Network of Excellence

Deliverable D8.3

First version of improved programming environments (including tools and prototypes) to support security
<table>
<thead>
<tr>
<th><strong>Project Number</strong></th>
<th>: 256980</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project Title</strong></td>
<td>: NESSoS</td>
</tr>
<tr>
<td><strong>Deliverable Type</strong></td>
<td>: Report</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Deliverable Number</strong></th>
<th>: D8.3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title of Deliverable</strong></td>
<td>: First version of improved programming environments (including tools and prototypes) to support security</td>
</tr>
<tr>
<td><strong>Nature of Deliverable</strong></td>
<td>: R</td>
</tr>
<tr>
<td><strong>Dissemination Level</strong></td>
<td>: Public</td>
</tr>
<tr>
<td><strong>Internal Version Number</strong></td>
<td>: 0.8</td>
</tr>
<tr>
<td><strong>Contractual Delivery Date</strong></td>
<td>: September 30, 2012</td>
</tr>
<tr>
<td><strong>Actual Delivery Date</strong></td>
<td>: October 15, 2012</td>
</tr>
<tr>
<td><strong>Contributing WPs</strong></td>
<td>: WP 8</td>
</tr>
<tr>
<td><strong>Editor(s)</strong></td>
<td>: Wouter Joosen</td>
</tr>
<tr>
<td><strong>Author(s)</strong></td>
<td>: Marianne Busch (LMU), Jorge Cuellar (SIEMENS), Philippe De Ryck (KUL), Maarten Decat (KUL), Lieven Desmet (KUL), Thomas Heyman (KUL), Bart Jacobs (KUL), Wouter Joosen (KUL), Boris Köpf (IMDEA), Bert Lagaisse (KUL), Fabio Martinelli (CNR), Nick Nikiforakis (KUL), Marchoa-Ronderos (SIEMENS), Animesh Pathak (INRIA), Pieter Philippearts (KUL), Frank Piessens (KUL), Steven Van Acker (KUL), Artsiom Yautsiukhin (CNR), Koen Yskout (KUL)</td>
</tr>
<tr>
<td><strong>Reviewer(s)</strong></td>
<td>: M. Carmen Fernandez Gago, Marinella Petrocchi, Michael Rusinowitch</td>
</tr>
</tbody>
</table>
Abstract

The aim of this work package is to study and incept new solutions that support service programming and com-
position, and to build and extend tools that assist programmers and application builders accordingly. The work
presented here is based on three large pillars: (1) security support for programming languages (Task 8.4), (2)
secure composition of services (Task 8.2), and (3) runtime support for the secure execution of services (Task 8.3).
This deliverable reports on the three subtopics listed above. Trying to build all-encompassing solutions for each
topic is impossible, so we have narrowed down our scope to specific topics in these domains. We believe that these
topics will be critical both for the near as well as for the long term research agenda. The deliverable summarizes
a selection of the many results that have been achieved by the NESSoS partners of this work package during the
second year of the project.

Keyword List

security, web services, run-time enforcement, component integration, secure service composition, program verifi-
cation, information flow
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Type of Change</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>initial setup</td>
<td>KUL</td>
</tr>
<tr>
<td>0.2</td>
<td>initial content for web app security, programming language support</td>
<td>KUL</td>
</tr>
<tr>
<td>0.3</td>
<td>section about secure navigation paths</td>
<td>LMU</td>
</tr>
<tr>
<td>0.4</td>
<td>added contents to Secure Service Composition</td>
<td>CNR</td>
</tr>
<tr>
<td>0.5</td>
<td>added contents to information flow section</td>
<td>IMDEA</td>
</tr>
<tr>
<td>0.6</td>
<td>added contents to Secure Service Composition</td>
<td>INRIA</td>
</tr>
<tr>
<td>0.7</td>
<td>incorporated reviewers comments</td>
<td>KUL</td>
</tr>
<tr>
<td>0.8</td>
<td>finished document approved by the internal reviewers</td>
<td>KUL</td>
</tr>
</tbody>
</table>
Table of Contents

LIST OF FIGURES ............................................................. 7

1 INTRODUCTION ................................................................. 8

2 WEB APPLICATION SECURITY .................................................. 9
  2.1 Defense against Malicious Cross-domain Requests ............................. 9
    2.1.1 High Level Overview .................................................. 9
    2.1.2 Disarming potentially malicious Cross-Domain Requests .............. 10
  2.2 Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications ............................................. 10
    2.2.1 Background ................................................................ 11
    2.2.2 FlashOver Prototype .................................................. 13
    2.2.3 Evaluation .................................................................. 14
  2.3 War Games ..................................................................... 16

3 SECURE SERVICE COMPOSITION AND COMPOSITION OF SECURITY SERVICES .......................................................... 17
  3.1 Metric-aware Secure Service Orchestration ..................................... 17
    3.1.1 Updated Formal model .................................................. 17
    3.1.2 C-semiring .................................................................. 18
    3.1.3 Example ..................................................................... 20
    3.1.4 Conclusion .................................................................. 21
  3.2 Modeling and Enforcing Secure Navigation Paths ............................ 21
  3.3 Federated authorization for SaaS applications ................................... 23
    3.3.1 Proposed solution ....................................................... 23
    3.3.2 Challenges .................................................................. 24
    3.3.3 Research methodology .................................................. 24
    3.3.4 Main contributions ....................................................... 24
  3.4 Composing Trust Models Towards Interoperable Trust Management .... 25
    3.4.1 Trust Model Description Language ..................................... 26
    3.4.2 Composing Trust Models ............................................... 27
    3.4.3 TMDL Tools ............................................................... 30
    3.4.4 Future Directions ......................................................... 30

4 PROGRAMMING LANGUAGE SUPPORT ....................................... 31
  4.1 Featherweight VeriFast ......................................................... 31
    4.1.1 Semantic Framework ..................................................... 31
    4.1.2 Formalization of VeriFast .............................................. 32
  4.2 Valorization ....................................................................... 32
    4.2.1 Problem Statement ....................................................... 33
    4.2.2 The Belgian Electronic Identity Card .................................. 33
    4.2.3 Embedded Linux Network Management Software ...................... 34
    4.2.4 Other Case Studies ....................................................... 36
    4.2.5 Contributions ............................................................. 37
List of Figures

Figure 2.1: ActionScript 2.0 source code of an example vulnerable Flash application ........................................... 12
Figure 2.2: Schematic overview of our FlashOver prototype .......................................................... 13

Figure 3.1: Abstract workflow for BestTravel ................................................................. 20
Figure 3.2: Implementation of BestTravel ................................................................. 20
Figure 3.3: Type of BestTravel ..................................................................................... 21
Figure 3.4: The Trust Enabler Architecture ......................................................................... 26
Figure 3.5: The Mediation Process ..................................................................................... 28

Figure 5.1: Upper bounds for the maximal leakage w.r.t. the adversary \textit{Adv}_{pre}. ................ 45
Figure 5.2: Upper bounds for the maximal leakage w.r.t. the adversary \textit{Adv}_{prob}. ............... 46
1 Introduction

Clearly, programmers and software developers are still struggling to write and combine code in a way that the end result meets the quality objectives with respect to security. Many research initiatives have addressed problems in this field, yet security of software (the code) remains a moving target and new problems keep popping up. Future Internet services present new demands and will inevitably lead to new security conundrums.

The aim of this work package is to study and incept new solutions that support service programming and composition, and to build and extend tools that assist programmers and application builders accordingly. The work presented here is based on three large pillars: (1) security support for programming languages (Task 8.4), (2) secure composition of services (Task 8.2), and (3) runtime support for the secure execution of services (Task 8.3).

Security support for programming languages is essential to build new services from scratch. By giving the programmer tools that can detect security errors early on, the quality of the resulting code can be guaranteed (or at least improved). These tools can be plugged into different steps of the software development lifecycle, or at multiple steps.

Secure composition of services deals with the specific security aspects of building compositions of software. Ensuring that the aggregate of security services is not less secure than each individual service is a key goal in this field. We put special emphasis on the composition of security services. Runtime support for the secure execution of services can be used to execute services that are not fully trusted. They may not be fully trusted because they come from an un-trusted location, are built by un-trusted developers, or do not offer the necessary security guarantees. By running these services in a secure runtime environment, the potential to harm a user can be dramatically decreased.

This deliverable reports on the three subtopics listed above. Trying to build all-encompassing solutions for each topic is impossible, so we have narrowed down our scope to specific topics in these domains. We believe that these topics will be critical both for the near as well as for the long term research agenda. The deliverable summarizes a selection of the many results that have been achieved by the NESSoS partners of this work package during the second year of the project. The report is structured as follows.

Chapter 2 elaborates on two contributions to the domain of secure program execution with a focus on web security. It also briefly mentions work in progress that is relevant in this context and that will be discussed more exhaustively in deliverable D13.3.

Chapter 3 describes the work that has been done in the context of secure composition of services. It studies how services can be composed in a secure fashion, and also how different security services (e.g. an authentication and an authorization service) can and should be composed.

Chapter 4 lists the contributions we have made on the programming languages support front, where we continued the work on the VeriFast tool (also presented in the previous deliverable - D8.2). In particular, work has been done to further formalize the verification process and to valorize the prototype with real world applications.

As a complement to the three chapters listed above, we have collected a set of results that focus on one critical security property - information flow. In this respect, Chapter 5 presents work that builds further on the notions that have been introduced in the previous deliverable (D8.2): dynamic information-flow analysis and quantitative information-flow analysis.

Finally, Chapter 6 summarizes the interactions between this work package and other work packages, and between the partners that have been contributing to this work package. Chapter 7 concludes the deliverable.
2 Web Application Security

Deliverable 8.2 has introduced some of the work that the consortium is doing to improve the security of web applications of the Future Internet. Rich and dynamic websites have become omnipresent on the web, but unfortunately the relevant standards are not always secure or are often used in ways that make implementations insecure.

In the previous deliverable, a number of countermeasures have been introduced to counter a number of these threats. These countermeasures have further been developed during the second year of the NESSoS NoE, and those results are presented in this section. In particular, this work builds on SessionShield (Section 2.1). These contributions are also partially supported by the FP7-project WebSand. This chapter also includes work that has been newly developed this year. Finally, a short overview of a new web security-related open competition is sketched.

In addition to the work presented here, KU Leuven has also worked together with a number of standard organizations to improve the quality of the standards that are related to this research. In particular, due to our analysis work (that was the precursor of this research), the standards have been updated with regards to the correct use of ports with Web Messaging, the potential use of null-Origin with CORS, and the potential of clickjacking attacks.

2.1 Defense against Malicious Cross-domain Requests

Two of the most popular platforms for providing enriched Web content are Adobe Flash and Microsoft Silverlight. Through their APIs, developers can serve data (e.g. music, video and online games) in ways that could not be traditionally achieved through open standards, such as HTML. The latest statistics show a 95% and 61% market penetration of Flash and Silverlight respectively, attesting towards the platforms’ popularity and longevity [6].

Unfortunately, history and experience have shown that functional expansion and attack-surface expansion go hand in hand. Flash, due to its high market penetration, is a common target for attackers. The last few years have been a showcase of “zero-day” Flash vulnerabilities where attackers used memory corruption bugs to eventually execute arbitrary code on a victim’s machine [17].

Apart from direct attacks against these platforms, attackers have devised ways of using legitimate Flash and Silverlight functionality to conduct attacks against Web applications that were previously impossible. One of the features shared by these two platforms is their ability to generate client-side cross-domain requests and fetch content from many remote locations. In general, this is an opt-in feature which requires the presence of a policy configuration. However, in case that a site deploys an insecure wildcard policy, this policy allows adversaries to conduct a range of attacks, such as leakage of sensitive user information, circumvention of CSRF countermeasures and session hijacking. Already, in 2007 a practical attack against Google users surfaced, where the attacker could upload an insecure cross-domain policy file to Google Docs and use it to obtain cross-domain permissions in the rest of Google’s services [128]. Even though the security implications of cross-domain configurations are considered to be well understood, three recent studies [94, 100, 60] showed that a significant percentage of websites still utilize highly insecure policies, thus, exposing their user base to potential client-side cross-domain attacks.

To mitigate this threat, we present DEMACRO, a client-side defense mechanism which can protect users against malicious cross-domain requests. Our system automatically identifies insecure configurations and reliably disarms potentially harmful HTTP requests through removing existing authentication information. Our system requires no training, is transparent to both the Web server and the user and operates solely on the client-side without any reliance to trusted third-parties. This system is partially based on the SessionShield work presented in Deliverable 8.2. It is done in collaboration with SAP, an associated partner of the NESSoS consortium. More details of this system can be found in [101].

2.1.1 High Level Overview

The general mechanism of our approach functions as follows: The tool observes every request that is created within the user’s browser. If a request targets a cross-domain resource and is caused by a plugin-based applet, the tool checks whether the request could potentially be insecure. This is done by examining the request’s execution context to detect two misuse cases: For one, the corresponding cross-domain policy is retrieved and checked for insecure wildcards. Furthermore, the causing applet is examined, if it exposes client-side proxy functionality. If one of these conditions is met, the mechanism removes all authentication information contained in the request. This way, the tool robustly protects the user against insecurely configured cross-domain mechanisms. Furthermore, as the request
itself is not blocked, there is only little risk of breaking legitimate functionality. While our system can, in principle, be implemented in all modern browsers, we chose to implement our prototype as a Mozilla Firefox extension.

2.1.2 Disarming potentially malicious Cross-Domain Requests

A cross-domain request conducted by a plug-in is not necessarily malicious as there are a lot of legitimate use cases for client-side cross-domain requests. In order to avoid breaking the intended functionality but still protecting users from attacks, it is crucial to eliminate malicious requests while permitting legitimate ones. The most vulnerable websites are those that make use of a wildcard policy and host access-controlled, personalized data on the same domain; a practice that is strongly discouraged by Adobe [80]. Hence, we regard this practice as an anti-pattern that carelessly exposes users to high risks. Therefore, we define a potentially malicious request as one that carries access credentials in the form of session cookies or HTTP authentication headers towards a domain that serves a wildcard policy. When the extension detects such a request, it disarms it by stripping session cookies and authentication headers. As the actual request is not blocked, the extension does not break legitimate application but only avoids personalized data to appear in the response. Furthermore, DEMACRO is able to detect attacks against vulnerable Flash proxies. If a page on a.net embeds an applet file served by b.net and conducts a same-domain request towards b.net, user credentials are also stripped by our extension. The rationale here is that a Flash-proxy would be deployed on a website so that the website itself can use it rather than allowing any third party domain to embed it and use it.

2.1.3 Evaluation

The implementation of DEMACRO has been evaluated on three points. For the full evaluation, we refer the reader to [101]. This section will give a short overview of the results.

Security  DEMACRO was tested against both publicly available exploits [11] and our own test-cases. We implemented Flash and Silverlight applets to test our system against all possible ways of conducting cross-domain requests across the two platforms and we also implemented several vulnerable Flash and Silverlight applets to test for misuse cases. In all cases, DEMACRO detected the malicious cross-domain requests and removed the authentication information. Lastly we tested our system against the exploits we developed for real-world use cases and were able to successfully prevent the attacks.

Compatibility  In order to test DEMACRO’s practical ability to stop potentially malicious cross-domain requests while preserving normal functionality, we conducted a survey of the Alexa top 1,000 websites. We used the Selenium IDE 9 to instrument Firefox to automatically visit these sites twice. The rationale behind the two runs is the following: In the first run, DEMACRO was deactivated and the sites and ad banners were populating cookies to our browser. In the second run, DEMACRO was enabled and reacting to all the insecure cross-domain requests by stripping-off their session cookies that were placed in the browser during the first run. In total, we were able to observe 79,093 HTTP requests, of which 1,105 were conducted by plug-ins across domain boundaries. When using DEMACRO, manual inspection showed that only a single site was affected by our countermeasure.

Performance  In order to evaluate the performance of our countermeasure we measured the time needed to perform a large number of cross-domain requests when issued by JavaScript and issued by a Flash applet. For JavaScript, the overhead that our system imposes is 0.00082 seconds for each cross-domain request. While this represents the best-case scenario, since none of the requests need to be checked against weak cross-domain policies, we believe that this is very close to the user’s actual everyday experience where most of the content served is done so over non-plugins and without crossing domain boundaries. For the Flash applet, our system added a 0.00173 seconds overhead to each plugin-based cross-domain request in order to inspect its origin, the policy of the remote-server and finally perform any necessary stripping of credentials.

2.2 Automated Discovery of Cross-site Scripting Vulnerabilities in Rich Internet Applications

Today’s Internet is teeming with dynamic web applications visited by numerous Internet users. During their visits, typical Web users will unknowingly use tens of Rich Internet Applications like Flash banners or media players. For
HTML-based web applications, it is well-known that Cross-site Scripting (XSS) vulnerabilities can be exploited to steal credentials or otherwise wreak havoc, and there is a lot of research into solving this problem. An aspect of this problem that seems to have been mostly overlooked by the academic community, is that XSS vulnerabilities also exist in Adobe Flash applications, and are actually easier to exploit because they do not require an enclosing HTML ecosystem.

In this section we present FLASHOVER, a system to automatically scan Rich Internet Applications for XSS vulnerabilities by using a combination of static and dynamic code analysis that reports no false positives. FLASHOVER was used in a large-scale experiment to analyze Flash applications found on the top 1,000 Internet sites, exposing XSS vulnerabilities that could compromise 64 of those sites, of which six are in the top 50.

### 2.2.1 Background

In this section we give a brief overview of Cross-site Scripting attacks and of the Adobe Flash platform. We also present a motivating example showing how a vulnerable Flash application can be used to inject malicious JavaScript that will be executed by user’s browser in the context of the domain hosting the vulnerable Flash application. While the techniques presented in the rest of this section are specific to the Flash platform, they are, in principle, applicable to other similar content-delivering platforms, such as Microsoft Silverlight [110].

#### Cross-site Scripting

Cross-site Scripting (XSS) attacks belong to a broader range of attacks, collectively known as code injection attacks. In code injection attacks, the attacker inputs data that is later on perceived as code and executed by the running application.

In XSS attacks, an attacker adds malicious JavaScript code on a page of a vulnerable website that will be executed by a victim’s browser when that vulnerable page is visited. Malicious JavaScript running in the victim’s browser and in the context of the vulnerable website can access, among others, the session cookies of that website and transfer them to an attacker-controlled server. The attacker can then replay these sessions to the vulnerable website effectively authenticating himself as the victim. The injected JavaScript can also be used to alter the page’s appearance to perform phishing or steal sensitive input as it is typed-in by the user.

