Model-based Security Guarantees and Change

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technische Universität Dortmund
am Fachbereich Informatik von

MSc. Martín Ochoa Ronderos

Dortmund

2012

Tag der mündlichen Prüfung: 16 Juli 2012
Dekanin: Prof. Dr. Gabriele Kern-Isberner
Erster Gutachter: Prof. Dr. Jan Jürjens
Zweiter Gutachter: Prof. Dr-Ing. Luca Viganò

"O frati," dissi, "che per cento milia
perigli siete giunti a l'occidente,
a questa tanto picciola vigilia

d'i nostri sensi ch'è del rimanente
non vogliate negar l'esperïenza,
di retro al sol, del mondo sanza gente.

Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza" [1]

Dante Alighieri, *Inferno, Canto XXVI*

---

[1] In English, as translated in rhyme by Seth Zimmerman [138] :

"Brothers," I said, "you who through a hundred thousand
Perils have reached the west, do not deny
To the brief vigil of your senses this final errand:

Before the time remaining to you goes by,
Seek out the uninhabited world beyond the sun;
Make it your last experience before you die.

Think of your origins: you're not just anyone
You weren't born to live like brutes;
To pursue knowledge and virtue is your mission!"

Abstract

Achieving security in practical systems is a hard task. As it is the case for other critical system properties (i.e. safety), security should be a concern through all the phases of software development, starting with the very early phases of requirements and design, because of the potential impact of unwanted behaviour. Moreover, it remains a critical concern throughout a system's life-span, because functionality driven updates or re-engineering of a system can have an impact on its security. The cost of using formal methods is clearly justified for critical applications. But in the context of a wider class of industrial applications answers to two questions are important: What are the gains and limitations of light-weight formal security guarantees achieved at different abstraction levels? What are the advantages of those techniques for reasoning about change?

For the first question, we discuss different detailed modelling techniques, ranging from UML models to CPU cache modelling at the level of binary code. To tackle the second question, we discuss results on compositionality and incremental verification techniques which, besides being useful tools for verification in general, allow re-utilization of existing verification results in case of changes in the models. We apply these techniques to exemplary security properties with focus on confidentiality, and pin down security assumptions and guarantees of information flow control across levels of abstraction.

Disclaimer This thesis is based on the results of the following peer-reviewed publications (most recent first):

1. Automatic Quantification of Cache Side-channels [89]
   Joint work with B. Köpf and L. Mauborgne
   Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012), Springer LNCS, 2012

2. Non-interference on UML state-charts [111]
   Joint work with J. Jürjens and J. Cuéllar
   Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe), Springer LNCS, 2012

3. A Sound Decision Procedure for the Compositionality of Secrecy [112]
   Joint work J.Jürjens and D. Warzecha
   Proceedings of the 4th International Symposium on Engineering Secure Software and Systems (ESSoS), Springer LNCS, 2012.

4. Security Guarantees and Evolution: From models to reality [110]
   Proceedings of the First ESSoS Doctoral Symposium, Eindhoven, CEUR, 2012.

5. Model-Based Security Verification and Testing for Smart-cards [55]
   Joint work with E.Fourneret, J. Botella, F. Bouquet, J.Jürjens and P. Yousefi
   Proceedings of the 6th International Conference on Availability, Reliability and Security (AReS 2011), IEEE, 2011.

6. Incremental Security Verication for Evolving UMLsec models [80]
   Joint work with J.Jürjens, L.Marchal and H. Schmidt.
   Proceedings of 7th European Conference on Modelling Foundations and Applications (ECMFA 2011), Springer LNCS, 2011.

where I have substantially contributed. In collaborative efforts, it is difficult to exactly draw the line between the contributions of the single authors. In this thesis however, I have tried to focus as much as possible on my contributions to those publications. Other peer-reviewed publications I have co-authored during my PhD studies are [54, 48] but those results are not included in this document, for the sake of narrowness of scope.

Martín Ochoa

Acknowledgements I would like to thank very warmly my supervisors Jan Jürjens at the TU Dortmund and Jorge Cúellar at Siemens AG. Jan was always very open for all kinds of discussions and encouraging during the inevitable beginner's pitfalls. Jorge very friendly welcomed me at Siemens in Munich for the last year of the PhD, which was an enriching experience from many points of view. Together they managed to be both challenging and motivating in the right measure, which I believe is essential for managing a successful project.

I would also like to thank all my co-authors during this period for interesting discussions and productive collaborative work, in particular Boris Köpf at IMDEA who has also made very helpful comments for improving this document. Prof. Marialuisa J. de Resmini deserves also my most sincere gratitude. She has been a kind friend and a wise guide since my Bachelor studies. I'm also very thankful with my family for their continuous support, patience and love, specially my mother Margarita and my girlfriend Diana.

Special thanks go also to the European Union, the German research foundation, the TU Dortmund and Siemens AG for partially sponsoring this research in the context of the SecureChange (EU, TU Dortmund) and NESSoS projects (EU, Siemens AG) and the MoDelSec project (DFG and TU Dortmund).

