Network of Excellence

Deliverable D8.4

Update of research results
<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>D8.4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title of Deliverable</strong></td>
<td>Update of research results</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>1.0</td>
</tr>
<tr>
<td><strong>Contractual Delivery Date</strong></td>
<td>September 30, 2013</td>
</tr>
<tr>
<td><strong>Actual Delivery Date</strong></td>
<td>October 15, 2013</td>
</tr>
<tr>
<td><strong>Contributing WPs</strong></td>
<td>WP 8</td>
</tr>
<tr>
<td><strong>Editor(s)</strong></td>
<td>Wouter Joosen (KUL), Pieter Agten (KUL), Marianne Busch (LMU), Maarten Decat (KUL), Lieven Desmet (KUL), Wouter Joosen (KUL), Boris Köpf (IMDEA), Fabio Martinelli (CNR), Job Noorman (KUL), Marin Ochoa-Ronderos (Siemens), Animesh Pathak (Inria), Pieter Philippaerts (KUL), Frank Piessens (KUL), Steven Van Acker (KUL),</td>
</tr>
<tr>
<td><strong>Author(s)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Reviewer(s)</strong></td>
<td>Michael Rusinowitch (INRIA), Francisco Moyano (UMA), Marinella Petrocchi (CNR),</td>
</tr>
</tbody>
</table>
Abstract

This deliverable reports on the work that was done for work package 8 during the third year of the network of excellence. WP8 is a broad work package that spans multiple disciplines. We classify the contributions under three main pillars that we see as the important categories where research can improve the state of the art. These pillars are (1) security support for programming languages (Task 8.4), (2) runtime support for the secure execution of services (Task 8.3), and (3) secure composition of services (Task 8.2). Each contribution presented in this deliverable belongs to at least one of these categories.

It is clear that the topics covered in this deliverable are very much alive in the scientific community. This is witnessed by the number of NESSoS contributions in this work package that were accepted at top conferences. A list of the individual papers can be found in Section A. A selection of this work is presented in this deliverable.

Keyword List

security, web services, run-time enforcement, component integration, secure service composition, program verification, 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>Wouter Joosen (KUL)</td>
</tr>
<tr>
<td></td>
<td>Added chapter ‘Complete client-side sandboxing of third-party JavaScript without browser modifications’</td>
<td>Pieter Agten, Steven Van Acker, Pieter Philippaerts (KUL)</td>
</tr>
<tr>
<td>0.3</td>
<td>Added chapter ‘Federated Authorization for Software-as-a-Service Applications’</td>
<td>Maarten Decat, Bert Lagaisse, Pieter Philippaerts (KUL)</td>
</tr>
<tr>
<td>0.4</td>
<td>Added chapter ‘Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base’</td>
<td>Job Noorman, Pieter Agten, Raoul Strackx (KUL)</td>
</tr>
<tr>
<td>0.5</td>
<td>Added chapter ‘Secure Navigation Paths’</td>
<td>Marianne Busch (LMU)</td>
</tr>
<tr>
<td>0.6</td>
<td>Added chapter ‘CacheAudit’</td>
<td>Boris Köpf (IMDEA)</td>
</tr>
<tr>
<td>0.7</td>
<td>Added chapter ‘Privacy and Access Control’</td>
<td>Animesh Pathak (INRIA)</td>
</tr>
<tr>
<td>0.8</td>
<td>Interactions and publications</td>
<td>Pieter Philippaerts, Wouter Joosen (KUL), Marianne Busch (LMU), Boris Köpf (IMDEA), Fabio Martinelli (CNR), Animesh Pathak (INRIA)</td>
</tr>
<tr>
<td>0.9</td>
<td>Finishing work</td>
<td>Wouter Joosen, Pieter Philippaerts (KUL)</td>
</tr>
<tr>
<td>1.0</td>
<td>Incorporated reviewer remarks</td>
<td>All</td>
</tr>
</tbody>
</table>

NESSoS - 256980
# Table of Contents

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

1 **INTRODUCTION** .................................................. 10

2 **COMPLETE CLIENT-SIDE SANDBOXING OF THIRD-PARTY JAVASCRIPT WITHOUT BROWSER MODIFICATIONS** ................................................................. 12
   2.1 JSand Security Architecture ........................................ 13
   2.2 Evaluation .......................................................... 14

3 **FEDERATED AUTHORIZATION FOR SOFTWARE-AS-A-SERVICE APPLICATIONS** ................................................................. 15
   3.1 Problem Illustration and Elaboration ................................ 15
   3.2 Solution: Federated Authorization .................................... 17
      3.2.1 Key Features .................................................. 17
      3.2.2 Generic Middleware Architecture for Federated Authorization ................................................................. 18
      3.2.3 Extensions to Current Policy Languages .......................... 20
   3.3 Performance Evaluation ................................................ 20
   3.4 Discussion ........................................................... 21

4 **SECURE NAVIGATION PATHS** ........................................ 22
   4.1 Problem Statement ................................................... 22
   4.2 Background .......................................................... 22
      4.2.1 Secure Navigation Paths ......................................... 22
      4.2.2 UWE .......................................................... 23
   4.3 Secure Navigation paths with UWE .................................... 23
      4.3.1 Modeling Approach ............................................... 24
      4.3.2 Testing ........................................................ 25
      4.3.3 Tool Support ................................................... 26
   4.4 Case Study: SmartGrid Bonus Application ............................ 27
   4.5 Related Work ........................................................ 29

5 **SANCUS: LOW-COST TRUSTWORTHY EXTENSIBLE NETWORKED DEVICES WITH A ZERO-SOFTWARE TRUSTED COMPUTING BASE** .................................................. 31
   5.1 Problem statement .................................................... 31
      5.1.1 System model .................................................... 31
      5.1.2 Attacker model .................................................. 32
      5.1.3 Security properties .............................................. 33
   5.2 Design of Sancus ....................................................... 33
      5.2.1 Overview ........................................................ 33
   5.3 Evaluation ............................................................. 35

6 **CACHEAUDIT: A TOOL FOR THE STATIC ANALYSIS OF CACHE SIDE CHANNELS** ................................................................. 38
   6.1 Illustrative Example .................................................... 39
   6.2 Tool Design and Implementation ........................................ 40
      6.2.1 Control Flow Reconstruction ..................................... 40
List of Figures

Figure 2.1: the JSand architecture. Inside the browser, all access from JSand sandboxes to the JavaScript environment is mediated according to server-supplied policies. ........................................ 13

Figure 3.1: Large organizations such as hospitals have started to adopt SaaS applications. These organizations often employ on-premise infrastructures for managing their users efficiently. Access control for SaaS should integrate with these on-premise infrastructures, which still poses challenges... 16

Figure 3.2: The case study that inspired this work: a home patient monitoring system offered to hospitals as a SaaS application. .......................................................... 16

Figure 3.3: The generic architecture for federated authorization. $P_P$ and $P_T$ are the provider and tenant policy sets respectively and $A_O$, $A_{S,P}$, $A_{E,P}$, $A_{S,T}$ and $A_{E,T}$ are as defined in Section 3.2.2. ........................................ 18

Figure 3.4: The resulting access control flow for the generic architecture. C.H. is context handler, Ob.S. is obligation service, $P_P$ and $P_T$ are the provider and tenant policy set respectively, and $A_O$, $A_{S,P}$, $A_{S,T}$, $A_{E,P}$ and $A_{E,T}$ are as defined in Section 3.2.2. For readability reasons, the provider attribute service is not shown explicitly. ........................................ 19

Figure 4.1: Example of expressing SNPs with UWE ........................................ 24

Figure 4.2: Test generation: Follow SNP and request illicit nodes ......................... 25

Figure 4.3: SNPpolicyTester: Average benchmark result .................................... 27

Figure 4.4: SmartGrid Bonus Application: Basic Rights Model .......................... 28

Figure 4.5: SmartGrid Bonus Application: Navigation State Model .................... 29

Figure 5.1: Overview of our system model. $IP$ provides a number of nodes $N_i$ on which software providers $SP_j$ can deploy software modules $SM_{j,k}$. ........................................ 32

Figure 5.2: Overview of the keys used in Sancus. The node key $K_N$ is only known by $IP$ and the hardware. When $SP$ is registered, it receives its key $K_N,SP$ from $IP$ which can then be used to create module specific keys $K_{N,SP,SM}$. ........................................ 34

Figure 5.3: Relative overhead, in function of the number of cycles used for calculations, of Sancus on the macro benchmark. The nth run is significantly faster due to the secure linking optimization discussed in [100]. ........................................ 36

Figure 6.1: The architecture of CacheAudit. The solid boxes represent modules. Black-headed arrows mean that the module at the head is an argument of the module at the tail. White-headed arrows represent is-a relationships. ........................................ 40

Figure 6.2: Effect of replacement strategy on the security guarantee, for $C^{acc}$ and $C^{accd}$, with and without preloading (PL). The horizontal axis gives the cache size, and the vertical axis gives the leakage in bits. ........................................ 42
Figure 6.3: Effect of attacker model and preloading (PL) on the security guarantee, for the LRU replacement strategy. The horizontal axis gives the cache size, and the vertical axis gives the leakage in bits.

Figure 6.4: Effect of cache line size on the security guarantee, for $C_{acc}$ and $C_{accd}$, and LRU replacement strategy without preloading. Different curves correspond to different cache line sizes. The horizontal axis gives the cache size, and the vertical axis gives the leakage bits.

Figure 6.5: The table illustrates the security guarantees derived by CacheAudit for the implementations of BubbleSort, SelectionSort, and InsertionSort, for trace-based, timing-based, and access-based adversaries, for LRU caches of 4KB and line sizes of 32B.

Figure 7.1: the Yarta-FSN architecture.
EXECUTIVE SUMMARY

This deliverable reports on the work that was done for work package 8 during the third year of the network of excellence. WP8 is a broad work package that spans multiple disciplines. We classify the contributions under three main pillars that we see as the important categories where research can improve the state of the art. These pillars are (1) security support for programming languages (Task 8.4), (2) runtime support for the secure execution of services (Task 8.3), and (3) secure composition of services (Task 8.2). Each contribution presented in this deliverable belongs to at least one of these categories.

It is clear that the topics covered in this deliverable are very much alive in the scientific community. This is witnessed by the number of NESSoS contributions in this work package that were accepted at top conferences. A list of the individual papers can be found in Section A. A selection of this work is presented in this deliverable.

Chapter 2 presents an improvement on WebJail\textsuperscript{1} that offers the same benefits as the original implementation, but does not require intrusive browser modifications. Chapter 3 describes a generic middleware architecture for federated authorization in which the XACML policy language is extended and a supporting distributed execution environment is defined. Chapter 4 proposes a methodology that allows one to specify Secure Navigation Paths (SNP) using UML models and automatically generate a server-side monitor enforcing such policies. Chapter 5 proposes a security architecture that supports secure third-party software extensibility for a network of low-end processors. Chapter 6 proposes a method and a tool for statically determining the degree of isolation between computations of mutually distrusting parties that are carried out on the same execution platform, which is a common scenario in the cloud. Chapter 7 presents initial results on access control and interoperability between online social networks, which forms the basis of collaborative work between KUL and Inria on privacy and access control in federated social networks. Finally, Chapter 8 summarizes the interactions between this work package and other work packages, and between the partners that have been contributing to this work package.

---

\textsuperscript{1}This work was presented in deliverable 8.2.
1 Introduction

Over the past decade, the amount of security vulnerabilities that have been found in software has rapidly increased. This year, over 9% more vulnerabilities were discovered in the first seven months of the year compared to the same period last year, and compared to ten years ago the number of vulnerabilities has increased fivefold. It is clear that this rising trend must quickly be countered, so it is not surprising that software security is a hot topic in both the research and commercial domains.

In order to mitigate the proliferation of vulnerabilities, this work package suggests a number of defense mechanisms that try to tackle the root of particular security problems. It proposes tools that aid programmers to write more secure software, and to provide (sometimes provable) security against certain known vulnerabilities. The work is based on three large pillars: (1) security support for programming languages (Task 8.4), (2) runtime support for the secure execution of services (Task 8.3), and (3) secure composition of services (Task 8.2).

Security support for programming languages aspires to improve programming languages such that writing insecure code is not possible in such a language. Of course, this is still a long-term goal, but intermediate success stories have been presented. Two scenarios that are of particular interest to the NESSoS consortium, is (1) the extension of a language with formal annotations in order to be able to prove certain properties of the resulting code, and (2) the creation of tools that will convert source or binary code into a more secure version than obtained by an ordinary compiler.

Software platforms can be amended to include extra security checks before or during the execution of programs that are running on top of them. Such checks can prevent or detect (and potentially repair) unintended program behavior, or can ensure the compliance of an application to a certain security policy. A lot of research has already been done on this topic, which has led to commercial success stories, but these systems only protect against a small subset of errors, are not fine-grained, or have a number of other drawbacks (such as a lack of performance). Thus, it remains necessary to keep investing resources to further explore this topic and develop systems with new (and better) trade-offs.

Different software services might have different security mechanisms, so combining them is not always easy. The third pillar of this deliverable, secure composition of services tries to solve the problems in this field. It can be split in two similar, but separate fields: (i) the composition of security services (e.g. an authentication service with an authorization service), and (ii) the secure composition of (non-security) services. The latter is the general case, but the former deserves special attention due to its security implications.

It is the aim of this work package to build solutions to specific problems that are encountered in practice. A full list of contributions of the past year can be found in Appendix A. This deliverable has selected some of these contributions, and offers a brief summary of them. The report is structured as follows.

Chapter 2 presents an improvement on WebJail that offers the same benefits as the original implementation, but does not require intrusive browser modifications. This alleviates the maintenance problems that were present in the original version of WebJail.

Chapter 3 describes a generic middleware architecture for federated authorization in which the XACML policy language is extended and a supporting distributed execution environment is defined. The architecture supports requesting an access control decision from the tenant, handling local and remote attributes, and handling local and remote obligations.

Chapter 4 proposes a methodology that allows one to specify Secure Navigation Paths (SNP) using UML models and automatically generate a server-side monitor enforcing such policies. We also discuss how those models can be used to generate tests in case a monitor is absent. We report on tool support for this methodology and on applications to a SmartGrid usage scenario.

Chapter 5 proposes a security architecture that supports secure third-party software extensibility for a network of low-end processors. The architecture enables mutually distrusting parties to run their software modules on the same nodes in the network, while each party maintains strong assurance that its modules run untampered.

Chapter 6 proposes a method and a tool for statically determining the degree of isolation between computations of mutually distrusting parties that are carried out on the same execution platform, which is a common scenario in the cloud.

Chapter 7 presents ongoing work on assuring privacy and access control in federated social networks, where users of one online social network are free to interact with those of another social platform. This is an initial report on collaborative work between KUL and Inria.