#### Adobe Flash

Adobe Flash is a proprietary multimedia platform which is used to create Rich Internet Applications. To be able to run Flash applications on a desktop, a Flash player must be installed which takes the form of a browser plugin. According to the latest statistics, Adobe’s Flash player is installed on more than 99% of desktops connected to the Internet [70, 127]. Over the years, the amount of functionality available to Flash applications has increased with each new version of the Flash player. Today, a Flash application can combine audio, video, images and other multimedia elements.

Flash applications are contained in SWF files (i.e. files with the .swf extension) which bundle multimedia elements together with byte-code-compiled ActionScript (AS) code. When loaded into the Flash player, the Flash application is rendered and, if present, the AS byte-code is interpreted and executed. ActionScript is a scripting language developed by Adobe which allows the programmer to handle events, design the interaction between multimedia elements and communicate with both the embedding browser and remote Web servers. The current version of ActionScript is ActionScript 3.0 with legacy support for prior versions.

#### Using SWF files

SWF files are typically embedded in HTML using the `<object>` or `<embed>` tags, but it is also possible to load an SWF file into the browser directly, without embedding it into HTML, either by requesting it as is from a browser’s URL bar or providing it as the source argument to an `<iframe>` tag in an existing HTML page.

Flash, like many other technologies, allows for the provision of load-time input next to hard-coded values specified at compile-time and present in the resulting SWF file. For instance, YouTube videos are displayed on webpages that each embed the same Flash video player. Data specific to the displayed video-file is passed to the Flash player at load-time through variables embedded in the enclosing HTML page. Flash supports two methods of passing values to Flash objects:
• **FlashVars directive**: When embedding a SWF file using the `<object>` or `<embed>` tags, the FlashVars parameter can be used to pass values to specific variables.

• **GET parameters**: A web developer can also utilize GET-parameters to pass arguments to a Flash application. For instance, when the URI: http://example.com/myFlashMovie.swf?var1=Hello&var2=World is invoked, the Flash application will initialize its internal variables var1 and var2 with their respective values. This method is usually overlooked by web developers who believe that the Flash application hosted on their page can only receive the parameters that they have hard-coded in the embedding HTML page and thus in many cases do not perform input validation within the Flash application itself.

**Execution context of SWF files**

In the previous section, we briefly examined the two ways that a SWF file can be loaded by a browser (using special HTML tags or a direct reference). While in both cases, the Flash Player loads the SWF file and starts executing it, there is a very important difference in the way that the two Flash applications interact with the surrounding page when the Flash applications requests the execution of JavaScript code from the browser.

The *allowScriptAccess* [16] runtime parameter arbitrates the access a Flash application has to the embedding page. There are three possible values: ‘always’, ‘sameDomain’ and ‘never’, with ‘sameDomain’ being the default. This value has the effect that access is only allowed when both the SWF application and the embedding page are from the same domain.

When an SWF file is embedded using the embed tag, and Flash requests the execution of JavaScript code from the browser, the code will execute within the origin of the embedding site, assuming a suitable value for the *allowScriptAccess* parameter. That is, if a SWF file hosted on the web server of foo.com is embedded in an HTML page on bar.com, the origin of the Flash-originating JavaScript is now bar.com. The origin is defined using the domain name, application layer protocol, and port number of the HTML document embedding the SWF.

If however, bar.com loads the SWF file of foo.com using an `<iframe>`, the browser creates an empty HTML page around the Flash application and any JavaScript initiated from the application will retain the origin of foo.com. Additionally, since the default value for *allowScriptAccess* is ‘sameDomain’, this means that the Flash application will be able to access data in the same origin as foo.com.

**XSS in Flash**

```actionscript
movie ‘ad.swf’ {
    button 42 {
        on (release) {
            getURL(_root.clickTag, '_blank');
        }
    }
}
```

**Figure 2.1: ActionScript 2.0 source code of an example vulnerable Flash application**

Consider a Flash advertising banner of which the ActionScript 2.0 source code is listed in Figure 2.1. The banner includes a button which, when clicked and released, triggers the execution of the `getURL()` function. The `getURL(url, target)` directs the browser to load a URL in the given target window. In this example, the URL is obtained from the variable `clickTag` in the global scope, and loaded into a new window (`_blank`).

When used legitimately, the banner is located on http://company.com/ad.swf and is embedded on one of company.com’s web pages. The value of the `clickTag` variable is provided by the embedding page using the FlashVars directive and, in our example, suppose that it would redirect the clicking user to e.g. http://company.com/new_product.html.

As described in earlier sections, a SWF file can be directly referenced and any GET parameters will be provided to the Flash application itself, exactly as in the FlashVars case. Thus, if the banner was directly requested through http://company.com/ad.swf?clickTag=http://www.evil.com, the `clickTag` variable would now hold the value http://www.evil.com instead of the value intended by company.com. This behavior could be abused by attackers in order to send malicious requests with the correct Referrer header towards

NESSoS - 256980 12
Web applications that use Referrer checking as a means of protection against CSRF attacks [135]. While this is definitely a misuse scenario, the vulnerable code unfortunately allows for a much greater abuse. Instead of providing a website URL as the value for `clickTag`, an attacker could provide a JavaScript URL, such as `javascript: alert('XSS')`. A JavaScript URL is a URL that causes the browser to execute the specified JavaScript code in the context of the current-page (`alert('XSS')` in our aforementioned example) instead of making a remote request, as is the case in HTTP(S) URLs. In this scenario, when that banner is clicked, the user’s browser will execute attacker-supplied JavaScript code instead of redirecting the user.

All an attacker needs to do in order to exploit this vulnerability, is to lure a victim into visiting a website which loads the vulnerable SWF file in an `iframe` and insert a `javascript:` URL containing malicious JavaScript code into the query string of the SWF file URL. Since the SWF file is loaded in an `iframe`, it will retain the origin of `company.com` and thus when the user clicks on the banner, the JavaScript code will execute in the context of `company.com` instead of the attacker’s site. This will allow the malicious JavaScript code to access, among other things, the user’s cookies for `company.com` and steal his session identifiers. If a click on the vulnerable Flash banner is required to trigger the execution of the injected JavaScript, the user can be tricked into clicking the banner, either using social engineering or clickjacking techniques [27]. In cases where the vulnerable code is triggered after a predetermined amount of time, all that the attacker needs to do is to make sure to keep the user on his malicious site for the appropriate amount of time.

While the example ActionScript in Figure 2.1 appears to be a contrived one, many websites unfortunately have similarly vulnerable banners.

### 2.2.2 FlashOver Prototype

The description of the general FlashOver approach omits implementation details, because each of the steps in FlashOver can be implemented in a number of ways with varying degrees of thoroughness. We purposefully chose to implement a minimalistic version of FlashOver to investigate the level of effort and skill required by an attacker to automatically detect XSS vulnerabilities in SWF files.

Our FlashOver prototype is schematically illustrated in Figure 2.2. The following subsections discuss the implementation details of each step in our FlashOver prototype.

![Figure 2.2: Schematic overview of our FlashOver prototype](image)

**Static analysis**

This first step in the FlashOver process requires static analysis of the SWF file. We chose to decompile the SWF file and then perform a simple static analysis on the resulting ActionScript source code.

There are many SWF decompilers, but not all of them support ActionScript 3.0. Choosing a decompiler, such as the freely available flare [93], that does not support the latest version of ActionScript, would mean that there would be a blind-spot in our analysis. For that reason, we chose a commercial decompiler with support for ActionScript 3.0 [139].
To reduce the complexity of our prototype, we opted for a simple regular-expression extraction of the PEVs instead of using more complicated analysis methods. Using this method, the resulting ActionScript source code is searched for patterns indicating potentially exploitable variables.

**Attack URL construction**

Based on the variable names identified in the previous step, attack URLs are constructed that, when the attack payload is triggered, will report in what way the given SWF file is vulnerable to XSS.

Exploitable variables can be used in ActionScript in a number of different ways. Through our review of JavaScript injection techniques, we identified a non-exhaustive list of nine ways in which an attacker-specified payload can ultimately be injected into a JavaScript context, through exploitable variables in an SWF file. As a control, we also use an injection template that injects no JavaScript code. For each of these injection templates, a separate attack URL is constructed.

The attack URL should encode information about entry point, attack vector and payload type into a unique identifier. The entry point is encoded by a unique hex-encoded 256-bit number that identifies the SWF file being analyzed. The attack vector, or the exploitable variable used to inject the payload, is encoded as an index into the list of identified potentially exploitable variables. Finally, the payload type is encoded as an index into the list of nine injection templates specified earlier.

From the given SWF file identifier (swfid), injection template index (type id) and exploitable variable index (var id), a unique identifier is constructed for this specific attack URL, by concatenating these three values, separated by a ‘x’ character. This unique identifier is appended to the URL for the log-server, forming the logging URL. The logging URL is then used in a JavaScript code fragment that, when executed, will trigger a request to the log-server, logging the unique identifier. This piece of JavaScript code is then inserted into the selected injection template, forming the payload of the attack URL, in this case a simple `javascript: URL`. Finally, the payload is assigned to the exploitable variable in a query string of the attack URL.

**Automated interaction**

The final step of **FLASHOVER**, involves passing the crafted attack URL to a simulated victim and let that victim interact with it, potentially triggering the execution of the injected JavaScript. Based on our personal experience and the analysis of many Flash applications, we make the assumption that most interactions with Flash applications are achieved through mouse clicks. For that reason, we only consider this type of interaction in our prototype implementation.

The Flash application is loaded into a real Firefox browser. The browser itself is started in Xvfb, a virtual framebuffer X server ¹ and the virtual mouse attached to this Xvfb session is controlled through the `xte` program ². The Xvfb server is set up to offer a virtual framebuffer of 640x480 pixels with 24-bit color to any program running inside. Firefox, running inside Xvfb is started full-screen (so 640x480) in kiosk mode. This means that all toolbars and menus are removed, and undesirable functionality, like printing, is disabled.

Once Firefox has started and loaded the Flash application, a list with 10,000 random (x,y) locations is generated and passed to `xte`, which moves the mouse to those locations and issues a click. After these 10,000 clicks, the automated clicker pauses to give the Flash application time to process the input, which could involve executing the injected JavaScript payload.

If the execution of the injected JavaScript is triggered as a result of one or more mouse-clicks, this will be recorded in our logging server. The detection of the injected codes’ execution effectively creates a new set of actually exploitable variables which is a subset of the original potentially exploitable variables set, as that was generated in the first stage of **FLASHOVER**. The entries of the logging server can then be used, as previously explained, to pinpoint the exact place in the Flash application and the specific attack vector that can be used for a XSS attack.

**2.2.3 Evaluation**

We evaluated our **FLASHOVER** prototype with a large-scale experiment to determine how many SWF files vulnerable to XSS are hosted on the Alexa top 1,000 Internet sites [21].

---

¹ [http://www.xfree86.org/4.0.1/Xvfb.1.html](http://www.xfree86.org/4.0.1/Xvfb.1.html)
² [http://linux.die.net/man/1/xte](http://linux.die.net/man/1/xte)
Experimental setup

For each of the domains in the Alexa top 1k, a list of publicly exposed SWF files was retrieved from Altavista using the query “site:domain.com filetype:swf” where domain.com would be a domain in our experiment.

The SWF files discovered through these queries were downloaded onto a local web server. Although the experiment could have been conducted using the SWF hosted on their original locations, we feared that it might potentially harm the targeted site. In addition, storing the SWF locally improved performance by reducing the time it took to load the SWF file into the browser.

After the non-SWF or otherwise invalid SWF files were removed from the set of downloaded files, they were processed by FlashOver. The static analysis and attack URL construction steps of FlashOver were performed on all SWF files in advance to reduce overhead for the entire experiment. The final step, using an automated clicker, was performed in parallel on 70 dual-core computers.

Because the automated clicker clicks on random positions on the Flash application, each run of the automated clicker can yield different results. To increase the odds that the payload in the attack URLs was triggered, the entire dataset was processed by the automated clickers 20 times. The total experiment ran for approximately five days, approximately six hours per run.

Results

From Altavista, 18,732 URLs were retrieved. After downloading, 3,800 SWF files did not contain a valid Flash application. Of the remaining 14,932 SWF files, 35 caused our decompiler to destabilize and crash. From the 14,897 SWF files that were decompiled successfully, 8,441 were determined to have exploitable variables. For each of these 8,441 SWF files, 10 attack URLs were generated: one for each injection template. The final generated dataset contained a list of 84,410 attack URLs. All of these were processed in parallel by the automated clickers.

After analysis of the log files, 523 SWF files were found to load content from an attacker-supplied URL (i.e. URL injection) and 286 SWF files allowed the execution of attacker-supplied JavaScript code. These 286 vulnerable SWF files can be traced back to 64 Alexa domains, of which six are in the top 50.

What we noticed while doing this research is that vulnerable variables often have the same names. Interestingly, the two most commonly vulnerable variables are responsible for more than 69% of all vulnerabilities found. The fact that many different Flash applications are vulnerable to the same attack and through the same variables, suggests the use of automated tools for the creation of Flash applications that generate code in a vulnerable way. At the same time, our results highlight the need for scanning of variables and code-paths beyond the ones commonly associated with vulnerabilities.

Discussion

When one considers the number of vulnerable Flash applications found on the Internet’s top websites, it becomes clear that XSS attacks through Flash applications are indeed a problem. Although Adobe advocates security best practices [18], stating that user-input should be sanitized where needed, this advice seems to be overlooked by Flash application developers.

The required effort and skill to automatically discover these XSS vulnerabilities is limited. As discussed in Section 2.2.2, our FlashOver prototype uses suboptimal static analysis and randomized clicking to simulate a user. For the static analysis part, a more precise taint-analysis system would produce better results since it could identify more variables influenced by user-input and thus produce a longer list of potentially exploitable variables. Moreover, a determined attacker can easily uncover additional vulnerabilities using a manual static analysis. Likewise, the randomized clicker is lacking the cognitive ability of an actual human user: it does not understand typical GUI widgets that a human would click and it can not interact with e.g. a game like a human would. This means that there may be vulnerabilities that our clickers could not trigger but that a human victim would. Therefore, the amount of vulnerable Flash applications detected in this experiment is a lower bound: the actual amount of vulnerable applications is most likely higher, making the security threat an even bigger issue.

An interesting property of FlashOver is that it detects successful JavaScript injection by actually simulating a victim who triggers the use of the injected JavaScript code in one or more potentially exploitable variables. Thus, while FlashOver may miss some vulnerabilities (false negatives), it has practically zero false positives. While one can construct examples where FlashOver would report a false positive, e.g. an application that is vulnerable to XSS but inspects the injected payload and only allows it if it is “not dangerous”, we believe that these are unrealistic examples and thus would not be encountered in the analysis of real-life Flash applications.
2.3 War Games

An important target for the NESSoS network of excellence is to foster new relationships between consortium members as well as with members outside of the consortium. An important element in setting up these collaborations is to promote the NoE and to get as many people as possible to interact with it. One way of doing this has been explored last year by organizing a first open competition aiming at promoting the understanding of methods and frameworks for modeling and analyzing security requirements.

In the second year of the project, a new type of open competition is introduced. Aimed at helping people understand the intricacies of web server security, we have developed a Web Security War Game. War games are popular with hackers and professional security experts alike, and offer a valuable training experience. In addition, the interest of different players can be boosted by introducing a competition element in the game and allow the players to play against each other.

The war game is a level-based game, where players start in a specific level and work their way up towards more difficult levels. A level can be completed by successfully exploiting a vulnerability in the system and retrieving the credentials to go to a next level. The first levels focus on easily exploitable vulnerabilities where more background information about the underlying systems is available. The player gradually gets less information needed to break the system, and the exploits become harder. In the most advanced levels, a number of exploits must be combined to finish a level.

As mentioned before, the war game focuses on server side security. Some example vulnerabilities that are included in the war game are described here.

1. Snooping around Important information is sometimes unprotected, and an attacker can find it by simply snooping around through the system
   (a) robots.txt often contains pages that should be hidden from search engines, which makes them potentially interesting for attackers
   (b) parent directories may contain important files and backups

2. Cookies/sessions Cookies are used for various things, including session management. Vulnerabilities in the cookie management of a website might allow an attacker to impersonate users or access restricted areas.
   (a) cookie data might store important data (for example 'loggedin=1')
   (b) referrers should be checked when a user is being authenticated
   (c) session IDs must be hard to guess
   (d) session poisoning exploits the lack of input validation on the server side

3. PHP A number of vulnerabilities are related to the programming platform that is used.
   (a) local file inclusion can allow directory traversals of not properly sanitized by the server
   (b) remote file inclusion can lead to code execution on the client or server side
   (c) uploaded files should never be able to execute
   (d) system() calls must be properly sanitized to prevent command execution

4. SQL The database language used by most websites on the internet can be used in an exploit if the website does not properly validate input.
   (a) simple SQL injections might allow attackers to bypass certain logic such as authentication policies
   (b) information leakage can be used to gradually extract secret data from a website
   (c) advanced SQL injections might add or update data in the database

The war game can be expanded by interested members of the community. Levels can be plugged-in with relative ease. A more detailed description of the war game can be found in the deliverable of WP13.
3 Secure Service Composition and Composition of Security Services

Reducing a business requirement to a set of disjoint web services and then composing them together helps to manage the complexity of modern systems. However, it also introduces a number of new problems: the composed system must still be secure, and managing it must be effortless. In addition, the performance must also be more than acceptable. Unfortunately, these requirements are conflicting and cannot be optimized simultaneously. A trade-off is required to tackle this problem, as we had already noted in Deliverable 8.2.

3.1 Metric-aware Secure Service Orchestration

Orchestration of complex web services is a multidimensional problem. Various criteria must be considered when different alternatives exist. Typically, one of such criteria is security. Recently, the security issues of service composition are receiving major attention [114, 129, 35, 42, 117, 47]. Among them, formal methods have been successfully applied for modelling and analysing several different aspects of service security. In practice, these techniques generate a formal abstraction of the services under analysis. Then, a verification procedure is applied to find a formal proof of compliance between the model and the security specifications.