Contents

1 Introduction **11**

2 Methodology **17**
2.1 The vertical view 17
2.2 The horizontal view 19
2.3 Information flow control and access control 20

3 Security requirements **23**
3.1 UMLseCh: Supporting Evolution of UMLsec Models 25
3.2 Verification Strategy 31
3.3 Application examples 33
3.4 Tool support 35
3.5 Related Work 37

4 Non-interference on UML state-charts **39**
4.1 Preliminaries 40
4.2 Verification Strategy 43
4.3 Object interaction 48
4.4 Validation 50
4.5 Related Work 53

5 Security protocols **57**
5.1 Preliminaries 58
5.2 Decision procedure 60
5.3 An insecure variant of the TLS protocol 64
5.4 Validation and Efficiency 68
5.5 Related Work 69

6 Cache side-channel analysis **71**
6.1 Preliminaries 73
6.2 Cache Channels 75

6.3 Counting Cache States 78
6.4 Implementation 82
6.5 Case Study 84
6.6 Related work 85

7 Model-Based Testing 89
7.1 Consistency verification with UMLsec 90
7.2 Handling change 93
7.3 Validation 95
7.4 Related work 97

8 Conclusions 99

A Omitted proofs 115
A.1 Chapter 4 115
A.2 Chapter 6 115

B Code snippets 117
B.1 Chapter 4 117
B.2 Chapter 6 118

Chapter 1

Introduction

In recent years, information security has gained increasing attention from the general public and there is a consensus about its paramount importance in society. Examples include recent scandals on users private data [125], leaks of government secret documents and public threats from anonymous hacker groups to corporate and governmental IT systems worldwide [10, 11]. Long gone are the days where the term 'computer security' was associated exclusively with spies, conspirational theories and cryptography. Today most successful attacks exploit vulnerabilities related to problems in design or implementation rather than vulnerabilities in cryptographic mechanisms. And most of the attackers are motivated teenagers, typically not interested in the mathematical aspects of cryptography[1]. In the words of Anderson [23]:

> "(...) in practice, security is compromised most often not by breaking dedicated mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being used."

There are several reasons why security is difficult to achieve in practice. On the one hand, the complexity of modern system architectures is constantly increasing: software logic evolves, often driven by the market pressure to deliver new functionalities, and different operating systems and hardware configurations make each instantiation of an IT system unique. Moreover, software components from different producers delivered without accurate (if at all) security guarantees are used together to achieve customized solutions. On the other hand, attacks can be performed by exploiting design failures at different levels of abstraction, ranging from vulnerable cryptographic protocol logic [26] to leakage due to micro-architectural configurations [31].

[1] Also social aspects of security result in attacks in many cases, but those are very difficult to control by technological means and are out of the scope of this work.

Techniques to tackle this ever 'moving target' exist in different areas of computer science and engineering: from software and hardware formal verification to testing, but also at the level of business-processes modelling and risk-analysis. As observed before, security plays a role at different levels of abstraction and at different phases of the development cycle, and if one wants to have a high degree of assurance about the security of a system one should consider them all.

In general, formal methods deliver strong guarantees about software and hardware behaviour, due to their rigour and precision. It is thus natural to consider formal methods to validate a system with respect to well-defined, mathematical descriptions of security. Typically this effort is justified in critical applications, where unwanted behaviour could have a big impact. Nevertheless, formal methods are difficult to apply in most commercial software-development scenarios because they require highly specialized designers and programmers in order to carry out the formalizations and interpret the verification results. For less critical systems, it has been recognized that formal methods offer a good cost-benefit relation if used 'correctly': focusing on light-weight methods such as formal specification, requirements validation and test generation from specifications rather than on full code, machine-code and hardware verification [92, 66].

Light-weight methods are usually applied at the level of system design, where models are abstracted from implementation details and are more amenable to automated verification. Since some security problems can be detected already at the specification level, the cost-benefit of applying this methodology is typically better than repairing design problems at later stages of development [92]. The de-facto standard for system modelling in industry is the Universal Modelling Language (UML), thus it is natural to perform this formal verification on UML models, if one aims at industry acceptance. It is important to have then a formally defined fragment of UML and a collection of easy to use tools to partially verify the correctness of models with respect to security properties as it had been presented by [77, 30].

Because of the necessary abstractions made, it is however not enough to have strong security guarantees on system models to be able to judge the overall security of a deployed system, which is the ultimate goal. As argued before, it is difficult to verify the code because of its size and the huge variety of programming languages and paradigms. Moreover it can be the case that libraries or components from third parties are used whose source code is unavailable. Therefore, conformance testing, where the expected system behaviour is compared to the actual behaviour for selected runs is useful in gaining confidence about a given system. The main disadvantage of this method is that it is only statistically accurate: in general it is computationally infeasible to test all possible inputs of a system.

Despite the confidence won with formal or informal validation of a system, security is a never ending open loop, since regularly new exploits appear

that consider properties or interfaces that were not considered at design. This is prominently illustrated by side-channels: here the attacker exploits information such as electro-magnetic radiation, timing and shared memory behaviour to gain possession of confidential data. Many of these side-channels are difficult to capture since they rely on micro-architectural configurations such as the duration of processor instructions or the cache behaviour. Closing these interfaces is sometimes impossible or costly, because of the impact to other system requirements (i.e. efficiency).

