7-PEOPLE-COFUND Project AMAROUT (229599), FP7-ICT Project HATS (231620), DESAFIOS-10 (TIN2009-14599-C03-02), and by Comunidad de Madrid Program PROMETIDOS-CM (S2009TIC-1465).


## References


[1] M. S. Alvim, M. Andrés, K. Chatzikokolakis, P. Degano, and C. Palamidessi. Differential privacy: on the trade-off between utility and information leakage. *CoRR*, abs/1103.5188v1, 2011.

[2] M. S. Alvim, K. Chatzikokolakis, P. Degano, and C. Palamidessi. Differential privacy versus quantitative information flow. *CoRR*, abs/1012.4250v1, 2010.

[3] M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. IEEE Symp. on Security and Privacy (S&P '09)*, pages 141–153. IEEE, 2009.

[4] C. Braun, K. Chatzikokolakis, and C. Palamidessi. Quantitative notions of leakage for one-try attacks. *Electr. Notes Theor. Comput. Sci.*, 249:75–91, 2009.

[5] C. Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zürich, 1997.

[6] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Inf. Comput.*, 206(2-4):378–401, 2008.

[7] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[8] M. R. Clarkson and F. B. Schneider. Quantification of integrity. Cornell Computing and Information Science Technical Reports, 2011. http://hdl.handle.net/1813/22012.

[9] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*. Elsevier Science, 1997.

[10] G. E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, 1975.

[11] C. Dwork. Differential Privacy. In *Proc. 33rd Intl. Colloquium on Automata, Languages and Programming (ICALP '06)*, volume 4052 of *LNCS*, pages 1–12. Springer, 2006.

[12] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proc. 3rd Theory of Cryptography Conference (TCC '06)*, volume 3876 of *LNCS*, pages 265–284. Springer, 2006.

[13] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4), 2010.

[14] S. R. Ganta, S. P. Kasiviswanathan, and A. Smith. Composition attacks and auxiliary information in data privacy. In *Proc. 14th ACM Conference on Knowledge Discovery and Data Mining (KDD '08)*, pages 265–273. ACM, 2008.

[15] J. W. Gray. Toward a Mathematical Foundation for Information Flow Security. *Journal of Computer Security*, 1(3-4):255–294, 1992.

[16] S. Hamadou, V. Sassone, and C. Palamidessi. Reconciling belief and vulnerability in information flow. In *Proc. 31st IEEE Symposium on Security and Privacy (S&P '10)*, pages 79–92. IEEE Computer Society, 2010.

[17] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[18] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *26th Annual Computer Security Applications Conference, (ACSAC '10)*, pages 261–269. ACM, 2010.

[19] L. Hörmander. *The Analysis of Linear Partial Differential Operators II: Differential Operators with Constant Coefficients.* Springer, 1983.

[20] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proc. ACM Conf. on Computer and Communications Security (CCS '07)*, pages 286–296. ACM, 2007.

[21] B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *Proc. IEEE Computer Security Foundations Symposium (CSF '09)*, pages 324–335. IEEE, 2009.

[22] B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 3–14. IEEE, 2010.

[23] B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *Proc 23rd. IEEE Computer Security Foundations Symposium (CSF '10)*, pages 44–56. IEEE, 2010.

[24] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *Proc. 29th ACM Symposium on Principles of Database Systems (PODS '10)*, pages 123–134. ACM, 2010.

[25] G. Lowe. Quantifying Information Flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 18–31. IEEE, 2002.

[26] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI '08)*, pages 193–205. ACM, 2008.

[27] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. International Conference on Management of Data (SIGMOD '09)*, pages 19–30. ACM, 2009.

[28] J. K. Millen. Covert Channel Capacity. In *Proc. IEEE Symp. on Security and Privacy (S&P '87)*, pages 60–66. IEEE, 1987.

[29] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pages 73–85. ACM, 2009.

[30] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *Proc. 39th Annual ACM Symposium on Theory of Computing (STOC '07)*, pages 75–84. ACM, 2007.

[31] A. Rényi. On measures of entropy and information. In *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability 1960*, pages 547–561, 1961.

[32] A. Roth and T. Roughgarden. Interactive privacy via the median mechanism. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC '10)*, pages 765–774. ACM, 2010.

[33] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *Proc. 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, pages 297–312. USENIX Association, 2010.

[34] G. Smith. On the foundations of quantitative information flow. In *Proc. Intl. Conf. of Foundations of Software Science and Computation Structures (FoSSaCS '09)*, LNCS 5504, pages 288–302. Springer, 2009.

[35] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. Univ. of California Press, 2nd edition, 1951.

[36] X. Xiao, G. Wang, and J. Gehrke. Differential privacy via wavelet transforms. In *Proc. 26th International Conference on Data Engineering (ICDE '10)*, pages 225–236. IEEE, 2010.

## Appendix

**Theorem 1** ([23])**.** *The maximal leakage of a channel $P_{Y|X}$ can be computed by*

$$ML(P_{Y|X}) = \log_2 \sum_y \max_x P_{Y|X}(y, x) \ ,$$

*where the maximum is assumed (e.g.) for uniformly distributed input.*

*Proof:* Assume a fixed distribution $P_X$. Then

$$\begin{aligned} L &= \log_2 \frac{\sum_y P_Y(y) \max_x P_{X|Y}(x,y)}{\max_x P_X(x)} \\ &\overset{(*)}{=} \log_2 \frac{\sum_y \max_x (P_X(x) P_{Y|X}(y,x))}{\max_x P_X(x)} \\ &\overset{(**)}{\leq} \log_2 \frac{\sum_y \max_x P_{Y|X}(y,x)(\max_x P_X(x))}{\max_x P_X(x)} \\ &= \log_2 \sum_y \max_x P_{Y|X}(y,x)) \ , \end{aligned}$$

where (*) is Bayes' rule. Note that (**) is an equality if $P_X$ is uniformly distributed, from which the assertion follows. ∎

**Lemma 1.** *A channel $P_{Y|X}$ guarantees $\epsilon$-differential privacy if and only if, for all $x \in \{0,1\}^n$, all $i \in \{1,\ldots,n\}$, all distributions $P_{\dot{X}}$ of $\dot{X}$, and all $y \in Range(Y)$,*

$$P[\dot{X}=1] \leq e^{\epsilon} P[\dot{X}=1 \mid Y=y \wedge X=x[i/\dot{X}]] \ . \tag{5}$$

*Proof:* Assume that $P_{Y|X}$ is $\epsilon$-differentially private but does not satisfy (5). Then there are $x \in \{0,1\}^n$, $i \in \{0,\ldots,n\}$, $P_{\dot{X}}$ and $y \in Range(Y)$ such that

$$\frac{P[\dot{X}=1]}{P[\dot{X}=1 \mid Y=y \wedge X=x[i/\dot{X}]]} > e^{\epsilon}$$

Applying Bayes' rule to the denominator we obtain

$$\frac{P[Y=y \wedge X=x[i/\dot{X}]]}{P[Y=y \wedge X=x[i/\dot{X}] \mid \dot{X}=1]} > e^{\epsilon} \ . \tag{6}$$

We observe that the denominator in (6) is equivalent to

$$P[Y=y \wedge X=x[i/1]]$$

and conclude that

$$\frac{P[Y=y \wedge X=x[i/0]]}{P[Y=y \wedge X=x[i/1]]} > e^{\epsilon} \ . \tag{7}$$

Here (7) follows by observing that the numerator in (6) is equivalent to

$$P[\dot{X}=0]P[Y=y \wedge X=x[i/0]] + P[\dot{X}=1]\,P[Y=y \wedge X=x[i/1]] \ .$$

Note that (7) contradicts the assumption that $P_{Y|X}$ is $\epsilon$-differentially private because $x[i/0]$ and $x[i/1]$ differ only in one bit.