Service usages are often based on security metrics. Metrics conveniently use mathematical values to represent some “qualities” of a service. Notions like reputation and trust are becoming more and more common in the context of web services. Contemporary, several authors, e.g., see [151, 104], proposed mathematical models for the definition and composition of security metrics.

In this chapter we propose an extension of our previous work [56, 57] (based on the work of Bartoletti et al. [35]) on secure service orchestration integrating facilities for composing and verifying security metrics. Our previous work was presented in D8.2. We extend our model by introducing metric checks and metric annotations on existing abstractions. We use a mathematical structure, called c-semiring, in order to generalise our model and be independent from the metrics used for the analysis, but still be able to reason on these metrics. Metric annotations are obtained through a new, improved type and effect system. In this way, we generate metric-annotated abstractions which contain both security and metric requirements. All the requirements are applied to different portions of the service orchestration through a local scope.

Thus, the main advantage of this approach is the possibility to model and compose both security and metric requirements in a single framework.

3.1.1 Updated Formal model

We start with the description of the modifications made in the syntax of $\lambda^{req}$. These modifications are intended to make our approach more service oriented and make possible wider range of analysis.

Table 3.1 shows the modified syntax of $\lambda^{req}$. Next to the old part of the syntax the table contains two new constructs. First, we added a possibility to express parallel execution to the syntax, i.e., \texttt{fork e and e'}. In fact, we were able to express this structural activity with the existing syntax and checked the correctness of its usage. Nevertheless, we decided to define a special shortcut for this operation for better visibility. Now, our model is able to capture all four basic structural activities used for defining a business process: sequence, flow (parallel execution), choice, and loop.

Second modification concerns usage of security metrics, i.e., metric framing. This framing is required for restriction of the part of a complex service where a quantitative property is check. Currently, we consider simple quantitative expressions: the value of the framed part of composition must be better (e.g., higher or lower) than a threshold.

Similar modifications were made in the historical expressions. We added a special operation for parallel execution and metric check framing (Table 3.2). Moreover, we also added a special construct which describes the metric for a historical expression. This construct is used to indicate the value of the metric.

Service execution is driven by the operational semantics defined in Table 3.3 which also was correspondingly modified. In particular we modified configurations. Now, configurations are tuples $\langle \eta, d, e \rangle$ where $\eta$ is an execution trace, i.e., the sequence of events performed so far ($\varepsilon$ denotes the empty execution trace); $d$ is the current metric value; and $e$ is a $\lambda^{req}$ term, which describes the part of the service under evaluation. The operational semantics is driven by a composition plan $\pi$ which is responsible for providing a mapping between each service request and an
to draw attention to the rule \(\text{details} \[40\]).

Formally, a c-semiring is defined as follows (see the work of S. Bistarelli et. al., for more details).

Since we provide a generic approach for service composition, we would like quantitative operations to be generic as well. For this purpose we use the mathematical structure called c-semirings [40]. A c-semiring consists of a set of values \(D\) (e.g., natural or real numbers), and two types of operators: multiplication (\(\otimes\)) and summation (\(\oplus\)) of values and constraints. Formally, a c-semiring is defined as follows (see the work of S. Bistarelli et. al., for more details [40]).

**Definition:** A c-semiring \(T\) is a tuple \((D, \oplus, \otimes, 0, 1)\) where

- \(D\) is a (possibly infinite) set of elements and \(0, 1 \in D\);
- \(\oplus\), being an addition defined over \(D\), is a binary, commutative (i.e., \(d_1, d_2 \in D \Rightarrow d_1 \oplus d_2 = d_2 \oplus d_1\)) and associative (i.e., \(d_1, d_2, d_3 \in D \Rightarrow d_1 \oplus (d_2 \oplus d_3) = (d_1 \oplus d_2) \oplus d_3\)) operator such that \(0\) is its unit element (i.e., \(d_1 \oplus 0 = d_1\));
- \(\otimes\), being a multiplication over \(D\), is a binary, commutative and associative operator such that \(1\) is its unit element and \(0\) is its absorbing element (i.e., \(d_1 \otimes 0 = 0 \otimes d_1\));
- \(\otimes\) is distributive over additive operator \((d_1 \otimes (d_2 \oplus d_3)) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3)\);

In this work we focus on a special subset of c-semirings:

| \(\epsilon, \epsilon'\) ::= | unit | \(\lambda \cdot x . e \equiv \lambda \cdot x . e\) with \(z \not\in f v(e)\)
| --- | --- | --- |
| \(\epsilon\) | resource | \(\lambda . e \equiv \lambda . e\) with \(x \not\in f v(e)\)
| \(\alpha(e)\) | access event | \(\epsilon; \epsilon' \equiv (\lambda . e')e\) fork \(e\) and \(\epsilon' \equiv (\epsilon'; \lambda . x . e)\)
| \(\text{if } b \text{ then } e \text{ else } e'\) | branch | \(\langle \text{req}_\nu , \tau \rightarrow \tau' \rangle e \equiv \nu \{ \langle \text{req}_\nu , \tau \rightarrow \tau' \rangle e \}\) with \(f v\) the standard function returning the set of free variables of an expression \(e\).
| \(\varphi[e]\) | application | Security framing
| \(\gamma(e)\) | metric framing | Metric check
| \(\text{req}_\nu , \tau \rightarrow \tau'\) | service request | 

**Table 3.1:** Syntax of \(\lambda^\nu e\) and abbreviations

**Table 3.2:** Syntax of \(\lambda^\nu e\) and abbreviations

actual service, in symbols \(\pi(\rho) = \ell\) where \(\rho\) and \(\ell\) are request and service identifiers, respectively. In the following we also use \(\rightarrow^*_\gamma\) for the transitive closure of \(\rightarrow_{\text{def}}\).

Although, most of the rules simply take into account the changes in the structure of the configurations, we want to draw attention to the rule \(\text{(S-Ev}_2\)). The rule says that if the action target reduces to a resource \(r\), the action takes place and the current history \(\eta\) is extended with the corresponding event \(\alpha(r)\) and the current metric is updated with the metric value for the event \(\alpha(r)\).

We also added corresponding modifications to the type and effect system, but these changes are not crucial for the main topic of the current work and we omit them for simplicity and brevity.

### 3.1.2 C-semiring

Since we provide a generic approach for service composition, we would like quantitative operations to be generic as well. For this purpose we use the mathematical structure called c-semirings [40]. A c-semiring consists of a set of values \(D\) (e.g., natural or real numbers), and two types of operators: multiplication (\(\otimes\)) and summation (\(\oplus\)) of values and constraints. Formally, a c-semiring is defined as follows (see the work of S. Bistarelli et. al., for more details [40]).

**Definition:** A c-semiring \(T\) is a tuple \((D, \oplus, \otimes, 0, 1)\) where

- \(D\) is a (possibly infinite) set of elements and \(0, 1 \in D\);
- \(\oplus\), being an addition defined over \(D\), is a binary, commutative (i.e., \(d_1, d_2 \in D \Rightarrow d_1 \oplus d_2 = d_2 \oplus d_1\)) and associative (i.e., \(d_1, d_2, d_3 \in D \Rightarrow d_1 \oplus (d_2 \oplus d_3) = (d_1 \oplus d_2) \oplus d_3\)) operator such that \(0\) is its unit element (i.e., \(d_1 \oplus 0 = d_1\));
- \(\otimes\), being a multiplication over \(D\), is a binary, commutative and associative operator such that \(1\) is its unit element and \(0\) is its absorbing element (i.e., \(d_1 \otimes 0 = 0 \otimes d_1\));
- \(\otimes\) is distributive over additive operator \((d_1 \otimes (d_2 \oplus d_3)) = (d_1 \otimes d_2) \oplus (d_1 \otimes d_3)\);
Therefore, in order to conduct the quantitative analyse it is enough to show that the applied example of the security metrics which could be considered as λ service should be expressed with and the appearing in history expressions. The rules in Table 3.4 define the correspondence between the history expressions the scope of the approach.

Finally, in Table 3.4 we propose a set of equivalences that we use to move and compose metric annotations appearing in history expressions. The rules in Table 3.4 define the correspondence between the history expressions and the c-semiring operators.

Now, we have all required machinery in order to perform secure composition of services. First, the composite service should be expressed with \( \lambda^{eq} \) and the atomic services as well. Then, the analyst must define the (security and quantitative) properties he/she would like to check. The next step is to define the corresponding c-semiring and find the values for the atomic operations (we assume these values are specified in the service level agreements).

\[ F(a, r) = d' \]

\[ \langle \eta, d, \alpha(r) \rangle \rightarrow_{\ast} \langle \eta(a(r), d \oplus d', \ast) \rangle \]

\[ e_2 : \tau \rightarrow \tau' \in \text{Sr}\]

\[ \langle \eta, d, (x \in \text{Sr}, \tau \rightarrow \tau') v \rangle \rightarrow_{\ast} \langle \eta, d, e_1 v \rangle \]

\[ \langle \eta, d, e_2 \rangle \rightarrow_{\ast} \langle \eta, d, e_1 e_2 \rangle \]

\[ \langle \eta, d, e_2 \rangle \rightarrow_{\ast} \langle \eta, d, e_1 e_2 \rangle \]

\[ \gamma \langle d\#H \rangle \equiv \gamma \langle H \rangle \quad \text{where} \quad \gamma = T \geq T \ d' \quad \text{and} \quad d = d' \oplus d' \]

\[ \mu h.H \equiv d\#h.H' \quad \text{where} \quad d = \bigoplus_n^{-1} \Phi^n(0) \quad \text{and} \quad \Phi(d) = d' \iff \begin{cases} H[d\#h/H] \equiv d'\#H' \\ \bigwedge \end{cases} \]

\[ \begin{array}{l}
\text{Definition:} \quad \text{c-semiring is a c-semiring with} \oplus \text{ satisfying the following condition: } \forall d_1, d_2 \in D \quad d_1 \oplus d_2 = d_1 \text{ or } d_1 \oplus d_2 = d_2 \\
\text{Definition:} \quad \leq_T \text{ is a total order over the set } D, \text{ such that } d_1 \leq_T d_2 \text{ iff } d_1 \oplus d_2 = d_2. \\
\text{In this work we need a reverse operation for summation } \oplus^{-1} \text{ which is defined as follows.} \\
\text{Definition:} \quad d_1 \oplus^{-1} d_2 = d_1 \text{ if } d_1 \oplus d_2 = d_2. \\
\text{In words, this operation always returns the worst possible value.} \\
\text{Property:} \quad \text{Operation } \oplus^{-1} \text{ is associative, commutative, idempotent, distributive over } \otimes, \text{ and monotone}.1 \\
\end{array} \]

The properties of c-semiring automatically infer whether the selected composition plan satisfies or fails the quantitative properties. Therefore, in order to conduct the quantitative analyse it is enough to show that the applied metric satisfies the conditions to be a c-semiring and follow the defined rules for aggregation of the metric. The example of the security metrics which could be considered as c-semirings are: cost, risk, trust, downtime, etc. Note, that assigning of the metric values to atomic services is a metric- and context-dependent task which is left behind the scope of the approach.

\[ \text{Table 3.3: Operational semantics of } \lambda^{eq} \]

\[ \text{Table 3.4: Equational rules.} \]

\[ \text{A link with proofs:} \text{http://www.iit.cnr.it/staff/artsiom.yautsiukhin/Resources/ICE-Proofs.pdf.} \]
with the help of type and effect system the atomic services from a service repository are mapped to the corresponding service requires and the historical expression (i.e., possible execution paths) is defined. Using the equational rules the analyst is able to aggregate metrics for the same path (using $\otimes$ operator) and select the worst one (using $\otimes^{-1}$ operator) for the specified metric frames. During the execution, applying the operational semantics the approach is able to follow the execution and compute a real value of the metric. If the value exceeds the limits of the quantitative property the execution may be halted (or other action, e.g., reporting, is performed.)

### 3.1.3 Example

The travel agency BestTravel offers a travel planning service to its customers. Figure 3.1 shows the abstract workflow of BestTravel. First the service searches for a direct flight and books it if one is found. If there is no a direct flight then an itinerary is found and booked. In parallel, a hotel is found and booked. Finally, the receipt is signed line-by-line.

A requirement of BestTravel is to have risk level of the performed tasks (in particular, flight booking, hotel reservation and receipt signature) less than 75. Risk, computed as annualised loss expectancy, could be seen as the following $c^*$-semiring $(\mathbb{N}^+ \cup \{\infty\}, min, +, \infty, 0)$, known as tropical semiring.

![Figure 3.1: Abstract workflow for BestTravel.](image)

We assume the existence of the resources: $A = \{\text{AIRPORT}\}$ and $C = \{\text{CITY}\}$, $I = \{\text{ITINERARY}\}$, $F = \{\text{FLIGHT\_No, NO\_FLIGHT}\}$, $H = \{\text{HOTEL\_RESV}\}$, $B = I \cup F \cup H$ and $D = \{\text{RCPT, SIGNED\_DOC}\}$. In Figure 3.2 we propose a $\lambda^{eq}$ implementation of the workflow of the BestTravel service, called $e_B$.

A repository contains a number of services with 10 services relevant for the BestTravel with pre-computed values of the metric (Table 3.5). We type the BestTravel implementation $e_B$ as in Figure 3.3. We call $H_B$ the latent effect labelling the arrow type of $e_B$.

Now we are able to apply Equational rules and compute the metric values for the three sub-process in question:

![Figure 3.2: Implementation of BestTravel.](image)
<table>
<thead>
<tr>
<th>Abstract service</th>
<th>concrete service</th>
<th>type mapping</th>
<th>Risk value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search direct flights</td>
<td>$H_1$</td>
<td>$A \rightarrow F$</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>$H_2$</td>
<td>$A \rightarrow F$</td>
<td>15</td>
</tr>
<tr>
<td>Search itinerary</td>
<td>$H_3$</td>
<td>$A \rightarrow I$</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>$H_4$</td>
<td>$A \rightarrow I$</td>
<td>15</td>
</tr>
<tr>
<td>Search a hotel</td>
<td>$H_5$</td>
<td>$C \rightarrow H$</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>$H_6$</td>
<td>$C \rightarrow H$</td>
<td>50</td>
</tr>
<tr>
<td>Book ...</td>
<td>$H_7$</td>
<td>$B \rightarrow D$</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>$H_8$</td>
<td>$B \rightarrow D$</td>
<td>25</td>
</tr>
<tr>
<td>Sign a line</td>
<td>$H_9$</td>
<td>$D \rightarrow D$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$H_{10}$</td>
<td>$D \rightarrow D$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.5: Concrete services

\[
e_B : \text{unit} \rightarrow \mathcal{D}
\]

\[
\gamma \left( (H_1 + H_2) \cdot \left( \frac{(H_7 + H_8)}{(H_5 + H_6) \cdot (H_7 + H_8)} \right) \right) \cdot \gamma \left( \frac{(H_5 + H_6)}{(H_7 + H_8)} \right) \cdot \gamma (\mu h.((H_9 + H_{10}) \cdot h + \epsilon))
\]

Figure 3.3: Type of BestTravel.

- $M(\text{Flight booking}) = \max(20, 15) + \max(\max(28, 25), (\max(25, 15) + \max(28 + 25))) = 73$;
- $M(\text{Hotel reservation}) = \max(40, 50) + \max(28, 25) = 78$
- $M(\text{Receipt signature}) = \max(1, 0) \times \infty = \infty$

We can see, that only the flight booking sub-process satisfies the required property (i.e., $73 < 75$). For other sub-processes we need to establish monitoring process, which is at run-time will decide whether the concrete execution fails the property or it does not. Note, that for both sub-processes exist the execution path which satisfy the property.

3.1.4 Conclusion

The present work is a first step toward a complete model for the specification and verification of quantitative and qualitative, non functional requirements for web services. The result is a unified framework for (i) the definition and application of security and metric policies within service implementation, (ii) the automatic extraction of history expressions carrying metric annotations and (iii) the computation (through an equational theory) of metric values which safely predict the expected behaviour of services.

Further effort is requested in order to generalise our approach. In particular, we aim at defining a procedure for generating orchestration plans starting from the history expressions produced by our type and effect system. Another limitation of the current model is our static description of metric value for the events. Even though we think that assigning metric values to events is a reasonable way to model the actual behaviour of services, it is not always correct to assume these values to keep unchanged in time. Indeed, many metrics aim at modelling dynamic evolution of some property, e.g., reputation or number of system failures, which we cannot model with our approach.

3.2 Modeling and Enforcing Secure Navigation Paths

The success of web applications is fundamentally based, among other factors like high-fidelity and user-friendliness, upon the guarantee of data protection. It creates confidence to the web user and enables protected data transfers of sensitive data over a basically unsafe medium like the Internet. Therefore, new security frameworks have been designed and developed to better protect web applications from unauthorized access. In general, security frameworks already apply the four pillars of security and data protection for web applications:

i. Authentication (the process of proving the identity of a user to gain access to a protected resource)
ii. Authorization / Access control (the process of authorization determines what a subject (e.g., an user or a program) is allowed to access, especially what it can do with specific objects (e.g., files) [22])

iii. Encryption (the conversion of data into a form that cannot be understood by unauthorized subjects)

iv. Session management (the process of tracking user-specific application data during the time he is authenticated to the system)

Examples of such security frameworks include Spring Security\(^2\) and Apache Shiro\(^3\). Both were designed to provide a high standard of security as comfortable as possible for already implemented web applications. Therefore, they gained popularity and they are used and recommended by a large community of application providers.

However, there is one important aspect of security in the area of data protection which has not been considered concretely yet: A kind of navigational access control which guarantees that every web user with a certain role has only a limited number of navigation paths inside the application context. We call them Secure Navigation Paths (SNPs). Suppose a web application procedure managed to open an online-banking-account, which consists of about ten steps. What happens when a user, whether intentionally or not, jumps from step two, “indicating the personal data”, to the last step, “confirmation”, simply by calling the corresponding URL, and confirms the transaction? This can not only lead to fatal inconsistencies on the application state, but may also cause a worst case scenario like losing money or damaging the image of the application provider.

Regarding the addressed issue, our goal is to fill the gap of missing possibilities to model and control SNPs to enhance data protection within web applications. We start by developing a first approach on how to design SNPs. In addition, we implement an innovative and generic monitor module which is capable to provide Role Based Access Control (RBAC) considering SNPs for JSF-based web applications. Thus safety-critical jumps through the different context views of the application, called navigation nodes, should be avoided. Finally, we provide a new plugin for the Computer-Aided Software Engineering (CASE) tool MagicDraw\(^4\). Basically, this plugin validates the designed security models which contain SNP semantics and it is able to extract the corresponding navigation rules.