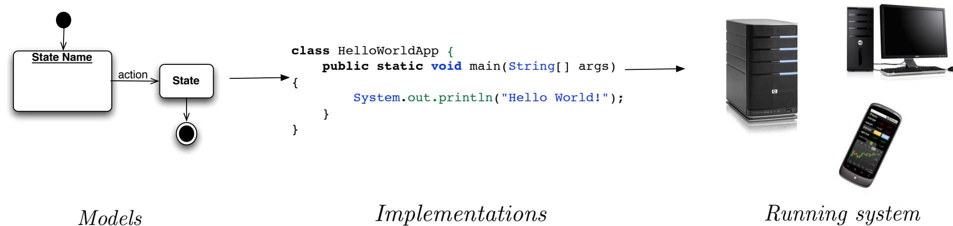


Figure 1: Roadmap from models to running systems

In the following we summarize the goals of this research and the achieved results. Security is typically described as the conjunction of one or more security requirements, abstractly classified as Confidentiality, Availability and Integrity. In this work we consider mainly Confidentiality: we want to understand how private information is flowing from a group of users to another [2]. We also discuss some preliminary steps towards reasoning about availability in the context of model-based testing.

There are basically two ways to look at the problem and our contributions: a) according to the abstraction level modelled and b) according to the security properties considered. First, we will take the first point of view (see Fig. 1).

UML Models: Requirements We consider the problem of specification evolution for security at the level of UML models. We extend the UMLsec [77] notation and verification techniques to reason about changes in the specification by means of the novel UMLseCh notation in Chapter 3. For some properties defined in static UML diagrams, we describe sufficient conditions that soundly preserve the security of already verified models by analysing the delta implied by the modifications. This fine-grained incremental technique is an appropriate choice for structural properties because of their locality: changes of parts of the model usually affect the property in a clearly identifiable, small subset of the specification.

[2] Nevertheless integrity is the dual of confidentiality when reasoning about information flow, and some of our results could be applied to reason about integrity.

UML Models: System logic For behavioural models, incremental changes can affect security in non-trivial ways. Therefore we propose to focus on compositionality results: if one or more components of a given system is substituted, the overall security of the system can be decided by re-verifying only the new components given a compositionality condition. In Chapter 4 we extend previous work on verifying non-interference in UML state-charts and derive compositionality results for interacting objects. We consider a fragment of UML State-charts that allow one to define a class behaviour, by taking advantage of the compact representation offered by hierarchical states and variables for representing the history of the state.

In particular, cryptographic protocols are an important building block for establishing secure and authentic channels between parties connected over an insecure channel. For the very specific task of specifying and verifying such protocols, we follow the approach of Jürjens [77] of using a domain specific language (DSL) associated to UML sequence diagrams. In Chapter 5 we describe a sound decision procedure for the compositionality of Dolev-Yao secrecy on processes exchanging cryptographic messages, building on the aforementioned DSL.

Micro-architecture models At this detailed abstraction level, we focus on cache configurations and how they can act as a leaking channel for different adversaries. Configurations of the CPU play a determinant role on the security of the system with respect to side-channel attacks, and the change in configurations is a typical phenomenon of system's evolution. Avoiding the use of caches conflicts with efficiency requirements and is therefore not realistic for a wide range of systems. A promising technique to achieve formal guarantees about countermeasures striving for a trade-off between security and other conflicting requirements is quantitative information flow analysis (for example [88]). In Chapter 6 we formalize heuristic countermeasures proposed in the literature and give strong security guarantees for arbitrary programs under a well-defined attacker model, and validate our approach using an automatic tool chain that evaluates compiled programs for various architectures.

Implementations: Model-based testing Recently, the use of models to describe the expected behaviour of a system have been proposed in the context of conformance testing for security properties [35, 78, 135]. However, the consistency of those models with respect to the properties to be tested is usually neglected. In Chapter 7 we discuss preliminary results about a methodology to verify UML models used for model-based testing based on a black box model of the system, extending previous work on security testing. We also discuss conditions under which the security relevant properties of the model are preserved under incremental changes, by using the techniques

discussed in 3.

In summary, in this work we propose a set of tools, easy to use by end users, and addressing the aforementioned problems on Models, Implementations and Micro-architectures. We believe that it is unrealistic to attempt to build complex industrial systems with a 100% guarantee of security because logical or physical interfaces that are vulnerable to attacks are often only exposed after real attacks take place. To date there is also no standard single methodology, tool, or model to tackle the whole Software Development Life-Cycle. It is thus necessary to provide tools that help to cope with evolution problems at all levels of abstractions and during the whole lifecycle. It is also our belief that the idea that systems are going to be secure because we commit to a single approach (for example rigorous security requirements elicitation, industrial best-practices, strict patch update policies etc.) is fallacious: we have to work on all phases and at different abstraction levels, sometimes using different models and different methodologies. Moreover, there is a necessary trade-off between security and costs associated to formalization efforts, and we advocate for the use of automatic tools and intuitive specification languages that allow non-experts to use them.

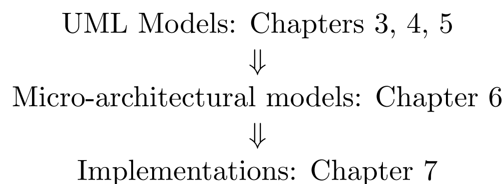
On the theoretical side, the contributions of this thesis are incremental and compositional analysis of exemplary security properties at different levels of abstractions and in different model types, ranging from structural and behavioural UML diagrams to CPU cache models. Practically, we apply these results in concrete examples from diverse domains, including Smart-Grids models for non-interference and a full-blown AES implementation for cache side-channel analysis. We have implemented proof-of-concept tools to validate our approach, always focusing in easy to use, intuitive modelling languages and automatic verification. Moreover we have preliminary evidence that our methods are efficient in the sense that they can handle models of reasonable size in little time. In the next chapter we discuss in detail the relation between the different chapters of the thesis, and how do the techniques in each of them combine methodologically to achieve a high degree of confidence on the security of systems.