1 source: http://nvd.nist.gov/
2 This work was presented in deliverable 8.2.
Finally, Chapter 8 summarizes the interactions between this work package and other work packages, and between the partners that have been contributing to this work package. Chapter 9 concludes the deliverable.
2 Complete Client-side Sandboxing of Third-party JavaScript without Browser Modifications

In the last decade, the web platform has become the number one platform on the Internet. There is a clear paradigm shift from desktop applications and proprietary client-server solutions towards web-enabled services. An important catalyst for this paradigm shift has been the power of JavaScript as well as the advent of HTML5, giving web developers the tools to build rich and interactive websites.

To enrich the functionality and interaction of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. Recent studies [124] have shown that 96.9% of websites include scripts from external sources, and on average each website includes scripts from 2.56 external sources. For example, websites integrate among others JavaScript-enabled advertisements (such as Google AdSense and adBrite), Web analytics frameworks (such as Google Analytics, Yahoo! Web Analytics and Tynt), web widgets and buttons (such as Google Maps, addToAny button and Google +1 button), and JavaScript programming libraries (such as jQuery and Dojo). The popular news site nytimes.com for instance, includes 17 pieces of third-party JavaScript code, hosted on 6 unique domains.

The de facto browser security model today is defined by the Same-Origin Policy (SOP). The SOP restricts access of client-side scripts to resources belonging to the same origin. For instance, the SOP ensures that document data and cookies from one origin cannot be read by scripts belonging to another origin. However, the SOP includes some important relaxations with respect to navigation and content inclusion (e.g. embedded images, scripts) [125]. In particular, if a page from one origin includes a script from another origin, the included script is treated as if it belongs to the including origin, and hence it inherits all the capabilities and permissions of the hosting page. This makes malicious script inclusion a very powerful attack vector.

Several countermeasures have been proposed to limit the capabilities of third-party JavaScript, including (1) the introduction of safe subsets of JavaScript [113, 46, 86], (2) client-side reference monitors [90, 118], and (3) server-side transformations of the JavaScript to be included [93, 112]. However, all existing countermeasures have at least one of the following limitations.

First, some approaches [90, 118] require intrusive browser modifications, in particular to the JavaScript engine and the binding between browser and JavaScript engine. Such intrusive browser modifications hinder short-term deployment of the countermeasure. Browser vendors are not very keen on adding new (unproven) countermeasures to the base code of the browser, so maintenance of the countermeasure integration code is a burden for its author.

Second, some approaches do not support client-side script inclusion: in order to perform server-side pre-processing (e.g. source-to-source translation or filtering) of the scripts, the scripts have to pass through the web server [93, 113, 46]. This effectively changes the architectural model of client-side script inclusion to server-side script inclusion.

Third, some approaches do not provide complete mediation between different scripts on the same page, or to all resources exposed in the browser. Self-Protecting JavaScript (SPJS) [103, 87] assumes that all scripts included on a hosting page need identical security constraints. It does not differentiate between different external scripts nor between local and remote inclusions. AdJail [112] successfully isolates untrusted advertisements from the Document Object Model (DOM) of the hosting page, but since it uses iframes as isolation units, it cannot fully protect security-sensitive APIs such as XHR, Geolocation and local storage.

Inspired by recent advances in achieving object-capability guarantees for JavaScript [93, 85, 91], this section introduces JSand, a novel security architecture to securely integrate third-party JavaScript. We improve upon the state-of-the-art with the following contributions:

1. JSand is the first JavaScript sandbox that (1) does not need browser modifications, (2) supports client-side script inclusion and (3) completely mediates different scripts and the browser APIs.

2. We show evidence that JSand is secure, compatible with complex and widely used scripts (such as Google Maps, Google Analytics and jQuery) and performs sufficiently well

This chapter summarizes the details of the prototype implementation. More information about this work can be found in [22].

---

1 An origin is defined as a (protocol, domain name, port) tuple.
2.1 JSand Security Architecture

The JSand architecture empowers the owner of a website to securely integrate third-party scripts, without the need for disruptive change to either server-side or client-side infrastructure. We first give a high-level overview of the architecture and then discuss the architectural choices under the hood.

Architectural overview Figure 2.1 depicts the JSand architecture. A website owner deploys JSand by including the JSand JavaScript library in his web pages. When one of these pages is loaded in a visitor’s browser, the third-party scripts to be sandboxed are fetched directly from the servers of the script provider. The JSand library confines each script to its own secure sandbox, which isolates the script from other scripts and from the DOM.

Under the hood The JSand architecture is based around the secure confinement of third-party JavaScript. JSand realizes this through the use of an object-capability environment. Such environment provides an appropriate device for isolating untrusted JavaScript: without an explicit and unforgeable reference to a security-sensitive object or function, a script is unable to access the resource or make use of its capabilities. The object-capability model is at the basis of Caja [93], and many other safe subsets of JavaScript [85].

The JSand library invokes third-party JavaScripts with an initially minimal set of capabilities (i.e. unforgeable references). To maintain control over all references acquired by a sandboxed script, JSand applies the Membrane pattern proposed by Miller [92]. Our implementation of this pattern consists of placing policy-enforcing wrappers around objects that potentially provide security-sensitive operations. Whenever one of these objects returns a reference to another object, the membrane is extended to cover that object as well. This ensures a sandboxed script never has direct access to a security-sensitive operation.

The membrane’s wrappers implement the decision points for the security policy, since they intercept all operations performed on the objects they wrap. On each decision point, the wrapper consults the security policy to determine whether or not the corresponding operation is permitted. If not, this will be indicated by the security policy and the operation will be blocked. The architecture is not bound to any specific type of security policy, which gives website owners the freedom to enforce arbitrarily complex policies.

Since any interaction between a script and the browser is performed by using a DOM method, it suffices to wrap all DOM objects in order to enforce a policy on all security-sensitive operations. Such operations include not only operations to read or modify content of the hosting page, but also include communication with other scripts and the use of JavaScript APIs such as XHR, Geolocation, etc.

In conclusion, the JSand architecture provides an end-to-end solution for securely integrating third-party JavaScripts on a website. The website owner is able to define and enforce security policies on third-party scripts, which puts him back into the driver’s seat. JSand does not require disruptive change to the architecture of the web: it does not break direct script delivery towards the browser, and can be deployed without additional server-side or client-side
infrastructure. The combination of the object-capability model and the Membrane pattern ensures that all access to security-sensitive operations passes through the membrane’s wrappers, which enforce the security policy.

2.2 Evaluation

JSand was evaluated against a number of requirements. This section gives a short overview of the results.

Complete mediation All sandboxed scripts are executed in an object-capability environment, set up by the SES library. Our implementation of the Membrane pattern ensures that each DOM access and JavaScript API call made by a sandboxed script is assessed by the security policy. Based on the theory of object-capability systems, this provides complete mediation.

Backwards compatibility We have extensively and successfully tested our prototype on a variety of script inclusions. In [20] we report and discuss in detail three of the most widespread included JavaScripts around: Google Analytics, Google Maps and the jQuery library. Google Analytics is included from more than 68% of all domains from the Alexa Top 10 000, making it the most included script on this list. Google Maps is the most included web mashup API according to [104], being used in 17.41% of registered mashups. jQuery is the most popular JavaScript library in use today, included in more than 57% of the top 10 000 web sites to date [35].

Performance To evaluate the runtime overhead of our prototype, we have conducted micro- and macro-benchmarks. All benchmarks were run using Google Chrome v20.0.1132.21 on an Intel Core 2 Duo T8300 2.4GHz processor with 4GB RAM. The micro-benchmarks showed that the framework took on average 48.5 ms to load on the clientside. The loading overhead of the jQuery library increased with 1350.6 ms. A function call crossing the membrane goes from about 1µs to 8µs. The most important metric that counts when executing JavaScript in a browser, is the user experience. Ideally, the user should not notice that JSand is being used at all. The overall performance of a JSand sandbox is quite acceptable. The overhead when loading a reasonably-sized SES-compliant JavaScript library inside the sandbox, is about 203%. For legacy scripts, JSand requires a code transformation step that results in a total overhead of about 365%, but it is expected that this step can be removed or at least sped up significantly for future JavaScript code in future browsers.
3 Federated Authorization for Software-as-a-Service Applications

Software-as-a-Service or SaaS is a form of cloud computing in which the tenant rents access to a shared application hosted by the provider [88]. The tenant is an organization representing multiple end-users, who use the application through a thin client, typically a web browser. A SaaS application is used by multiple tenant organizations and each organization can be tenant for multiple SaaS applications. While traditional SaaS applications such as Google Apps (an office suite) and Salesforce (CRM) mainly target small enterprises, large enterprises have recently started to adopt SaaS as well, for example in the domains of document processing [15], workforce management [15] or e-health [13, 14]. This evolution stresses key challenges, such as the increased importance of security in general and access control in particular.

While the data in a SaaS application is hosted by the provider, the tenant should still be able to control access to it based on tenant-specific access control policies. While large organizations often employ on-premise access control infrastructures for managing their users centrally and efficiently (illustrated in Fig. 3.1), state-of-practice SaaS applications offer tenants an application-specific access control configuration interface. As a result, the tenant policies are stored and evaluated provider-side. This causes two major problems: (i) This approach fails to integrate with the on-premise infrastructures of the tenant and again distributes and scatters its access control management. Since a single organization can be tenant of multiple SaaS applications, this leads to large administrative overhead and eventually to inconsistencies and security holes. (ii) This approach forces the tenant to disclose to the provider all access control data required for evaluating its policies. While SaaS applications outsource specific functionality, the organization-wide policies of the tenant apply. These policies reason about data of the organization stored in the on-premise infrastructures, such as attributes of a patient or a physician. Although the tenant may trust the provider with the data in the application, it does not necessarily want to trust the provider with this sensitive on-premise data needed for access control. Moreover, regulatory requirements such as the European DPD [42] even forbid the tenant to share the data.

This chapter first introduces the concept of federated authorization for SaaS applications. It then presents a generic middleware architecture for federated authorization in which the XACML policy language is extended and a supporting distributed execution environment is defined. This architecture has three key features: (i) requesting an access control decision from the tenant, (ii) handling local and remote attributes and (iii) handling local and remote obligations. A short summary of the evaluation in terms of performance and security concludes this chapter. A detailed description and evaluation of this work can be found in [50].

3.1 Problem Illustration and Elaboration

As stated in the introduction, large enterprises have started to adopt SaaS. An example of such an application inspired by a number of research projects [13, 14] is a home patient monitoring system provided to hospitals as a service (see Fig. 3.2). The system allows patients to be monitored continuously after leaving the hospital, for example by a chest band or a wrist sensor. The measurements (the application data) are sent from the patients to the application back-end using a smartphone as an intermediary device. The measurements are stored and processed by the provider: telemedicine operators continuously check the patient’s status and the patient’s physician at the hospital is notified in case of important evolutions. A patient’s status can also be viewed by other physicians and nurses and by the patients themselves. The hospitals are the tenants of the application and each tenant represents multiple patients and physicians, i.e., the end-users of the application. Next to the monitoring system, the hospital employs other on-premise applications, e.g., for patient management, and other SaaS applications, e.g., for medical imaging.

**Example policy.** A typical example of an organization-wide policy employed by the hospital in this case study can be summarized as follows: “a physician can only view or alter patient data when currently treating the patient who owns the data and if that patient has given consent or when the data is relevant to his specialization or when the patient is in a life-threatening situation”.

It is clear that the provider wants to control access to the application. The provider constrains its own end-users, i.e., the telemedicine operators, and constrains its tenants, e.g., to ensure a tenant has enough credit to perform a certain action or is credited afterward. However, e-health applications are subject to regulatory requirements (e.g., the European DPD [42]) and even if the data is hosted by the provider, the hospital is still accountable for it. Therefore, the hospital has to be able to control access to the application as well and the hospital-wide policy
Figure 3.1: Large organizations such as hospitals have started to adopt SaaS applications. These organizations often employ on-premise infrastructures for managing their users efficiently. Access control for SaaS should integrate with these on-premise infrastructures, which still poses challenges.

The above presented also applies to the SaaS application. In this work we take the point of view of the tenant and focus on the following three requirements:

**Scalable administration.** Hospitals can have a medical staff of thousands and treat even more patients each day, continuously producing large amounts of new application data. Manual security administration on this scale would incur too much administrative overhead and would quickly lead to inconsistencies and security holes. To avoid this, hospitals have extensive centralized security infrastructures at hand, e.g., organization-wide patient management systems. When employing SaaS applications such as the patient monitoring system, this degree of centralization should be upheld and security administration should remain centralized at the hospital.

**Complex and large policies.** Current e-health policies require detailed and extensive information. For example, the small policy presented above requires user roles, treating relationships, patient consent, resource content and more. Furthermore, the complexity of current policies makes it impossible to statically determine the required data up-front.

**Sensitive access control data.** Some of the access control data required for evaluating the policy presented above is sensitive in nature, such as the list of patients being treated by a physician, patient diseases or patient consent. While the hospital may trust the provider with the monitoring data in the application, it does not necessarily want to trust the provider with this sensitive access control data. Moreover, regulatory requirements such as the European DPD [42] even forbid the hospital to share this data.

The hospital policies are evaluated by the provider in state-of-the-art access control for SaaS. As a consequence, the hospital policies are still distributed and fragmented over the multitude of applications it uses. Moreover, all

Figure 3.2: The case study that inspired this work: a home patient monitoring system offered to hospitals as a SaaS application.
tenant data required for evaluating the tenant policies has to be shared with the provider. Therefore, state-of-the-art federated access control does not fulfill the requirements stated above: it leads to limited administrative scalability and forces the tenant to disclose sensitive access control data.

3.2 Solution: Federated Authorization

We address the problems identified in the previous section by externalizing tenant policy evaluation from the provider to the tenant. This effectively centralizes tenant access control management and enables the tenant to enforce policies on the SaaS application without having to share the access control data needed for evaluating these. In analogy to federated authentication, we call this federated authorization.

Section 3.2.1 first describes the key features required for realizing federated authorization and their impact on the XACML reference architecture and current policy languages. Section 3.2.2 presents our generic architecture for federated authorization and finally Section 3.2.3 illustrates the required policy language extensions using the XACML policy language.

3.2.1 Key Features

With federated authorization, the provider evaluates its own policies while the evaluation of tenant policies is externalized from the provider to the tenant. With respect to the XACML reference architecture, federated authorization adds three key features: (i) requesting an access control decision from the tenant, (ii) handling local and remote attributes and (iii) handling local and remote obligations. For each of these new features, we determine what should be added to the XACML reference architecture and to current policy languages such as XACML [96], Ponder [48] or EPAL [25].

Requesting tenant access control decisions.

As a first key feature, the provider is able to request an access control decision from the tenant concerning the SaaS application. Afterward, the provider combines the tenant response with its own policy results so that the requested action is only permitted if both parties permit it. Allowing the provider to request a tenant access control decision impacts both the architecture and the policy language.

Architecture. The tenant should provide a service to receive decision requests from the providers of the SaaS applications it employs. Such a request should identify the provider and should contain information about the subject, the object, the action and the environment, similar to a request from PEP to PDP. Using this information, the tenant determines and evaluates the applicable policies and returns its response. This response contains the decision itself (permit or deny) and possibly obligations, similar to a response from PDP to the PEP. Notice that the provider does not reference a particular tenant policy, but rather the tenant as a whole behaving like a PDP.