In order to develop our approach on how to design access control considering SNPs we decided to use the UML-based Web Engineering (UWE)\(^5\) approach. UWE extends the UML profile by a large set of useful security and web features. Basically, we use UWE’s Navigation State Model, a UML state machine, to be able to design RBAC for web applications considering SNPs using states and transitions.

Our new CASE tool plugin MagicSNP for MagicDraw is a tool to validate the designed security model and moreover to extract the corresponding access-control-semantics. By iterating recursively through all hierarchical states and by analyzing the incoming transitions, state names and tags, this tool fetches and converts the relevant information into a semi-structured data format like JSON. Consequently, this plugin secures, accelerates and reduces the complexity of the handover between the modeling and the implementation progress of the web application development.

In order to be capable to provide RBAC with SNPs for web applications considering the modeled access-control-semantics we develop a new generic monitor module approach. This module is responsible to decide whether or not a user is allowed to get access to a protected resource. The decision making is based on the web user’s session information (e.g., previously visited location, assigned user roles etc.) and the extracted access-control rule file, generated by our new MagicDraw plugin. In order to ensure robustness, this monitor module also handles any kind of access-constraint violation: The web user gets redirected to a corresponding error-page including an appropriate error-message with possibility to go back to its previously visited navigation-context.

By using our new modeling approach, we design RBAC considering SNPs in the design-phase. Then we use our new MagicDraw plugin to validate our security-model and to extract the corresponding navigation-rule file. Finally, we apply our generic monitor module and inject the navigation-rule file in the implementation-phase. As a result we get a secure and robust web application which excels the security standard of modern web applications by using and applying our approaches.

Our tools can be downloaded from the MagicSNP website\(^6\)\(^\text{a}\) and more information about the tools and our case study can be found in [133].

\(\text{\textsuperscript{b}}\)Apache Shiro. http://shiro.apache.org/
\(\text{\textsuperscript{d}}\)UWE. http://uwe.pst.ifi.lmu.de
\(\text{\textsuperscript{e}}\)MagicSNP. http://uwe.pst.ifi.lmu.de/toolMagicSNP.html
3.3 Federated authorization for SaaS applications

Software-as-a-Service (SaaS) applications are a part of cloud computing in which centrally hosted web-based applications are offered to a large number of tenants (customer organizations each representing multiple end-users), each using multiple applications. From the point of view of the tenant, cloud computing and SaaS are a form of outsourcing using a pay-per-use billing scheme. From the point of view of the provider, SaaS is the next step in Application Service Provider (ASP) evolution, trying to cut operational costs by sharing resources of the application and offering it on a larger scale.

As mentioned in the previous paragraph, SaaS applications are a form of outsourcing: the application remotely hosts and processes data actually belonging to the tenant. Therefore, access control in SaaS applications is mainly about protecting the tenant’s data located at the provider’s side. The tenant controls the administration and information about the users of the application (e.g., employees of the tenant) and the access control rules and policies.

A first problem with access control for remote applications is administrative scalability. From the tenant’s point of view, every application has to be provided with the necessary user information. Therefore, federated authentication techniques such as WS-Federation [26], OpenID [10] and SAML [9] have been developed. These techniques allow users to be authenticated in their home domain, after which the needed user information is securely exchanged with the application. This centralizes user management with the tenant and offers control over the shared information and the authentication means used.

With the evolution towards core-business SaaS applications, more control over the application and the data it uses is required, raising the need for authorization. While federated authentication techniques allow tenants to centralize user management, similar techniques for authorization currently do not exist and all access control policies are evaluated at the provider’s side. Because of this, access control policies are distributed and fragmented over the multiple SaaS applications the tenant uses. This increases the administrative load, eventually leading to inconsistencies and security holes.

A second problem with this approach, is the necessary disclosure of confidential tenant information. Because the access control policy is evaluated at the provider’s side, all necessary information has to be shared with the provider. While the tenant trusts the provider with the information in the application itself, it is likely that more information is needed for authorization. For example, in the patient monitoring system, the list of all patients the physician is currently treating or the fact that the patient is currently being treated by an oncologist is confidential information. Moreover, it can be the case the tenant does not want to share the policy itself, for example how the hospital handles its competitors when requesting access to its data.

The work in this section describes initial research in order to reach a solution for these problems. The work in this deliverable is described in more detail in [62] and was presented at the NESSoS Doctoral Symposium.

3.3.1 Proposed solution

In this subsection we summarize an initial solution we envision to solve the two problems described above: their limited administrative scalability and the necessary disclosure of confidential information. The solution we envision consists of externalizing policy evaluation from the SaaS application to the tenant. With each request, the SaaS application would ask the tenant for an access control decision, after which the tenant himself evaluates the necessary policies and returns his decision. This way, tenant policies remain centralized and confidential information does not have to be shared with the application provider. Since this achieves for authorization what federated authentication did for authentication, we call this federated authorization.

The objective of this research is to incept and evaluate support for federated authorization in security middleware. A high-level technical overview of how we currently envision this middleware is given in [62] using XACML [119] terminology for the components. When a user sends a request to the SaaS application to perform an action, the Policy Enforcement Point (PEP) is called from the application in order to determine whether the user is allowed to do so. It therefore calls the local Policy Decision Point (PDP), which will evaluate tenant policies loaded from the Policy Administration Point (PAP) using information stored in the Policy Information Point (PIP). The provider’s PDP also forwards the request to the tenant PDP, which in turn evaluates the tenant’s policies using local data and returns its decision. The requested action is only allowed in case both parties allow it.

In order to achieve this objective, both the policy language and the execution environment will have to be extended. The policy has to support the necessary constructs in order to declare the needed properties of the federated authorization. Future research will have to show which constructs are minimally needed. For the execution environment, the communication between the provider and the tenant will most likely be the crucial part, since the
information about the request, the subject, the object and the system are now distributed over the two parties.

3.3.2 Challenges

Federated authorization as described above builds upon and extends externalized authorization, a concept that has been researched in the past (e.g., [87]). Moreover, the XACML access control standard already defines a profile for integration with SAML for transport of both user information and authorization decisions [102]. However, no research about the practical feasibility of combining externalized authorization with federation into federated authorization is present to the best of our knowledge. An important aspect of this are the security trade-offs implicitly made in federated authorization. For example, externalizing access control to the tenant increases administrative scalability, but also makes the system more susceptible to denial of service attacks and traffic analysis.

One major challenge we predict is the performance and scalability of the security middleware. This challenge is increased by taking into account concurrency and distributed data updates. Related research has led to techniques for improving access control performance (e.g., Wei [149] proposes decision recycling and inference, Brucker and Petritsch [43] focus on the retrieval of the necessary user information), some of which can also be applied for scalability. However, SaaS applications frequently use aggressive scalability techniques [136] and their applicability on access control still has to be investigated.

3.3.3 Research methodology

In this research, a case-study driven methodology is used. Two major case studies are used: (i) a home patient monitoring system and (ii) an electronic document processing system. The security analyzes of both case studies offer requirements and policies to be applied in our prototypes. The former offers insights in the stringent legal requirements concerning medical data and the complex, fine-grained access control policies in e-health, the latter offers a tenant hierarchy, a concept which should be supported by federated authorization.

Currently, this research is in its early phase. Involvement in the case studies stated above has shown the need for solving this problem, after which a literature study of existing authorization techniques was made. As the next step in this research, our concept of federated authorization will be refined into a reference architecture. In order to build upon existing work, we will base this on widely-used standards such as XACML [119] as much as possible. This reference architecture will then be instantiated into a proof of concept using the case studies in order to evaluate the feasibility of federated authorization. The next step will be to apply and empirically evaluate performance and scalability techniques on this using metrics such as the impact of the complexity of policies (e.g., the number of subject and object attributes needed) on the response time of the whole.

3.3.4 Main contributions

Next to SaaS applications, federated access control has been researched in other domains as well, such as web applications (e.g., OpenId [10]), grid computing (e.g., PERMIS [48]) and web services (e.g., WS-Federation [26]). These domains offer relevant techniques, but their purpose differs from SaaS. For example, access control in grid computing is mainly about access to on-premise resources from other domains. Moreover, the described techniques still limit themselves to local policy enforcement based on federated authentication.

Some literature has been published about access control for multi-party systems. Zhang et al. [152] describe a framework for usage control in collaborative systems, taking into account access control data updates. Stihler et al. [140] also describe an architecture which enables the involved parties to declare access control policies on the application. However, both still employ provider-side policy enforcement, resulting into the problems described before.

Existing technologies for access control also do not support federated authorization. Products such as IBM Tivoli Access Manager allow to centrally enforce access control policies on multiple applications. Moreover, by acting as an XML gateway these policies can be applied on external applications. This set-up is however limited to the level of messages, thereby limiting the complexity of supported policies. Moreover, this limits the mobility of the users of the external application, making it inapplicable for SaaS applications.
3.4 Composing Trust Models Towards Interoperable Trust Management

In D8.2, we had sketched the basic motivations behind trust interoperability, which has inspired our recent work on TMDL (Trust Model Description Language) to express and compose a wide range of trust management systems and thereby support trust management across heterogeneous networked systems. The composition of the trust models of these systems is specified in terms of mapping rules between roles of the original models. Rules are then processed by a set of mediation algorithms to overcome the heterogeneity of the trust metrics, relations and operations associated with the composed trust models.

During the past year, we enhanced and improved the TMDL language and provided its complete and final XML representation. We also implemented several dedicated tools that (i) guide developers to check and create a valid TMDL description; (ii) automatically generate from such description the Java code of the corresponding trust management system; and (iii) enable the composition of any given trust management system according to given mapping rules.

Our work involves 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 to specify 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.

Networked Systems (NSs) are heterogeneous and often implement different trust management systems, which need to be enriched continuously to be able to interact and interoperate with the trust management system of discovered NSs. As part of the CONNECT project, we have been working on a Trust Enabler to support and continually adapt and enhance the CONNECT trust management system. The Trust Enabler is not responsible for monitoring the behavior of NSs, but instead provides an interface to other Enablers to be warned of any monitored misbehavior, for instance: the Security Enabler warns the Trust Enabler if any policy is violated and hence the trustworthiness of the corresponding NS is decreased.

In Figure 3.4, we show an overview of the CONNECT Trust Management System. It is mainly made up of the following roles:

- The Trust Enabler Role ($R_{TE}$): It represents the entry point of the trust management system. It provides to the CONNECT Enablers all trust operations that allow to retrieve and update the trustworthiness value of the discovered NSs.

- The NS Role ($R_{NS}$): It is the role given by the Trust Enabler to all discovered NSs. To do so, each discovered NS is considered by the CONNECT trust management system as a participant that plays the $R_{NS}$ role.

- The Trust Enabler Mediation Role ($R_{mTE}$): this role performs a mediation between the CONNECT trust management system and the NS trust management system. In other words, it plays the role of a recommender that can be requested by the $R_{TE}$ to retrieve NS’s trustworthiness values and also propagate any CONNECT feedback.

- The NS mediation Role ($R_{mNS}$): Similarly to $R_{mTE}$, the $R_{mNS}$ is a mediator that plays the roles of trusted recommenders to bridge the trust management systems of the CONNECTed NSs, transparently.

The Trust Enabler is built with specific modules that are triggered automatically when an NS is discovered (i.e., the “TMDL Loader”) and when a CONNECTor is deployed (i.e., the “Mapping Loader”). The “TMDL loader” processes the given TMDL description of a discovered NS and produces a TMDL description of a mediator($R_{mTE}$) that enables the $R_{TE}$ role to interact with the trust management system of the discovered NSs. Similarly, the “Mapping Loader” processes the TMDL description of the CONNECTed NSs and generates a TMDL description of a trust mediator ($R_{mNS}$) that is able to compose the trust management systems of these NSs.

Then comes the role of the “Trust Role Generator” module which processes the TMDL description of the mediators and generates the corresponding Java code. Instances of these mediators are then deployed within the CONNECT Trust Management System and bound to the Interaction Enabler to be able to interact using any middleware protocol handled by that Enabler.

As described above, the Trust Enabler requires some engines able to process TMDL descriptions, generate trust mediators and deploy instances of those mediators into the CONNECT trust management system.
3.4.1 Trust Model Description Language

In order to recall the functioning of the Trust Model Description Language, let us dissect the following very generic definition of a trust relationship: *A trustor trusts a trustee with regard to its ability to perform a specific action or to provide a specific service* [72].

- **Trustor and Trustee**: are participants of the trust model and can be identified by the *Roles* they play.
- **Trustor trusts a trustee**: represents a trust *Relationship* that links a participant (trustor) to another (trustee). The action *trust* is a *Decision* that is based on the evaluation of a specific trust *Metric*.
- **Perform a specific action or Provide a specific service**: represent the *Context* where a specific trust relationship is established.

In summary, to define a trust model, we have to identify the following trust elements: (i) The *Domains* where trust elements are defined; (ii) The *Roles* played by the participants involved; (iii) The *Metrics* for assessing the trustworthiness of the participants; (iv) The *Contexts* where trust is required; (v) The trust-related *Relations* that link participants, and finally (vi) The *Decisions* provided by the model from the assessment of trust relationships.

All the TMDL trust elements can be enriched semantically by making reference to specific ontologies. Since TMDL is an XML-based language, semantic references are annotated with SAWSDL attributes, namely: (i) *modelReference* to identify the corresponding ontological concept, (ii) *liftingSchemaMapping* and (iii) *loweringSchemaMapping* to specify the mappings between semantic data and trust element data (see Table 3.6).

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Type</th>
<th>Optional</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotation</td>
<td>TMDL Element</td>
<td>Tag</td>
<td>Yes</td>
<td>One</td>
</tr>
<tr>
<td>-modelReference</td>
<td>SAWSDL Attribute</td>
<td>URL</td>
<td>No</td>
<td>One</td>
</tr>
<tr>
<td>-loweringSchemaMapping</td>
<td>SAWSDL Attribute</td>
<td>URL</td>
<td>Yes</td>
<td>One</td>
</tr>
<tr>
<td>-liftingSchemaMapping</td>
<td>SAWSDL Attribute</td>
<td>URL</td>
<td>Yes</td>
<td>One</td>
</tr>
</tbody>
</table>

Table 3.6: SAWSDL Annotation: XML Definition
3.4.2 Composing Trust Models

The aim of composing two different trust models ($TM_X$ and $TM_Y$) is to provide participants of a given model (which we call target model $TM_T$) the ability to interact (i.e., create, assess and retrieve relationships) with participants that come from another model (which we call source model $TM_S$).

In our previous work [131], we introduced a trust-centric composition process that relies on a set of mapping rules $\Psi_{xy}$ defining how roles (i.e., $r_S$) from the source model are mapped ($\triangleright$) to roles (i.e., $r_T$) from the target model, as follows:

$$\Psi_{xy} = \{ \psi^k_{ST} | \psi^k_{ST} \triangleright r_S \triangleright r_T \}$$

Where $S, T \in \{x, y\}, S \neq T$.

During the past year we enhanced and adapted the trust composition process according to the actual enhanced formalization of TMDL. We focused on a concrete implementation of trust tools to support given TMDL descriptions and to perform mediation. This required enhancement of the algorithms introduced last year to provide a concrete implementation of the composition process. In more detail, we identified and fully implemented the two following composition processes:

- The first process is to merge trust models. It aims at creating a new trust model that results from merging the two given ones. We call this process “Merging Composition”. The composition is hence permanent and leads to extension of the behavior of the trust models’ roles according to the given mapping rules.

- The second process is to create a cooperation that might happen for a limited period of time. The objective of this process is to not modify the behavior of existing roles, but create a set of mediation roles capable of bridging the two given models. We call this process “Mediation Composition”.

In order to define the composition process formally, in the table below, we illustrate all the notations that will be used:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tor_l$</td>
<td>Trustor role of the relation $l$</td>
</tr>
<tr>
<td>$tee_l$</td>
<td>Trustee role of the relation $l$</td>
</tr>
<tr>
<td>$req_l$</td>
<td>The trustor role of the access relation defined for operations of the relation $l$ or $null$ if no access relation is defined</td>
</tr>
<tr>
<td>$D_x$</td>
<td>The set of $Domains$ defined in the trust model $x$</td>
</tr>
<tr>
<td>$R_x$</td>
<td>The set of $Roles$ defined in the trust model $x$</td>
</tr>
<tr>
<td>$M_x$</td>
<td>The set of $Metrics$ defined in the trust model $x$</td>
</tr>
<tr>
<td>$C_x$</td>
<td>The set of $Contexts$ defined in the trust model $x$</td>
</tr>
<tr>
<td>$L_x$</td>
<td>The set of $Relations$ defined in the trust model $x$</td>
</tr>
<tr>
<td>$S_x$</td>
<td>The set of $decisions$ defined in the trust model $x$</td>
</tr>
<tr>
<td>$l_x \approx l_y$</td>
<td>The relation $l_x$ is defined to be similar to $l_y$</td>
</tr>
<tr>
<td>$l_x \leftarrow l_y$</td>
<td>The relation $l_x$ is derived from the relation $l_y$</td>
</tr>
</tbody>
</table>

Table 3.7: TMDL Formal Notations

Merging composition process

In order to merge two exiting trust models, we define two mapping operators, namely: the extension ($\triangleright$) and the equivalence ($\equiv$) operators (i.e., $\triangleright \in \{\triangleright, \equiv\}$). As consequence:

- The extension rule $r_S \triangleright r_T$ leads to extend the inheritance attribute of the source role $r_S$ with the target role $r_T$.

- The equivalence rule, $r_S \equiv r_T$ leads to define a new role ($r_{new}$) that inherits both the source $r_S$ and the target $r_T$ roles.