Chapter 2

Methodology

As discussed in the introduction, the problem of constructing secure software poses many challenges. On the one hand, the complexity of modern system architectures is constantly increasing, and spans from software logic, operating systems and diverse hardware configurations, making security design and verification difficult. On the other hand, new exploits and security holes are found on a regular basis, due to an increasing interest in the subject by practitioners, researchers and of course, attackers themselves. Techniques to tackle this ever 'moving target' have been proposed from different corners of computer science: from formal verification and theoretical computer science to software-testing (and even hardware testing and verification to ensure secure systems as a whole). In this chapter we discuss how the technical results of the subsequent chapters of this thesis are methodologically organized. There are different dimensions and points of view from which this analysis can be performed. We will discuss at least three of them: the vertical dimension (referring to the abstraction levels of the models), the horizontal dimension (discussing techniques for verification under change) and the relationship between the security properties considered.

2.1 The vertical view



In the next chapters we will thus introduce the results of this thesis for tackling the complex task of designing and verifying secure systems. Our goal is to explore three main levels of abstraction and to advance the state

of the art of techniques for tackling exemplary security properties in each of them. Indeed, the ultimate goal is to achieve a secure running system, that is a deployed software in a concrete hardware environment. To accomplish this, we start by considering models of the envisaged system and gradually refine them. For each step in the software development process, it is important to evaluate security with appropriate techniques.

The UML model level is particularly interesting for the construction of secure software, since already at design some security flaws can be spotted, and are less expensive to correct at that stage [92]. For example in distributed systems that communicate using cryptographic protocols, the Dolev-Yao attacker model, where perfect cryptography is assumed, has been a successful model to reason about the correctness with respect to security properties. This line of research has been followed by Jürjens [77] who has also developed a method to reason systematically about software components communicating through insecure channels in UML, called UMLsec.

In Chapter 3 we explore an extension to UMLsec to deal with changes in the system model and to re-use previous verification results on the original model called UMLseCh. The verification is based mainly on the differences between model versions and is defined for different UMLsec diagrams, aiming at sufficient conditions respecting security. This is useful to tackle the continuous changes in the system model that software usually undergoes. This is however more challenging when it comes to behavioural models, where an incremental analysis is less feasible.

We consider the question of non-interference and its compositionality on Chapter 4, which is a fine-grained analysis that can detect leaks of secret information that arise in the system logic despite assuming perfect access control. In Chapter 5 we reason about the composition of processes exchanging cryptographic messages specified in Sequence Diagrams, which is also an important methodology when deciding on the evolution of a system. This is because a compositional decision procedure can be helpful when adding or replacing components of an existing system due to evolution.

As discussed in the introduction, not even full verification is enough to have exhaustive security on the system for different reasons. One of them is that they both assume ideal protection of some ‘security primitives’ like cryptographic functions against leakage. However, micro-architecture configuration can be determinant on the presence (and the capacity) of unwanted side-channels. In Chapter 6 we analyse the role of the cache in confidentiality in general way, and describe how to formalise countermeasures to known attacks by means of quantitative information flow.

Since the model level abstracts from implementation details, it is necessary to have some guarantees on the implementation level. It is however not easy to verify implementations for different reasons: the availability of source-code in all components used and the scalability problem of modern model-checkers. We consider thus the most popular industrial methodology

to obtain guarantees on implementation: Testing. In Chapter 7 we describe a well-founded methodology generating test sequences from UMLseCh models, thus re-using results of the previous chapter on system evolution, and linking them to test-specifying languages. We exemplary show how to test security properties related to preventing Denial of Service attacks (DoS) and access control, but also comment on how this can be extended to other properties.

Notice that we do not aim at a formal integration of the security guarantees obtained at the discussed levels of abstraction: such a task, although interesting, goes outside the scope of this work. On the other hand, current industrial environments do not have a formally justified software development process. We consider different layers of abstraction mainly due to necessity: any error found in any of these layers would invalidate the over-all security of the system.

2.2 The horizontal view

From the perspective of model evolution, in this thesis we explore two possible solutions: on the one hand we consider incremental analysis, were small changes to a monolithic model are evaluated with respect to a security property. On the other hand we also consider a less fine grained analysis by reasoning about compositionality of security properties. Both analysis have however advantages depending on the security property considered as we discuss in the following.

Incremental analysis For some model consistency properties related to security, where the property quantifies on local model elements, it is possible to give quite precise sufficient conditions that preserve security in the presence of small modifications. This modifications can be additions, substitutions or deletions of different model elements, and can be clearly identified by using for example the UMLseCh language proposed in Chapter 3. There sufficient conditions for exemplary UMLsec properties are discussed. Also in Chapter 7, an incremental analysis of the model consistency properties defined for model-based security testing is discussed.