For the other direction, assume that $P_{Y|X}$ satisfies (5) but is not $\epsilon$-differentially private. I.e. there are $x \in \{0,1\}^n$, $i \in \{1,\ldots,n\}$, and $y \in Range(Y)$ such that

$$\frac{P[Y=y \mid X=x[i/0]]}{P[Y=y \mid X=x[i/1]]} > e^{\epsilon} \tag{8}$$

We obtain a contradiction by showing that

$$\frac{P[\dot{X}=1]}{P[\dot{X}=1 \mid Y=y \wedge X=x[i/\dot{X}]]} > e^{\epsilon} \tag{9}$$

for some distribution $P_{\dot{X}}$. To this end, observe that the left-hand side of (9) is equal to

$$\frac{P[Y=y \wedge X=x[i/\dot{X}]]}{P[Y=y \wedge X=x[i/1]]} \tag{10}$$

which is obtained by applying Bayes rule to the denominator of (9). We define $P_{\dot{X}}$ such that $P_{\dot{X}}(0) = \alpha$. Then

$$\begin{aligned} P[Y=y \wedge X=x[i/\dot{X}]] = (1-\alpha)\, &P[Y=y \wedge X=x[i/1]] \\ &+ \alpha\, P[Y=y \wedge X=x[i/0]] \end{aligned}$$

By inserting this expansion into (10) and applying (8) we obtain

$$\frac{P[Y=y \mid X=x[i/\dot{X}]]}{P[Y=y \mid X=x[i/1]]} > (1-\alpha) + \alpha \frac{P[Y=y \mid X=x[i/0]]}{P[Y=y \mid X=x[i/1]]}$$

which becomes larger than $e^{\epsilon}$ if $\alpha$ converges to 1 and concludes this proof. ∎

**Lemma 3.** *Let $C_i$ be channels that are $\epsilon_i$-differentially private ($i = 1, 2$). Then*

1) *$C_1 + C_2$ is $\epsilon_1 + \epsilon_2$-differentially private, and*
2) *$ML(C_1 + C_2) \leq ML(C_1) + ML(C_2)$.*

*Proof:* For 1), choose an arbitrary $(y_1, y_2) \in Y_1 \times Y_2$ and consider $x, x' \in X$ that differ exactly in one position. Then we have

$$\begin{aligned} &\left| \log \frac{P_{Y_1 \times Y_2|X}((y_1,y_2),x)}{P_{Y_1 \times Y_2|X}((y_1,y_2),x)} \right| \\ &= \left| \log \frac{P_{Y_1|X}(y_1,x) P_{Y_2|Y_1 \times X}(y_2,(y_1,x))}{P_{Y_1|X}(y_1,x') P_{Y_2|Y_1 \times X}(y_2,(y_1,x'))} \right| \\ &\leq \left| \log \frac{P_{Y_1|X}(y_1,x)}{P_{Y_1|X}(y_1,x')} \right| + \left| \log \frac{P_{Y_2|Y_1 \times X}(y_2,(y_1,x))}{P_{Y_2|Y_1 \times X}(y_2,(y_1,x))} \right| \\ &\leq \epsilon_1 + \epsilon_2 \ , \end{aligned}$$

from which the assertion follows directly. A proof of 2) is given in the body of the paper. ∎

**Lemma 4.** *Let $C_i$ be channels that are $\epsilon_i$-differentially private ($i = 1, 2$). Then*

1) *$C_1 \times C_2$ is $\max\{\epsilon_1, \epsilon_2\}$-differentially private, and*
2) *$ML(C_1 \times C_2) = ML(C_1) + ML(C_2)$.*

*Proof:* For the proof of 1), consider arbitrary $y = (y_1, y_2) \in Y_1 \times Y_2$ and $x = (x_1, x_2), x' = (x'_1, x'_2) \in X_1 \times X_2 = \{0,1\}^n$ that differ in one single bit. Consider first the case that the differing bit is in the first component, i.e. $x_1 \neq x'_1$ and $x_2 = x'_2$. Then

$$\begin{aligned} \left| \log \frac{P_{Y_1 \times Y_2|X}(y,x)}{P_{Y_1 \times Y_2|X}(y,x')} \right| &= \left| \log \frac{P_{Y_1|X_1}(y_1,x_1) P_{Y_2|X_2}(y_2,x_2)}{P_{Y_1|X_1}(y_1,x'_1) P_{Y_2|X_2}(y_2,x'_2)} \right| \\ &\overset{(*)}{=} \left| \log \frac{P_{Y_1|X_1}(y_1,x_1)}{P_{Y_1|X_1}(y_1,x'_1)} \right| \\ &\overset{(**)}{\leq} \epsilon_1 \ , \end{aligned}$$

where $(*)$ follows because $x_2 = x_2'$ and $(**)$ follows because $C_1$ is $\epsilon_1$-differentially private. For the case that $x_1 = x_1'$ and $x_2 \neq x_2'$, we obtain a symmetric bound in terms of $\epsilon_2$. Combining both bounds yields the assertion. A proof of 2) can be found in the body of the paper. ■

# Non-Uniform Distributions in Quantitative Information-Flow

Michael Backes
Saarland University &
MPI-SWS
Saarbrücken, Germany
backes@mpi-sws.org

Matthias Berg
Saarland University
Saarbrücken, Germany
berg@cs.uni-saarland.de

Boris Köpf
IMDEA Software
Madrid, Spain
boris.koepf@imdea.org

## ABSTRACT

Quantitative information-flow analysis (QIF) determines the amount of information that a program leaks about its secret inputs. For this, QIF requires an assumption about the distribution of the secret inputs. Existing techniques either consider the worst-case over a (sub-)set of all input distributions and thereby over-approximate the amount of leaked information; or they are tailored to reasoning about uniformly distributed inputs and are hence not directly applicable to non-uniform use-cases; or they deal with explicitly represented distributions, for which suitable abstraction techniques are only now emerging. In this paper we propose a novel approach for a precise QIF with respect to non-uniform input distributions: We present a reduction technique that transforms the problem of QIF w.r.t. non-uniform distributions into the problem of QIF for the uniform case. This reduction enables us to directly apply existing techniques for uniform QIF to the non-uniform case. We furthermore show that quantitative information flow is robust with respect to variations of the input distribution. This result allows us to perform QIF based on approximate input distributions, which can significantly simplify the analysis. Finally, we perform a case study where we illustrate our techniques by using them to analyze an integrity check on non-uniformly distributed PINs, as they are used for banking.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*; H.1.1 [**Models and Principles**]: Systems and Information Theory—*Information theory*

## General Terms

Security

## Keywords

Quantitative information flow

## 1. INTRODUCTION

The goal of an information-flow analysis is to keep track of sensitive information during computation. If a program does not expose any information about its secret inputs to unauthorized parties, it has secure information flow, a property that is often formalized as noninterference. In many cases, achieving noninterference is expensive, impossible, or simply unnecessary: Many systems remain secure as long as the amount of exposed secret information is sufficiently small. Consider for example a password checker. A failed login attempt reveals some information about the secret password. However, for well-chosen passwords, the amount of leaked information is so small that a failed login-attempt will not compromise the security of the system.

Quantitative information-flow analysis (QIF) is a technique for establishing bounds on the information that is leaked by a program. The insights that QIF provides go beyond the binary output of Boolean approaches, such as non-interference analyzers. This makes QIF an attractive tool to support gradual development processes, even without explicitly specified policies. Furthermore, because information-theory forms the foundation of QIF, the quantities that QIF delivers can be directly associated with operational security guarantees, such as lower bounds for the expected effort of uncovering secrets by exhaustive search.

Technically, a quantitative information-flow analysis requires an assumption about the probability distribution of the confidential inputs. Existing approaches deal with this assumption in four fundamentally different ways. We briefly present all four alternatives and discuss their implications on applicability and automation of quantitative information-flow analyses.

The first kind of approach focuses on computing the channel capacity, which is the maximum leakage with respect to all possible input distributions [7,8,17,23,24,27,31]. Maximizing over all possible input distributions is a safe, but often overly pessimistic assumption: Consider a password checker with two possible observable outcomes, `succeed` and `fail`. The capacity of the channel from secret passwords to observables is 1 bit, corresponding to a distribution that assigns probability 0.5 to both outcomes. A naive security analysis will infer that an $n$-bit password can be leaked in as few as $n$ login attempts and conclude that the system is insecure. However, if the passwords are well-chosen (e.g. drawn uniformly from a large set), a login attempt will reveal much less than one bit of information, which is the reason why the password checker is in fact secure.