Policy Language. The policy language should allow the provider to refer to the access control decision service of the tenant and specify how the result should be processed, similar to on-premise policies.

Handling local and remote attributes.

As a second key feature, all required attributes are made available to the respective policies. As mentioned before, the provider reasons about its end-users (e.g., the telemedicine operators in the case study) and its tenants (e.g., the hospital) and the tenant reasons about its end-users (e.g., the hospital physicians and nurses). For the provider policies, the subjects are therefore its end-users and tenants, the objects are the application data and the environment is the SaaS application. Thus, all attributes required by the provider policies are available locally. However, for a tenant policy, the subjects are its end-users, the objects are the application data and the environment comprises both the SaaS application and the local infrastructure. Data about the tenant end-users and local infrastructure is stored in its on-premise applications, while the rest is hosted by the provider. Thus, the attributes required by the tenant policies are distributed over tenant and provider. Allowing the tenant to handle both local and remote attributes impacts both the architecture and the policy language.

Architecture. For evaluating the tenant policies tenant-side, the attributes of the objects in the SaaS application and the attributes of the provider-side environment have to be made available to the tenant. In addition, while attributes can be added to the initial request to the tenant, the complexity of current policies makes it impossible to
Figure 3.3: The generic architecture for federated authorization. $P_P$ and $P_T$ are the provider and tenant policy sets respectively and $A_O$, $A_{S,P}$, $A_{E,P}$, $A_{S,T}$ and $A_{E,T}$ are as defined in Section 3.2.2.

Policy statically determine the required attributes up-front. Therefore, the provider should provide a service to the tenant to dynamically fetch required attributes.

Policy Language. When evaluating the tenant policies, the context handler should know where to find the required attributes. Therefore, the policy language should allow to define the location of each attribute referenced in the policies.

Handling local and remote obligations.

As a third key feature, the tenant is able to handle both tenant-side obligations, e.g., for logging, and provider-side obligations, e.g., for updating the access control history of an object. Allowing the tenant to handle local and remote obligations impacts both the architecture and the policy language.

Architecture. The response from the tenant policy evaluation service should contain the obligations specified by the tenant which will be fulfilled by the provider. This was already mentioned before. The tenant response should not contain locally fulfilled obligations.

Policy Language. The policy language should allow the tenant to specify where obligations should be fulfilled: locally or remotely.

3.2.2 Generic Middleware Architecture for Federated Authorization

Based on the architectural requirements listed above, we now present a generic architecture for federated authorization. We first describe the decomposition of the architecture aligned to the XACML reference architecture and then illustrate the resulting access control flow.

Architectural Decomposition.

Figure 3.3 shows the decomposition of the architecture. First of all, the provider hosts the SaaS application and therefore also the PEP and no application components are located at the tenant side. Since both parties evaluate policies and process obligations, both have a PAP, a PDP, a context handler, one or more PIPs and an obligation service. The provider PIP contains the attributes of the objects in the SaaS application ($A_O$), the attributes of the provider policy subjects ($A_{S,P}$) and the attributes of the provider-side environment ($A_{E,P}$). The tenant PIP contains the attributes of the tenant policy subjects ($A_{S,T}$) and the attributes of the tenant-side environment ($A_{E,T}$). For handling decision requests, the tenant offers a Remote Policy Decision Point (RPDP) to the provider. For handling attribute requests, the provider offers an attribute service to the tenant. The provider PDP is extended with functionality to contact the RPDP and the tenant context handler is extended with functionality to contact the provider.
attribute service. To summarize, the resulting interface between provider and tenant consists of two services: the tenant policy evaluation service and the provider attribute service. Notice that the architecture still allows the tenant to use its infrastructure for on-premise applications as well, which is not shown explicitly.

Access Control Flow.

Figure 3.4 illustrates the resulting access control flow. The presented flow starts after the provider and tenant policies are loaded from their respective PAPs and the end-user has been successfully authenticated. The remainder of the flow is as follows: (1) With each request an end-user makes to the application, the PEP constructs an access control request and forwards this to the local context handler. (2) The context handler collects statically known attributes from the local PIPs (only one is shown in Fig. 3.4), adds these to the request and sends it to the local PDP. (3) The PDP determines the applicable provider policies and evaluates them, thereby dynamically requesting attributes from the context handler. When the PDP encounters a remote policy reference, it asks the context handler to determine how and where to contact the tenant. The PDP then constructs a decision request and sends it to the tenant RPDP. (4) From the tenant point of view, the RPDP acts similarly to a PEP and forwards the request to the tenant context handler. (5) The context handler collects statically known attributes from the local PIPs (only one is shown in Fig. 3.4), adds these to the request and forwards it to the PDP. (6) The PDP determines the applicable tenant policies and evaluates them, thereby dynamically requesting attributes from the context handler. The context handler fetches tenant attributes locally and contacts the provider attribute service for provider attributes. The tenant PDP eventually returns its response (i.e., decision and obligations) to the context handler. (7) The context handler returns the response to the RPDP. (8) The RPDP fulfills tenant obligations using the tenant obligation service and removes these from the response. (9) The RPDP returns the resulting response to the provider PDP. (10) The provider PDP combines the tenant decision with the result of the provider policies so that the request is only allowed if both parties allow it and returns the overall response to the provider context handler. (11) The provider context handler returns the response to the PEP. (12) The PEP fulfills the obligations using the provider obligation service and enforces the decision.
3.2.3 Extensions to Current Policy Languages

The three key enablers to support federated authorization identified in Section 3.2.1 all required policy language extensions. To illustrate the practical impact of these requirements, we have extended the XACML policy language v2 [96]. We opted for XACML because of its wide-spread use in both academia and industry and its active development. In this section we describe the extensions we introduced, their specifications are provided on-line. We start by introducing XACML first.

The XACML policy language. In addition to the access control reference architecture, the XACML standard also defines a policy language [96], hereafter simply referred to as “XACML”. XACML expresses policies using XML. Three main elements are defined: <PolicySet>, <Policy> and <Rule>. A policy set can contain multiple policies and a policy can contain multiple rules. A rule specifies an effect (permit or deny), attribute-based conditions for this to hold and obligations to fulfill with it. A policy combines the results of its rules using a rule-combining algorithm (e.g., deny overrides), a policy set combines the results of its policies using a similar policy-combining algorithm. Each of these elements also contains an attribute-based target specifying in which situations it applies.

Referencing remote policies. For referencing remote policies, the <RemotePolicyReference> element is introduced. As mentioned before, the tenant behaves as a remote policy from the provider point of view. Therefore, evaluating a remote policy reference returns a policy evaluation result and the element can be part of a policy set, similarly to the <Policy> element. The PolicyId attribute specifies the id of the remote policy. Following the XACML design principles, it is left to the context handler to determine how and where to contact it. A remote policy reference can also contain a description, a target and obligations for local use.

Handling local and remote attributes. For differentiating between local and remote attributes, we follow the XACML design choice of having the context handler infer the location of an attribute based on its id instead of defining attribute properties declaratively. Thus, XACML is not extended for this requirement.

Handling local and remote obligations. For differentiating between tenant and provider as obligation targets, the <Obligation> element is extended with the optional FulfillWhere attribute. This attribute specifies whether the obligation should be fulfilled locally (with the tenant) or remotely (with the provider), local fulfillment being the default. The new attribute is only to be used in tenant policies. It is the responsibility of the tenant to remove local obligations from its response so that the provider can interpret all obligations as before.

3.3 Performance Evaluation

Following the description of the supporting middleware, this section evaluates the concept of federated authorization in terms of performance based on a prototype of the middleware. The performance evaluation explores the impact of federation on the time it takes for the provider to reach an access control decision. Because federated authorization only affects the evaluation of the tenant policies, the provider policies are not taken into account.

The tests compare federated authorization against the two strategies for SaaS authorization in the state-of-the-art. Thus, three different cases of authorization are compared: full provider-side authorization, provider-side authorization with federated authentication and federated authorization. Notice that only federated authorization keeps sensitive access control data of the tenant confidential.

The tests employ policies which require 10, 20 and 30 attributes. Access control studies in a number of research projects [13, 14, 15] show that these numbers represent modest to large policies. Because multiple strategies for fetching attributes exist, ranging from one-at-a-time to combined requests, the tests also compare the two extreme strategies: fetching all required attributes separately and fetching all attributes at the same time as one multi-valued attribute. All attributes are fetched just-in-time and no attribute caching is used.

The prototype contains four main components: (i) the provider PDP, (ii) the tenant PDP (iii) the provider attribute web service and (iv) the tenant attribute web service. Each of these components is run on a separate machine with 4GiB RAM and two cores of 2.40GHz running Ubuntu 12.04. Local databases are run on the same node as the services that use them. To simulate the distance between tenant and provider in a realistic SaaS setting, a fixed single-way network delay of 5ms between tenant and provider is applied. Tests are run sequentially and PDP evaluation is done single-threaded. Each test starts with five warm-up requests and is repeated until the confidence interval lies within 1% of the sampled mean for a confidence level of 95%.

The whole set of results is publicly available1 and a detailed description of the results can be found in [50].

---

However, from the results, we can conclude that federated authorization comes with a performance penalty compared to full provider-side authorization. Depending on the relative amount of tenant attributes in the tenant policies, federated authorization can achieve better performance than provider-side authorization with federated authentication. To illustrate a realistic case, the example policy rules from the e-health case study presented in [50] require significantly more tenant attributes than provider attributes: the tenant hosts the subject roles, treating relationships, patient consent and patient diseases while the provider hosts ownership relations and the application data itself.

3.4 Discussion

In this chapter, we described federated authorization. The two main goals of federated authorization were (i) to enable scalable access control management for SaaS applications by integrating it with the on-premise tenant infrastructures and (ii) to lower the required trust in the provider by removing the need to share sensitive access control data. Federated authorization effectively achieves both by allowing the tenant to evaluate its policies locally, but does come with a performance penalty, as shown in Section 3.3. Full provider-side authorization achieves better performance, but does force the tenant to disclose sensitive access control data to the provider. Therefore, we can conclude that federated authorization poses a clear trade-off between security and performance.

While federated authorization removes the need to share sensitive access control data, a potential threat lies in the fact that the provider could still infer information about this data using the collection of access control requests and responses from the tenant collected over time. We argue (i) that the provider has no business incentive for this, since very little can be gained w.r.t. the contract with the tenant and (ii) that the possibly inferred knowledge is limited since both the tenant policies and the access control data used for evaluating these are kept unknown. However, future work is required to answer this question more quantitatively, for example using techniques such as logical abduction. Also notice that externalizing policy evaluation to the tenant allows it to lie or provide incorrect results. However, the tenant has no incentive to do this, since the policies control access to its own data.

Also, while lowering the required trust in the provider, federated authorization does introduce new security threats. From the point of view of a network attacker, the main difference between provider-side and federated authorization is the externalization of the policy evaluation process. This results in a new communication channel between tenant and provider, which introduces a denial of service threat: since a tenant decision is required for every request to the SaaS application, blocking the channel or the services is a way to deny the use of the application for a specific tenant. This threat cannot be easily mitigated. The channel should also be secured against the threats of information disclosure, tampering and spoofing and against attacks such as replay. For SOAP web services, WS-Security [81] provides the necessary security primitives.

Towards the future, the performance of federated authorization can be improved. First of all, the performance evaluation shows that the overhead of federated authorization is highly dependent on the number of requests between provider and tenant. Therefore, a good strategy for fetching attributes is essential to improve performance. For example, combining multiple attributes in a single request is able to lower the number of required requests. In the extreme case, this approaches the tests with a single multi-valued attribute. Secondly, employing caching also has the ability to lower the required provider-tenant communication. However, caching also has an impact on security, as discussed by Gheorghe et al. [62]. Next to attributes, access control decisions can also be cached or even inferred, as shown by Wei et al. [121]. We plan to investigate both in future work. Finally, a more fine-grained and dynamic policy deployment strategy also has the ability to improve performance while maintaining the trust advantages. This chapter explored two extremes: the tenant policies are either completely evaluated by the tenant or by the provider. Based on the location of the required data and its sensitivity, the tenant policies could be split and distributed for optimal performance while maintaining the confidentiality of the tenant data. We recently explored this strategy [49] and plan to refine this work in the near future.
4 Secure Navigation Paths

Although there exist robust solutions for managing authentication, authorization and session management in web applications, the question of effectively controlling the navigation flow for different users remains challenging. In Deliverable D8.3 we reported about first results regarding secure navigation flow in web applications modeled with UWE (UML-based modeling language). In this chapter we go into more detail and add information about the runtime behavior of our monitor and about how to generate tests in case a monitor is absent. Additionally, we report on tool support for this methodology and on applications to a SmartGrid usage scenario [39].

4.1 Problem Statement

In the era of modern information technologies, where sensitive personal data is stored and managed over the Internet on databases or remote servers, software engineers are constantly faced with new and challenging tasks in the field of security and user guidance. In general, there exist security frameworks that provide means to enforce security and data protection for web applications: (1) Authentication, which is the process of proving the identity of a user to gain access to a protected resource; (2) Authorization, which is the process that determines what a subject (e.g., a user or a program) is allowed to access, especially what it can do with specific objects (e.g., files) [23]; (3) Encryption, which is the conversion of data into a format that cannot be understood by unauthorized subjects and (4) Session management, which is the process of keeping user-specific application data while a user is authenticated to the system. Examples of such security frameworks include Spring Security [9] and Apache Shiro[2]. Both were designed to provide a high standard of security as comfortably as possible.

However, there is one important aspect of security in the area of data protection which usually remains challenging: A kind of navigational access control which guarantees that users with a certain role have only a limited number of Secure Navigation Paths (SNPs) inside an application’s context. 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 trigger a worst case scenario like losing money or seriously damaging the image of the application provider.

Our contributions is twofold: on the one hand we propose a methodology to define simultaneously RBAC and SNP policies for an application by modeling the integrity of its business workflows using UML-based Web Engineering (UWE) [36] and for safeguarding them at runtime by a monitor, which also enforces navigational access control. On the other hand we provide means for automatically testing SNP policies for applications not running an enforcing monitor based on UML policies.

The remainder of this chapter is structured as follows: Section 4.2 provides information about secure navigation paths and summarizes previous work on modeling web applications with UWE. Section 4.3 is the central section. It shows how to model SNPs with UWE and how to test them. Additionally, we present tool support developed to validate our approach. Section 4.4 illustrates a usage scenario from the SmartGrid domain. We review related work in Section 4.5.

4.2 Background

In this section we briefly recall the addressed challenge of enforcing secure navigation paths and the chosen modeling methodology, UML-based Web Engineering, which is part of our solution. UWE was chosen because it already enables the web developer to model web applications including security features such as access control or authentication.