Compositional analysis The incremental analysis is appropriate when small parts of the model need to be re-verified. In general, this is not the case for security properties on behavioural models such as security protocol interaction or non-interference: small changes of the model may imply a complete reverification. Therefore, we have tackled this problem by considering a global perspective of the system: if the system model is built up on interacting components (like most modern system architectures) then the effects of changes to a small number of components to the overall system security can be easily decided if there exist decision procedures for the composition of

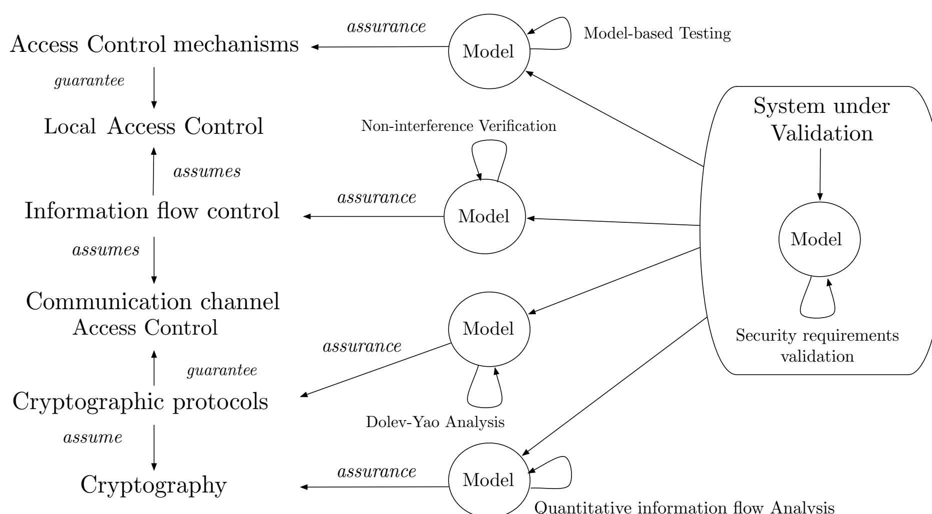


Figure 1: Assumptions and guarantees for information flow control

verification results on the single components. In general, security properties are not compositional: if A respects a given property and B respects it as well, is not said that $A \otimes B$ (their compositions) does. This is for example the case with secrecy on cryptographic protocols as we discuss on Chapter 5, but also for information-flow properties [95]. In Chapter 5 we discuss a sound and efficient[1] decision procedure that given proof artefacts (dependency trees) on two components can establish whether their composition will be security preserving or not. In Chapter 4 we prove a compositionality theorem for non-interference on our system model (UML state-charts) for one-way composition without call-backs.

2.3 Information flow control and access control

Although we do not provide a formal connection, it is nevertheless interesting to informally discuss our contributions from the perspective of the security properties considered and the assumptions they rely on at different abstraction levels. In fact, when analysing information-flow at the model level, we are assuming perfect mechanisms for access control, and among others, perfect cryptography, which guarantees access control when information is shared through insecure networks. To gain confidence in the correctness of those mechanisms, we validate them locally using model-based testing and performing a Dolev-Yao analysis for the cryptographic protocol logic. To gain even more confidence about primitives, we consider the micro-architectural abstraction level, using quantitative information-flow related techniques. We

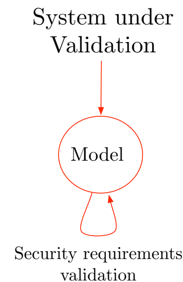
[1] In the sense that it can handle models of reasonable size in little time.

can see this as an (informal) chain of assumptions and guarantees across abstraction levels, as depicted in Fig. 1.

The assumptions and guarantees discussed here are by no means complete: for example we are not considering operating system access control mechanisms or semantic security properties of the cryptographic primitives. As already discussed we believe that a complete and formally justified methodology is unrealistic. It is nevertheless essential to consider different levels of abstraction and development phases to achieve a good degree of confidence in the system, since an error in any of them would invalidate the results at higher abstraction levels or previous phases. For example a faulty implementation of access control would invalidate a secure abstract design w.r.t non-interference, and a cryptographic implementation with side-channels would make a formally verified protocol against the Dolev-Yao adversary meaningless.

Chapter 3

Security requirements



The task of evolving software systems such that the desired security requirements are preserved through a system's lifetime is of great importance in practice. In this chapter we discuss a model-based approach to support the evolution of secure software systems. Our approach allows us to verify of potential future evolutions using an automatic analysis tool. An explicit model evolution implies the transformation of the model and defines a difference Δ between the original model and the transformed one. The proposed approach supports the definition of multiple evolution paths, and provides tool support to verify evolved models based on the delta of changes. This idea is visualized in Fig. 1: The starting point of our approach is a Software System Model which was already verified against certain security properties. The model can then evolve within a range of possible evolutions (the evolution space). We consider the different possible evolutions as evolution paths each of which defines a delta Δ_i. The result is a number of evolved system models. The main research question is: which of the evolution paths leads to a target model that still fulfils the security properties of the source model?

Theoretically, one could simply re-run the security analysis done to establish the security of the original model on the evolved model to decide whether these properties are preserved after evolution. This would, however, result in general in a high resource consumption for models of realistic size,

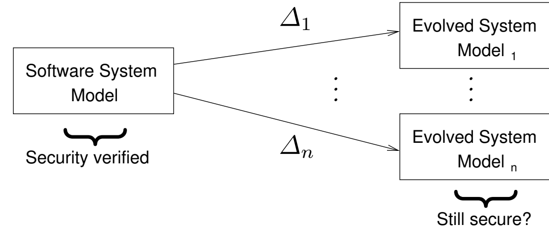


Figure 1: Model verification problem for n possible evolution paths

in particular since the goal in general is to investigate the complete potential evolution space (rather than just one particular evolution) in order to determine which of the possible evolutions preserve security. Also, verification efficiency is very critical if a continuous verification is desired (i.e. it should be determined in real-time and in parallel to the modelling activity whether the modelled change preserves security).