The second kind of approach considers the maximum leakage with respect to a subset of possible input distributions, where the subset is specified in terms of bounds on the entropy of the input variables [9, 11]. While entropy bounds are an attractive way of specifying interesting subsets of input distributions, precise reasoning about the leakage of programs in terms of such bounds turns out to be challenging. In particular, deriving tight bounds for the leakage of programs with loops in terms of entropy bounds for their


*ASIACCS '11,* March 22–24, 2011, Hong Kong, China.

input variables is still an open problem.

The third kind of approach analyzes programs with respect to uniformly distributed inputs [2, 20, 22]. Under the uniformity assumption, computing the information-theoretic characteristics of deterministic programs can be reduced to computing the programs' preimages and determining their numbers or sizes [20]. It has been shown that these tasks can be performed using symbolic [2] and randomized algorithms [22], allowing one to analyze large state-spaces with precision guarantees. However, the applicability of those techniques has so far been restricted to domains with uniformly distributed inputs.

Finally, the fourth kind of approach analyzes programs with respect to an arbitrary, but fixed, probability distribution on the secret inputs [12, 25, 30]. The first automated approach delivers precise results [30], but is limited to programs with small state-spaces due to the explicit representation of the input distribution. An abstraction technique [29] addresses this problem by partitioning the (totally ordered) domain into intervals, on which a piecewise uniform distributions is assumed. However, it is an open problem how to choose the initial partition of the domain in order to allow for an analysis that is precise and efficient at the same time.

In summary, it has been an open problem to perform quantitative information-flow analysis with non-uniform distributions in a precise, scalable, and general way. In this paper, we make the following contributions towards this goal.

Our first contribution is a technique for reducing the problem of QIF with respect to non-uniform distributions to the problem of QIF with respect to uniform distributions. Our reduction enables one to directly apply existing tools for uniform QIF [2, 20, 22] to the non-uniform case. The main idea of the reduction is to represent a non-uniform distribution in terms of a generator program that receives uniformly distributed input and produces outputs according to the desired distribution. We exhibit and prove a connection between the information-flow properties of the target program and the sequential composition of the target program with the generator program. This connection enables us to analyze the composed program with respect to uniform inputs, and to draw conclusions about the information flow of the target program with respect to the non-uniform distribution. Our reduction is motivated by a number of examples that occur in practice. For example, the (non-uniformly distributed) PINs used in electronic banking are derived from uniform bit-strings, e.g., using decimalization techniques [13]. Another example are the keys of a public-key cryptosystem, which are typically produced by key generation algorithms that operate on uniformly distributed input.

Our second contribution is to show that QIF is robust with respect to small variations in the input distribution. This allows us to replace actual input distributions with approximate distributions. Based on the quality of the approximation, we give upper and lower bounds on the error this approximation introduces in the analysis. Focusing on approximate distributions can simplify the information-flow analysis, e.g. by allowing to replace "almost uniform" distributions by uniform distributions.

Finally, we give examples of how our two results can be used for the quantitative information-flow analysis of realistic systems. We use our reduction technique to estimate the information leaked by an integrity check on non-uniformly distributed PINs, and we use our robustness result to bound the error that is introduced by assuming uniformly distributed PINs.

The paper is organized as follows. In Section 2 we introduce basic notions of information flow. The reduction of non-uniform analysis to the uniform case is shown in Section 3. The robustness results are presented in Section 4. Section 5 contains our experiments where we apply our results to analyze an integrity check on non-uniformly distributed PINs. We present related work in Section 6 and conclude in Section 7.

# 2. PRELIMINARIES

## 2.1 Programs

A program $P = (I, F, R)$ is a triple consisting of a set of *initial states* $I$, a set of *final states* $F$, and a transition relation $R \subseteq I \times F$. We consider programs that implement total functions, i.e., we require that for all $s \in I$ there is exactly one $s' \in F$ with $(s, s') \in R$, and we use the shorthand notation $P(s) = s'$ for $(s, s') \in R$.

Given a final state $s' \in F$, we define its *preimage* $P^{-1}(s')$ to be the set of all input states from which $s'$ is reachable, i.e.,

$$P^{-1}(s') \quad \equiv \quad \{s \mid (s, s') \in R\} .$$

The preimage of an unreachable state is the empty set.

## 2.2 Qualitative Information Flow

We assume that the initial state of each computation is secret. We consider an attacker that knows the program, in particular its transition relation, and the final state of each computation.

We characterize partial knowledge about the elements of $I$ in terms of *partitions* of $I$, i.e., in terms of a family $\{B_1, \ldots, B_r\}$ of pairwise disjoint *blocks* such that $\bigcup_{i=1}^{r} B_i = I$. A partition of $I$ models that each $s \in I$ is known up to its enclosing block $B_i$. We compare partitions using the (im-)precision order $\sqsubseteq$ defined by

$$\begin{aligned} &\{B_1, \ldots, B_r\} \sqsubseteq \{B'_1, \ldots, B'_{r'}\} \\ &\quad \equiv \quad \forall i \in \{1, \ldots, r\}\ \exists j \in \{1, \ldots, r'\} : B_i \subseteq B'_j . \end{aligned}$$

The knowledge gained by an attacker about initial states of computations of the program $P$ by observing their final states is given by the partition $\Pi$ that consists of the preimages of reachable final states, i.e.,

$$\Pi \quad \equiv \quad \{P^{-1}(s') \mid s' \in F\} .$$

The partition $\{I\}$ consisting of a single block corresponds to the case where no information leaks, and $\{\{s\} \mid s \in I\}$ where each block is a singleton set captures the case that $P$ fully discloses its input. Partitions $\Pi$ with $\{\{s\} \mid s \in I\} \sqsubset \Pi \sqsubset \{I\}$ capture that $P$ leaks partial information about its input.

More generally, one can assume that initial and final states are pairs of *high* and *low* components, i.e., $I = I_H \times I_L$ and $F = F_H \times F_L$, and that the observer can access the low components of the initial and final states of a given computation. For a low input $l$ and a low output $l'$ we then define the *low-preimage* $P_l^{-1}(l')$ of $l'$ to be the set of all high components of input states with low component $l$, from which a final state with low component $l'$ is reachable, i.e.

$$P_l^{-1}(l') \quad \equiv \quad \{h \mid \exists h' \in F_H : ((h, l), (h', l')) \in R\} .$$

As before, we can characterize the knowledge an attacker gains about the high components of initial states states in terms of a partition of $I_H$. However, the exact shape of this partition strongly depends on the role of the low input. For example, when the low input is controlled by an attacker who can exhaustively run the program with all possible low inputs, then the knowledge the attacker gains about the high input is characterized by partition corresponding to the intersection of all low-preimages, i.e.,

$$\Pi \quad \equiv \quad \bigcap_{l \in I_L} \{P_l^{-1}(l') \mid l' \in F_L\} ,$$

where the intersection of partitions $\Pi_1, \Pi_2$ is defined by pairwise intersection of their blocks, i.e. $\Pi_1 \cap \Pi_2 = \{A \cap B \mid A \in \Pi_1, B \in \Pi_2\}$.

Several approaches in the literature consider weaker attackers, e.g. those that run the program with a single, fixed low input [25], or those that can observe a bounded number of program runs with adaptively chosen low inputs [20]. While the precise definition of $\Pi$ depends on the considered attacker model, the characterization of attacker knowledge in terms of a partition (or an equivalence relation) is universal.

The results of this paper rely only on such a partition-based characterization of attacker knowledge and can hence be used in conjunction with all of the aforementioned attacker models. For the sake of presentation, we focus on the simplest scenario, namely programs in which the entire initial state is high, and the entire final state is low (and hence $\Pi = \{P^{-1}(s') \mid s' \in F\}$).

## 2.3 Quantitative Information Flow

We use information theory to characterize the information that $P$ reveals about its input. This characterization has the advantage of being compact and easy to compare. Moreover, it yields concise interpretations in terms of the effort needed to determine $P$'s input from the revealed information, e.g., by exhaustive search.