4.2.1 Secure Navigation Paths

Among the most challenging web application vulnerabilities are the ones involving the misuse of the application logic itself. As stated by the Common Weakness Enumeration (CWE) [3]:

Errors in business logic can be devastating to an entire application. They can be difficult to find automatically, since they typically involve legitimate use of the application’s functionality.
Exploiting flaws in the business workflow is a common attack to the application logic. These exploits typically consist of jumping to certain URLs, bypassing critical controls of an intended flow or manipulating the parameters of legal requests. The consequences of those attacks can be diverse: bypassing log-in controls result in authentication breaches whereas skipping certain controls in a trading operation might result in monetary benefits to the attacker. Examples of documented vulnerabilities in popular web applications include the Yahoo SEM Logic Flaw [12]: if one deposited USD $30 into an advertising account, Yahoo would then add an additional USD $50 to that account. The sign-up process was able to be circumvented such that failing to deposit the USD $30 still allowed to receive the additional USD $50. Other examples include bypassing of age restrictions in Youtube, access to private photos in MySpace (resulting in attacks on celebrities) among others (see [4]).

In recent years, much attention has been given to validating user input to web applications to prevent code-injection (i.e. SQL, XSS), but very few tools and methodologies are available to prevent and test logical errors (some attempts include [58, 97]). This is due to the almost endless possible logical flaws that could be present in an application ranging from obvious to very subtle coding vulnerabilities. In this work we focus on one of the most commonly abused logic vulnerabilities: the integrity of the navigation paths as intended by the application owner, that is, the order in which authorized resources of an application should be accessed by a given user role.

4.2.2 UWE

In the following, we outline UML-based Web Engineering (UWE) [36], the security-aware engineering approach we have chosen for modeling web applications.

One of the cornerstones of the UWE language is the “separation of concerns” principle, which is implemented by separate models for different views. However, we can observe that security features are cross-cutting concerns which cannot be separated completely:

The Requirements Model defines (security) requirements for a project.

The Content Model contains the data structure used by the application.

The UWE Role Model describes a hierarchy of user groups to be used for authorization and access control issues. It is usually part of a User Model, which specifies basic structures, as e.g., that a user can take on certain roles simultaneously.

The Basic Rights Model describes access control policies. It constrains elements from the Content Model and from the Role Model.

The Presentation Model sketches the web application’s user interface.

The Navigational State Model defines the navigation flow of the application and navigational access control policies. The former shows which possibilities of navigation exist in a certain context. The latter specifies which roles are allowed to navigate to a specific state and the action taken in case access cannot be granted. In a web application such actions can be, e.g., to logout the user and to redirect to the login form or just to display an error message. Furthermore, secure connections are modeled here.

For each view, an appropriate type of UML diagram is used, e.g., a state machine for the Navigational Model. In addition, the UWE Profile adds a set of stereotypes, tag definitions and constraints, which can be downloaded from the UWE website [83]. Further details of our modeling approach can be found in the following section.

4.3 Secure Navigation paths with UWE

In this section we propose a solution for controlling the intended navigation paths in web applications. The idea is to specify navigation policies by means of UML state-chart diagrams (Section 4.3.1). Each node in the state machine represents a basic interaction step, i.e., (a part of) a web page. By using the UWE approach, it is also possible to annotate transitions with necessary permissions required to access a resource using a Role-Based Access Control (RBAC) scheme. In this way, a monitor can be automatically generated that enforces the desired navigation policies for multiple roles, separating the navigation control from the application itself. We also discuss how to automatically generate tests for a given policy in case the monitor is absent, as is the case for legacy applications (Section 4.3.2). In Section 4.3.3 we present tools we have written for (1) supporting the user while modeling SNPs; (2) for exporting graphical model of SNPs as text; (3) for monitoring web applications and (4) for testing them.
4.3.1 Modeling Approach

We focus on how to model basic SNPs, on a notation for specifying parameters for web pages and on the relation of SNPs and RBAC.

**Basic SNPs.** As introduced in Section 4.2.2, UWE provides several views. For each view, an appropriate UML diagram is selected. In UWE, UML state machines are used to model the navigational structure of a web application. In our case, states correspond to navigational nodes that are implemented as web pages. Transitions define all possibilities to navigate from one page to another and thus specify a policy for the navigation.

Building on UWE’s Navigational State Model, we define SNPs as follows: in case a transition leads from state $A$ to $B$, a user can visit page $B$ after having visited page $A$. If it should be possible to go back (e.g. with the back-button in the browser), another transition has to connect $B$ with $A$. For more than one option, an arbitrary number of transitions can be used.

We use the behavior of UML composite states to model links which should be available within a certain area at any time. “Composite” means states can be nested within a composite state. If state $Y$ and $Z$ and an initial node are nested inside a state $A$ and the initial node is connected to $Y$, then transitions to $A$ activate $Y$ (because of the inner initial node). A transition that leads from $A$ to an arbitrary new state $B$ can be fired from inside $A$, no matter if $Y$ or $Z$ is active. Consequently, $A$ does not correspond to a web page itself, but groups others. An example of a UWE navigational diagram expressing SNPs is depicted in our case study (Section 4.4, Figure 4.5).

In this way we model SNPs at a high level of abstraction, as no technical details have to be given.

**SNPs with Checked Parameters.** However, many web pages use parameters to pass, e.g., user input or session IDs to the next page. To model allowed parameters specifically, we extend UWE’s Navigational State Model so that a minimum of technical information can be specified, if needed. Our extension is inspired by Braun et al. [32], who came up with a textual control-flow definition language for MVC (Model-View-Controller)-based web applications. As our approach does not specify method names, but page names, it is not restricted to MVC-based applications. Furthermore, information about allowed parameters can easily be added to existing Navigational State diagrams, so that related information about authentication, secure connections, navigational control as well as SNPs can be overseen immediately. We add parameters to transitions using a guard like $\{ \text{param = GET(par1:type1, par2:type2)} \}$. GET or POST are allowed and types can be bool, numeric or string (c.f. part a of Figure 4.1).

Sometimes, a parameter should be added to requests within a certain area, using the same value as at the first occurrence. For instance, a session ID is not allowed to change during a session and selected items should not change during the payment process. This can be modeled by a composite state which comprises all transitions that should use fixed parameters. All navigational states in UWE inherit from the stereotype «NavigationalState», for which a tag called $\{ \text{fixedParam = POST(par1:type1, par2:type2)} \}$ can be set. Global parameters are applied to all transitions where the target state is located within the composite state. The choice of GET / POST for the composite state and affected transitions has to be coherent in case inner transitions specify further parameters. When leaving and entering the composite state again, the values of the fixed parameters can of course be different than before.

![Figure 4.1: Example of expressing SNPs with UWE](image)

Part b of Figure 4.1 depicts this behavior: the bold transitions inherit the fixed parameter. This means the value of item is set by the transition targeting BuyEnergy. Afterwards, it cannot be changed until the OrderProcess is left. The stereotype «collection» denotes that several Offers are shown. Each offer is of the type EnergyOffer, which is defined by the tag $\{\text{itemType} \}$. If an offer is bought, the confirmation cannot be shown for another one.
For simplicity, in the rest of this deliverable we focus on modeling and enforcing SNPs on web applications without imposing constraints on the parameters. This is reasonable since important security parameters such as the session ID are already handled automatically by development frameworks.

**Relation of SNPs and RBAC.** Within UWE, modeling SNPs can easily go hand in hand with modeling RBAC, where RBAC is twofold: on the one hand we specify navigational access control, i.e., the behavior of a web application if an unauthorized user accesses a web page. On the other hand, we express access control on classes that are used in the implementation.

We also use the Navigational State Model of UWE to specify navigational access control. For each «session» stereotype, denoting a user’s session, a tag called {roles} can point to a set of roles from UWE’s Role Model. In order to be able to access the web page represented by a state, a user has to have at least one of the roles that are allowed to access this state. If this is not the case, the tag {unauthorizedAccess} specifies which state should be used instead. This state can then represent, e.g., a page with an error message or an advertisement for a more expensive account.

In order to provide a full picture of access control in UWE, we briefly introduce how to model RBAC on data objects with UWE. For the Content- and the Basic Rights Model UML class diagrams are used. In the Content Model, classes and their relationships are defined. These classes can then be reused in the Basic Rights Model, where tagged UML dependencies connect role instances to them for modeling RBAC (an example is shown for our case study in Section 4.4, Figure 4.4). These dependencies can be tagged by «create», «readAll», «updateAll» or «delete», which represents the common CRUD functions. For example, «updateAll» means that a role can update all attributes of a class. Dependencies can also directly point to an attribute («update», «read») or to a method («execute»). Generally, the Basic Rights Model is equally expressive as SecureUML [84], while the representation is less bulky, as discussed in [36].

In theory, it is possible to export both kinds of access control to XACML (eXtensible Access Control Markup Language), as described in [38]. However, up to now only the access control on data objects has been explored in practice. Our approach complements this by using a monitor on the server side, which enforces not only SNPs but also navigational access control.

### 4.3.2 Testing

In this subsection, we describe how to generate navigational test cases for web applications with access control policies regarding SNPs. The main goal is to cover every possible navigation context considering navigation history, current navigation node, user role and access permission result. Comparing these results with given access control policies, we can detect possible access control misbehavior issues.

In order to build navigational test cases we use the following approach based on an UWE SNP policy given input: for each user role we walk through every available SNP starting at the entry node which is marked as **isHome**. We navigate from node to node inside the SNP until we come back to an already visited node. This way we can test if the application behaves according to the specification (we can identify false negative access control behavior by collecting occurring access denials). In order to detect possible violations of the integrity of the policy, we simply try to leave the SNP on every node by requesting each node which is currently not accessible by an outgoing transition and thus not allowed. In this context, every granted access represents an incorrect behavior. Figure 4.2 depicts such a navigation through a SNP including illicit node requests on every node.

![Figure 4.2: Test generation: Follow SNP and request illicit nodes](image)

However, with every violating request we cause a navigation history which does not correspond to the original SNP, and we possibly harm the state of the application. Therefore, we need to reconstruct the previous SNP-valid
navigation context to go ahead: First, we have to fall back to the current entry node to clear the navigation history. Second, we have to repeat the navigation progress on the SNP until we achieve the previously visited node inside the SNP to reconstruct the navigation context.

This testing process is efficient, since it has a complexity of $O(rn^3)$. Assuming there are $n$ possible navigation nodes and $r$ user roles: Every navigation node has up to $n - 1$ neighbors which are not accessible by an outgoing transition. By testing all roles, we get an amount of $rn(n-1)$ test cases which gives an upper bound of $O(rn^2)$ tests. Considering the backtracking behavior to reconstruct the navigation context we have to visit up to $n - 1$ additional nodes for every forbidden node. Consequently, we get a final complexity of $O(rn^3)$. However, testing parameters cannot be exhaustive, as parameters can contain arbitrary values. Extending our approach to consider constraints on parameters is thus left as future work.

### 4.3.3 Tool Support

This subsection presents tool support that we developed to validate our approach. Only the tool MagicUWE existed before, which is used to model UWE diagrams. We have implemented MagicSNP to export navigational access control rules that can be enforced by our SNPmonitor. For tests, our SNPpolicyTester is employed.

**MagicUWE.** UWE models can be built using any UML CASE tool that enables the import of UML profiles. We use the MagicUWE plugin [37] implemented for MagicDraw that provides additional support for the developer so that repetitive tasks can be avoided. Thus, instead of creating a basic element, as a class, and applying a stereotype to it, UWE’s stereotyped elements can be inserted directly from a toolbar. Besides, transformations between UWE models can be performed semi-automatically.

**MagicSNP.** In order to validate a UML Navigational State Model and moreover to extract the corresponding access control semantics we developed a CASE tool plugin for MagicDraw called MagicSNP. By iterating through all hierarchical states and analyzing incoming transitions, state names and tags, our tool fetches relevant information about navigational access control and SNPs. The JSON-structured result can be taken as input for a security framework for a specific web application. In our case, the exported rules are read by the server-side monitor. An example of a result file can be found in Section 4.4.

**SNPmonitor.** The SNPmonitor is our generic monitor module approach which provides RBAC with SNPs for web applications considering modeled access control semantics. Basically, 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 a policy file (e.g., generated by MagicSNP). In order to ensure robustness, our monitor module also handles any kind of access constraint violation: the web user is redirected to a corresponding error page including an appropriate error message with possibility to go back to its previously visited navigation context.

Technically, our SNPmonitor is implemented as a Java EE application, using the Spring Framework. [9] The code of a client application which should be safeguarded does not have to be touched, the monitor just has to be added as a filter to the Java EE deployment descriptor. Using a URL-pattern, it is also possible to shield a certain part of a web application, e.g., web pages stored in a protected/* directory.

**SNPpolicyTester.** In order to test already defined SNP policies for a specific web application, as mentioned in Section 4.3.2, we developed a testing tool called SNPpolicyTester. It parses a given security policy file and searches for false positive and false negative access control behavior. Therefore, it navigates through every available SNP with every defined user role trying to leave the SNP on each navigation node by requesting every possible illegal node in this context. As a result, we get a detailed log file which allows a quick identification of traces that are possible although they should be prohibited.

Additionally, we analyzed the runtime performance of our testing tool using a benchmark client based on the TPC-W Benchmark[11]. Consequently, we are able to compare the result with the complexity of our test generation process according to Section 4.3.2: Figure 4.3 depicts the average result of benchmarks we performed with two user roles regarding ten, twenty, thirty and forty navigation nodes. The result corresponds to the expected complexity of $O(rn^3)$. 

NESSoS - 256980 26
4.4 Case Study: SmartGrid Bonus Application

Smart grids use information and communication technology (ICT) to optimize the transmission and distribution of electricity from suppliers to consumers, allowing smart generation and bidirectional power flows – depending on where generation takes place. With ICT the Smart Grid enables financial, informational, and electrical transactions among consumers, grid assets, and other authorized users [98]. The Smart Grid integrates all actors of the energy market, including the customers, into a system which supports, for instance, smart consumption in cars and the transformation of incoming power in buildings into heat, light, warm water or electricity with minimal human intervention. Smart grid represents a potentially huge market for the electronics industry [107]. Two basic reasons why the attack surface is increasing with the new technologies are: a) The Smart Grid will increase the amount of private sensitive customer data available to the utility and third-party partners and b) Introducing new data interfaces to the grid through meters, collectors, and other smart devices create new entry points for attackers. For a more detailed discussion on security issues arising in this context see D11.2 [47]. See also [69] for a current version of proposed technologies to solve this power systems management and associated information exchange issues. In the following we model a scenario in this domain, the SmartGrid Bonus Application.

Basically, our SmartGrid Bonus Application represents a prototype of an energy offer management including optional bonus handling. It provides two different user roles namely Provider and Customer: Providers manage and sell energy packages including optional bonus programs for customers. Customers have the possibility to buy energy packages. Therefore, our application lists all available energy offers and the customer selects a specific offer which includes a bonus code. After buying an energy package, the application shows the corresponding bonus code which contains a gift voucher, e.g., for online shops. Finally, the customer gets a confirmation for the ordered energy.