Formally, $r_S \equiv r_T \models (r_{new} \triangleright r_S) \land (r_{new} \triangleright r_T)$.
Thus, the merging composition, denoted by $\Psi_{xy}$, of two trust models $TM_x$ and $TM_y$, introduces a new trust model $TM_z$, as follows:

\[
TM_z = TM_x \bigoplus_{\Psi_{xy}} TM_y
= < D_x, R_x, M_x, C_x, L_x, S_x > \bigoplus_{\Psi_{xy}} < D_y, R_y, M_y, C_y, L_y, S_y >
\]

\[
\begin{align*}
D_z &= D_x \cup \{d_{xy}\} \\
R_z &= R_x^+ \cup R_y^+ \cup R_{xy} \\
M_z &= M_x \cup M_y \\
C_z &= C_x \cup C_y \\
L_z &= L_x \cup L_y \\
S_z &= S_x \cup S_y
\end{align*}
\]

(3.2)

Where the $R_x^+$ and $R_y^+$ notations means that, in addition to the roles of each models, each set contains also the roles that are extended according to the given extension rules. Finally, the $R_{xy}$ is the set of all new roles issued from an equivalence rule. These new roles are hence defined within a new domain $d_{xy}$.

Figure 3.5: The Mediation Process
Mediation composition process

Since the cooperation is temporary, the mediation process preserves the existing trust models, and can be applied at runtime by deploying some recommenders (i.e., mediation roles) that are able to perform operations across models (i.e., mediation). This mediation process hence generates a new trust mediation model where the required mediation roles are defined.

In the context of cooperation, we express mapping rules with the play operator (▷). The play rule \( r_S ⊸ r_T \) basically defines which role form the target model the participants of the source model (identified by their role) are going to play when they visit the target model. Thus, in order to help roles of the target model to establish relationships with the source role as if it is a target role, for each mapping rule we define a new mediation role. This role is going to work as (i) a recommender to enable roles of the target model that want to assess the source role to request the mediation that works also as (ii) a requestor able to retrieve and update the trustworthiness of the source role from the target model requests. As illustrated in Figure 3.5, applying a play rule is performed in four steps:

1. **Build the target relation set (\( L_t^{r_t} \)):** It represents all the relations (\( l_t \)) that assess the target role (\( r_t \)) and give a remote access to their get or set operations. In this case the mediator can inherit from the trustor role of that relation and hence be a recommender, Formally:
   \[
   L_t^{r_t} = \{ l \in L_t \mid (tee_l = r_t) \land (req_l \neq \text{null}) \}
   \]  
   (3.3)

2. **Build source relation set (\( L_s^{r_s} \)):** It represents all the relations (\( l_s \)) that assess the source role (\( r_s \)) and provide a remote access to their operation. The mediator thus can be a requestor able to answer/propagate any request that come from the target models. To do so, the mediation role will have to inherit from the requestor role (\( req_{l_s} \)) of the relation. Formally:
   \[
   L_s^{r_s} = \{ l \in L_s \mid (tee_l = r_s) \land (req_l \neq \text{null}) \}
   \]  
   (3.4)

3. **Build the mapping set \( \mathcal{M} \):** in order to allow the mediation role to be requested by the role of the target model and answer by requesting roles of the source model, the mediation process build the mapping set that contains pairs of similar relations, i.e., it has to find for each target relation a corresponding similar source relation. The similarity is mainly based on the trust context and have to be validated by the administrator. However, if relations are semantically annotated, similarity can be automatically states by interrogating their corresponding ontology. Formally:
   \[
   \mathcal{M} = \{ (l_t, l_s) \in L_t^{r_t} \times L_s^{r_s} \mid l_t \approx l_s \}
   \]  
   (3.5)

4. **Create the trust mediation model:** Now that all inputs are found, we are able to create the trust mediation model. It defines the new mediator role \( \mu_{new} \) that inherits both the trustor roles of the target relations (\( \text{tor}_{l_t} \)) and the requestor roles \( \text{req}_{l_s} \) of the source relations. All target relations that are managed by the mediation role have to be derived from source relations since the mediator will request the source relation to assess the target relation. Therefore, whenever the mediator is solicited for a target relation, it will perform the corresponding operation on the derived source relation. Formally:
   \[
   r_S ⊸ r_T \models \forall (l_t, l_s) \in \mathcal{M} \mid (\mu_{new} \triangleright \text{req}_{l_s}) \land (\mu_{new} \triangleleft \text{tor}_{l_t}) \land (l_t \leftarrow l_s)
   \]  
   (3.6)

Summarizing, the mediated composition, denoted by \( \otimes \), of two trust models \( TM_x \) and \( TM_y \), which produces the trust mediation model \( \mu TM_{xy} \), is defined as follows:

\[
\mu TM_{xy} = TM_x \otimes TM_y
\]

\[
\mu TM_{xy} = \langle \mathcal{D}_{xy}, \mathcal{R}_{xy}, \mathcal{M}_{xy}, \mathcal{C}_{xy}, \mathcal{L}_{xy}, \mathcal{S}_{xy} \rangle \overset{\Phi_{xy}}{=} \langle \mathcal{D}_{xy}, \mathcal{R}_{xy}, \mathcal{M}_{xy}, \mathcal{C}_{xy}, \mathcal{L}_{xy}, \mathcal{S}_{xy} \rangle
\]

\[
\begin{align*}
\mathcal{D}_{xy} &= \mathcal{D}_x \cup \mathcal{D}_y \cup \{ d_m \} \\
\mathcal{R}_{xy} &= \mathcal{R}_x \cup \mathcal{R}_y \cup \mu R \\
\mathcal{M}_{xy} &= \mathcal{M}_x \cup \mathcal{M}_y \\
\mathcal{C}_{xy} &= \mathcal{C}_x \cup \mathcal{C}_y \\
\mathcal{L}_{xy} &= \mathcal{L}_x \cup \mathcal{L}_y \\
\mathcal{S}_{xy} &= \emptyset
\end{align*}
\]  
(3.7)
where $D_x, D_y, R_x, R_y, M_x, M_y, C_x, C_y, L_x, L_y$ are sub sets of the original models’ trust sets and contain only the trust attributes that lead to the creation of the mediation roles $\mu_{r_{new}}$. $d_m$ represents the mediation domain where all the mediation roles included in $R$ are defined.

### 3.4.3 TMDL Tools

Our work on the TMDL language has given rise to two software tools. The former is a TMDL editor plugin for Eclipse, generated with EMF (Eclipse Modeling Framework\(^7\)). The latter is a Java application that we called $iMTrust$ (grey boxes). $iMTrust$ allows developers to (i) generate Java code of trust management systems from TMDL descriptions, (ii) emulate, test and deploy a network of participants that plays the roles described in the corresponding TMDL and finally (iii) compose heterogeneous trust management systems. We refer the reader to the $iMTrust$ website (https://www.rocq.inria.fr/arles/software/tmdl/) for detailed documentation and downloads.

### 3.4.4 Future Directions

As an interesting area of trust model interoperability, two NESSoS partners (KUL and Inria) are currently in the process of investigating its applicability for federated social networks, especially in the health-care domain. See Section 6.2 for more details.

---

\(^7\)EMF: www.eclipse.org/emf
4 Programming Language Support

The core components of secure Future Internet services are the individual building blocks that make up these services. In order to make sure that these base components offer the required guarantees that are needed to build secure services, new tools are needed to help the developers write code that fulfills these needs. Thus, in order to solve the bigger picture, the consortium also needs to focus on these individual parts that are composed into a Future Internet service.

To this end, Work Package 8 developed a strategy of software verification, where a verification tool can mathematically prove that an annotated application will never violate a predefined set of security requirements. In deliverable 8.2, we introduced the VeriFast program verifier, a sound and modular program verifier for common programming languages such as C and Java. It 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.

During the past year, this work has been improved and new results have been published. There are in particular two new research directions: (1) the theoretical underpinnings of the software have been formalized and the soundness of the symbolic execution w.r.t. the concrete execution has been proven, and (2) the prototype has been valorized on a number of industrial test cases.

Section 4.1 presents a short overview of the results of the formalization of VeriFast. Section 4.2 sketches the first results of the valorization of the current VeriFast prototype.

4.1 Featherweight VeriFast

VeriFast [82] is a static verification tool for C and Java. It supports both single-threaded and multithreaded [81] code and it has been used in real-world applications [124, 83]. The tool requires the code to be annotated using separation logic [126] formulae and ghost commands. In this section, a short overview of the formalization of a subset of VeriFast with a proof of its soundness is given.

Ideally, verification of a program would consist of simply executing this program while taking into account all possible external inputs (i.e. forking execution into $N$ paths whenever an interaction with the outside world can yield $N$ different results) and checking that no failures (i.e. violations of the targeted safety properties such as accesses of unallocated memory) are encountered. This straightforward approach is represented in this section by the concrete execution. Due to external input, it is nondeterministic and a single execution potentially consists of many different paths, each yielding a different result. Since each path must succeed, we say this nondeterminism is demonic.

However, the infinite number and length of such paths makes this approach infeasible. In order to make verification practicable, we need a finite simulation of this concrete execution, which we call the symbolic execution. One important property of this execution we wish to emphasize here is that it introduces a second, angelic kind of nondeterminism. This follows from the fact that the symbolic execution requires extra annotations to be added to the program code which in some cases are ambiguous in meaning. Each time such an ambiguity is encountered, execution forks, but, contrary to demonic nondeterminism, only one path is required to succeed. While this angelic nondeterminism adds complexity, it has the advantage of allowing us to formalize the execution semantics in a very elegant way.

In [146], we prove that the symbolic execution is sound with respect to the concrete execution, more specifically, that it is safe to use for the purposes of verification. This section summarizes this result.

4.1.1 Semantic Framework

To succinctly define concrete and symbolic execution of programs, we build on a semantic framework. This framework is a kind of monadic domain-specific language similar in spirit to the monadic approach to denotational semantics pioneered by Moggi [111] in the early nineties.

We defined an algebra that models outputs of executions taking into account angelic and demonic choice. Several implementations of this algebra have been formalised in Coq [55]. Then, we defined a monadic DSL on top of that algebra that handles most of the issues related to non-determinism, failure and nontermination under the hood. The operators in this DSL are used to conveniently define concrete and symbolic execution. Operators are composable functions on program states which allow us to considerably simplify the process of formally specifying both concrete
and symbolic executions as they hide the complexity of having to deal with failure, nontermination and two kinds of nondeterminism. While it is possible to express our execution semantics using these primitive operators, it can quickly become a syntactic mess. For this reason we extend operators with return values and slightly adapt the binding, so as to become something similar to Haskell’s monads.

4.1.2 Formalization of VeriFast

In this section, we present a formalization of VeriFast’s core. For this, we first need a small language on which to operate. The small imperative language (SIL) is a minimal language on which we’ll perform our verification. It yields very few surprises: cons and dispose allocate and deallocate memory, respectively. \( x := [y] \) and \([x] := y\) represent reads and writes to the heap, respectively. The full definition is as follows:

\[
e ::= n \mid x \mid e + e \mid e - e \quad n \in \mathbb{N}, x \in \text{Vars}
\]

\[
c ::= \text{skip} \mid x := e \mid [x] := x \quad b ::= e \equiv b \mid e < b \mid \neg b
\]

\[
| \text{if } b \text{ then } c \text{ else } c \mid x := \text{cons} \quad \text{Routine} ::= \text{routine } r(x) = c
\]

\[
| \text{dispose}(x) \mid c; c \mid r(x) \quad \text{Program} \in \mathcal{P}(\text{Routine})
\]

Next, we need to formalize the concrete execution, which describes the semantics of a SIL program. To keep the language minimal, the only source of nondeterminism is memory allocation: the newly allocated memory can be located anywhere in memory (as long as it does not overlap with previously allocated memory) and it is initialized to arbitrary values. Using this, we can simulate external input.

While the concrete execution is rather straightforward, it suffers from the fact that it involves demonic choice over \( \mathbb{N} \). The symbolic execution is a finite approximation to the concrete execution, which makes it suitable for implementation. This finiteness is achieved in three steps: routine contracts, heap abstraction and symbols.

**Routine Contracts** First, we deal with nontermination. Using concrete execution, our verification algorithm consists of running the program until it finishes. However, programs might not terminate at all. In our SIL, the only way to encounter nontermination is through infinite recursion. We can thus solve this issue by using routine contracts, i.e. preconditions and postconditions. To express these, we make use of an assertion language.

**Heap abstraction** Consider a destroy routine for singly-linked lists, which frees every node in the list. This routine needs to be able to work on lists of any length. However, we cannot express this in our current assertion language. We solve this problem by introducing custom chunks. With the addition of custom chunks, it can make sense to have multiple instances of the same chunk on the heap. For this reason, we upgrade the heap from a set to a multiset of chunks.

**Symbols** Lastly, we introduce symbols. It often occurs that we deal with a demonic choice between states which only differ in the values assigned to variables. We can represent all these states by a single state by making use of symbols. A single symbol represents a demonic choice over \( \mathbb{N} \).

Now that we have defined both the concrete and symbolic execution, we can proceed to proving that the symbolic execution can be trusted for verification purposes. We say the symbolic execution is sound if, for all programs, successful symbolic execution implies that the concrete execution also succeeds. For a sketch of the soundness proof, we refer the reader to [146].

4.2 Valorization

The field of software verification has made great strides in the last decade, but even the most advanced technologies typically limit themselves to verifying single algorithms or toy applications. In this section we present a series of four non-trivial case studies in software verification. We employ VeriFast to two Java Smart Card applets, a Linux device driver, and an embedded Linux network management component, which are written in C. Our case studies have been carefully selected so as to evaluate the industrial applicability of VeriFast.
4.2.1 Problem Statement

Software verification is still a very time-consuming process. Existing or new source code must be annotated in order to express assumptions and invariants, and to let the verifier reason about the code. Minimizing these required annotations is an active field of research where a lot of work remains to be done. For current verification technologies the overhead of annotating code is far from negligible, so it is not (yet) economically profitable to try to annotate and verify every piece of code. Large, non-critical code bases are examples where the effort probably is not justifiable.

However, there are a number of areas where software verification potentially does make sense. Smart card applications for example have a number of properties that do make them ideal candidates for software verification. First of all, they are typically small, in the order of a few thousand lines of code. Secondly, they are critical, in the sense that they usually offer some kind of security service. And last but not least, it is extremely difficult to update the code once it has been deployed. If a serious bug is discovered in the code, it might be necessary to recall all the deployed smart cards and issue new ones, which could be a commercial disaster.

Another example are operating system drivers that have to be extremely stable because any bug can crash the entire system. In addition, drivers typically run with elevated privileges, so bugs in driver code may have very significant security implications. Moreover, multiple threads can execute concurrently inside a driver, making it difficult to find bugs through testing and to reliably reproduce them once they are found.

The rest of this section reports on four case studies. We detail our experience with a large open source Java Card applet, and a commercial C implementation of a Policy Enforcement Point for embedded Linux gateways. We also briefly describe two other case studies, one on a Linux device driver and one on an industrial Java Card applet. The commercial Policy Enforcement Point and the industrial Java Card applet originate from industry partners of the SECURECHANGE project. A more detailed analysis of the case studies can be found in [123], [120] and [122].

4.2.2 The Belgian Electronic Identity Card

The Belgian Electronic Identity Card (eID) was introduced in 2003 as a replacement for the existing non-electronic identity card. Its purpose is to enable e-government and e-business scenarios where strong authentication is necessary. The card has the size of a standard credit card and features an embedded chip. In addition to containing a machine readable version of the information printed on the card, the chip also contains the address of the owner and two RSA key pairs with the corresponding X509 certificates. One key pair is used for authentication, whereas the other key pair can be used to generate legally binding electronic signatures.

The card is implemented on top of the Java Card platform (Classic Edition) and implements the smart card commands as defined in the ISO7816 standard. Unfortunately, the actual code that runs on the eID cards is not publicly available. For our case study, we used an official open source testbed version of the eID applet that implements the same functionality as the real eID card. It is aimed at developers who wish to interact with eID cards as an easy to use and customizable testing platform.

The eID implementation consists of one large class called EidCard and a few other small helper classes. The EidCard class inherits from the Applet class and encapsulates about 80% of the entire code base. It is a complex class of about 900 lines of code and no less than 38 fields.

The main goal of this case study was to see how practical it is to use VeriFast to annotate a Java Card applet that is 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.

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 applet. Both modi operandi are supported by the tool. For this Java Card 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.

The eID applet and helper classes consist of 1,004 lines of Java Card code. In order to verify the project, we added 684 lines of VeriFast annotations (or about seven lines of annotations for every ten lines of code). The majority of these annotations were requires/ensures pairs (88 pairs, one for each method). Remarkably, only 8 predicates

---

1 Background information on SECURECHANGE is available online at http://www.securechange.eu/.
2 The source code of the eID applet can be downloaded from http://code.google.com/p/eid-quick-key-toolset/. An annotated version of this source code is included in the examples section of the VeriFast distribution.
are defined throughout the entire code base, reflecting the design decision of the authors of the applet to write most of it as one huge class file.

During the past months, a lot of progress has been made to reduce the annotation overhead by automatically inferring open and close statements. This progress can be clearly seen when we compare the first annotated version of the eID applet with the latest version. In the first version, presented in [123], the verification needed 99 open and 112 close statements. With the latest version of VeriFast, the number of required statements has been reduced to 26 and 17 respectively.

Another type of annotation overhead is the time it took to actually write the annotations. The verification of the eID 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. This includes time spent learning the VeriFast tool and the Java Card API specifications.

Bugs and Other Problems in the Applet

Because the eID applet in our case study is aimed at developers, the authors did not spend a lot of time worrying about card tearing. This is demonstrated by the fact that they did not use the Java Card transaction system at all. Using VeriFast, we found 25 locations where a card tear could cause the persistent memory to enter an inconsistent state.

Three locations were found where a null pointer dereference could occur. An additional three class casting problems were found, where a variable holding a reference to the selected file (of type File) was cast to an ElementaryFile instance. These bugs could be triggered by sending messages with invalid file identifiers to the smart card. Seven potential out of bounds operations were also found in the code. These bugs could be triggered by sending illegal messages to the smart card.

4.2.3 Embedded Linux Network Management Software

The second case study presented in this deliverable is on applying VeriFast to an implementation of a Policy Enforcement Point (PEP) for Network Admission Control scenarios. The case study originates from an industry partner of the SECURECHANGE project. Due to a non-disclosure agreement with that partner, the case study, in particular the source code of PEP, cannot be revealed in full.

The PEP program consists of a total of 1612 lines of C code, including comments. It is designed to run on embedded Linux-based gateways and facilitates the application of security policies in Network Admission Control scenarios. More specifically, for an authenticated network device, PEP will receive an access policy from a Policy Decision Point. This policy is then put in place by configuring the gateway’s network interfaces and firewall rules accordingly.