We begin by introducing necessary notation. Let $A$ be a finite set and $p\colon A \to \mathbb{R}$ a probability distribution. For a random variable $X\colon A \to B$, we define $p_X\colon B \to \mathbb{R}$ as $p_X(x) = \sum_{a \in X^{-1}(x)} p(a)$, which we will also denote by $\Pr(X = x)$.

The *(Shannon) entropy* [34] $H(X) = -\sum_{x \in B} p_X(x) \log_2 p_X(x)$ of $X$ is a lower bound for the average number of bits required for representing the results of independent repetitions of the experiment associated with $X$. Thus, in terms of guessing, the entropy $H(X)$ is a lower bound for the average number of questions with binary outcome that need to be asked to determine $X$'s value [6]. Given another random variable $Y\colon A \to C$, we write $H(X|Y = y)$ for the entropy of $X$ given that the value of $Y$ is $y$. The *conditional entropy* $H(X|Y)$ of $X$ given $Y$ is the expected value of $H(X|Y = y)$ over all $y \in C$; it captures the remaining uncertainty about $X$ when $Y$ is observed.

For analyzing programs, we assume a probability distribution $p$ on $I$ and we suppose that it is known to the attacker. For analyzing the program $P$, we define two random variables. The first random variable $D\colon I \to I$ models the choice of an input in $I$, i.e., $D$ is the identity $D(s) = s$. The second random variable captures the input-output behavior of $P$. We overload notation and also denote it by $P$. Formally, we define $P\colon I \to F$ by $P(s) = s'$ whenever $(s, s') \in R$.

The conditional entropy $H(D|P)$ captures the remaining uncertainty about the program's input when the output is observed. We will use $H(D|P)$ as a measure of information flow in this paper, because it is associated with operational security guarantees: one can give lower bounds for the effort for determining a secret by exhaustive search in terms of $H(D|P)$, see [21, 26].

We mention for completeness that several approaches in the literature (e.g. [11, 25, 30]) focus on computing the *mutual information* $I(D; P)$ between the input and the output of a program, which is defined as the reduction in uncertainty about the input when the output is observed, i.e. $I(D; P) = H(D) - H(D|P)$. For given $H(D)$, the value of $I(D; P)$ can be immediately be derived from that of $H(D|P)$, and vice versa. We present our results in terms of the remaining uncertainty $H(D|P)$ because of its more direct connection to operational security guarantees.

# 3. NON-UNIFORM QUANTITATIVE INFORMATION-FLOW ANALYSIS

In this section we first show how to reduce the problem of QIF w.r.t. non-uniform distributions to the problem of QIF w.r.t. uniform distributions. We then show how this reduction allows us to leverage existing QIF techniques for programs with uniformly distributed inputs for the QIF-analysis of programs with arbitrarily distributed inputs.

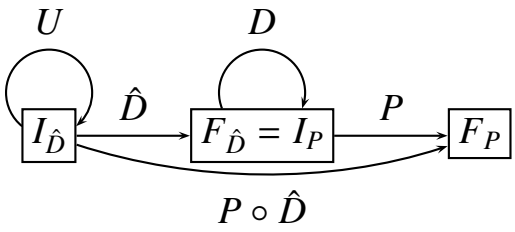



**Figure 1: Overview of the random variables required for reducing non-uniform QIF analysis to the uniform case. The input to $P$ is distributed according to the variable $D$. The input to $\hat{D}$ is given by the uniformly distributed random variable $U$; the output distribution of $\hat{D}$ matches that of $D$.**

## 3.1 Reducing the Non-uniform Case to the Uniform Case

The main idea behind our reduction is to express a probability distribution $p$ as a program $\hat{D}$ that takes input that is uniformly distributed and produces output that is distributed according to $p$. We prove an assertion that connects the remaining uncertainty about the inputs of the sequentially composed program $\hat{D}; P$ (with respect to uniform distributions) to the remaining uncertainty about the inputs of the program $P$ with respect to the distribution $p$.

Our reduction is motivated by a number of examples that occur in practice. For example, the Personal Identification Numbers (PINs) used in electronic banking are often not uniformly distributed, but derived from uniform bitstrings using decimalization techniques [13] (We will apply our techniques to analyze a program handling such PINs in Section 5). Another example are the keys of a public-key cryptosystem, which are typically not uniformly distributed bitstrings. However, they are produced by a key generation algorithm that operates on uniformly distributed input. More generally, a large number of randomized algorithms expect uniformly distributed randomness. E.g. in complexity theory, randomized algorithms are based on probabilistic Turing machines that work with uniformly distributed random tapes. For a language-based perspective on distribution generators, see [32].

Formally, let $P = (I_P, F_P, R_P)$ be a program and $p$ an arbitrary distribution on $I_P$. Let $\hat{D} = (I_{\hat{D}}, F_{\hat{D}}, R_{\hat{D}})$ be a program that maps to $P$'s initial states, i.e., $F_{\hat{D}} = I_P$, and let $u$ be the uniform distribution on $I_{\hat{D}}$. We require that the distribution produced by $\hat{D}$ matches the distribution on $P$'s inputs, i.e., $u_{\hat{D}} = p$. We define the random variables $D$ and $U$ as the identity functions on $I_P$ and $I_{\hat{D}}$, respectively, and we use them for modeling the choice of an input according to $p$ and $u$, respectively. Figure 1 depicts these mappings and their connections.

The setup is chosen such that the uncertainty about the output of the composed program $P \circ \hat{D}$ matches the uncertainty about the output of $P$, i.e. $H(P \circ \hat{D}) = H(P)$. Similarly, we have $H(\hat{D}) = H(D)$. As a consequence, we can express the remaining uncertainty about the input of $P$ in terms of a difference between the remaining uncertainties about the (uniformly distributed) inputs of $P \circ \hat{D}$ and $\hat{D}$.

Lemma 1. *Let $P, \hat{D}, D, U$ be as defined in Section 3.1. Then*

$$H(D|P) = H(U|P \circ \hat{D}) - H(U|\hat{D}).$$

Proof. The output of $P$ is determined by the output of $D$, namely $H(P, D) = H(D)$. Therefore it holds $H(D|P) = H(D) - H(P)$, and hence by construction

$$H(D|P) = H(\hat{D}) - H(P \circ \hat{D}) \,. \tag{1}$$

Similarly, the outputs of $\hat{D}$ and $P \circ \hat{D}$ are determined by the output of $U$, hence

$$H(U|P \circ \hat{D}) - H(U|\hat{D}) \;=\; H(U) - H(P \circ \hat{D}) - (H(U) - H(\hat{D})) \,. \tag{2}$$

The assertion then follows by combining (1) and (2). □

Lemma 1 shows how the remaining uncertainty about the (non-uniform) input of $P$ can be expressed as a difference of remaining uncertainties about (uniform) inputs of $\hat{D}$ and $P \circ \hat{D}$. In the following, we show how this result can be exploited for automating the quantitative information-flow analysis w.r.t. non-uniform distributions using established tools for uniform QIF.

## 3.2 Automation of QIF for Non-uniform Distributions

We summarize two kinds of techniques for automatically analyzing the information-flow of programs with respect to uniform distributions. The first kind of technique allows for the accurate, but possibly expensive QIF of a given program [2, 20], and the second kind of technique uses a randomized algorithm for obtaining approximate results with quality guarantees [22]. As we will show next, Lemma 1 allows one to use both kinds of techniques for analyzing programs with respect to non-uniform distribution.

### 3.2.1 Accurate Quantification

The following proposition from [20] connects the combinatorial characteristics of the partition $\Pi_Q$ induced by a program $Q$ with the remaining uncertainty about the uniformly distributed input of $Q$.

Proposition 1 (see [20]). *Let $U = id$ be uniformly distributed and let $Q$ be a program taking input distributed according to $U$. Then*

$$H(U|Q) \;=\; \frac{1}{\#(I_Q)} \sum_{B \in \Pi_Q} \#(B) \log_2 \#(B) \,.$$

Proposition 1 can be turned into an algorithm for computing $H(U|Q)$: Enumerate all blocks $B$ in the partition $\Pi_Q$, determine their sizes, and use these data for computing $H(U|Q)$. The algorithm described in [20] uses this approach for a partition $\Pi$ that reflects the knowledge gained by an attacker that can adaptively provide input to the program. The algorithm described in [2] extracts a logical representation of $\Pi$ by computing weakest preconditions and employs model counting techniques for determining the sizes of individual blocks from this logical representation.