In order to model the data structure managed by our case study, we use UWE’s Content Model. Basically, it comprises two domain classes, EnergyOffer and BonusProgram, which are also used in Figure 4.4. An instance of the class EnergyOffer represents a specific energy offer launched by an energy provider including start and end date. Each object of EnergyOffer can include an arbitrary number of BonusProgram instances. A BonusProgram instance stands for an additional bonus customers get, after they have bought the corresponding EnergyOffer.

In order to model RBAC constraints we use UWE’s Basic Rights Model, depicted in Figure 4.4. Basically, it uses classes of the Content Model on the left-hand side in combination with user roles on the right-hand side. Access permissions were defined by stereotyped dependencies: for our application, a provider has no restricting constraints. By contrast, there is only a limited set of permissions for users taking on the role of a customer: they are only allowed to read instances of the class EnergyOffer and to call the methods buyOffer() and generateBonusCode(). These permissions represent the basis for a customer to list all available energy offers, to buy a specific offer and to eventually get a bonus code. In order to define constraints like “a customer can only get access to a bonus code after he bought an energy package” we now have to define navigational access control policies using SNPs.
Therefore, we use UWE’s Navigational State Model as described in Section 4.3.1. For our web application, the navigational structure should start at a login page. After completing the authentication successfully, customers should be redirected to an internal page where they can have a look at a list of offers. If they decide to accept an offer they have to give their consent, before a confirmation is shown. In case the energy offer was connected to a bonus program, a page containing the bonus code is displayed before the final confirmation.

Notice that for our case study we make the assumption that names of pages correspond to names of states and we do not model parameters – our monitor then just forwards given parameters, if any.

Figure 4.5 depicts our Navigational State Model which contains the following information: The outermost state SmartGridBonusApplication stands for the whole application. The tag transmissionType="cif" sets the overall type of data transmission during the session to cif, which stands for confidentiality, integrity, and freshness. Thus, the implementation should prevent eavesdropping, replaying, or altering transmitted data. As we can see, navigational nodes, represented by the states on the innermost level, are grouped by three main areas or parent states: LoginArea, ProviderArea and CustomerArea.

Every web user can access the login context node loginViaPasswordForm which is inside the LoginArea indicated by the {isHome} tag. Inner states are tagged by {unauthorizedAccess=Error} which represents the default violation node. Logged in users with the role customer can access the whole CustomerArea indicated by the inherited {roles} tag. In addition, they must follow the SNPs as defined by the transitions between the navigation states to be allowed to request a protected node. This means, e.g., to get access to showBonusCode the user has to be associated to the role customer and he must have been on buyEnergy right before. In order to get access to customerHome the user needs to have the same role but must have been on one of the nodes loginViaPasswordform, showEnergyOffers or showConfirmation right before and so on. Otherwise, the user gets redirected to the error state as defined in the tag {unauthorizedAccess}. Each user which enters the constraint violation node error gets logged out automatically as indicated by the entry event entry / logout().

SNPs for providers are modeled in an analogous manner as depicted in the lower-left corner of Figure 4.5.

Listing 4.1 shows an excerpt from the navigation rule file which is generated by our tool MagicSNP from the state machine shown in Figure 4.5.

It contains information for a generic monitor to provide navigational access control including SNPs for a web application: The default violation node is error, defined by the attribute default_violation on the outermost level. Furthermore, every single location entry holds the corresponding violation node and a list of access rules. Each rule entry represents a user role that is allowed to access the current navigation node. In addition, the attribute pre_visited specifies navigation nodes the user is allowed to come from.

This rule file can be imported in our SNPmonitor as well as in our SNPpolicyTester.
Figure 4.5: SmartGrid Bonus Application: Navigation State Model

Listing 4.1: Excerpt from extracted JSON-structured navigation rule file

4.5 Related Work

In this section we present approaches in the area of SNPs, which are related to business workflow integrity and summarize our contribution using UWE.
As already mentioned, [32] recently published a robust approach for SNPs for MVC-based web applications where policies are specified using an ad-hoc textual notation. They also tackle race-conditions and handling of multiple tabs within a browser, which is currently outside of the scope of our approach. The parameter constraint in our approach was partially inspired by their textual policy language, although our approach is mainly based on web pages, not on methods. In UWE, some problems are inherently solved, as e.g., superstates exist so that all transitions can be easily specified and there is no need to invent extra notations for the ability to change decisions later or for the availability of the back button.

In 2002, Scott et al. [108] described a system which is also based on a solution using a monitor. A textual policy specifies validation constraints, mainly for parameters and cookies, in a language called Security Policy Description Language (SPDL). This policy is then compiled to code which is executed by the monitor when a page is accessed. Additionally, Message Authentication Codes (MACs) can be added by the monitor when delivering a page so that, e.g., hidden form fields can be secured from changes at the client-side.

Halle et al. [66] define a navigation state machine with session traces with a focus on a formal model. However, the state machine is only a simple one with no further information than a sequence of states which does not include parallel states. If desired, their formal approach might be extended to describe UWE’s navigational states model, including information given by the stereotypes, tags and parallel states.

In [97] a method for secure design of business application logic is sketched. It comprises strategies such as analyzing weaknesses caused by misconfiguration of server-side components or by errors in the application logic. They suggest to test several kinds of parameters, however they do not provide tool support. Furthermore, it is recommended to define a clear design of the architecture, especially for components which update session data. The authors aim to provide a good practice which certainly can be combined with our approach.

Additionally, a tool for servlet-based web applications is provided by Felmetsger et al. [58]. The tool, called Waler, uses a composition of dynamic analysis and symbolic model checking: Regarding the dynamic analysis, it observes the normal operation of a web application in order to infer behavioral specifications that are filtered to reduce false positives. Afterwards, symbolic model checking is used to identify program paths that are likely to violate these specifications. Compared to our approach, models do not have to be created manually, which is convenient, especially for legacy applications. However, the price is that flaws in the navigation paths can only be detected with a certain possibility. SPaCiTE [34] is a tool that generates concrete attack tests based on model checking and mutation operators. It has been applied so far for testing RBAC and XSS, but not for business workflow integrity.

To the best of our knowledge, no web framework exists that provide mechanisms to specify rules for SNPs externally, i.e., without diving into the implementation of the web application. Even frameworks that focus on security, as e.g., Apache Shiro Web-Features [2], Spring Security [9] or jGuard Web [6] do not tackle the issue of SNPs.

For this work we have used UWE. Other web engineering methods do not include a navigational model and security aspects.

WebML [106] offers a so called hypertext model, but it is less fine grained than UWE’s Navigational State Model so that SNPs cannot be ensured. Furthermore, WebML includes no navigational access control. ActionGUI [27] is an approach for generating complete, but simplified, data-centric web applications from models. It provides an OCL specification of all functionalities, so that navigation is only modeled implicitly by OCL constraints. Unfortunately, it would be difficult to model SNPs with those constraints, because this would require to model the whole web application, which can be tedious. SecureUML [84] is a UML-based modeling language for secure systems. A dialect for classes (called components) provides modeling elements for RBAC which are as expressive as UWE’s Basic Rights Model. A similar approach is UACML [109] which also comes with a UML-based meta-metamodel for access control, which can be specialized into various meta-models for, e.g., RBAC or mandatory access control (MAC). Conversely to UWE, the resulting diagrams of SecureUML and UACML are overloaded, as SecureUML uses association classes instead of dependencies and UACML do not introduce a separate model to specify user-role hierarchies. UMLsec [71] provides a UML extension with emphasis on secure protocols. Similarly, SecureMDD [95] allows one to define security properties in UML diagrams and generate code focusing primarily on security protocols for smart-cards.
5 Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base

Computing devices and software are omnipresent in our society, and society increasingly relies on the correct and secure functioning of these devices and software. Two important trends can be observed. First, network connectivity of devices keeps increasing. More and more (and smaller and smaller) devices get connected to the Internet or local ad-hoc networks. Second, more and more devices support extensibility of the software they run – often even by third parties different from the device manufacturer or device owner. These two factors are important because they enable a vast array of interesting applications, ranging from over-the-air updates on smart cards, over updateable implanted medical devices to programmable sensor networks. However, these two factors also have a significant impact on security threats. The combination of connectivity and software extensibility leads to malware threats. Researchers have already shown how to perform code injection attacks against embedded devices to build self-propagating worms [63, 61]. Viega and Thompson [119] describe several recent incidents and summarize the state of embedded device security as “a mess”.

For high-end devices, such as servers or desktops, the problems of dealing with connectivity and software extensibility are relatively well-understood, and there is a rich body of knowledge built up from decades of research. However, for low-end, resource-constrained devices, no effective low-cost solutions are known. Many embedded platforms lack the standard security features (such as privilege levels or advanced memory management units that support virtual memory) present in high-end processors. Depending on the overall system security goals, as well as the context in which the system must operate, there may be more optimal solutions than just porting the general-purpose security features from high-end processors. Several recent results show that researchers are exploring this idea in a variety of settings. For instance, El Defrawy et al. propose SMART, a simple and efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor [55], and Strackx et al. propose a simple program-counter based memory access control system to isolate software components [111].

In this chapter we build on these primitives to propose a security architecture that supports secure third-party software extensibility for a network of low-end processors (the prototypical example of such a network is a sensor network). The architecture enables mutually distrusting parties to run their software modules on the same nodes in the network, while each party maintains strong assurance that its modules run untampered. This kind of secure software extensibility is very useful for applications of sensor networks (for instance in the logistics and medical domains), and fits under NESSoS Task 8.4. The main distinguishing feature of our approach is that we achieve these security guarantees without any software in the TCB on the device, and with only minimal hardware extensions.

Our attacker model assumes that an attacker has complete control over the software state of a device, and even for such attackers our security architecture ensures that any results a party receives from one of its modules can be validated to be genuine. Obviously, with such a strong attacker model, we cannot guarantee availability, so an attacker can bring the system down, but if results are received their integrity and authenticity can be verified. To guarantee the reproducibility and verifiability of our results, all our research materials, including the hardware design of the processor, and the C compiler are publicly available at https://distrinet.cs.kuleuven.be/software/sancus/. A detailed presentation of the architecture can be found in [100].

5.1 Problem statement

5.1.1 System model

We consider a setting where a single infrastructure provider, $IP$, owns and administers a (potentially large) set of microprocessor-based systems that we refer to as nodes $N_j$. A variety of third-party software providers $SP_j$ are interested in using the infrastructure provided by $IP$. They do so by deploying software modules $SM_{j,k}$ on the nodes administered by $IP$. Figure 5.1 provides an overview.

This abstract setting is an adequate model for many ICT systems today, and the nodes in such systems can range from high-performance servers (for instance in a cloud system), over smart cards (for instance in GlobalPlatform-based systems) to tiny microprocessors (for instance in sensor networks). In this chapter, we focus on the low end of this spectrum, where nodes contain only a small embedded processor.

Any system that supports extensibility (through installation of software modules) by several software providers must implement measures to make sure that the different modules cannot interfere with each other in undesired ways (either because of bugs in the software, or because of malice). For high- to mid-end systems, this problem
Figure 5.1: Overview of our system model. \( IP \) provides a number of nodes \( N \) on which software providers \( SP_j \) can deploy software modules \( SM_{j,k} \).

is relatively well-understood and good solutions exist. Two important classes of solutions are (1) the use of virtual memory, where each software module gets its own virtual address space, and where an operating system or hypervisor implements and guards communication channels between them (for instance shared memory sections or inter-process communication channels), and (2) the use of a memory-safe virtual machine (for instance a Java VM) where software modules are deployed in memory-safe bytecode and a security architecture in the VM guards the interactions between them.

For low-end systems with cheap microprocessors, providing adequate security measures for the setting sketched above is still an open problem, and an active area of research [57]. One straightforward solution is to transplant the higher-end solutions to these low-end systems: one can extend the processor with virtual memory, or one can implement a Java VM. This will be an appropriate solution in some contexts, but there are two important disadvantages. First, the cost (in terms of required resources such as chip surface, power or performance) is non-negligible. And second, these solutions all require the presence of a sizable trusted software layer (either the OS or hypervisor, or the VM implementation).

The problem we address in this chapter is the design, implementation and evaluation of a novel security architecture for low-end systems that is inexpensive and that does not rely on any trusted software layer. The TCB on the networked device is only the hardware. More precisely, a software provider needs to trust only his own software modules; he does not need to trust any infrastructural or third-party software on the nodes, only the hardware of the infrastructure and his own modules.

### 5.1.2 Attacker model

We consider attackers with two powerful capabilities.

First, we assume attackers can manipulate all the software on the nodes. In particular, attackers can act as a software provider and can deploy malicious modules to nodes. Attackers can also tamper with the operating system (for instance because they can exploit a buffer overflow vulnerability in the operating system code), or even install a completely new operating system.

Second, we assume attackers can control the communication network that is used by \( IP \), software providers and nodes to communicate with each other. Attackers can sniff the network, can modify traffic, or can mount man-in-
the-middle attacks.

With respect to the cryptographic capabilities of the attacker, we follow the Dolev-Yao attacker model [52]: attackers cannot break cryptographic primitives, but they can perform protocol-level attacks.

Finally, attacks against the hardware are out of scope. We assume the attacker does not have physical access to the hardware, cannot place probes on the memory bus, cannot disconnect components and so forth. While physical attacks are important, the addition of hardware-level protections is an orthogonal problem that is an active area of research in itself [77, 78, 30, 24]. The addition of hardware-level protection will be useful for many practical applications (in particular for sensor networks) but does not have any direct impact on our proposed architecture or on the results of this work.

5.1.3 Security properties

For the system and attacker model described above, we want our security architecture to enforce the following security properties:

- **Software module isolation.** Software modules on a node run isolated in the sense that no software outside the module can read or write its runtime state, and no software outside the module can modify the module’s code. The only way for other software on the node to interact with a module is by calling one of its designated entry points.

- **Remote attestation.** A software provider can verify with high assurance that a specific software module is loaded on a specific node of IP.

- **Secure communication.** A software provider can receive messages from a specific software module on a specific node with authenticity, integrity and freshness guarantees. For simplicity we do not consider confidentiality properties in this chapter, but our approach could be extended to also provide confidentiality guarantees.

- **Secure linking.** A software module on a node can link to and call another module on the same node with high assurance that it is calling the intended module. The runtime interactions between a module $A$ and a module $B$ that $A$ links to cannot be observed or tampered with by other software on the same node.

Obviously, these security properties are not entirely independent of each other. For instance, it does not make sense to have secure communication but no isolation: given the power of our attackers, any message could then simply be modified right after its integrity was verified by a software module.

5.2 Design of Sancus

The main design challenge is to realize the desired security properties without trusting any software on the nodes, and under the constraint that nodes are low-end resource constrained devices. An important first design choice that follows from the resource constrained nature of nodes is that we limit cryptographic techniques to symmetric key. While public key cryptography would simplify key management, the cost of implementing public key cryptography in hardware is too high [82].

We present an overview of our design, and then we zoom in on the most interesting aspects.