We use models specified using the Unified Modeling Language (UML) [1] and the security extension UMLsec [76]. The UMLsec profile offers new UML language elements (i.e., stereotypes, tags, and constraints) to specify typical security requirements such as secrecy, integrity, and authenticity, and other security-relevant information. Based on UMLsec models and the semantics defined for the different UMLsec language elements, possible security vulnerabilities can be identified at an early stage of software development. One can thus verify that the desired security requirements, if fulfilled, enforce a given security policy. This verification is supported by a tool suite [2] [82].

In this chapter we present a general approach for the incremental security verification of UML models against security requirements inserted as UMLsec stereotypes. We discuss the possible atomic (i.e. single model element) evolutions annotated with certain security requirements according to UMLsec. Moreover, we present sufficient conditions for a set of model evolutions, which, if satisfied, ensure that the desired security properties of the original model are preserved under evolution. We demonstrate our general approach by applying it to a representative UMLsec stereotype, «secure dependency». As one result of our work, we demonstrate that the security checks defined for UMLsec account for significant efficiency gains by considering this incremental verification technique.

Our technique is based basically in a two-sided type distinction: on the one hand we reason about the UML type of an atomic change and on the other hand we distinguish on its evolution type. This technique allows for a comprehensive class of sufficient conditions if the security properties in ques-

[1] The Unified Modeling Language `http://www.uml.org/`
[2] Available online via `http://www-jj.cs.tu-dortmund.de/jj/umlsectool`

tion are of a mainly local nature. In other words, the incremental technique works well if a small change has an impact in a small neighbourhood. In subsequent chapters we will discuss compositional techniques which allows one to reason about more complex security properties.

To explicitly specify possible evolution paths, we have developed a further extension of the UMLsec profile (called UMLseCh) that allows to precisely define which model elements are to be *added*, *deleted*, and *substituted* in a model. Constraints in first-order predicate logic help to coordinate and define more than one evolution path (and thus obtain the deltas for the analysis).

Note that UMLseCh is not intended as a general-purpose evolution modeling language: it is specifically intended to model the evolution in a security-oriented context in order to investigate the research questions with respect to security preservation by evolution (in particular, it is an extension of UMLsec and requires the UMLsec profile as prerequisite profile). Thus, UMLseCh does not aim to be an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [69, 24, 33, 120, 86]), but as a mean to easily reason about the delta induced by a transformation within the UMLsec framework.

3.1 UMLseCh: Supporting Evolution of UMLsec Models

In this section we present a further extension of the UML security profile UMLsec to deal with potential model evolutions, called UMLseCh (that is, an extension to UML which itself includes the UMLsec profile). Figure 2 shows the list of UMLseCh stereotypes, together with their tags and constraints, while Fig. 3 describes the tags.

The UMLseCh tagged values associated to the tags {`add`} and {`substitute`} are strings, their role is to describe possible future model evolutions. UMLseCh describes **possible future changes**, thus conceptually, the substitutive or additive model elements are not actually part of the current system design model, but only an attribute value inside a change stereotype[3]. At the concrete level, i.e. in a tool, this value is either the model element itself if it can be represented with a sequence of characters (for example an attribute or an operation within a class), or a namespace containing the model element.

Note that the UMLseCh notation is complete in the sense that any kind of evolution between two UMLsec models can be captured by adding a suitable number of UMLseCh annotations to the initial UMLsec model. This can be seen by considering that for any two UML models M and N there exists a sequence of deletions, additions, and substitutions through which the model M can be transformed to the model N. In fact, this is true even when only

[3]The type change represents a type of stereotype that includes «change»,«substitute», «add» or «delete».

Stereotype	Base Class	Tags	Constraints	Description
change	all	ref, change	FOL formula	execute sub-changes in parallel
substitute	all	ref, substitute,	FOL formula	substitute a model element
add	all	ref, add,	FOL formula	add a model element
delete	all	ref, delete	FOL formula	delete a model element
substitute-all	all	ref, substitute,	FOL formula	substitute a group of elements
add-all	all	ref, add,	FOL formula	add a group of elements
delete-all	all	ref, delete	FOL formula	delete a group of elements

Figure 2: UMLseCh stereotypes

Tag	Stereotype	Type	Multip.	Description
ref	change, substitute, add, delete, substitute-all, add-all, delete-all	list of strings	1	List of labels identifying a change
substitute	substitute, substitute-all	list of pairs of model elements	1	List of substitutions
add	add, add-all	list of pairs of model elements	1	List of additions
delete	delete, delete-all	list of pairs of model elements	1	List of deletions
change	change	list of references	1	List of simultaneous changes

Figure 3: UMLseCh tags

considering deletions and additions: the trivial solution would be to sequentially remove all model elements from M by subsequent atomic deletions, and then to add all model elements needed in N by subsequent additions. Of course, this is only a thereotical argument supporting the theoretical expressiveness of the UMLseCh notation, and this approach would neither be useful from a modelling perspective, nor would it result in a meaningful incremental verification strategy. This is the reason that the substitution of model elements has also been added to the UMLseCh notation, and the incremental verification strategy explained later in this work will crucially rely on this.