The following theorem enables us to directly apply both techniques to programs with non-uniform input distributions.

Theorem 1. *Let $P, \hat{D}, D$ be as defined in Section 3.1. Then*

$$H(D|P) = \frac{1}{\#(I_{\hat{D}})} \left( \sum_{B \in \Pi_{P \circ \hat{D}}} \#(B) \log_2 \#(B) - \sum_{B' \in \Pi_{\hat{D}}} \#(B') \log_2 \#(B') \right).$$

Proof. The statement is obtained by applying Proposition 1 to both terms on the right hand side of Lemma 1. □

### 3.2.2 Randomized Quantification

The direct computation of $H(D|P)$ on basis of Theorem 1 requires the enumeration of all blocks in the partitions $\Pi_P$ and $\Pi_{P \circ \hat{D}}$, respectively. Each partition may have as many elements as $I_P$, which severely limits scalability. The following proposition from [22] is an extension of a result from [3] and addresses this limitation: it implies that, for uniformly distributed inputs, one can give tight bounds for $H(U|Q)$ by considering only a small subset of randomly chosen blocks.

Proposition 2 (see [22]). *Let $U = id$ be uniformly distributed and let $Q$ be a program taking input distributed according to $U$. Let $B_1, \ldots, B_n$ be drawn randomly from $\Pi_Q$ with respect to the distribution $p(B) = \frac{\#(B)}{\#(I_Q)}$. Then*

$$\frac{1}{n} \sum_{i=1}^{n} \log \#(B_i) - \delta \;\leq\; H(U|Q) \;\leq\; \frac{1}{n} \sum_{i=1}^{n} \log \#(B_i) + \delta$$

*holds with probability of more than $1 - \frac{(\log \#(\Pi_Q))^2}{n\delta^2}$.*

As described in [22], Proposition 2 can be turned into a randomized algorithm for quantitative information-flow analysis. To this end, observe that the random choice of blocks can be implemented by executing the program on a (uniformly chosen) input $s \in I_Q$ and determining the preimage $B = Q^{-1}(s')$ of $s' = Q(s)$. If this preimage is represented by a logical assertion, one can compute the size $\#(B)$ of $B$ using model counting techniques [15]. In this way, $H(U|Q)$ can be approximated with high confidence levels using a number of samples $n$ that is only polylogarithmic in the size of the state space.

The following theorem enables us to leverage these techniques for analyzing programs with non-uniform inputs.

Theorem 2. *Let $P, \hat{D}, D$ be as defined in Section 3.1. Let $B_1, \ldots, B_n$ be drawn randomly from $\Pi_{P \circ \hat{D}}$ and let $B'_1, \ldots, B'_n$ be drawn randomly from $\Pi_{\hat{D}}$ with respect to the distribution $p(B) = \frac{\#(B)}{\#(I_{\hat{D}})}$. Then*

$$\frac{1}{n} \sum_{i=1}^{n} \log \frac{\#(B_i)}{\#(B'_i)} - 2\delta \;\leq\; H(D|P) \;\leq\; \frac{1}{n} \sum_{i=1}^{n} \log \frac{\#(B_i)}{\#(B'_i)} + 2\delta$$

*with a probability of more than $\left(1 - \frac{(\log \#(\Pi_{\hat{D}}))^2}{n\delta^2}\right)^2$.*

Proof. Apply Proposition 2 to both terms on the right hand side of Lemma 1. For the confidence levels, observe that the blocks $B_i$ are drawn independently from the blocks $B'_i$, hence the probabilities that the inequalities hold multiply. Observing that $\#(\Pi_{\hat{D}}) \geq \#(\Pi_{P \circ \hat{D}})$, we replace the larger probability by the smaller one, which concludes this proof. □

Finally, the exact computation of the blocks $B_i$ can be prohibitively expensive. Fortunately, one can avoid this expensive computation by resorting to under- and over-approximations $\underline{B_i}$ and $\overline{B_i}$ of $B_i$, i.e. subsets of initial states with $\underline{B_i} \subseteq B_i \subseteq \overline{B_i}$. The computation of $\underline{B_i}$ and $\overline{B_i}$ can be done using existing techniques for symbolic execution and abstract interpretation, see [22]. In this paper, we simply assume the existence of such approximations.

Corollary 1. *Let $B_1, \ldots, B_n$ and $B'_1, \ldots, B'_n$ be chosen as in Theorem 2. Let $\underline{B_i} \subseteq B_i \subseteq \overline{B_i}$ and $\underline{B_i'} \subseteq B'_i \subseteq \overline{B'_i}$, for all $i \in \{1, \ldots n\}$. Then*

$$\frac{1}{n} \sum_{i=1}^{n} \log \frac{\#(\underline{B_i})}{\#(\overline{B'_i})} - 2\delta \;\leq\; H(D|P) \;\leq\; \frac{1}{n} \sum_{i=1}^{n} \log \frac{\#(\overline{B_i})}{\#(\underline{B'_i})} + 2\delta$$

*with a probability of more than $\left(1 - \frac{(\log \#(\Pi_{\hat{D}}))^2}{n\delta^2}\right)^2$.*

Corollary 1 follows directly from Theorem 2 by replacing all blocks that occur in the numerator (denominator) of the right (left) hand side and on the denominator (numerator) on the left (right) hand side by their over-(under-)approximating counterparts.

# 4. ROBUSTNESS

In this section, we show that the remaining uncertainty about a secret is robust with respect to small variations in the input distribution. This allows us to replace actual input distributions with approximate distributions. Based on the quality of the approximation, we give upper and lower bounds on the error this approximation introduces in the analysis. Focusing on approximate distributions can simplify the information-flow analysis, e.g. by allowing to replace "almost uniform" distributions by uniform distributions.

We say that two distributions $p$ and $q$ on some set $S$ are $\gamma$-*close* if the probabilities they assign to each value differ at most by a factor of $\gamma$.

$$p \overset{\gamma}{\approx} q \quad \equiv \quad \forall x \in S : \ \frac{1}{\gamma}\, q(x) \leq p(x) \leq \gamma\, q(x)$$

In the following we will consider a random variable with respect to different probability distributions on its input domain. We introduce the notation $X^p$ emphasize that we consider variable $X$ with respect to the underlying distribution $p$. The following lemma states that, for $\gamma$-close distributions $p$ and $q$, the distributions of $X^p$ and $X^q$ are also $\gamma$-close.

Lemma 2. *Let $X$ be a random variable and let $p$ and $q$ be distributions on the domain of $X$. Then $p \overset{\gamma}{\approx} q$ implies $p_{X^p} \overset{\gamma}{\approx} p_{X^q}$.*

Proof. For all $x$ we have

$$\frac{1}{\gamma} \cdot p_{X^q}(x) = \sum_{a \in X^{-1}(x)} \frac{1}{\gamma} \cdot q(a) \leq \sum_{a \in X^{-1}(x)} p(a) = p_{X^p}(x)\,.$$

The proof for the upper bound follows along the same lines. □

We next show that the entropy of a random variable is robust with respect to small changes in its input distribution. Formally, we show that for two random variables $X$ and $Y$ with $\gamma$-close distributions, i.e., $p_X \overset{\gamma}{\approx} p_Y$, the Shannon entropy $H(X)$ can be bounded in terms of the entropy $H(Y)$.