5.2.1 Overview

**Nodes.** Nodes are low-cost, low-power microcontrollers (our implementation is based on the TI MSP430). The processor in the nodes uses a von Neumann architecture with a single address space for instructions and data. To distinguish actual nodes belonging to IP from fake nodes set up by an attacker, IP shares a symmetric key with each of its nodes. We call this key the node master key, and use the notation $K_N$ for the node master key of node $N$. Given our attacker model where the attacker can control all software on the nodes, it follows that this key must be managed by the hardware, and it is only accessible to software in an indirect way.
Figure 5.2: Overview of the keys used in Sancus. The node key $K_N$ is only known by IP and the hardware. When SP is registered, it receives its key $K_{N,SP}$ from IP which can then be used to create module specific keys $K_{N,SP,SM}$.

Software Providers. Software providers are principals that can deploy software to the nodes of IP. Each software provider has a unique public ID $SP$. IP uses a key derivation function $kdf$ to compute a key $K_{N,SP} = kdf(K_N, SP)$, which $SP$ will later use to setup secure communication with its modules. Since node $N$ has key $K_N$, nodes can compute $K_{N,SP}$ for any $SP$. The node will include a hardware implementation of $kdf$ so that the key can be computed without trusting any software.

Software Modules. Software modules are essentially simple binary files containing two mandatory sections: a text section containing protected code and constants and a protected data section. As we will see later, the contents of the latter section are not attested and are therefore vulnerable to malicious modification before hardware protection is enabled. Therefore, the processor will zero-initialize its contents at the time the protection is enabled to ensure an attacker cannot have any influence on a module’s initial state. Next to the two protected sections discussed above, a module can opt to load a number of unprotected sections. This is useful to, for example, limit the amount of code that can access protected data. Indeed, allowing code that does not need access to protected data increases the attack surface and this increases the possibility of bugs that could be used to steal protected data. In other words, this gives developers the opportunity to keep the trusted code of their own modules as small as possible. Each section has a header that specifies the start and end address of the section.

The identity of a software module consists of (1) the content of the text section and (2) the start and end addresses of the text and protected data sections. We refer to this second part of the identity as the layout of the module. It follows that two modules with the exact same code and data can coexist on the same node and will have different identities as their layout will be different. We will use notations such as $SM$ or $SM_1$ to denote the identity of a specific software module.

Software modules are always loaded on a node on behalf of a specific software provider $SP$. The loading proceeds as expected, by loading each of the sections of the module in memory at the specified addresses. For each module loaded, the processor maintains the layout information in a protected storage area inaccessible from software. It follows that the node can compute the identity of all modules loaded on the node: the layout information is present in protected storage and the content of the text section is in memory.

An important sidenote is here that the loading process is not trusted. It is possible for an attacker to intervene and modify the module during loading. However, this will be detected as soon as the module communicates with its provider or with other modules.

Finally, the node computes a symmetric key $K_{N,SP,SM}$ that is specific to the module $SM$ loaded on node $N$ by provider $SP$. It does so by first computing $K_{N,SP} = kdf(K_N, SP)$ as discussed above, and then computing $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$. All these keys are kept in the protected storage and will only be available to software indirectly by means of new processor instructions we discuss later. Section 5.2 gives an overview of the keys used by Sancus.

Note that the provider $SP$ can also compute the same key, since he received $K_{N,SP}$ from IP and since he knows the identity $SM$ of the module he is loading on $N$. This key will be used to attest the presence of $SM$ on $N$ to $SP$ and to protect the integrity of data sent from $SM$ on $N$ to $SP$.

Memory protection on the nodes. The various modules on a node must be protected from interfering with each other in undesired ways by means of some form of memory protection. We base our design on the recently proposed program-counter based memory access control [111], as this memory access control model has been shown to support strong isolation [110] as well as remote attestation [55]. Roughly speaking, isolation is implemented by

$K_N = \text{Known by IP}$

$K_{N,SP} = kdf(K_N, SP)$

$K_{N,SP,SM} = kdf(K_{N,SP}, SM)$
restricting access to the protected data section of a module such that it is only accessible while the program counter is in the corresponding text section of the same module. Moreover, the processor instructions that use the keys $K_{N,SP,SM}$ will be program counter dependent. Essentially the processor offers a special instruction to compute a MAC. If the instruction is invoked from within the text section of a specific module $SM$, the processor will use key $K_{N,SP,SM}$ to compute the MAC. Moreover, the instruction is only available after memory protection for module $SM$ has been enabled. It follows that only a well-isolated $SM$ installed on behalf of $SP$ on $N$ can compute MACs with $K_{N,SP,SM}$, and this is the basis for implementing both remote attestation and secure (integrity-protected) communication to $SP$. 

Secure linking. A final aspect of our design is how we deal with secure linking. When a software provider sends a module $SM_1$ to a node, this module can specify that it wants to link to another module $SM_2$ on the same node, so that $SM_1$ can call services of $SM_2$ locally. $SM_1$ specifies this by including a MAC of (the identity of) $SM_2$ computed using the key $K_{N,SP,SM_1}$ in an unprotected section. The processor includes a new special instruction that $SM_1$ can call to check that (1) there is a module loaded (with memory protection enabled) at the address of $SM_2$ and (2) the MAC of the identity of that module has the expected value. This initial authentication of $SM_2$ is needed only once.

We currently do not incorporate caller authentication in our design. That is, $SM_2$ can not easily verify that it has been called by $SM_1$. Note that this can in principle be implemented in software: $SM_1$ can call $SM_2$ providing a secret nonce as parameter. $SM_2$ can then call-back $SM_1$, passing the same nonce, asking for acknowledgement that it had indeed been called by $SM_1$. Future work will include caller authentication in the core of Sancus’ design to make it more efficient and transparent.

Separating the various uses of MACs. Sancus uses MACs for a variety of integrity checks as well as for key derivation. Our design includes a countermeasure to avoid attacks where an attacker replays a MAC computed on data. In order to achieve separation between the different applications of MAC functions, we make sure the first byte of an input to the MAC function is different for each use case: 01 for the derivation of $K_{N,SP}$, 02 for the derivation of $K_{N,SP,SM}$, 03 for attestation and 04 for MAC computations on data.

Confidentiality. As mentioned in Section 5.1.3, we decided to not include confidentiality of communication in our design. However, since we provide attestation of modules and authentication of messages, confidentiality can be implemented in software if necessary. One possibility is deploying a module with the public key of $SP$ and a software implementation of the necessary cryptographic primitives. Another possibility is establishing a shared secret after deployment using a method such as Diffie-Hellman key exchange with authenticated messages. Note that implementing this last method is non-trivial due to the lack of a secure source of randomness. However, in the context of wireless sensor networks, methods have been devised to create cryptographically secure random number generators using only commonly available hardware [60].

Since the methods outlined above are expensive in terms of computation time and increase the TCB of modules, we are currently considering adding confidentiality to the core of Sancus’ design. Exploring this is left as future work.

This completes the overview of our design. The interested reader can find the details of key management, memory access control, secure communication, remote attestation and secure linking in [100].

5.3 Evaluation

The architecture that was proposed in the previous section has been implemented and tested. In this section, we present the evaluation of the prototype, running on a Xilinx XC6SLX9 Spartan-6 FPGA (running at 20MHz). Details of the implementation can be found in [100].

Performance A first important observation from the point of view of performance is that our hardware modifications do not impact the processor’s critical path. Hence, the processor can keep operating at the same frequency, and any code that does not use our new instructions runs at the same speed. This is true independent of the number of software modules $N_{SM}$ supported in the processor. The performance results below are also independent of $N_{SM}$.

\footnote{Note that since this MAC depends on the load addresses of $SM_1$ and $SM_2$, it may not be known until $SM_1$ has been deployed. If this is the case, $SP$ can simply send the MAC after $SM_1$ is deployed and the load addresses are known.}

\footnote{We verified this experimentally for values of $N_{SM}$ up to 8.}
Table 5.1: The number of cycles needed by the new instructions for various input sizes. The input for the instructions is as follows: protect: the text section of the software module being protected; MAC-seal: the data to be signed; and MAC-verify: the text section of the software module to be verified.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>256B</th>
<th>512B</th>
<th>1024B</th>
</tr>
</thead>
<tbody>
<tr>
<td>protect</td>
<td>30,344</td>
<td>48,904</td>
<td>86,016</td>
</tr>
<tr>
<td>MAC-seal</td>
<td>24,284</td>
<td>42,848</td>
<td>79,968</td>
</tr>
<tr>
<td>MAC-verify</td>
<td>24,852</td>
<td>43,416</td>
<td>80,536</td>
</tr>
</tbody>
</table>

Figure 5.3: Relative overhead, in function of the number of cycles used for calculations, of Sancus on the macro benchmark. The nth run is significantly faster due to the secure linking optimization discussed in [100].

To quantify the impact on performance of our extensions, we first performed microbenchmarks to measure the cost of each of the new instructions. The get-id and unprotect instructions are very fast: they both take one clock cycle. The other three instructions compute hashes or key derivations, and hence their run time cost depends linearly on the size of the input they handle. We summarize their cost in Section 5.1. Note that since MAC-seal and MAC-verify both compute the HMAC of the input data, one might expect that they would need the same number of cycles. However, since MAC-verify includes the layout of the module to be verified in the input to HMAC, it has a fixed overhead of 568 cycles.

To give an indication of the impact on performance in real-world scenarios, we performed the following macro benchmark. We measured the time it takes from the moment a request arrives at the node until the response is ready to be sent back. More specifically, the following operations are timed:

1. The original request is passed, together with the nonce, to $SM_i$;
2. $SM_i$ requests $SM_S$ for sensor data;
3. $SM_i$ performs some transformation on the received data; and
4. $SM_i$ signs its output together with the nonce.

The overhead introduced by Sancus is due to a call to MAC-verify in step 2 and a call to MAC-seal in step 4 as well as the entry and exit code introduced by the compiler. Since this overhead is fixed, the amount of computation performed in step 3 will influence the relative overhead of Sancus. Note that the size of the text section of $M_S$ is 218 bytes and that nonces and output data signed by $M_i$ both have a size of 16 bits.

By using the Timer_A module of the MSP430, we measured the fixed overhead to be 28,420 cycles for the first time data is requested from the module. Since the call to MAC-verify in step 2 is not needed after the initial verification, we also measured the overhead of any subsequent requests, which is 6,341 cycles. Given these values, the relative overhead can be calculated in function of the number of cycles used during the computation in step 3. The result is shown in Section 5.3.

We believe that these numbers are clear evidence of the practicality of our approach.
Area  The unmodified Spartan-6 FPGA implementation of the openMSP430 uses 998 slice registers and 2,322 slice LUTs. The fixed overhead\(^4\) of our modification is 586 registers and 1,138 LUTs. For each protected module, there is an additional overhead of 213 registers and 307 LUTs.

There are two easy ways to improve these numbers. First, if computational overhead is of lesser concern, the module key may be calculated on the fly instead of storing it in registers. Second, in applications with lower security requirements, smaller keys may be used reducing the number of registers used for storage as well as the internal state of the SPONGENT implementation. Exploring other improvements is left as future work.

Power  Our static power analysis tool\(^5\) predicts an increase of power consumption for the processor of around 6\% for the processor running at 20MHz. We measured power consumption experimentally, but could not detect a significant difference between an unmodified processor and our Sancus prototype. Of course, since Sancus introduces a runtime overhead, the total overhead in energy consumption will be accordingly.

Security  We provide an informal security argument for each of the security properties Sancus aims for (see Section 5.1.3).

First, software module isolation is enforced by the memory access control logic in the processor. Both the access control model as well as its implementation are sufficiently simple to have a high assurance in the correctness of the implementation. Moreover, Agten et al. [21] have shown that higher-level isolation properties (similar to isolation between Java components) can be achieved by compiling to a processor with program-counter dependent memory access control. Sancus does not protect against vulnerabilities in the implementation of a module. If a module contains buffer-overflows or other memory safety related vulnerabilities, attackers can exploit them using well-known techniques [56] to get unintended access to data or functionality in the module. Dealing with such vulnerabilities is an orthogonal problem, and a wide range of countermeasures for addressing them has been developed over the past decades [122].

The security of remote attestation and secure communication both follow from the following key observation: the computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key (and of course by the software provider of the module). As a consequence, if an attacker succeeds in completing a successful attestation or communication with the software provider, he must have done it with the help of the actual module. In other words, within our attacker model, only API-level attacks against the module are possible, and it is indeed possible to develop modules that are vulnerable to such attacks, for instance if a module offers a function to compute MACs with its module key on arbitrary input data. But if the module developer avoids such API-level attacks, the security of Sancus against attackers conforming to our attacker model follows.

The security of secure linking is the most intricate security property of Sancus. It follows again from the fact that computation of MACs with the module key is only possible by a module with the correct identity running on top of a processor configured with the correct node key, or by the software provider of the module. Hence, an attacker can not forge MACs of other modules that a module wants to link to and call. Because of our technique for separation of uses of MACs (Section 5.2.1), he can also not do this by means of an API level attack against the module. As a consequence, if a module implements a MAC-verify check for any module it calls\(^6\), this verification can only be successful for modules for which the software provider has deployed the MAC. Hence the module will only call modules that its provider has authorized it to call.

\(^4\)That is, the overhead when \(N_{SM} = 0\).
\(^5\)We used Xilinx XPower Analyzer.
\(^6\)Note that our compiler automatically adds these checks.
6 CacheAudit: A Tool for the Static Analysis of Cache Side Channels

Side-channel attacks recover secret inputs to programs from non-functional characteristics of computations, such as time [75], power [76], or memory consumption [70]. Typical goals of side-channel attacks are the recovery of cryptographic keys and private information about users. Side-channel attacks (in particular those based on timing) can be carried out remotely and across virtual machines, which severely undermines the security of cloud-based services and platforms [33, 105, 127]. Developing defenses against this kind of attacks is hence a priority for the NESSoS project.

Processor caches are a particularly rich source of side-channels because their behavior can be monitored in various ways, which is demonstrated by three documented classes of side-channel attacks: (1) In time-based attacks [75, 29] the adversary monitors the overall execution time of a victim, which is correlated with the number of cache hits and misses during execution. Time-based attacks are especially daunting because they can be carried out remotely over the network [17]. (2) In access-based attacks [102, 101, 64] the adversary probes the cache state by timing its own accesses to memory. Access-based attacks require that attacker and victim share the same hardware platform, which is common in the cloud and has already been exploited [105, 127]. (3) In trace-based attacks [16] the adversary monitors the sequence of cache hits and misses. This can be achieved, e.g., by monitoring the CPU’s power consumption and is particularly relevant for embedded systems.

A number of proposals have been made for countering cache-based side-channel attacks. Some proposals focus entirely on modifications of the hardware platform; they either solve the problem for specific algorithms such as AES [5] or require modifications to the platform [120] that are so significant that their rapid adoption seems unlikely. The bulk of proposals relies on controlling the interactions between the software and the hardware layers, either through the operating system [64, 126], the client application [29, 101, 43], or both [73]. Reasoning about these interactions can be tricky and error-prone because it relies on the specifics of the binary code and the microarchitecture.