3.1.1 Description of the Notation

In the following we give an informal description of the notation and its semantics.

substitute

The stereotype «substitute» attached to a model element denotes the possibility for that model element to evolve over time and defines what the possible changes are. It has two associated tags, namely ref and substitute. These tags are of the form { ref = CHANGE-REFERENCE } and

$$\{ \mathsf{substitute} = (\mathsf{ELEMENT}_1,\ \mathsf{NEW}_1),\ \ldots,\ (\mathsf{ELEMENT}_n,\ \mathsf{NEW}_n) \}$$

with $n \in \mathbb{N}$. The tag ref takes a list of sequences of characters as value, each element of this list being simply used as a reference of one of the changes modeled by the stereotype «substitute». In other words, the values contained in this tag can be seen as labels identifying the changes. The values of this tag can also be considered as predicates which take a truth value that can be used to evaluate conditions on other changes (as we will explain in the following). The tag substitute has a list of pairs of model element as value, which represent the substitutions that will happen if the related change occurs. The pairs are of the form $(e,\ e')$, where e is the element to substitute and e' is the substitutive model element [4]. For the notation of this list, two possibilities exist: The elements of the pair are written textually using the abstract syntax of a fragment of UML defined in [76] or alternatively the name of a namespace containing an element is used instead. The *namespace notation* allows UMLseCh stereotypes to graphically model more complex changes (cf. Sect. 3.1.2).

If the model element to substitute is the one to which the stereotype «substitute» is attached, the element e of the pair $(e,\ e')$ is not necessary. In this case the list consists only of the second elements e' in the tagged value, instead of the pairs (this notational variation is just syntactic sugar). If a change is specified, it is important that it leaves the resulting model in a syntactically consistent state. In this work however we focus only on the preservation of security.

Example We illustrate the UMLseCh notation with the following example. Assume that we want to specify the change of a link stereotyped «Internet» so that it will instead be stereotyped «encrypted». For this, the following three annotations are attached to the link concerned by the change (cf. Figure 4):

«substitute», { ref = encrypt-link }, { substitute = («encrypted», «Internet») }

The stereotype «substitute» also has a list of optional constraints formulated in first order logic. This list of constraints is written between square brackets and is of the form $[(\mathsf{ref}_1,\ \mathsf{CONDITION}_1),\ \ldots,\ (\mathsf{ref}_n,\ \mathsf{CONDITION}_n)]$,

[4] More than one occurrence of the same e in the list is allowed. However, two occurrences of the same pair $(e,\ e')$ cannot exist in the list, since it would model the same change twice.

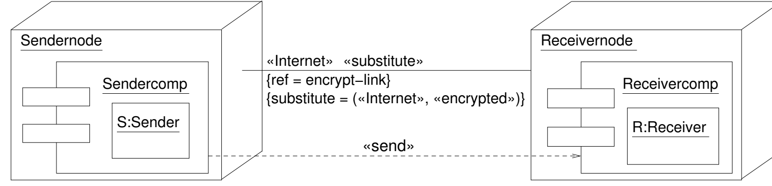


Figure 4: Example of stereotype substitute

$n \in \mathbb{N}$, where, $\forall i : 1 \leq i \leq n$, ref_i is a value of the list of a tag ref and $\mathsf{CONDITION}_n$ can be any type of first order logic expression, such as $A \wedge B$, $A \vee B$, $A \wedge (B \vee \neg C)$, $(A \wedge B) \Rightarrow C$, $\forall x \in N.P(x)$, etc. Its intended use is to define under which conditions the change is allowed to happen (i.e. if the condition is evaluated to true, the change is allowed, otherwise the change is not allowed). As mentioned earlier, an element of the list used as the value of the tag ref of a stereotype «substitute» can be used as an atomic predicate for the constraint of another stereotype «substitute». The truth value of that predicate is true if the change represented by the stereotype «substitute» to which the tag ref is associated occurred, false otherwise.

To illustrate the use of the constraint, the previous example can be refined. Assume that to allow the change with reference encrypt-link, another change, simply referenced as change for the example, has to occur. The constraint [change] can then be attached to the link concerned by the change. To express for example that two changes, referenced respectively by change_1 and change_2, have to occur first in order to allow the change referenced encrypt-link to happen, the constraint [$\mathsf{change}_1 \wedge \mathsf{change}_2$] is added to the stereotype «substitute» modeling the change.

add and delete

Both «add» and «delete» can be seen as syntactic sugar for «substitute». The stereotype «add» attached to a parent model element describes a list of possible sub-model elements to be added as children to the parent model element. It thus substitutes a collection of sub-model elements with a new, extended collection.