The PEP implementation is split into 9 C source files and 8 C header files. In total, 53 functions are implemented. The core module of PEP is the file pep.c, which comprises 658 lines of code in 13 functions. Although PEP itself is relatively small, it involves a range of Linux libraries – namely libpcap, libdumbnet, libssl, and the POSIX threads API – that increase the complexity of the verification effort substantially.

For a thorough verification that proves the absence of runtime errors and functional correctness, the entire PEP code, including the Linux system libraries would have to be annotated. While, having such a verified software stack would certainly be a great contribution, doing so is beyond our scope. Thus, we restrict this case study to annotating a subset of the above libraries’ APIs and the PEP program’s core component, pep.c.

An important goal of the exercise was to conduct the verification with as few modifications to the sources as possible. This was important for communicating bugs reported by our verification team to the developers of the PEP program.

With respect to verification properties, we aimed at proving that the PEP implementation does not perform illegal operations such as dividing by zero or illegal memory accesses. PEP also exploits concurrency, using the POSIX threading API. Thus, a second objective was to verify that the PEP implementation is free of data races. Considering that PEP implements essential security functionality on a gateway by enforcing policies that protect network resources from unauthorized access, violations of the above safety properties do have security implications. Examples for such cases are exploitable buffer overflows (i.e., illegal memory access) or thread synchronization errors that may render the PEP program unresponsive. The consequences of such bugs may be severe: the gateway could be incapable of updating security policies or, even worse, set up policies forged by an attacker.

The key result of the verification exercise reported on in this section certainly is that we have proven a relatively
complex example for an embedded network management software correct with respect to a number of safety properties, which substantially increases our confidence in the overall safety and security of this software component.

Since the PEP program is production code that implements a security module which has potentially been deployed with thousands of home gateways, our expectations to find critical bugs were relatively low.

**Errors Found and Error Severity** Surprisingly, we discovered a total of 41 bugs in code. In more detail, our verification effort revealed 16 program locations at which a null-pointer may be de-referenced, 6 memory leaks, 6 out-of-bounds access on buffers, 6 global variables that are involved in race conditions, 4 cases in which errors in input/output operations are not handled and 3 unconfirmed functional errors.

The high number of NULL-pointer errors is largely due to the fact that, throughout the code, the return value of calls to malloc() is typically not checked to be unequal to NULL. Arguably, the programmer’s assumption is that the PEP program allocates so few memory that malloc() will never fail. Yet, the program allocates new buffers regularly throughout its execution, which, in combination with the memory leaks we discovered, will inevitably lead to memory exhaustion. Especially since the program is intended to run on a low-cost embedded device with rather limited resources, we consider the NULL-pointer errors and memory leaks as a serious problem with respect to the reliability of the PEP program.

The next group of severe bugs are those we report as buffer overflows. Out of the six buffer overflows identified during verification, five are related to reading a malformed configuration file. From a security perspective, this could be abused if an attacker gains access to the configuration file. Evaluating the chances for this is outside of the scope of our work.

More severe, is the 6th out-of-bounds access, actually a buffer overwrite, we detected. It is related to parsing a network package received by the PEP program. That is, the PEP program employs the function pcap_next() provided by libpcap to read network packets from an interface. pcap_next() returns a pointer to a char buffer containing the raw packet data and assigns a struct pcap_pkthdr with status information of the buffer, including the size of the packet buffer. The PEP implementation only checks whether this packet buffer is at least 14 bytes long. Yet, under some conditions up to 18 bytes of the buffer are read. The content of the overread 4 bytes is used to specify the length and assignment of another buffer, which gives rise to an overflow. We did not investigate whether this bug may be directly abused to manipulate PEP’s control flow so as to gain access or to crash PEP.

Furthermore, we discovered a number of race conditions on global variables. PEP spawns a number of threads that listen for particular packets on the several network interfaces the program manages. Yet, all these threads’ operations share some global variables to store data, such as MAC addresses, host names, interface names, and file descriptors, that are assigned by one thread and used by the others. We discovered that six of these globals were not protected by appropriate mechanisms to prevent undefined program behavior due to data races. However, the developers of PEP put some sleep() statements in place to ensure that one thread receives sufficient CPU time to perform a set of critical operations before the next thread is started. Obviously, the developers were aware of synchronization issues but decided, for unknown reasons, against the use of locks or mutexes.

**Effort and Overhead** To prove the PEP program correct, we attempted to fix all the above errors. That is, we introduced a number of NULL-checks, free() statements, and locks into the program and re-verified it to be correct. Importantly, our fixes make the program safe with respect to our API specifications. Yet, it may still not be functionally correct. That is, e.g., adding and initializing a global lock and pthread_spin_lock() and pthread_spin_unlock() directives around each access to the particular data object the lock is intended to protect, prevents that data object from being read from and written to at the same time. Doing so does, however, not enforce the intended order of execution. In a similar manner, failure of malloc() is handled by safely terminating the program, which is most probably not what a security engineer’s advice would be: terminating the PEP program might result in the gateway being unable to grant or revoke privileges on demand. Thus, the decision on how to finally address the programming errors discovered in our verification effort is left to the developers and security engineers of our industry partner.

Initially, pep.c consisted of 658 lines (429 non-empty LOC). The file was modified by removing a number of printf() statements which were only executed when the program runs in a debug mode but not in actual deployment scenarios. We further added 70 LOC to work around issues in VeriFast. Another 91 LOC were added to fix bugs in the code, resulting in pep.c containing 508 non-empty LOC. Finally, verifying pep.c required 801 lines of annotations in that file, and another 215 lines of annotations for internal header files. A relatively high effort of approximately two man-months was spent to verify the program, producing 1.57 lines of annotations per line of
source code for PEP’s main source file. This annotation overhead is roughly twice as high as overheads reported in the previous case study (c.f. Section 4.2.2), which is due to the involvement of thread management and resource locking in PEP. The fully annotated version of pep.c verifies in VeriFast in just under 20s on an 800 MHz AMD Turion machine running Linux. The peak memory consumption of VeriFast is 31 MBytes.

4.2.4 Other Case Studies

Other nontrivial case studies have also been performed with VeriFast. This section describes our experience with two additional case studies: a Linux driver and an industrial Java Card applet.

The Linux USB BP Keyboard Driver  Kernel and driver code are particularly challenging types of software to verify. They contain a lot of low-level code to interact with hardware, and typically also have strict synchronization and security requirements. However, because they are of paramount importance to an operating system and because they are typically self-contained modules, they are excellent targets for software verification.

There is not much published work that shows whether or not verification of real-world kernel code is feasible. To work towards addressing this question, we applied VeriFast on a device driver taken from the Linux kernel. The driver code subject to verification is Linux’s USB Boot Protocol keyboard driver (usbkbd). While being small, this driver contains a bigger than expected subset of kernel driver complexity. It involves asynchronous callbacks, dynamic allocated memory, synchronization and usage of complex APIs. During verification, we identified and fixed a number of bugs. For these bugs we submitted patches that have been accepted by the driver’s maintainer and included in Linux 3.3.

Verification of the driver is against the original API. Wrapper functions are only used in a few cases where API functions return a struct (i.e. not a pointer to a struct) because this is currently not supported by VeriFast. The APIs that usbkbd uses are the USB API, the input API, spinlocks, and some generic functions like memcpy. Verification thus consists of (1) writing formal specifications for these APIs, based on official documentation and reading the API implementation for the underspecified or undocumented parts, and (2) of adding annotations to usbkbd.c. These annotations consist of contracts (pre- and postconditions written in separation logic), predicates to describe data structures, predicate family instances to instantiate callback function contracts, lemmas (i.e. ghost functions), and ghost-code like folding and unfolding predicates.

The verified properties are absence of data races in the presence of concurrent callbacks, absence of illegal memory accesses, and correct API usage. This does not include a formal proof of correctness of the hand-written API formalization.

usbkbd is one of the smallest Linux kernel drivers. It consists of 426 lines of C code (including blanks and comments). VeriFast reports 329 lines of actual code and 822 lines of annotations. The API specifications count up to 769 lines of code. On an Intel L9400 1.86GHz running the verifier takes about one second. Writing annotations, studying the API documentation, studying the API implementation for undocumented parts and studying the driver implementation is estimated to sum up to about 56 working days. More details about this work can be found in Penninckx et al. [120].

An Industrial Java Card Applet  VeriFast was also used to verify the code of a Java Card applet that was developed and supplied by a commercial smart card vendor in the context of the SECURECHANGE project.

The applet consists of 251 lines of Java Card code, which we annotated with 205 lines of VeriFast annotations. There were 13 requires/ensures pairs, 25 open statements and 29 close statements. It was annotated by a VeriFast specialist and took about 5 man-days, excluding the time it took to add some new required features to VeriFast.

We found a number of bugs in the commercial applet, even though it had already been verified with another verification technology previously. We found an unsafe API call, a handful of unchecked assumptions about incoming APDUs, and four locations where transactions were not used properly. Clearly, the tool used earlier was not sound or was not used in a sound way.

3The annotated sources of usbkbd, specifications for the used APIs and the patches submitted to the driver’s maintainer are available at http://people.cs.kuleuven.be/~willem.penninckx/usbkbd/.
4.2.5 Contributions

This section reports on four industrial case studies with the VeriFast program verifier. We present results for an open-source version of a Java Card applet that implements the Belgian electronic ID card, and a commercial C-implementation of a Policy Enforcement Point (PEP) for embedded Linux gateways. We further summarize two case studies, one on a Linux device driver and one on an industrial Java Card applet. All four programs have been verified for correctness with respect to the absence of certain common programming errors. In particular, the verification checked that the applications do not contain transaction errors, synchronization or multi-threading errors, performed no out-of-bounds operations on buffers, and never dereferenced null pointers or leak memory.

In all case studies, a number of bugs were discovered that could have an impact on the stability and security of these systems. Notably, the PEP implementation that had been deployed on numerous home gateways, was found to contain a surprisingly high number of memory safety bugs and race conditions, some of which might have severe implications on the reliability and security of the gateway. Also, the industrial Java Card applet that was already verified with another verification technology was found to still contain a number of bugs, indicating that either the other verification technology was unsound or was used in an unsound way.

The results of the case study are encouraging: the annotation overhead differs from case study to case study, but never proved prohibitively large. The annotation overhead varies between 0.69 and 1.57 lines of annotation per line of code, consuming between one and two man-months to develop. Given the strong guarantees that VeriFast provides in return, and the empirical evidence of the many bugs we discovered in the four case studies presented in this article, we are coming closer to the point where the approach might in some projects be cost-effective. This is especially true in the domain of security critical and safety critical applications, where bugs may have severe consequences. We hope that the details we present on the verification process, our annotations, and the advantages and pitfalls of using VeriFast will be helpful for future industrial or academic verification projects.
5 Information Flow for Secure Services

In the previous deliverable, we discussed research results in two subfields of information flow: Dynamic Information-Flow Analysis and Quantitative Information-Flow Analysis. These two techniques have been further developed and are the basis of the four new contributions that are discussed in this topic. Section 5.1 improves the technique described in the dynamic information-flow section of the previous deliverable in order to be able to securely execute applications through secure multi-execution. The base idea of this technique is implemented as a prototype, described in Section 5.2. The other two sections use quantitative information-flow techniques to prevent cache-attacks (Section 5.3) and to reason about quantitative confidentiality properties (Section 5.4).

5.1 Secure Multi-execution Through Static Program Transformation

Deliverable 8.2 already mentioned that IMDEA and KUL were jointly working on a new technique to implement secure multi-execution through static program transformations. The collaborative effort had just started back then, and was finished during the second year of the NESSoS project. Even though it has been completed and published, the collaboration line between IMDEA and KUL remains open. This section will summarize the results of the work. A full description can be found in [30].

5.1.1 Introduction

Information flow policies are confidentiality and integrity policies that constrain the propagation of data in programs. For instance, such policies can limit how public outputs can depend on confidential inputs, or how high integrity outputs can be influenced by low integrity inputs. A baseline confidentiality 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 possibly different confidential inputs) that lead to different public outputs. This definition of non-interference generalizes from two security levels H and L to an arbitrary partially ordered set of security levels.

Enforcing non-interference and other information flow policies is a challenging problem. Ideally, enforcement mechanisms should achieve potentially conflicting goals, including: i. soundness: no illicit flows should arise during execution; ii. precision: the execution of secure programs should not be prevented or altered; iii. practicality: the cost of the mechanism should be acceptable. Costs can be incurred at development time (for instance additional code annotations), at deployment time (for instance modifications to standard runtime environments) or at run time (for instance performance cost). Despite substantial attention from the research community for several decades, enforcement mechanisms achieving these goals simultaneously have remained elusive.

There are two main classes of enforcement mechanisms for information flow policies. Static mechanisms include security type systems [147, 77, 112], and verification-based approaches [32]. Dynamic techniques, which have received renewed interest in recent years, include run-time monitors [73, 132, 23, 51], and more recently secure multi-execution (SME) [64, 45]. The cited dynamic techniques are sound, and can be more precise than some static techniques.

SME offers perfect precision (at the cost of potentially modifying the behavior of insecure programs); it is also practical for developers, since there is no need for security annotations of the code. However, SME is not easy to deploy, as all existing implementations of SME require modifications to the underlying computing infrastructure (OS [45], browser [38, 24], virtual machine [64], trusted libraries [85]). Specifically, it is hard to deploy SME for distributed and heterogeneous infrastructures, such as the web. The key contribution presented in this section is a new implementation technique for SME based on static program transformation that eliminates the need to modify the computing infrastructure, while retaining its appealing theoretical properties.

5.1.2 Secure Multi-Execution: the operational approach

The central insight of SME is that non-interference can be enforced by executing programs once per security level. In order to guarantee non-interference, the execution at security level $l$ only performs inputs and outputs to channels at level $l$; moreover, inputs from channels with security levels $l'$ such that $l' \leq l$ are replaced by default values and inputs from channels of security levels $l'$ such that $l' < l$ are delayed until the execution corresponding to security level $l'$ reads from them—the result is then available to be reused at security level $l$. 
The precision of SME intuitively follows from the fact that for non-interferent programs, the behavior of the program visible at a level \( l \) is by definition not influenced by changes to information at levels not lower than \( l \). Therefore, the execution at any level \( l \) will still produce the same behavior at level \( l \) as the standard execution of the program, since it receives the same input on all levels lower than \( l \).

5.1.3 Secure Multi-Execution by program transformation

The semantics of the previous section provides a direct, operational interpretation of the effect of secure multi-execution on programs. In this section, we explore an alternative approach in which a program \( P \) of the source language is transformed into a program \( P' \) whose behavior matches the behavior of \( P \) under SME execution. Our results show that one can achieve soundness and precision without modifying the runtime environment.

Informally, one defines for each program \( P \) and security level \( l \) a transformed program \( \text{Tr}(P, l) \) and defines \( \text{Tr}(P) \) as the sequential composition of the commands \( \text{Tr}(P, l) \), where \( l \) ranges over security levels from low to high. This mimics execution under the SME semantics with the select\_lowprio scheduler, as defined in [63]. We assume that this sequential composition is done in the same order as the order in which the select\_lowprio scheduler selects executions. For a totally ordered \( L \), this order is fixed, but non-comparable levels can be scheduled in different ways.

SME requires the buffering of inputs so that these inputs can be reused by executions running at higher security levels. We implement these buffers as global lists (\( \text{list}\_c \)) and the global input pointer as well as local input pointers are represented as global integer variables (\( \text{count}\_c \) and \( \text{count}\_c,l \) respectively).

For commands that do not perform input/output operations, the command \( \text{Tr}(P, l) \) executes \( P \) “locally”. Specifically, for each variable \( x \) of the source program, we introduce variables \( x_l \), where \( l \) ranges over security levels; informally, \( x_l \) is the local copy of \( x \) for the execution corresponding to security level \( l \). Then, we ensure that \( \text{Tr}(P, l) \) reads and writes only from/to variables indexed by \( l \). The definition of the transformation extends recursively to sequences, branching statements, and loops. In the case of dynamic code evaluation, \( \text{Tr}(\text{eval}(e), l) \) should informally compute the value of \( e \) locally at level \( l \), decode the resulting value into a command \( c \), compute \( c' = \text{Tr}(c, l) \), encode \( c' \) into an integer \( n' \), and return \( \text{eval}(n') \).

5.1.4 Implementation

In order to validate our approach, we have developed two prototype implementations. Our first implementation considers a restricted fragment of Python; the fragment essentially corresponds to our exemplary language, with I/O functions input and print added as built-in functions. It does not support any of Python’s more advanced features, but was useful to provide a baseline implementation. Our second implementation supports a fragment of JavaScript including \( \text{eval}() \). Both implementations were tested for security and for precision by means of small test scenarios. More details about these implementations can be found in [31]

5.1.5 Transformation to a concurrent language

The transformation defined earlier simulates SME with the select\_lowprio scheduler. Kashyap et al. [88] have shown that other scheduling strategies can be useful too. In this section, we present a variant of our transformation towards a language that supports concurrency in order to enable the use of more scheduling strategies.

This revised transformation still takes programs in the sequential subset of the language as input. The concurrency features are only used in the output of the transformation.

We extend our command language with the following syntax:

\[
\begin{align*}
c &::= \ldots | \text{await } b \text{ then } c \\
P &::= \| (id, c)^* 
\end{align*}
\]

Intuitively, the command \( \text{await } b \text{ then } c \) executes \( c \) atomically, provided \( b \) holds, and blocks otherwise. Then, a program is simply a set of threads; for convenience, we assume that each thread is tagged with a unique identifier. In what follows, we write \( \text{atomic } c \) as a shorthand for \( \text{await } \text{true} \text{ then } c \).

The operational behavior of programs is modeled as a transition between configurations. A configuration is a 5-tuple consisting of a program \( P \), a waiting queue \( \text{wq} \) mapping guards to commands, an input pointer \( p \), a program input \( I \) and a program output \( O \). The thread-local semantics is similar to our sequential language; note however that we introduce another rule for sequence in order to propagate the emission of signals induced by await commands.
The rules for the latter are standard; if the guard holds, then the body of the command is executed atomically. Otherwise, the command blocks and emits a signal, namely the guard in which its blocked. Upon the emission of a signal, the global semantics then inserts the blocked thread associated with the guard into the waiting queue. Further changes in global state trigger the re-evaluation of guards, and threads associated with guards that become true are moved back to the ready list.