In this chapter we present CacheAudit, a tool for the static exploration of the interactions of a program with the cache. CacheAudit takes as input a program binary and a cache configuration and delivers formal security guarantees that cover all possible executions of the corresponding system. The security guarantees are quantitative upper bounds on the amount of information that is contained in the side-channel observations of timing-, access-, and trace-based adversaries, respectively. CacheAudit can be used to formally analyze the effect on the leakage of software countermeasures and cache configurations, such as preloading of tables or increasing the cache’s line size. The design of CacheAudit is modular and facilitates the extension with any cache model for which efficient abstractions are in place. The current implementation of CacheAudit supports caches with LRU, FIFO, and PLRU replacement strategies.

We demonstrate the scope of CacheAudit in case studies where we analyze the side-channel leakage of representative algorithms for symmetric encryption and sorting. We highlight the following two results: (1) For the reference implementation of the Salsa20 [28] stream cipher (which was designed to be resilient to cache side-channel attacks) CacheAudit can formally prove non-leakage on the basis of the binary executable, for all adversary models and replacement strategies. (2) For a library implementation of AES 128 [7]. CacheAudit confirms that the preloading of tables significantly improves the security of the executable: for most adversary models and replacement strategies, we can in fact prove non-leakage of the executable, whenever the tables fit entirely into the cache. However, for access-based adversaries and LRU caches, CacheAudit reports small, non-zero bounds. And indeed, with LRU (as opposed to, e.g., FIFO), the ordering of blocks within a cache set reveals information about the victim’s final memory accesses.

On a technical level, we build on the fact that the amount of leaked information corresponds to the number of possible side-channel observations, which can be over-approximated by abstract interpretation [44] and model counting [80]. To realize CacheAudit based on this insight, we propose three novel abstract domains (i.e., data structures that facilitate the sound approximation of program semantics) that keep track of the observations of access-based, time-based, and trace-based adversaries, respectively. In particular:

1. We propose an abstract domain that tracks relational information about the memory blocks that may be cached. Opposed to existing abstract domains used in worst-case execution time analysis [59], our novel domain can retain analysis precision in the presence of array accesses to unknown positions.

2. We propose an abstract domain that tracks the traces of cache hits and misses that may occur during execution. We use a technique based on prefix trees and hash consing to compactly represent such sets of traces, and to count
their number.

3. We propose an abstract domain that tracks the possible execution times of a program. This domain captures timing variations due to control flow and caches by associating hits and misses with their respective latencies and adding the execution time of the respective commands. We formalize the connection of these domains in an abstract interpretation [44] framework that captures the relationship between microarchitectural state and program code. We use this framework to formally prove the correctness of the derived upper bounds on the leakage to the corresponding side-channel adversaries.

In summary, our main contributions are both theoretical and practical: On a theoretical level, we define novel abstract domains that are suitable for the analysis of cache side channels, for a comprehensive set of adversaries. On a practical level, we build CacheAudit, the first tool for the automatic, quantitative information-flow analysis of cache side-channels, and we show how it can be used to derive formal security guarantees from binary executables of sorting algorithms and state-of-the-art cryptosystems.

6.1 Illustrative Example

In this section, we illustrate on a simple example program the kind of guarantees CacheAudit can derive. Namely, we consider an implementation of BubbleSort that receives its input in an array \( a \) of length \( n \). We assume that the contents of \( a \) are secret and we aim to deduce how much information a cache side-channel adversary can learn about the relative ordering of the elements of \( a \).

```c
void BubbleSort(int a[], int n)
{
    int i, j, temp;
    for (i = 0; i < (n - 1); ++i)
        for (j = 0; j < n - 1 - i; ++j)
            if (a[j] > a[j+1])
                {
                    temp = a[j+1];
                    a[j+1] = a[j];
                    a[j] = temp;
                }
}
```

To begin with, observe that the conditional swap in lines 6–11 is executed exactly \( \frac{n(n-1)}{2} \) times. A trace-based adversary that can observe, for each instruction, whether it corresponds to a cache hit or a miss is likely to be able to distinguish between the two alternative paths in the conditional swap, hence we expect this adversary to be able to distinguish between \( 2^{\frac{n(n-1)}{2}} \) execution traces. A timing-based adversary who can observe the overall execution time is likely to be able to distinguish between \( \frac{n(n-1)}{2} + 1 \) possible execution times, corresponding to the number of times the swap has been carried out. For an access-based adversary who can probe the final cache state upon termination, the situation is more subtle: evaluating the guard in line 6 requires accessing both \( a[j] \) and \( a[j+1] \), which implies that both will be present in the cache when the swap in lines 8–10 is carried out. Assuming we begin with an empty cache, we expect that there is only one possible final cache state.

CacheAudit enables us to perform such analyses (for a particular \( n \)) formally and automatically, based on actual x86 binary executables and different cache types. CacheAudit achieves this by tracking compact representations of supersets of possible cache states and traces of hits and misses, and by counting the corresponding number of elements. For the above example, CacheAudit was able to precisely confirm the intuitive bounds, for a random selection of \( n \) in \( \{2, \ldots, 64\} \).

In terms of security, the number of possible observations corresponds to the factor by which the cache observation increases the probability of correctly guessing the secret ordering of inputs. Hence, for \( n = 32 \) and a uniform distribution on this order (i.e. an initial probability of \( \frac{1}{32!} = 3.8 \cdot 10^{-36} \)), the bounds derived by CacheAudit imply that the probability of determining the correct input order from the side-channel observation is \( 1 \) for a trace-based adversary, \( 3.7 \cdot 10^{-33} \) for a time-based adversary, and remains unchanged for an access-based adversary.
6.2 Tool Design and Implementation

In this section we describe the architecture and implementation of CacheAudit.

We take advantage of the compositionality of our framework (described in detail in [53]) and use a generic iterator module to over-approximate the program semantics, where we rely on independent modules for the abstract domains that correspond to the components of the $next^8$ operation. Figure 6.1 depicts the overall architecture of CacheAudit, with the individual modules described below. CacheAudit is implemented in currently about 7.5 KLOC of OCaml [10], which we plan to make publicly available.

![Figure 6.1: The architecture of CacheAudit. The solid boxes represent modules. Black-headed arrows mean that the module at the head is an argument of the module at the tail. White-headed arrows represent is-a relationships.](image)

6.2.1 Control Flow Reconstruction

The first stage of the analysis is similar to a compiler front end. The main challenge is that we directly analyze x86 executables with no explicit control flow graph, which we need for guiding the abstract interpretation.

For the parsing phase, we rely on Chlipala's parser for x86 executables [40], which we extended to a set of instructions that is sufficient for our case studies (but not yet complete). For the control-flow reconstruction, we consider only programs without dynamically computed jump and call targets, which is why it suffices to identify the basic blocks and link them according to the corresponding branching conditions and (static) branch targets. We plan to integrate more sophisticated techniques for control-flow reconstruction [74] in the future.

6.2.2 Iterator

The iterator module is responsible for the computation of the $next^8$ operator using adequate iteration strategies [45]. Our analysis uses an iterative strategy, i.e., it stabilizes components of the abstract control flow graph according to a weak topological ordering, which we compute using Bourdoncle’s algorithm [31].

The iterator also implements parts of the reduced cardinal power, based on the labels computed according to the control-flow graph: Each label is associated with an initial abstract state. The analysis computes the effect of the commands executed from that label to its successors on the initial abstract state, and propagates the resulting final states using the abstract domains described below. In order to increase precision, we expand locations using loop unfolding, so that we have a number of different initial and final abstract states for each label inside loops, depending on a parameter describing the number of loop unfoldings we want to perform. Most of our examples (such as the cryptographic algorithms) require only a small, constant number of loop iterations, so that we can choose unfolding parameters that avoid joining states stemming from different iterations.
6.2.3 Abstract Domains

We decompose the abstract domain used by the iterator into mostly independent abstract domains describing different aspects of the concrete semantics.

Value Abstract Domains A value abstract domain represents sets of mapping from variables to (integer) values. Value abstract domains are used by the cache abstract domain to represent ages of blocks in the cache, and by the flag abstract domain to represent values stored at the addresses used in the program. We implemented different value abstract domains, such as the interval domain, an exact finite sets domain (where the sets become intervals when they are growing too large) [94] and a relational set domain.

Flag Abstract Domain In x86 binaries, there are no high level guards: instead, most operations modify flags which are then queried in conditional branches. In order to deal precisely with such branches, we need to record relational informations between the values of variables and the values of these flags. To that purpose, for each operation that modifies the flags, we compute an over-approximation of the values of the arguments that may lead to a particular flag combination. The flag abstract domain represents an abstract state as a mapping from values of flags to an element of the value abstract domain. When the analysis reaches a conditional branch, it can identify which combination of flag values corresponds to the branch and select the appropriate abstract values.

Memory Abstract Domain The memory abstract domain associates memory addresses and registers with variables and translates machine instructions into the corresponding operations on those variables, which are represented using flag abstract domains as described above. One important aspect for efficiency is that variables corresponding to addresses are created dynamically during the analysis whenever they are needed. The memory abstract domain further records all accesses to main memory using a cache abstract domain, as described below.

Stack Abstract Domain Operations on the stack are handled by a dedicated stack abstract domain. In this way the memory abstract domain does not have to deal with stack operations such as procedure calls, for which special techniques can be implemented to achieve precise interprocedural analysis.

Cache Abstract Domain Cache abstract domains only keep information about the cache, using a representation of sets of mappings from blocks to ages in the cache. This is implemented using an instance of value abstract domains. Effects from the memory domain are passed to the cache abstract domain at the memory abstract domain level, so that the cache domain knows which addresses are touched during computation. The cache abstract domain passes information about the presence or absence of cache hits and misses to the trace abstract domain. The timings are then obtained as an abstraction from the traces.

6.3 Case Studies

In this section we demonstrate the capabilities of CacheAudit in case studies where we use it to analyze the cache side channels of algorithms for sorting and symmetric encryption. All results are based on the automatic analysis of corresponding 32-bit x86 Linux executables that we compiled using gcc with disabled stack canaries and without any compiler optimizations.

6.3.1 AES 128

We analyze the AES implementation from the PolarSSL library [7] with keys of 128 bits, where we consider the implementation with and without preloading of tables, for all attacker models, different replacement strategies, associativities, and line sizes. All results are presented as upper bounds of the leakage in bits; for their interpretation see [53]. In some cases, CacheAudit reports upper bounds that exceed the key size (128 bits), which corresponds to an imprecision of the static analysis. We opted against truncating to 128 bits to illustrate the degree of imprecision. The full data of our analysis are given in [33]. Here, we highlight some of our findings.

• Preloading almost consistently leads to better security guarantees in all scenarios, see e.g. Figure 6.3. However, the effect becomes clearly more apparent for cache sizes beyond 8KB, which is explained by the PolarSSL AES tables exceeding the size of the 4KB cache by 256B. For cache sizes that are larger than the preloaded tables, we can
prove noninterference for $C_{\text{acc}}$ and FIFO, $C_{\text{accd}}$ and LRU, and for $C_{\text{tr}}$ and $C_{\text{time}}$ on LRU, FIFO, and PLRU. For $C_{\text{acc}}$ with shared memory spaces and LRU, this result does not hold because the adversary can obtain information about the order of memory blocks in the cache.

- A larger line size consistently leads to better security guarantees for access-based adversaries, see e.g. Figure 6.4. This follows because more array indices map to a line which decreases the resolution of the attacker’s observations.

- In terms of replacement strategies, we consistently derive the lowest bounds for LRU, followed by PLRU and FIFO, where the only exception is the case of $C_{\text{accd}}$ and preloading (see Figure 6.2). In this case FIFO is more secure because with LRU the adversary can obtain information about the ordering of memory blocks in the cache.

- In terms of cache size, we consistently derive better bounds for larger caches, with the exception of $C_{\text{accd}}$. For this adversary model the bounds increase because larger caches correspond to distributing the table to more sets, which increases its possibilities to observe variations. The guarantees we obtain for $C_{\text{accd}}$ and $C_{\text{acc}}$ converge for caches of 4 ways and sizes beyond 16KB, see e.g. Figure 6.4. This is due to the fact that each cache set can contain at most one unique block of the 4KB table. In that way, the ability of $C_{\text{acc}}$ to observe ordering of blocks within a set does not give it power.

- In terms of precision, set-based analyses consistently match or improve over the bounds delivered by interval-based analyses. Notice that the improvement is given in bits, i.e. on a logarithmic scale.

The analysis time for the examples was typically below 60s and peaked at 365s for AES without preloading and 4KB cache.

Comparison to [79]: The PolarSSL AES implementation has already been analyzed in [79] with respect to access-based adversaries and LRU replacement. The results obtained by CacheAudit go beyond that analysis in that we derive bounds w.r.t. access-based, trace-based, and time-based adversaries, for LRU, FIFO, and PLRU strategies. For access-based adversaries and LRU, the bounds we derive are lower than those in [79]; in particular, for $C_{\text{accd}}$ we derive bounds of zero for implementations with preloading for all caches sizes that are larger than the AES tables—which is obtained in [79] only for caches of 128KB. While these results are obtained for different platforms (x86 vs ARM) and are hence not directly comparable, they do suggest a significant increase in precision. In contrast to [79], this is achieved without any code instrumentation.
Figure 6.4: Effect of cache line size on the security guarantee, for $C^{acc}$ and $C^{accd}$, and LRU replacement strategy without preloading. Different curves correspond to different cache line sizes. The horizontal axis gives the cache size, and the vertical axis gives the leakage bits.

6.3.2 Salsa20

Salsa20 is a stream cipher by Bernstein [28]. Internally, the cipher uses XOR, 32-bit addition mod $2^{32}$, and constant-distance rotation operation on an internal state of 16 32-bit words. The lack of key-dependent memory lookups intends to avoid cache side channels in software implementations of the cipher. With CacheAudit we could formally confirm this intuition by automated analysis of the reference implementation of the Salsa20 encryption, which includes a function call to a hash function. Specifically, we analyze the leakage of the encryption operation on an arbitrary 512-byte message for $C^{acc}$, $C^{ctr}$, and $C^{time}$, FIFO and LRU strategies, on 4KB caches with line size of 32B, where we consistently obtain upper bounds of 0 for the leakage. The time required for analyzing each of the cases was below 11s.

6.3.3 Sorting Algorithms

In this section we use CacheAudit to establish bounds on the cache side channels of different sorting algorithms. This case study is inspired by an early investigation of secure sorting algorithms [19]. While the authors of [19] consider only time-based adversaries and noninterference as a security property, CacheAudit allows us to give quantitative answers for a comprehensive set of side-channel adversaries, based on the binary executables and concrete cache models.

As examples, we use the implementations of BubbleSort, InsertionSort, and SelectionSort from [8], which are given in Section 6.1 and [53], respectively, where we use integer arrays of lengths from 8 to 64.

The results of our analysis are summarized in Figure 6.5. In the following we highlight some of our findings.