Lemma 3. *Let $X$ and $Y$ be random variables with $p_X \overset{\gamma}{\approx} p_Y$. Then we have*

$$H(X) \begin{cases} \leq \gamma \cdot H(Y) + \gamma \log_2 \gamma \\ \geq \frac{1}{\gamma} \cdot H(Y) - \frac{\log_2 \gamma}{\gamma} \end{cases} .$$

Proof. $H(X) = -\sum_x p_X(x) \log_2 p_X(x) \overset{(*)}{\leq} -\sum_x \gamma \cdot p_Y(x) \log_2 \frac{p_Y(x)}{\gamma} = \gamma \cdot H(Y) + \gamma \log_2 \gamma,$
where $(*)$ follows from $p_X \overset{\gamma}{\approx} p_Y$. The proof of the lower bound is analogous. □

We can use Lemma 3 together with Lemma 2 to obtain bounds on the remaining uncertainty of a program for distribution $p$ from an analysis with respect to a distribution $q$ with $p \overset{\gamma}{\approx} q$.

Theorem 3. *Let $p$ and $q$ be distributions with $p \overset{\gamma}{\approx} q$. Then we have*

$$H(D^p|P^p) \begin{cases} \leq \gamma \cdot H(D^q) - \frac{1}{\gamma} \cdot H(P^q) + \log_2 \gamma \left(\gamma + \frac{1}{\gamma}\right) \\ \geq \frac{1}{\gamma} \cdot H(D^q) - \gamma \cdot H(P^q) - \log_2 \gamma \left(\gamma + \frac{1}{\gamma}\right) \end{cases} .$$

Proof. Observe that $H(D^p|P^p) = H(D^p) - H(P^p)$ because $P^p$ is determined by $D^p$. Since $p \overset{\gamma}{\approx} q$, Lemma 2 yields $p_{D^p} \overset{\gamma}{\approx} p_{D^q}$. Applying Lemma 3 yields the assertion. □

In Section 5.2 we show an application of Theorem 3, where we analyze a program with respect to a uniformly distributed $q$ in order to derive bounds for the real, almost uniform, distribution $p$.

# 5. CASE STUDY

In this section we illustrate the techniques described in the previous sections. Namely, we will give an example of how an analysis with respect to non-uniform distributions can be reduced to the uniform case, which we handle using existing tool support [20]. Furthermore, we give an example of how our robustness result can be used to estimate the error introduced by replacing in the analysis an almost uniform distribution by a uniform one.

We consider a program for checking the integrity of Personal Identification Numbers (PINs) as used in electronic banking. Previous formal analyses of this program [19, 37] assume uniformly distributed PINs; they are not fully accurate because PIN generation methods typically produce a skewed distribution. Using the techniques presented in this paper, we perform the first formal analysis that takes this skew into account.

We analyze the integrity check with respect to PINs that stem from two different PIN generation algorithms. The first generation algorithm is easily expressed as a program, and we will use the techniques developed in Section 3 to perform a precise non-uniform QIF. The second generation algorithm produces PINs that are almost uniformly distributed, and we will use the techniques developed in Section 4 to perform an approximate QIF of the integrity check program. We begin by describing the integrity check and its use in practice.

## 5.1 PIN Integrity Check

When a customer authenticates himself at an Automated Teller Machine (ATM), he enters his PIN. This PIN is then sent to his bank for verification [1, 4]. Before sending, the PIN is XORed with the customer's Personal Account Number (PAN) and encrypted using a symmetric cryptosystem. In case the ATM cannot communicate directly with the customer's bank, the encrypted PIN⊕PAN will pass through a series of switches. Each of these switches decrypts and extracts the PIN, checks it for integrity, and re-encrypts the PIN using a key that is shared with the next switch. All operations on the PIN are performed in dedicated tamper-resistant Hardware Security Modules (HSMs), which protect the communication keys and the PINs even if the switch is compromised.

Unfortunately, HSMs fail to fulfill this purpose because the outcome of the PIN integrity check leaks information about the value of the PIN [13]: Upon receiving an encrypted pin, the HSM decrypts and XORs the result with a given account number to extract the PIN. The HSM then performs a simple integrity check on the PIN, namely it checks whether all PIN digits are $< 10$. Clearly, this check will succeed if the given account number is the customers PAN. However, the protocol does not forbid the use of the integrity check with an arbitrary account number PAN', in which case the HSM will reveal whether PIN⊕PAN⊕PAN' is a valid PIN. As the PAN itself is not secret, the integrity check can be seen as an oracle that, on input $m$, reveals whether all digits of PIN⊕$m$ are $< 10$.

We model the integrity check of a single PIN digit as a program $P = (I_P, F_P, R_P)$ with $I_P = \{0, \ldots, 9\}$, $F_P = \{0, 1\}$, and

$$P(s) \equiv s \oplus m < 10\,,$$

where $m \in \{0, \ldots, F\}$ is fixed. For example, for $m = F$, the integrity check is equivalent to the condition $s \geq 6$.

## 5.2 Non-uniform PINs from Decimalization Tables

The PIN generation algorithm described in [5] works as follows: In a first step the customer's account number is encrypted using DES under a fixed PIN derivation key. In a second step, the ciphertext is converted into a hexadecimal number. The first 4 digits of this number are taken and decimalized. The decimalization leaves digits $0-9$ unchanged, and maps $A-F$ to $0-5$.

We assume DES to be an ideal cipher, i.e., a random permutation. This assumption is known as the *Ideal Cipher Model* [35] and is commonly used in cryptography to abstract from the inner details of block ciphers. In this model, the output of the DES encryption is uniformly distributed and we can characterize the skew of the PIN as described in Section 3. More precisely, we capture the PIN generation algorithm as a program $\hat{D}$ which, given uniformly distributed input (i.e. the result of encrypting the PAN with DES), computes a PIN as described above. The purpose of this section is to illustrate our reduction to uniform distributions. For clarity of presentation, we will focus on a simplified scenario with one-digit PINs, i.e., $I_{\hat{D}} = \{0, \ldots, F\}$, $F_{\hat{D}} = \{0, \ldots, 9\}$, and

$$\hat{D}(s) = s \bmod 10$$

For computing $H(D|P)$ for $m = F$ using Theorem 1, we need to determine the partitions $\Pi_{\hat{D}}$ and $\Pi_{P \circ \hat{D}}$ induced on $I_{\hat{D}}$ by $\hat{D}$ and $P \circ \hat{D}$, respectively. It is easy to see that $\Pi_{\hat{D}}$ consists of blocks of values that are equal modulo 10 and that $\Pi_{P \circ \hat{D}}$ combines the blocks from $\Pi_{\hat{D}}$ with values $< 6$ or $\geq 6$ modulo 10, respectively.

$$\Pi_{\hat{D}} = \{\{0, A\}, \{1, B\}, \{2, C\}, \{3, D\}, \{4, E\}, \{5, F\}, \{6\}, \{7\}, \{8\}, \{9\}\}$$
$$\Pi_{P \circ \hat{D}} = \{\{0, 1, 2, 3, 4, 5, A, B, C, D, E, F\}, \{6, 7, 8, 9\}\}$$

We apply Theorem 1 to this data and obtain

$$\begin{aligned} & H(D|P) \\ &= \frac{1}{\#(I_{\hat{D}})} \left( \sum_{B \in \Pi_{P \circ \hat{D}}} \#(B) \log_2 \#(B) - \sum_{B' \in \Pi_{\hat{D}}} \#(B') \log_2 \#(B') \right) \\ &= \frac{1}{16} \left( (12 \log_2(12) + 4 \log_2(4)) - (6 \cdot 2 \log_2(2) + 4 \cdot 1 \log_2(1)) \right) \\ &\approx 2.4387 \end{aligned}$$

This result shows that, after a integrity check with account number PAN$\oplus F$, there are 2.4387 bits of uncertainty left about a single digit. A sightly more complex analysis (whose details we omit) reveals that, for a 4 pin digit, one check with PAN$\oplus FFFF$ leaves 12.9631 bits of uncertainty.

## 5.3 Automated Analysis of Adaptive Attacks

The analysis presented above considers an adversary that performs the PIN integrity check using a single fixed input $m = F$. In practice, however, an attacker can repeatedly perform PIN integrity checks with different values of $m$, thereby further narrowing down the possible values of the PIN. For assessing the security of a system it is necessary to take such repeated queries into account. We briefly report on experimental results where we use existing automated techniques for reasoning about this kind of attack.