The revised transformation again yields executions equivalent to secure multi-execution, now for any scheduling strategy. The proof relies on a simulation result and hinges on the assumption that (informally) schedulers pick the same threads to execute. For the proof, we refer to the extended version of this work [31].

5.2 FlowFox: a Web Browser with Flexible and Precise Information Flow Control

A web browser handles content from a variety of origins, and not all of these origins are equally trustworthy. Moreover, this content can be a combination of markup and executable scripts where the scripts can interact with their environment through a collection of powerful APIs that offer communication to remote servers, communication with other pages displayed in the browser, and access to user, browser and application information including information such as the geographical location, clipboard content, browser version and application page structure and content. With the advent of the HTML5 standards [79], the collection of APIs available to scripts has substantially expanded. An important consequence is that scripts can be used to attack the confidentiality or integrity of that information. Scripts can leak session identifiers [115], inject requests into an ongoing session [28], sniff the user’s browsing history, or track the user’s behavior on a web site [84]. Such malicious scripts can enter a web page because of a cross-site scripting vulnerability [86], or because the page integrates third party scripts such as advertisements, or gadgets. Barth et al. [29, 20] have proposed the gadget attacker, as an appropriate attacker model for this broad class of attacks against the browser.

The importance of these attacks has led to many countermeasures being implemented in browsers. The first line of defense is the same-origin-policy (SOP) that imposes restrictions on the way in which scripts and data from different origins can interact. However, the SOP is known to have holes [137], and all of the attacks cited above bypass the SOP. Hence, additional countermeasures have been implemented or proposed. Some of these are ad-hoc security checks added to the browser (e.g. to defend against history-sniffing attacks, browsers responded with prohibiting access to the computed style of HTML elements [150]), others are elaborate and well thought-out research proposals to address specific subclasses of such attacks (for example AdJail [141] proposes an architecture to contain advertisement scripts).

Several researchers [41, 103] have proposed information flow control as a general and powerful security enforcement mechanism that can address many of these attacks, and hence reduce the need for ad-hoc or purpose-specific countermeasures. Several prototypes that implement some limited form of information flow control have been developed. However, general, flexible, sound and precise information flow control is difficult to achieve, and so far nobody has been able to demonstrate a fully functional browser that enforces sound and precise information flow control for web scripts. As a consequence, there was no evidence for the practicality of this approach in the context of web applications, till now.

This section introduces FLOWFOX, the first fully functional web browser (implemented as a modified Mozilla Firefox) that implements a precise and general information flow control mechanism based on the technique of secure multi-execution [65]. FLOWFOX can enforce general information flow based confidentiality policies on the interactions between web scripts and the browser API. Information entering or leaving scripts through the API is labeled with a confidentiality label chosen from a partially ordered set of labels, and FLOWFOX enforces that information can only flow upward in a script. A more complete description of this prototype can be found in [61].

5.2.1 FlowFox Design

In this section we describe the design of FLOWFOX. An important design decision when implementing SME for web scripts is how to deal with the browser API exposed to scripts. A first option is to multi-execute the entire browser: the API interactions would become internal interactions and each SME copy of the browser would have its own copy of the DOM. Both Bielova et al. [39] and Capizzi et al. [46] applied this strategy in their implementations.

The alternate strategy is to only multi-execute the web scripts and to treat all interactions with the browser API as inputs and outputs. Both designs have their advantages and disadvantages. When multi-executing the entire browser,
the information flow policy has to label inputs and outputs at the abstraction level provided by the operating system. The policy can talk about I/O to files and network connections, or about windows and mouse events. Multi-execution can be implemented relatively easily by running multiple processes. However, at this level of abstraction, the SME enforcement mechanism lacks the necessary context information to give an appropriate label to e.g. mouse events. The operating system does not know to which tab, or which HTML element in that tab a specific mouse click or key press is directed. It can also not distinguish individual HTML elements that scripts are reading from or writing to.

When multi-executing only the scripts, the information flow policy has to label inputs and outputs at the abstraction level offered by the browser API. The policy can talk about reading from or writing to the text content of specific HTML elements, and can assign appropriate labels to such input and output operations. However, multi-execution is much harder to implement, as it now entails making cross-cutting modifications to the source code of the browser. Also, policies become more complex, as there are much more methods in the browser API than there are system calls.

FLOWFOX takes the second approach, as the first approach is too coarse grained and imprecise to counter relevant threats. The first approach (taken by [46, 39]) can e.g. not protect against a script leaking an e-mail typed by the user into a webmail application to any third party with whom the browser has an active session in another tab, because the security enforcement mechanism cannot determine to which origin the user text input is directed.

5.2.2 Security Policies

In FLOWFOX every DOM call is interpreted as an output message to the DOM (the invocation with the actual parameters), followed by an input call from the DOM (the return value). DOM events delivered to scripts are interpreted as inputs. The policy deals with events by giving appropriate labels to the DOM API calls that register handlers.

Hence a FLOWFOX policy must specify two things. First, it assigns security levels to DOM API calls. Second, a default return value must be specified for each DOM API call that could potentially be skipped by the SME enforcement mechanism.

A policy rule has the form $R[D] : C_1 \rightarrow l_1, \ldots, C_n \rightarrow l_n \rightarrow dv$ where $R$ is a rule name, $D$ is a DOM API method name, the $C_i$ are boolean expressions, the $l_i$ are security levels and $dv$ is a JavaScript value.

Policy rules are evaluated in the context of a specific invocation of the DOM API method $D$, and the boolean expressions $C_i$ are JavaScript expressions and can access the receiver object ($arg_0$) and arguments ($arg_i$) of that invocation. Given such an invocation, a policy rule associates a level and a default value with the invocation as follows. The default value is just the value $dv$. The conditions $C_i$ are evaluated from left to right. If $C_j$ is the first one that evaluates to true, the level associated with the invocation is $l_j$. If none of them evaluate to true, the level associated with the invocation is $L$.

Policies are specified as a sequence of policy rules, and associate a level and default value with any given DOM API invocation as follows. For an invocation of DOM API method $D$, if there is a policy rule for $D$, that rule is used to determine level and default value. If there is no rule in the policy for $D$, that call is considered to have level $L$, with default value undefined. The default value for invocations classified at $L$ is irrelevant, as the SME rules will never require a default value for such invocations.

Making API calls low by default, supports the writing of short and simple policies. The empty policy (everything low) corresponds to standard browser behavior. By selectively making some API calls high, we can protect the information returned by these calls. It can only flow to calls that also have been made high.

JavaScript properties that are part of the DOM API can be considered to consist of a getter method and a setter method. For simplicity, we provide some syntactic sugar for setting policies on properties: for a property $P$ (e.g. `document.cookie`), a single policy rule specifies a level $l$ and default value $dv$. The setter method then gets the level $l$ and default value $dv$ and the setter method gets the level $l$ and the default value `true` – for a setter, the return value is a boolean indicating whether the setter completed successfully.

5.2.3 Evaluation

FLOWFOX is implemented on top of Mozilla Firefox 8.0.1 and consists of about $\pm 1400$ new lines of C/C++ code. We evaluated our FLOWFOX prototype in three major areas: compatibility with major websites, security guarantees offered, and performance and memory overhead.

---

1For API methods that return void, this can be optimized; they can be considered just outputs, but we ignore that optimization in the discussion below.
Compatibility Since SME is precise [65], theory predicts that FLOWFOX should not modify the behavior of the browser for sites that comply with the policy. Moreover, SME can sometimes fix interferent executions by providing appropriate default values to the low execution. These two hypotheses are confirmed by two experiments in [61].

Policy Support FLOWFOX supports a wide variety of useful policies. We consider three classes of policies to be interesting for further investigation:

1. Policies that classify the entire DOM API low, except for some selected calls that return sensitive information. The three examples above fall in this category. Such policies could be offered by the browser vendor as a kind of privacy profile.

2. Policies that approximate the SOP, but close some of its leaks. Writing such a policy is an extensive task, as each DOM API method must receive an appropriate policy rule that ensures that information belonging to the document origin is high and other information is low. However, such a policy must be written only once, and should only evolve as the DOM API evolves.

3. Server-driven policies, where a site can configure FLOWFOX to better protect the information returned from that site.

Note that none of these cases requires the end-user to write policies. Policy writing is obviously too complex for browser end-users.

Performance and Memory Cost The FLOWFOX prototype was tested on a MacBook notebook with a 2GHz Intel®Core™2 Duo processor and 2GB RAM. Micro benchmarks modifications have the largest impact – even when not multi-executing – for applications that extensively exploit data structures. The results also confirm our expectations that our prototype implementation more or less doubles execution time when actively multi-executing with two security levels. Since web scripts can be I/O intensive, it is important to note that I/O intensive code has only a small performance overhead. Macro benchmarks show that FLOWFOX induces an overhead of around 20% in real-life web applications. Finally, a memory benchmark showed that FLOWFOX averaged around 88% memory overhead.

5.3 Automatic Quantification of Cache Side-Channels

Many modern computer architectures use caches to bridge the latency gap between the CPU and main memory. Caches are small, fast memories that store the contents of previously accessed main memory locations; they can improve the overall performance because typical memory access patterns exhibit locality of reference. On today’s architectures, an access to the main memory (i.e. a cache miss) may imply an overhead of hundreds of CPU cycles w.r.t. an access to the cache (cache hit).

While the use of caches is beneficial for performance reasons, it can have negative effects on security: An observer who can measure the time of memory lookups can see whether a lookup is a cache hit or miss, thereby learning partial information about the state of the cache. This partial information has been used for extracting cryptographic keys from implementations of AES [37, 116, 74], RSA [121], and DSA [12]. In particular AES is vulnerable to such cache attacks, because most high-speed software implementations make heavy use of look-up tables. Cache attacks are the most effective known attacks against AES and allow to recover keys within minutes [74].

A number of countermeasures have been proposed against cache attacks. They can be roughly put in two classes: (1) Avoiding the use of caches for sensitive computations. This can be achieved, e.g. by using dedicated hardware implementations (For example, recent Intel processors offer support for AES [4]), or by side-stepping the use of caches in software implementations [90]. Both solutions obviously defeat cache attacks; however they are not universally applicable (i.e. to arbitrary programs), e.g. due to lack of available hardware support, or for reasons of performance. (2) Mitigation strategies for eliminating attack vectors and reducing leakage. Proposals include disabling high-resolution timers, hardening of schedulers [74], and preloading [37, 116] of tables. Such strategies are implemented, e.g. in the OpenSSL 1.0 [8] version of AES, however, their effectiveness is highly dependent on the operating system and the CPU. Without considering/modeling all implementation details, such mitigation strategies necessarily remain heuristic. In summary, there is no general-purpose countermeasure against cache attacks that is backed-up by mathematical proof.
In this section, we propose a novel method for establishing formal security guarantees against cache-attacks that is applicable to arbitrary programs and a wide range of embedded platforms. The guarantees we obtain are upper bounds on the amount of information about the input that an adversary can extract by observing which memory locations are present in the CPU’s cache after execution of the program; they are based on the actual program binary and a concrete processor model and can be derived entirely automatically. At the heart of our approach is a novel technique for effective counting of concretizations of abstract states that enables us to connect state-of-the-art techniques for static cache analysis and quantitative information-flow analysis. The full details can be found in [97].

Technically, we build on prior work on static cache analysis [69] that was primarily used for the estimation of worst-case execution time by abstract interpretation [58]. There, two abstract domains for cache-states are introduced; one of them captures a superset of the memory locations that *may* be in the cache, the other captures a subset of the memory locations that *must* be in the cache. For the purpose of this paper it suffices to know that both abstract analyses are sound, i.e. that each of them computes a superset of the set of reachable cache-states. We also leverage techniques from quantitative-information-flow analysis that enable establishing bounds for the amount of information that a program leaks about its input. One key observation is that (an upper bound on) the number of reachable states of a program corresponds to (an upper bound on) the number of leaked bits [138, 98]. Such upper bounds can be obtained by computing super-sets of the set of reachable states by abstract interpretation, and by determining their sizes [98].

We develop a novel technique for counting the number of cache states represented by the abstract states of the static cache analyses described above. In particular, we give formulas and algorithms for computing the respective sizes of the set of states represented by the may- and must-analysis, and for their intersection. We give a concise implementation of our counting procedures in Haskell [7] and we connect this counting engine to AbsInt’s $\alpha^3$ [1], the state-of-the-art tool for static cache analysis. $\alpha^3$ efficiently analyzes binary code based on accurate models of several modern embedded processors with a wide range of cache types (e.g. data caches, instruction caches, or mixed) and replacement strategies. Using this tool-chain, we perform an analysis of a binary implementation of 128-bit AES from the PolarSSL library [5], based on a 32-bit ARM processor with a 4-way set associative data cache with LRU replacement strategy. We analyze this implementation with and without the preloading countermeasure applied, with different cache sizes, and for two different adversary models, obtaining the following results.

Without preloading, the derived upper bounds for the leakage (about the payload and the key) in one execution exceed the size of the key and are hence too large for practical use. With preloading and a powerful adversary model, however, the derived bounds drop to values ranging from 55 to 1 bits, for cache sizes ranging from 16KB to 128KB. With a less powerful but realistic adversary model, the bounds drop even further to ranges from 6 to 0 bits, yielding strong security guarantees. This case study shows that the automated, formal security analysis of realistic cryptosystems and accurate real processor models is in fact feasible.

In summary, our contributions are threefold. Conceptually, we show how state-of-the-art tools for quantitative information-flow analysis and static cache analysis can be combined for quantifying cache side-channels. Technically, we develop and implement novel methods for counting abstract cache states. Practically, we perform a formal cache-analysis of a binary AES 128 implementation on a realistic processor model.

### 5.3.1 Implementation

In this deliverable we report only on the implementation of a tool for quantifying cache leaks. See the full paper for the theoretical underpinnings [97]. The building blocks for our tool are the AbsInt $\alpha^3$ tool for static cache analysis, and a novel counting engine for cache-states.

**Abstract Interpreter**

The AbsInt $\alpha^3$ [1] is a suite of industrial-strength tools for the static analysis of embedded systems. In particular, $\alpha^3$ comprises tools (called aiT and TimingExplorer) for the estimation of worst-case execution times based on the static cache analysis by Ferdinand et al. [69]. The tools cover a wide range of CPUs, such as ERC32, TriCore, M68020, LEON3 and several PowerPC models (aiT), as well as CPU models with freely configurable LRU cache (TimingExplorer). We base our implementation on the TimingExplorer for reasons of flexibility.

The TimingExplorer receives as input a program binary and a cache configuration and delivers as output a control flow graph in which each (assembly-level) instruction is annotated by the corresponding abstract *may* and *must* information, where memory locations are represented by strings, abstract cache lines are lists of memory locations,
abstract sets are lists of abstract lines, and abstract caches are lists of abstract sets. We extract the annotations of the final state of the program, and provide them as input to the counting engine.

**Counting Engine**

We implemented an engine for counting the concretizations of abstract cache states according to the development in the full paper. Our language of choice is Haskell [7], because it allows for a concise representation of sums, products, and enumerations using list comprehensions. For brevity of presentation, we give only the procedures for exact counting.

We use the following data types for representing abstract cache sets, which matches the output of the TimingExplorer described above.

```haskell
type Loc = String
type ConcreteSet = [Loc]
type AbstractLine = [Loc]
type AbstractSet = [AbstractLine]
```

The function `allStates` is the core of the exact counting of concrete cache states in the intersection defined by may and must.

```haskell
allStates :: AbstractSet -> AbstractSet -> [ConcreteSet]
allStates may must = filter (checkMust must) (genAllMay may)
```

As described in the full paper, this is achieved by enumerating all concrete states that satisfy a given set of may-constraints (done by `genAllMay`), and keeping only those that also satisfy the must-constraints (done by filtering with `checkMust`). At the core of the function `genAllMay` is the following function `genMay` that returns all concretizations of the same length as the given abstract set,

```haskell
genMay:: AbstractSet -> [ConcreteSet]
genMay (a:as) = [c:cs| c<-a, cs<-genMay (carry (delete c a) as)]
genMay [] = [[]]
```

where it relies on a function `carry` that carries unused may-constraints to the next line of the abstract state.

Finally, the function `checkMust` tests whether a concrete set satisfies the must-constraints, by checking whether all elements in line number `n` (denoted by `as!!(n-1)`) of the abstract state also appear in the prefix of length `n` of the concrete state.

```haskell
checkMust :: AbstractSet -> ConcreteSet -> Bool
checkMust as cs = and [elem a (take n cs)| n<-[1..length as], a<-as!!(n-1)]
```

### 5.3.2 Case Study

We report on a case-study where we use the methods developed in this paper for analyzing the cache side-channel of a widely used AES implementation on a realistic processor model with different cache configurations.

**Target Implementations**

**Code.** We analyze the implementation of 128 bit AES encryption from the PolarSSL library [5], a lightweight crypto suite for embedded platforms. As is standard for software implementations of AES, the code consists of single loop (corresponding to the rounds of AES) in which heavy table lookups are performed to indices computed using bit-shifting and masking. We also analyze a modified version of this implementation, where we add a loop that loads the entire lookup table into the cache before encryption. This preloading has been suggested as countermeasure against cache attacks because, intuitively, all lookups during encryption will hit the cache.

**Platform.** We compile the AES C source code into a binary for the ARM7TDMI [2] CPU using the GNU ARM GCC compiler [3]. Although the original ARM7TDMI does not have any caches, the AbsInt TimingExplorer supports this CPU with the possibility of specifying arbitrary configurations of data/instruction/mixed caches with LRU strategy. For our experiments we use data caches with sizes of 16-128 KB, associativity of 4 ways, and a line size of 32 Bytes, which are common configurations in practice.
Improving Precision by Partitioning

The TimingExplorer can be very precise for simple expressions, but loses precision when analyzing array lookups to non-constant indexes. This source of imprecision is well-known in static analysis, and abstract interpretation offers techniques to regain precision, such as abstract domains specialized for arrays [59], or automatic refinement of transfer functions. For our analysis, we use results on trace partitioning [105], which consists in performing the analysis on a partition of all possible runs of a program, each partition yielding more precise results.

We have implemented a simple trace partitioning strategy using program transformations that do not modify the data cache (which is crucial for the soundness of our approach). For each access to the look-up table, we introduce conditionals on the index, where each branch corresponds to one memory block, and we perform the table access in all branches. As the conditionals cover all possible index values for the table access, we add one memory access to the index before the actual table look-up, which does not change the cache state for an LRU cache strategy, since the indices have to be fetched before accessing the table anyway.