- We obtain the same bounds for BubbleSort and SelectionSort, which is explained by the similar structure of their control flow. A detailed explanation of those bounds is given in Section 6.1. InsertionSort has a different control flow structure, which is reflected by our data. In particular InsertionSort has only $n!$ possible execution traces due to premature abortion of the inner loop, which leads to better bounds w.r.t. trace-based adversaries. However, InsertionSort leaks more information to timing-based adversaries, because the number of iterations in the inner loop varies and thus fewer executions have the same timing.

- For access-based adversaries we obtain zero bounds for all algorithms. For trace-based adversaries, the derived bounds do not imply meaningful security guarantees: the bounds reported for InsertionSort are in the order of $\log_2(n!)$, which corresponds to the maximum information contained in the ordering of the elements; the bounds reported for the other sorting algorithm exceed this maximum, which is caused by the imprecision of the static analysis.

- We performed an analysis of the sorting algorithms for smaller (256B) and larger (64KB) cache sizes and obtained the exact same bounds as in Figure 6.5, with the exception of the case of arrays of 64 entries and 256B caches: there the leakage increases because the arrays do not fit entirely into the cache due to their misalignment with the memory blocks.

6.3.4 Discussion and Outlook

A number of comments are in order when interpreting the bounds delivered by CacheAudit. First, we obtained all of the bounds for an empty initial cache. For access-based adversaries they immediately extend to bounds for arbitrary initial cache states, as long as the victim does not access any block that is contained in it. This is relevant, e.g. for
an adversary who can fill the initial cache state only with lines from its own disjoint memory space. For LRU, our bounds extend to arbitrary initial cache states without further restriction.

Second, while CacheAudit relies on more accurate models of cache and timing than any information-flow analysis we are aware of, there are several timing-relevant features of hardware it does not capture (and make assertions about) at this point yet, including pipelines, TLBs, and multiple levels of caches.

Third, for the case of AES and Salsa20, the derived bounds hold for the leakage about the key in one execution, with respect to any payload. For the case of zero leakage (i.e., noninterference), the bounds trivially extend to bounds for multiple executions and imply strong security guarantees. For the case of non-zero leakage, the bounds can add up when repeatedly running the victim process with a fixed key and varying payload, leading to a decrease in security guarantees. One of our prime targets for future work is to derive security guarantees that hold for multiple executions of the victim process. One possibility is to employ leakage-resilient cryptosystems [54, 123], where our work can be used to bound the range of the leakage functions.

6.4 Related Work

The work most closely related to ours is [79]. There, the authors quantify cache side channels by connecting a commercial, closed-source tool for the static analysis of worst-case execution times [1] to an algorithm for counting concretizations of abstract cache states. The application of the tool to side-channel analysis is limited to access-based adversaries and requires heavy code instrumentation. In contrast, CacheAudit provides tailored abstract domains for all kinds of cache side-channel adversaries, different replacement strategies, and is modular and open for further extensions. Furthermore, the bounds delivered by CacheAudit are significantly tighter than those reported in [79], see Section 6.3.

Zhang et al. [126] propose an approach for mitigating timing side channels that is based on contracts betweens software and hardware. The contract is enforced on the software side using a type system, and on the hardware side, e.g., by using dedicated hardware such as partitioned caches. The analysis ensures that an adversary cannot obtain any information by observing public parts of the memory; any confidential information the adversary obtains must be via timing, which is controlled using dedicated mitigate commands. Tiwari et al. [115] sketch a novel microarchitecture that facilitates information-flow tracking by design, where they use noninterference as a baseline confidentiality property. Other mitigation techniques include coding guidelines [43] for thwarting cache attacks on x86 CPUs, or novel cache architectures that are resistant to cache side-channel attacks [120]. The goal of our approach is orthogonal to those approaches in that we focus on the analysis of microarchitectural side channels rather than on their mitigation. Our approach does not rely on a specific platform; rather it can be applied to any language and hardware architecture, for which abstractions are in place.

Kim et al. put forward StealthMem [73], a system-level defense against cache-timing attacks in virtualized environments. The core of StealthMem is a software-based mechanism that locks pages of a virtual machine into the cache and avoids that they are evicted by other VMs. StealthMem can be seen as a lightweight variant of flushing/preloading countermeasures. As future work, we plan to use our tool to derive formal, quantitative guarantees for programs using StealthMem.

For the case of AES, there are efficient software implementations that avoid the use of data caches by bitslicing [72]. 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 [114]. In contrast, our analysis technique applies to arbitrary programs.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [41], where the automation by reduction to counting problems appears in [26, 99, 68, 89], and the connection to abstract interpretation
in [80]. Finally, our work goes beyond language-based approaches that consider caching [18, 67] in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.
7 Privacy and Access Control in Federated Social Networks

7.1 Motivation

Online social networks (OSNs) are increasingly turning mobile and further calling for decentralized social data management. This trend is only going to increase in the near future, based on the increased activity, both by established players like Facebook and new players in the domain such as Google, Instagram, and Pinterest. Modern smart phones can thus be regarded as social sensors, collecting data not only passively using, e.g., Bluetooth neighborhoods, but actively in the form of, e.g., "check-in"s by users to locations. The resulting (mobile) social ecosystems are thus an emergent area of interest.

The recent years have seen three major trends in the world of online social networks: i) users have begun to care more about the privacy of their data stored by large OSNs such as Facebook, and have won the right (at least in the EU) to remove it completely from the OSN if they want to; ii) OSNs are making their presence felt beyond casual, personal interactions to corporate, professional ones as well, starting with LinkedIn, and most recently with the purchase by Microsoft of Yammer, the enterprise social networking startup, and the launch of Google Plus for enterprise customers; and iii) users are increasingly using the capabilities of their (multiple) mobile devices to enrich their social interactions, ranging from posting cellphone-camera photos on Instagram to “checking-in” to a GPS location using foursquare.

In view of the above, we envision that in the near future, the use of ICT to enrich our social interactions will grow (including both personal and professional interactions2), both in terms of size and complexity. However current OSNs act like data silos, storing and analyzing their users’ data, while locking in those very users to their servers, with non-existent support for federation; this is reminiscent of the early days of email, where one could only email those who had accounts on the same Unix machine. The knee-jerk reaction to this has been to explore completely decentralized social networks such as Diaspora [51], which give the user complete control over and responsibility of their social data, while resorting to peer-to-peer communication protocols to navigate their social networks; there are few techniques available to reconcile with the fact that the same user might have multiple devices, or that it is extremely resource-consuming to perform complex analysis of social graphs on small mobile devices.

Our view lies somewhere in the middle of the two extremes, taking inspiration from the manner in which users currently use email. While their inboxes contain an immense amount of extremely personal data, most users are happy to entrust it to corporate or personal email providers (or store and manage it individually on their personal email servers) all the while being able to communicate with users on any other email server. The notion of Federated Social Networks (FSNs) — already gaining some traction3 — envisions a similar ecosystem where users are free to choose OSN providers which will provide storage and management of their social information, while allowing customers using different OSN providers to interact socially. Such a federation can be beneficial in three major ways, among others: i) it allows users to enjoy properties such as reliability, availability, and computational power of the hosting infrastructure of their choice, while not being locked down in terms of whom they can communicate with; ii) much like spam filtering services provided by modern email providers, that are tuned by feedback from their users, FSN users can benefit from the behavior of others sharing the same OSN provider; and iii) this fits perfectly with enterprise needs, where ad-hoc teams can be formed across corporate OSN providers of two organizations to work on a joint project.

7.2 Challenges

As part of our research in the past year, we have identified the following new and interesting challenges that will be encountered in order to enable the federated social networks of the future:

• **Foundational theory of social data and interactions.** In order to fully realize the potential of these large networks, a foundational theory of social data and interactions is needed; something that is sorely missing at

---

3This also gives an incentive to commercial OSN providers to provide value-added services.
the moment. The data models of most OSNs are fixed (app developers are restricted to using the data model of their host OSN, without the ability to semantically extend concepts), and sometimes not realistic (e.g., only symmetric “friend” links were possible until recently on Facebook). Similarly, while current approaches have tried to model all social interactions in terms of classical interaction paradigms such as publish/subscribe, they cannot directly represent, for example, the rich discussions that take place when a user posts a status message on his “wall”. This lack of standard abstractions renders application development for and interoperability among OSNs extremely difficult, thus being a key hurdle to federation.

- **Ensuring privacy and access control.** Expressive, flexible, yet easy-to-specify access control policies coupled with adequate security and trust mechanisms are needed so that users can feel safe as more and more of their (social) life is made available online, thanks in large part to mobile clients for OSNs. As we have discovered in the course of our recent work, most current policy and trust frameworks are unable to adequately address the complexity of social networks, resorting to simple role-based access control. The need of the hour is a policy and trust framework founded on the clear data and interaction models discussed earlier. Such a framework will also allow OSN providers to adequately evaluate whether or not a certain data of a user should be shared with another OSN provider (e.g., replies to a Facebook wall post have the privacy settings of the original post, while replies to a status message on Twitter have those of the user posting the reply, thus rendering them incompatible).

- **Supporting multiple devices for each user.** Most OSN users today access their social networks using more than one (mobile) device (e.g., smart phone, tablet, laptop). While today these devices serve as little more than thin clients to the (centralized) OSN, there is an immense potential for the ensemble of these devices to operate as a user’s personal cloud, smartly replicating social data among them as needed (e.g., a user’s smart phone can automatically pre-fetch vacation photos either from the user’s OSN provider or his home desktop before leaving for a party), while also increasingly the dependability of the social networking infrastructure. A significant amount of work needs to be done to effectively integrate these disparate mobile devices connected to different networks while maintaining properties such as data consistency and responsiveness.

### 7.3 Initial Results

For federation among social networks, our work is based upon the rich semantic models and algorithms for processing social data [117] and privacy policies [65] developed by us as part of the Yarta middleware for mobile social applications [116]. As an initial step to understand the heterogeneity inherent in existing OSNs, we started by studying the social functionalities provided by Facebook and Twitter, two commonly used OSNs, and created a bridge between them in order for their users to interact socially with each other across OSN boundaries. Figure 7.1 depicts the architecture of our middleware, called Yarta-FSN. A Facebook users needs only to install a Facebook app, after which he can make requests to “follow” users on Twitter. For a Twitter user, the middleware appears as a fellow Twitter user, which performs the relaying of status messages between the social networks. The middleware takes care of maintaining access control lists and delivering status updates only to the relevant users.

During our initial evaluation, we identified two immediate issues:

1. Since Twitter does not have the concepts of apps, the middleware had to masquerade as a user, followed by all Twitter users who want to interact with Facebook users. This leads to a large privacy risk, since any Twitter user following the middleware on Twitter can read all the messages relayed by it, even those meant for other Twitter users! While this can be resolved by creating a new middleware user on Twitter for each actual Twitter user, this approach is not scalable. However, this is more of an implementation issues, since if Twitter allowed apps like in Facebook, it would not arise.

2. The access control policies for Facebook and Twitter are not interoperable. Notably, any posts on Facebook are visible to the user’s followers on Twitter, but the replies from Twitter are visible only to the post’s author. This is more restricted than the default Facebook policy, where the replies are visible to everyone who can read the initial post. Unlike the previous issue, this is a more fundamental problem, since it is rooted in the differences between the two OSNs’ access control models.
7.4 Future Plans

This initial exercise has provided us with the early insights into the specifics of the challenges expected in creating FSNs. In the near future, KUL and Inria will be leveraging their experiences on mobile social networking middleware, multi-network service discovery and access, semantic technologies, and interoperability between software systems to address the above mentioned challenges in the federated social networks of the future. The community has already started looking into the area, and we expect the interest to grow. We will begin by defining an extensible model of social data and interactions, and validating it against the common commercial OSNs as well as prominent approaches proposed in the community. We will then use this to base our work on service oriented middleware for federated social networks, connecting the various OSN providers, as well as to the various mobile devices owned by each user. Our research will be continually incorporated into the Yarta middleware, with the release of both representative mobile applications based on our middleware as well as reference implementations of our research on federation for OSN providers, allowing us to rapidly evolve our research based on the feedback received.
8 Relation to other Work Packages

The work in Work Package 8 partially overlaps and links with work in other work packages. This section offers a short summary of these links.

- Work presented in WP7 describes a prototype tool that adapts the system at runtime when the access control policy is changed. This tool relies on middleware platforms that would also fit within the context of Task 8.3 ("Platform support for security enforcement").

- The work about synthesizing a mediator of Inria in WP9 is closely related to Task 8.2 ("Secure service composition for the Future Internet").

- The work about Secure Navigation Paths is joint work of LMU and Siemens and is related to Task 9.2 ("Assurance in Implementation") of WP9. The case study we use is related to the smart grid Scenario of WP11. The used notation, UWE, is extended in WP7.

In addition to the above, KUL and Inria are continuing their new collaboration, which started due to NESSoS-enabled interactions. This work deals with access-control on federated social networks, where users’ social data need not be hosted on the same online social network provider. It can potentially lay the foundation of a whole new paradigm of social networking, not only freeing regular users from the current limitations imposed by un-interoperable data-silos used by large commercial social networking providers, but also providing a valuable service to the burgeoning field of enterprise social networks, allowing users from different companies to collaborate on joint projects as needed.

New interactions with WP10 are also foreseen with respect to the monitoring of security indicators in order to support risk assessment.
9 Conclusion

Work package 8 delivers a broad range of work, which is split in three main topics: (1) security support for programming languages (Task 8.4), (2) runtime support for the secure execution of services (Task 8.3), and (3) secure composition of services (Task 8.2). This document presents some of the work that has been done for each of these topics in the third year of the NoE, and refers the interested reader to the other published work that is not summarized here. The topics that are dealt with in this work package are very much alive in the research community, as evidenced by the large number of publications in top conferences by the members of WP8.

The work that is presented here or is referenced in Appendix A consists of contributions from all partners in this work package:

- **CNR** contributed to Task 8.2 and Task 8.3
- **IMDEA** contributed to Task 8.3 and Task 8.4
- **Inria** contributed to Task 8.2
- **KUL** contributed to Task 8.2, Task 8.3 and Task 8.4
- **LMU** contributed to Task 8.3
- **Siemens** contributed to Task 8.3 and Task 8.4

There is a strong collaboration between the partners of WP8. In particular, this deliverable presented collaborative work between IMDEA and Siemens, between LMU and Siemens, and between KUL and UNITN. The collaboration between the partners of WP8 and with other work packages is ongoing, and will be continued further as we go into the final months of the project. Current collaborations include work between KUL and Inria on federated social networks, and work between CNR and UMA on secure service composition.

Because of the breadth of this work package, it is difficult to align the research agendas onto a single path. The three tasks are complementary but also orthogonal to each other, and even within tasks there are different classes of solutions (for different subproblems) that are difficult to align. However, NESSoS does provide us with tools to help us in this matter. One example is the SDE where we have a set of tools belonging to different tasks (and even different work packages) that are implemented in a similar uniform way. This allows bridges to be built between technologies, and is a first step in a fully integrated platform that offers solutions for the problems described in the tasks of WP8.
A Appendix - Relevant Papers

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


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 supported by NESSoS and that belong to this work package:


Bibliography