The basis for our analysis is the formal model for knowledge refinement in adaptive attacks described in [19, 20]. In this model, each attack strategy induces a partition on the set of secret inputs. An attack strategy of $n$ steps is *optimal* if the remaining uncertainty about the secret after an attack is minimal among all possible attack strategies of $n$ steps. Here, the remaining uncertainty is computed from the attack strategy's induced partition using Proposition 1. As a consequence, the automatic tool presented in [19, 20] requires that the inputs are uniformly distributed. Using our reduction techniques, we leverage this restriction: We can simply apply the tool to $P \circ \hat{D}$ and obtain $H(U|P \circ \hat{D})$. We then use Theorem 1 to obtain $H(D|P)$ from $H(U|P \circ \hat{D})$ and $H(U|\hat{D})$ (which is a constant). The results of the analysis are depicted in Figure 2. The value of $H(\hat{D}|P) = 1$ in the last row accounts for the fact that a single PIN digit can can be narrowed down to two equally likely alternative values.

## 5.4 Almost Uniform PINs

The Interbank PIN generation algorithm [18] works as follows.[1] In a first step, the PAN is encrypted, yielding a string of 16 hexadecimal numbers. This string is scanned from left to right, and the first four decimal digits that are encountered are used as the PIN. If there are less than 4 decimal digits in the string, A is subtracted from each digit in the string, and the process is repeated until four decimal digits are found. This second scan ignores the positions in the string that already yielded decimal digits in the first round. [2]

We use the techniques presented in Section 4 for analyzing the PIN integrity check with respect to PINs that are generated according to the Interbank PIN generation algorithm. To this end, we first determine $\gamma$ such that the PIN distribution is $\gamma$-close to the uniform distribution. Then we perform a uniform analysis of the PIN integrity check and use Theorem 3 with $\gamma$ to estimate the uncertainty about a PIN drawn from the skewed distribution.

For an analysis of the Interbank PIN distribution, let the random variable $X$ denote the number of decimal digits in the hexadecimal string obtained by encrypting the account number. We assume this string to be uniformly distributed. $\Pr(X = k)$ can be calculated as follows:

$$\Pr(X = k) = \binom{16}{k} \left(\frac{5}{8}\right)^k \left(\frac{3}{8}\right)^{16-k}$$

The probability that there are at least 4 decimal digits in the string is

$$\Pr(X \geq 4) = 1 - \sum_{k=0}^{3} \Pr(X = k) \approx 0.9995 \,,$$

in which case the resulting PINs are uniformly distributed.

Let the random variable $Y$ denote the output of the generation algorithm, i.e., the generated PIN. Then

$$\begin{aligned} \Pr(Y = a) &= \Pr(Y = a | X \geq 4) \Pr(X \geq 4) + \Pr(Y = a | X \leq 3) \Pr(X \leq 3) \\ &= \Pr(X \geq 4) \cdot 10^{-4} + \underbrace{\Pr(Y = a | X \leq 3)}_{\leq 6^{-4}} \Pr(X \leq 3) \end{aligned}$$

We set $\gamma = \Pr(X \geq 4) + \left(\frac{5}{3}\right)^4 \Pr(X \leq 3) \approx 1.003$. A simple calculation shows that $\frac{1}{\gamma} < \Pr(X \geq 4)$, which gives us the following lower bound on $\Pr(Y = a)$:

$$\Pr(Y = a) \geq \Pr(X \geq 4) \cdot 10^{-4} > \frac{1}{\gamma} \cdot 10^{-4}$$

Similarly, we can obtain an upper bound on $\Pr(Y = a)$ as follows:

$$\Pr(Y = a) \leq \Pr(X \geq 4) \cdot 10^{-4} + \Pr(X \leq 3) \cdot 6^{-4} = \gamma \cdot 10^{-4}.$$

[1] We thank Graham Steel for pointing us to this example.
[2] Additionally, a PIN of 0000 is replaced by 0100. We will ignore this detail in our analysis.

| #Steps | $\Pi_{P \circ \hat{D}}$ | $H(U\|P \circ \hat{D})$ | $H(\hat{D}\|P)$ |
|---|---|---|---|
| 0 | `[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]]` | 4 | 3.25 |
| 1 | `[[0,1,8,9,10,11],[2,3,4,5,6,7,12,13,14,15]]` | 3.05 | 2.30 |
| 2 | `[[0,1,10,11],[8,9],[2,3,12,13],[4,5,6,7,14,15]]` | 2.09 | 1.34 |
| 3 | `[[0,1,10,11],[8,9],[2,3,12,13],[4,5,14,15],[6,7]]` | 1.75 | 1.0 |

**Figure 2: Results of automatically analyzing the PIN integrity check with respect to multiple runs of an adaptive attacker.**

We conclude that the distribution of the PINs is $\gamma$-close to the uniform distribution $u$, i.e., $p_Y \overset{\gamma}{\approx} u$.

Consider again the PIN integrity check program $P$ from Section 5.1, generalized to 4 PINs, i.e., $I_P = \{0, \ldots, 9\}^4$, $F_P = \{0, 1\}$, and

$$P(s_1 s_2 s_3 s_4) = \bigwedge_{i=1\ldots4} s_i \oplus m_i < 10$$

Theorem 3 gives the following formula to bound $H(D^{p_Y}|P^{p_Y})$:

$$H(D^{p_Y}|P^{p_Y}) \leq \gamma \cdot H(D^u) - \frac{1}{\gamma} \cdot H(P^u) + \log_2 \gamma\left(\gamma + \frac{1}{\gamma}\right)$$

We set $m = FFFF$, which results in the following upper bound:

$$\begin{aligned} &H(D^{p_Y}|P^{p_Y}) \\ &\leq \gamma \cdot \log_2 10^4 + \frac{1}{\gamma} \cdot \left(\frac{4^4}{10^4} \log_2 \frac{4^4}{10^4} + \frac{10^4 - 4^4}{10^4} \log_2 \frac{10^4 - 4^4}{10^4}\right) \\ &\quad + \log_2 \gamma\left(\gamma + \frac{1}{\gamma}\right) \\ &\approx 13.1654 \end{aligned}$$

A lower bound of $H(D^{p_Y}|P^{p_Y}) \geq 13.0664$ follows along the same lines.

The small delta between the upper and lower bounds shows that the uniform analysis is (almost) precise for PINs generated according to the Interbank algorithm. We conclude by comparing this result with the result of the analysis with respect to PINs generated using decimalization tables presented in Section 5.2 where, for 4 digit PINs, we obtained a remaining uncertainty of 12.9631 bits. The difference between the remaining uncertainties gives an account of the security that is gained by using a better (i.e., less skewed) PIN generation algorithm.

# 6. RELATED WORK

Denning is the first to quantify information flow in terms of the reduction in uncertainty about a program variable [14]. Millen [28] and Gray [16] use information theory to derive bounds on the transmission of information between processes in multi-user systems. Lowe [24] shows that the channel capacity of a program can be over-approximated by the number of possible behaviors. The channel capacity corresponds to the maximal leakage w.r.t. to any input distribution and hence is an over-approximation of the information that is actually revealed.

Clark, Hunt, and Malacaria [10] connect equivalence relations to quantitative information flow, and propose the first type system for statically deriving quantitative bounds on the information that a program leaks [11]. The analysis assumes as input (upper and lower) bounds on the entropy of the input variables and delivers corresponding (upper and lower) bounds for the leakage of the program. For loops with high guards, the analysis always reports complete leakage of the guard.

Malacaria [25] shows how to characterize the leakage of loops in terms of the loop's output and the number of iterations. Closely related on this approach, Mu and Clark [30] propose a precise, automatic QIF based on a distribution transformer semantics, which can deal with non-uniform input distributions. Their approach relies on an explicit representation of the probability distribution transformed by the program (and hence the set of initial states), which prevents the direct application to programs with large state spaces. The problem is mitigated by an interval-based abstraction proposed in [29]. The abstraction splits a totally ordered domain into intervals, each of which assumed to be uniformly distributed. In our approach, the probability distribution is represented in terms of preimages of a generating program, which offers the possibility of a symbolic treatment of large state spaces.