Note that the same increase in precision could be achieved without program transformation if the trace partitioning were implemented at the level of the abstract interpreter, which would also allow us to consider instruction caches and cache strategies beyond LRU. Given that the TimingExplorer is closed-source, we opted for partitioning by code transformation.

Results and Security Interpretation

The results of our analysis with respect to the adversary $Adv_{prec}$ are depicted in Figure 5.1. For AES without preloading of tables, the bounds we obtained exceed 160 bits for all cache sizes. For secret keys of only 128 bits, they are not precise enough for implying meaningful security guarantees. With preloading, however, those bounds drop down to 55 bits for caches sizes of 16KB and to only 1 bit for sizes of 128KB, showing that only a small (in the 128KB case) fraction of the key bits can leak in one execution.

The results of our analysis with respect to the (less powerful, but more realistic) adversary $Adv_{prob}$ are depicted in Figure 5.2. As for $Adv_{prec}$, the bounds obtained without preloading exceed the size of the secret key. With preloading, however, they remain below 6 bits and even drop to 0 bits for caches of 128KB, giving a formal proof of noninterference for this implementation and platform.

To formally argue tightness of the non-zero bounds, we would need to show that this information can be effectively recovered (i.e. devise an attack), which is out of the scope of this paper. Manual inspection of the final cache states shows that the non-zero bounds stem from AES tables sharing the same set with other memory locations used by the AES code, which may indeed be exploitable.
5.3.3 Prior Art

Timing attacks against cryptosystems date back to [92]. They can be divided into those exploiting timing variations due to control-flow [92, 44] and those exploiting timing variations of the execution platform, e.g. due to caches [118, 37, 121, 116, 13, 15], or branch prediction units [14]. In this paper we focus solely on caching.

The literature on cache attacks is stratified according to a variety of different adversary models: In time-driven attacks [37, 15] the adversary can observe the overall execution time of the victim process and estimate the overall number of cache hits and misses. In trace-driven attacks [13] the adversary can observe whether a cache hit or miss occurs, for every single memory access of the victim process. In access-driven attacks [121, 116] the adversary can probe the cache either during computation (asynchronous attacks) or after completion (synchronous attacks) of the victim’s computation, giving him partial information about the memory locations accessed by the victim. Finally, some attacks assume that the adversary can choose the cache state before execution of the victim process [116], whereas others only require that the cache does not contain the locations that are looked-up by the victim during execution [15]. The information-theoretic bounds we derive hold for single executions of synchronous access-driven adversaries, where we consider initial states that do not contain the victim’s data. The derivation of bounds for alternative adversary models is left future work.

A number of mitigation techniques have been proposed to counter cache attacks. Examples include coding guidelines [54] for thwarting cache attacks on x86 CPUs, or novel cache-architectures that are more resistant to cache attacks [148]. One commonly proposed mitigation technique is preloading of tables [37, 116]. However, as first observed by [37], it is a non-trivial issue to establish the efficacy of this countermeasure. As [116] comments: “[...] it should be ensured that the table elements are not evicted by the encryption itself, by accesses to the stack, inputs or outputs. Ensuring this is a delicate architecture-dependent affair [...]”.

The methods developed in this paper enable us to automatically and formally deal with these delicate affairs based on an accurate model of the CPU.

For the case of AES, there are efficient software implementations that avoid the use of data caches by bit-slicing, and achieve competitive performance by relying on SIMD (Single Instruction, Multiple Data) support [90]. Furthermore, a model for statistical estimation of the effectiveness of AES cache attacks based on sizes of cache lines and lookup tables has been presented in [144]. For programs beyond AES that are not efficiently implementable using bit-slicing, our analysis technique allows to derive formal assertions about their leakage, based on the actual program semantics and accurate models of the CPU.

Technically, our work builds on methods from quantiative information-flow analysis (QIF) [52], where the automation by reduction to counting problems appears in [25, 113, 78, 108], and the connection to abstract interpretation in [98]. Prior applications of QIF to side-channels in cryptosystems [95, 96, 99] are limited to stateless systems. For the analysis of caches, we rely on the abstract domains from [69] and their implementation in the Ab-
Finally, our work goes beyond language-based approaches that consider caching \cite{19, 76} in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.

Finally, leakage-resilient cryptography (see e.g. \cite{109, 68}) aims at providing cryptographic primitives that remain secure even if their implementation leaks information. As future work, we plan to investigate whether the bounds derived by our technique can be used for justifying the leakage models underlying those approaches.

5.4 Probabilistic Relational Reasoning for Differential Privacy

5.4.1 Motivation

When dealing with collections of private data one is faced with conflicting requirements: on the one hand, it is fundamental to protect the privacy of the individual contributors; on the other hand, it is desirable to maximize the utility of the data by mining and releasing partial or aggregate information, e.g. for medical statistics, market research, or targeted advertising. Differential privacy \cite{67} is a quantitative notion of privacy that achieves an attractive trade-off between these two conflicting requirements: it provides strong confidentiality guarantees, yet it is permissive enough to allow for useful computations on private data. The key advantages of differential privacy over alternative definitions of privacy are its good behavior under composition and its weak assumptions about the prior knowledge of adversaries. For a discussion of the guarantees provided by differential privacy and their limitations, see \cite{89, 91}.

As the theoretical foundations of differential privacy become well-understood, there is momentum to prove privacy guarantees for real systems. Several authors have recently proposed methods for reasoning about differential privacy on the basis of different languages and models of computation, e.g. SQL-like languages \cite{107}, higher-order functional languages \cite{125}, imperative languages \cite{50}, the MapReduce framework \cite{130}, and I/O automata \cite{145}. The unifying basis of these approaches are two key results: The first is the observation that one can achieve privacy by perturbing the output of a deterministic program by a suitable amount of exponentially distributed noise, giving rise to the so-called Laplacian \cite{67} and Exponential Mechanisms \cite{106}. The second result are theorems that establish privacy bounds for the sequential and parallel composition of differentially private programs, see e.g. \cite{107}. In combination, both results form the basis for creating and analyzing programs by composing differentially private building blocks.

While approaches relying on composing these building blocks apply to an interesting range of examples, they fall short of covering the expanding frontiers of differentially private mechanisms and algorithms. Examples that cannot be handled by previous approaches include mechanisms that aim for weaker guarantees, such as approximate differential privacy \cite{66}, or randomized algorithms that achieve differential privacy without using any standard mechanism \cite{75}. Dealing with such examples requires fine-grained reasoning about the complex mathematical and probabilistic computations that programs perform on private input data. Such reasoning is particularly intricate and error-prone, and calls for principled approaches and tool support.

5.4.2 A Logic for Reasoning about Privacy

In this section we present a novel framework for formal reasoning about an expressive class of quantitative confidentiality properties, including (approximate) differential privacy and probabilistic non-interference. Our framework, coined CertiPriv, is built on top of the Coq proof assistant \cite{143} and goes beyond the state-of-the-art in the following three aspects:

Expressivity: CertiPriv enables reasoning about a general and parametrized notion of confidentiality that encompasses differential privacy, approximate differential privacy, and probabilistic noninterference.

Flexibility: CertiPriv enables reasoning about the outcome of probabilistic computations from first principles. That is, instead of being limited to a fixed set of pre-defined building blocks one can define and use arbitrary building blocks, or reason about arbitrary computations using sophisticated machinery, without any limitation other than being elaborated from first principles. Proofs in CertiPriv can be verified independently and automatically by the Coq type checker.

Extensibility: CertiPriv inherits the generality of the Coq proof assistant and allows modeling and reasoning using arbitrary domains and datatypes. That is, instead of being confined to a fixed set of datatypes, CertiPriv can be extended on demand (e.g. with types and operators for graphs).
We illustrate the scope of CertiPriv by giving machine-checked proofs of four representative examples, some of which fall out of the scope of previous language-based approaches: (i) we prove the correctness of the Laplacian and Exponential mechanisms within our framework (rather than assuming their correctness as a meta-theorem), (ii) we prove the privacy of a randomized approximation algorithm for the Minimum Vertex Cover problem [75], (iii) we prove the privacy of a randomized approximation algorithm for the k-medium problem [75], and (iv) we prove the privacy of randomized algorithms for continual release of aggregate statistics of data streams [49]. Taken together, these examples demonstrate the generality and versatility of our approach. The full details of this work can be found in [34].

As the first step in our technical development, we recast and generalize the definition of differential privacy. Informally, a probabilistic computation satisfies differential privacy if, independent of each individual’s contribution to the dataset, the output distribution is essentially the same. More formally, a probabilistic program $c$ is $(\epsilon, \delta)$-differentially private if and only if, given two initial memories $m$ and $m'$ that are adjacent (typically for a notion of adjacency that captures that $m$ and $m'$ differ in the contribution of one individual), the output distributions generated by $c$ are related up to a multiplicative factor $\exp(\epsilon)$ and an additive term $\delta$. That is, for every event $E$ one requires

$$\Pr[c(m) : E] \leq \exp(\epsilon) \Pr[c(m') : E] + \delta$$

where $\Pr[c(m) : E]$ denotes the probability of event $E$ in the distribution obtained by running $c$ on initial memory $m$. The case of $\delta = 0$ corresponds to the vanilla definition of differential privacy [67], whereas cases with $\delta > 0$ correspond to approximate differential privacy [66]. For our development, we generalize $(\epsilon, \delta)$-differential privacy in two ways: First, we define $(\epsilon, \delta)$-differential privacy with respect to arbitrary relations $\Psi$ on initial memories. The original definition is recovered by specializing $\Psi$ to capture adjacency of memories. Second, we introduce a notion of distance (called $\alpha$-distance) that generalizes statistical distance with a skew parameter $\alpha$, and we show that a computation $c$ is $(\epsilon, \delta)$-differentially private if and only if $\delta$ is an upper bound for the $\exp(\epsilon)$-distance between the output distributions obtained by running $c$ on two memories $m$ and $m'$ satisfying $\Psi$.

This generalization of differential privacy has the following two natural readings:

- The first reading is as an information flow property: if $\Psi$ is an equivalence relation and $\epsilon = \delta = 0$, the definition states that the output distributions obtained by executing $c$ in two related memories $m$ and $m'$ coincide, entailing that an adversary who can only observe the final distributions cannot distinguish between the two executions. Or, equivalently, by observing the output distributions, the adversary can only learn the initial memory up to its $\Psi$-equivalence class.

- The second reading is as a continuity property: if $\Psi$ models adjacency between initial memories, the definition states that $c$ is a continuous mapping between metric spaces, where $\alpha$ distance is used as a metric on the set of output distributions.

We leverage on both readings to provide a fresh foundation for reasoning about differentially private computations. For this, we build on the observation that differential privacy can be construed as a quantitative 2-property [142, 53]. Using this observation we define an approximate probabilistic Relational Hoare Logic (apRHL), following Benton’s seminal use of relational logics to reason about information flow [36]. Judgments in apRHL have the form

$$c_1 \sim_{\alpha, \delta} c_2 : \Psi \Rightarrow \Phi$$

and capture that $\delta$ is an upper bound for the $\alpha$-distance of the probability distributions generated by two probabilistic programs $c_1$ and $c_2$, modulo relational pre- and post-conditions $\Psi$ and $\Phi$ on program states. For the special case where $\Phi$ is the equality on states, $c_1 = c_2 = c$, and $\alpha = \exp(\epsilon)$, the above judgment entails that the output distributions obtained by executing $c$ starting from two initial memories related by $\Psi$ are at $\alpha$-distance at most $\delta$, and hence that $c$ is $(\epsilon, \delta)$-differentially private with respect to $\Psi$. This intuitive understanding of apRHL judgments extends to the important case where $\Phi$ is an equivalence relation: With the view that $\Phi$ captures the observational capabilities of an adversary, such judgments simultaneously generalize differential privacy and notions of confidentiality as encountered in information flow analysis.

We present a calculus for reasoning about the validity of apRHL judgments, including fundamental rules corresponding to (sequential and parallel) composition and bounded loops, as well as rules corresponding to the Laplacian and Exponential mechanisms. The proof of soundness of our calculus relies on the novel notion of $(\alpha, \delta)$-lifting of relations on states to relations on distributions, which crisply generalizes existing notions of lifting from probabilistic
process algebra [134, 63] and enjoys good closure properties. We exhibit a connection of \((\alpha, \delta)\)-liftings to maximal network flows, giving a means for the automatic computation of liftings of finite relations.

As bonus material, we present a variant of apRHL that supports reasoning about an asymmetric version of \(\alpha\)-distance. Asymmetric apRHL strictly generalizes apRHL, as any proof in apRHL can be replaced by two (symmetric) proofs in asymmetric apRHL. However, reasoning in the asymmetric logic can lead to increased precision (i.e. better bounds), as we demonstrate for the example of the Minimum Vertex Cover problem [75].

The basis of our formalization is CertiCrypt [33], a machine-checked framework to verify cryptographic proofs in the Coq proof assistant. The most outstanding difference between the two frameworks is that CertiPriv supports reasoning about a wide range of quantitative relational properties, whereas CertiCrypt is confined to baseline information flow properties. We refer to Section 5.3.3 for a more detailed comparison.

Our contributions are twofold. On the theoretical side, we lay the foundations for reasoning formally about an important and general class of approximate relational properties of probabilistic programs. Specifically, we introduce the notions of \(\alpha\)-distance and \((\alpha, \delta)\)-lifting, and an approximate probabilistic relational Hoare logic. On the practical side, we demonstrate the applicability of our approach by providing the first machine-checked proofs of differential privacy properties of fundamental mechanisms and complex approximation algorithms from the recent literature.
6 Interactions

6.1 Current Interaction with Other WPs

Work Package 8 has links to a number of other Work Packages. This section gives a short overview of these links.

Work of this Work Package has been integrated into the SDE of Work Package 2. For example, the VeriFast tool (see Chapter 4) is integrated into the SDE and can be used from there.

Work package 7 deals with secure architectures and middleware. This can be linked to the work described in Section 3.2 about secure navigation paths and the work by INRIA with reflexive middleware for model-driven access control.

As also mentioned in the deliverable of Work Package 9, task 9.2.1 of that work package is completely covered in this deliverable in Section 4. This task deals with secure programming by using formal verification techniques. In addition to this, the two quantitative information-flow topics presented in Chapter 5 are clearly related to quantitative security (task 9.4).

Finally, there are a number of links between WP8 and WP11. The case studies presented in work package 11 are used as test cases for work presented in this deliverable (e.g. work in secure service composition, where eHealth is a major scenario) and other work that was supported by NESSoS but not reported on in this deliverable.

6.2 New Initiatives

One of the shared areas of focus of KUL and UNITN is policy-based access control for Future Internet applications, which poses challenges at both the architectural level of security middleware and the policy level. This area of focus will serve as a basis for future cooperation. One interesting track for joint work on the middleware level is the relation between performance and security. This topic was first explored in past joint work by Gheorghe [71] which investigated attribute cacheability and dynamic reconfiguration of distributed authorizations system with changing security and performance requirements, which will serve as the basis for future exploration of this topic. This track can also be extended to performance techniques on the policy level such as policy optimization or distribution.

Thanks to NESSoS enabled interactions, KUL and INRIA have started to explore exploitation avenues for their complementary expertise. As a first step, KUL will use INRIA’s TMDL approach to model trust models in the health-case use case, which will then be used to explore trust model interoperability issues. They will also collaborate on modeling and distributed evaluation of access-control on federated social networks, where users’ social data need not be hosted on the same online social network provider. If successful, this collaboration can help lay the foundation of a whole new paradigm of social networking, freeing users from the current limitations imposed by un-interoperable data-silos used by large commercial social networking providers.

At the end of year 3, KUL and IMDEA will also start co-operating on the idea of architecture-driven verification of software. The new research will build a formal basis for the informally described patterns, idioms, abstractions and other forms of structure contained in software (typically called the architecture of the application), and the hope is that this will allow verification tools to verify software of real-world size and complexity. If successful, this approach will revolutionize the field of verifying complex software.
7 Conclusion

The work in WP8 has continuously addressed improving techniques and mechanisms to successfully implement secure services. New technologies allow developers to easily combine services together to implement new features that the individual services do not support. However, care must be taken that the combination of services happens in a secure way. In addition, services that are developed from the ground up must be supported by technologies to ensure that they achieve the security requirements. This can be either by helping developers to build secure services, or ensure that the security impact of bugs is negligible. The strategy in this work package is consistent with the approach that has been followed in project year one: these three essential elements in the service creation environment are and remain the cornerstones of this work package: secure execution environments, secure and security service composition, and security in programming environments.

This report gathers the results of new R&D that has been conducted in the second year of this NoE. The contributions in Chapter 2 improve the security of web services and includes work by individual partners as well as collaborative work. The contributions in Chapter 3 focus on the secure composition of services. The same chapter also focuses on composing services that are specifically used for security purposes (e.g. authentication). Chapter 4 further formalizes and valorizes the VeriFast research that was presented in the previous year. Finally, Chapter 5 introduced collaborative and non-collaborative work in the field of information flow analysis.

The range of topics of work package 8 is very broad, and this is clearly visible in the contributions described in this deliverable. An overview of finished work and work in progress has been presented, and an elaborate description of this work can be found in Appendix A of the extended version of this deliverable. Most of this work is ongoing and will be extended further into the third year of the NoE. In addition, we will continue to incubate new ideas that lead to the development of working prototypes. Both the collaboration between the partners in the work package, as well as the collaboration between this work package and others, is strong and ongoing as described in Chapter 6 of this deliverable.
Appendix - Relevant Papers

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


2. Van Acker, Steven and Nikiforakis, Nick and Desmet, Lieven and Joosen, Wouter and Piessens, Frank FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications AsiaCCS, May 2012


6. Willem Penninckx, Jan Tobias Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens Sound formal verification of Linux’s USB BP keyboard driver NASA Formal Methods, volume 7226, pages 210-215, Norfolk, Virginia, USA, 3-5 April 2012


In addition to the work from the above papers that has been summarized in this deliverable, there are a number of other papers that are also (partially) supported by NESSoS and that belong to this work package:


6. Yskout, K., Scandariato, R. and Joosen, W., Does Organizing Security Patterns Focus Architectural Choices?, In International Conference on Software Engineering (ICSE '12)

Bibliography