Köpf and Basin [20] show how to compute partitions on the secret input that represent what an attacker can learn in an adaptive attack. Backes, Köpf, and Rybalchenko [2] show how to determine the partitions corresponding to the information (with respect to a non-adaptive attacker) that a program leaks by computing weakest preconditions. Both approaches rely on counting the number and the sizes of the preimages in order to quantify the remaining uncertainty about the input w.r.t. uniform distributions. When used in conjunction with these approaches, the ideas presented in this paper can be used to weaken the requirement of a uniform distribution.

Köpf and Rybalchenko [22] propose approximation and randomization techniques to approximate the remaining uncertainty about a program's inputs for programs with unbounded loops. Their approach relies on approximating the sizes of blocks (but without their complete enumeration) and it delivers bounds w.r.t. uniformly distributed inputs. As we have shown, the reduction presented in this paper can be used for extending the techniques to programs with non-uniform input distributions.

McCamant and Ernst propose a dynamic taint analysis for quantifying information flow [27]. Their method does not assume a particular input distribution and provides over-approximations of the leaked information along a particular path. However, it does not yield guarantees for all program paths, which is important for security analysis. Newsome, McCamant, and Song [31] also use the feasible outputs along single program paths as bounds for channel capacity (i.e. the maximal leakage w.r.t. to all possible input distributions), and they apply a number of heuristics to approximate upper bounds on the number of reachable states of a program.

Chatzikokolakis, Chothia, and Guha [7] use sampling to build up a statistical system model. Based on this model, they compute the channel capacity, i.e. the maximum leakage w.r.t. all possible input distributions.

DiPierro, Hankin, and Wiklicky [33] consider probabilistic processes with given input distributions and (instead of information theory) use the distance of the produced output distributions to quantify information flow.

Clarkson, Myers, and Schneider [12] use non-uniform input distributions to model adversaries beliefs, which they update according to the program semantics. They do not discuss techniques for automation or abstraction.

Smith [36] proposes min-entropy as an alternative measure of

information flow. Min-entropy gives bounds on the probability of guessing a secret in one attempt, whereas Shannon-entropy gives bounds on the average number of guesses required for determining a secret. The investigation of a reduction from non-uniform to uniform QIF for min-entropy remains future work.

# 7. CONCLUSIONS AND FUTURE WORK

We have considered the problem of quantifying the information-flow in programs with respect to non-uniform input distributions. We have made the following contributions to solve the problem. First, we have shown how the problem of non-uniform QIF can be reduced to the uniform case. To this end, we represented the non-uniform input distribution as a program that receives uniform input, and we sequentially composed it with the target program. We have proved a connection between the information-theoretic characteristics of the target program and its composition with the distribution generator. This connection enables us to perform a precise non-uniform analysis using existing QIF techniques for the uniform case. Second, we have shown that the result of a QIF is robust with respect to small variations in the input distribution. This result shows that we can estimate the information-theoretic characteristics of a program by considering an approximate input distribution. This is useful in cases where the input distribution can only be approximated or an approximation simplifies the analysis. Finally, we have performed a case-study where we illustrated both techniques and demonstrated their usefulness in practice.

*Acknowledgments.*

Boris Köpf's research was partially done while at MPI-SWS and is partially supported by FP7-ICT Project NESSoS (256980), by FP7-PEOPLE-COFUND Project AMAROUT (229599), and by Comunidad de Madrid Program PROMETIDOS-CM (S2009TIC-1465).

# 8. REFERENCES

[1] American National Standards Institute. Banking - Personal Identification Number Management and Security - Part 1: PIN protection principles and techniques for online PIN verification in ATM & POS systems. ANSI X9.8-1, 2003.

[2] M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. 30th IEEE Symposium on Security and Privacy (S& P '09)*, pages 141–153. IEEE, 2009.

[3] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld. The complexity of approximating entropy. In *Proc. 34th Symposium on the Theory of Computing (STOC '02)*, pages 678–687. ACM, 2002.

[4] O. Berkman and O. M. Ostrovsky. The unbearable lightness of pin cracking. In *Financial Cryptography (FC '07)*, volume 4886 of *LNCS*, pages 224–238. Springer, 2008.

[5] M. Bond and P. Zieliński. Decimalisation table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, Feb. 2003.

[6] C. Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zürich, 1997.

[7] K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical Measurement of Information Leakage. In *Proc. 16th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, LNCS 6015, pages 390–404. Springer, 2010.

[8] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. *Inf. Comput.*, 206(2-4):378–401, 2008.

[9] D. Clark, S. Hunt, and P. Malacaria. Quantitative Analysis of the Leakage of Confidential Data. *Electr. Notes Theor. Comput. Sci.*, 59(3), 2001.

[10] D. Clark, S. Hunt, and P. Malacaria. Quantitative Information Flow, Relations and Polymorphic Types. *J. Log. Comput.*, 18(2):181–199, 2005.

[11] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[12] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in Information Flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '05)*, pages 31–45. IEEE, 2005.

[13] J. Clulow. The Design and Analysis of Cryptographic Application Programming Interfaces for Security Devices. Master's thesis, University of Natal, SA, 2003.

[14] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[15] C. Gomez, A. Sabharwal, and B. Selman. Chapter 20: Model counting. In *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[16] J. W. Gray. Toward a Mathematical Foundation for Information Flow Security. *Journal of Computer Security*, 1(3-4):255–294, 1992.

[17] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proc. Annual Computer Security Applications Conference (ACSAC '10)*. ACM, 2010.

[18] IBM Corporation. Interbank pin generation algorithm. `https://publib.boulder.ibm.com/infocenter/zos/v1r9/topic/com.ibm.zos.r9.csfb400/inbkal.htm`.

[19] B. Köpf and D. Basin. Automatically Deriving Information-theoretic Bounds for Adaptive Side-channel Attacks. *Journal of Computer Security (to appear)*.

[20] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proc. ACM Conference on Computer and Communications Security (CCS '07)*, pages 286–296. ACM, 2007.

[21] B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *Proc. 22rd IEEE Computer Security Foundations Symposium (CSF '09)*, pages 324–335. IEEE, 2009.

[22] B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 3–14. IEEE, 2010.

[23] B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF '10)*, pages 44–56. IEEE, 2010.

[24] G. Lowe. Quantifying Information Flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 18–31. IEEE, 2002.

[25] P. Malacaria. Risk assessment of security threats for looping constructs. *Journal of Computer Security*, 18(2):191–228, 2010.

[26] J. L. Massey. Guessing and Entropy. In *Proc. IEEE International Symposium on Information Theory (ISIT '94)*, page 204. IEEE, 1994.

[27] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. Conf. on Programming*

*Language Design and Implementation (PLDI '08)*, pages 193–205. ACM, 2008.

[28] J. K. Millen. Covert Channel Capacity. In *Proc. IEEE Symposium on Security and Privacy (S&P '87)*, pages 60–66. IEEE, 1987.

[29] C. Mu and D. Clark. An Interval-based Abstraction for Quantifying Information Flow. *ENTCS*, 253(3):119–141, 2009.

[30] C. Mu and D. Clark. Quantitative Analysis of Secure Information Flow via Probabilistic Semantics. In *Proc. 4th International Conference on Availability, Reliability and Security (ARES '09)*, pages 49–57. IEEE, 2009.

[31] J. Newsome, S. McCamant, and D. Song. Measuring Channel Capacity to Distinguish Undue Influence. In *Proc. 4th ACM Workshop on Programming Languages and Analysis for Security (PLAS '09)*. ACM, 2009.

[32] S. Park, F. Pfenning, and S. Thrun. A Probabilistic Language based upon Sampling Functions. In *Proc. ACM Symposium on Principles of Programming Languages (POPL '05)*, 2005.

[33] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate Non-Interference. In *Proc. IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 3–17. IEEE, 2002.

[34] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[35] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[36] G. Smith. On the foundations of quantitative information flow. In *Proc. Intl. Conference of Foundations of Software Science and Computation Structures (FoSSaCS '09)*, LNCS 5504, pages 288–302. Springer, 2009.

[37] G. Steel. Formal analysis of PIN block attacks. *Theoretical Computer Science*, 367(1-2):257–270, Nov. 2006.