The stereotype «delete» attached to a (sub)-model element marks this element for deletion. Deleting a model element could be expressed as the substitution of the model element by the empty model element $\emptyset$. Both stereotypes «add» and «delete» may also have associated constraints in first order logic.

substitute-all

The stereotype «substitute-all» is an extension of the stereotype «substitute». It denotes the possibility for **a set of model elements of same type and sharing common characteristics** to evolve over time. In this case, «substitute-all» will always be attached to the super-element to which the sub-elements concerned by the substitution belong. As the stereotype «substitute», it has the two associated tags ref and substitute, of the form { ref = CHANGE-REFERENCE } and

$$\{ \mathsf{substitute} = (\mathsf{ELEMENT}_1,\ \mathsf{NEW}_1),\ \ldots,\ (\mathsf{ELEMENT}_n,\ \mathsf{NEW}_n) \}.$$

The tags ref has the same meaning as in the case of the stereotype «substitute». For the tag substitute the element e of a pair representing a substitution does not represent one model element but **a set of model elements** to substitute if a change occurs. This set can be, for example, a set of classes, a set of methods of a class, a set of links, a set of states, etc. All the elements of the set share common characteristics. For instance, the elements to substitute are the methods having the integer argument *"count"*, the links being stereotyped «Internet» or the classes having the stereotype «critical» with the associated tag secrecy. Again, in order to identify the model element precisely, we can use, if necessary, either the UML namespaces notation or, if this notation is insufficient, the abstract syntax of UMLseCh.

Example To replace all the links stereotyped «Internet» of a subsystem so that they are now stereotyped «encrypted», the following three annotations can be attached to the subsystem: «substitute-all», { ref = encrypt-all-links }, and { substitute = («Internet», «encrypted») }. This is shown in Figure 5.

A pair (e, e') of the list of values of a tag substitute here allows us a parameterization of the values e and e' in order to keep information of the different model elements of the subsystem concerned by the substitution. To allow this, variables can be used in the value of both the elements of a pair. The following example illustrates the use of the parameterization in the stereotype «substitute-all». To substitute all the tags secrecy of stereotypes «critical» by tags integrity, but in a way that it keeps the values given to the tags secrecy (e.g. { secrecy = d }), the following three annotations can be attached to the subsystem containing the class diagram: «substitute-all», { ref = secrecy-to-integrity }, and:

$$\{ \mathsf{substitute} = (\{ \mathsf{secrecy} = \mathsf{X} \},\ \{ \mathsf{integrity} = \mathsf{X} \}) \}.$$

The stereotype «substitute-all» also has a list of constraints formulated in first order logic, which represents the same information as for the stereotype «substitute».

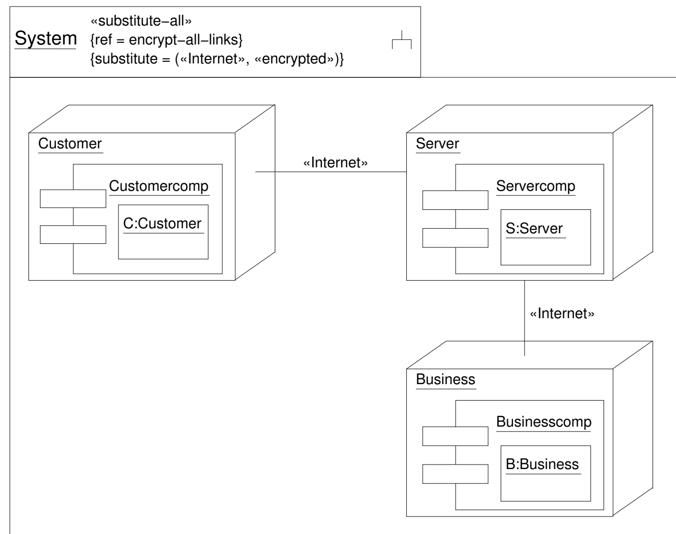


Figure 5: Example of stereotype substitute-all

change

The stereotype ⟪change⟫ is a particular stereotype that represents a *composite change*. It has two associated tags, namely ref and change. These tags are of the form { ref = CHANGE-REFERENCES } and { change = CHANGE-REFERENCES$_1$, ..., CHANGE-REFERENCES$_n$ }, with $n \in \mathbb{N}$. The tag ref has the same meaning as in the case of a stereotype ⟪substitute⟫. The tag change takes a list of lists of strings as value. Each element of a list is a value of a tag ref from another stereotype of type change.[5] Each list thus represents the list of *sub-changes* of a *composite change* modeled by the stereotype ⟪change⟫. Applying a change modeled by ⟪change⟫ hence consists in applying all of the concerned *sub-changes* **in parallel**.

Any change being a *sub-change* of a change modeled by ⟪change⟫ **must** have the value of the tag ref of that change in its condition. Therefore, any change modeled by a *sub-change* can only happen if the change modeled by the *super-stereotype* takes place. However, if this change happens, the *sub-changes* will be applied and the *sub-changes* will thus be removed from the model. This ensures that *sub-changes* cannot be applied by themselves, independently from their *super-stereotype* ⟪change⟫ modeling the *composite change*.

[5]By type change, we mean the type that includes ⟪substitute⟫, ⟪add⟫, ⟪delete⟫ and ⟪change⟫.

3.1.2 Complex Substitutive Elements

As mentioned above, using a complex model element as substitutive element requires a syntactic notation as well as an adapted semantics. An element is complex if it is not represented by a sequence of characters (i.e. it is represented by a graphical icon, such as a class, an activity or a transition). Such complex model elements cannot be represented in a tagged value since tag definitions have a string-based notation. To allow such complex model elements to be used as substitutive elements, they will be placed in a UML namespace. The name of this namespace being a sequence of characters, it can thus be used in a pair of a tag substitute where it will then represent a reference to the complex model element. Of course, this is just a notational mechanism that allows the UMLseCh stereotypes to graphically model more complex changes. From a semantic point of view, when an element in a pair representing a substitution is the name of a namespace, the model element concerned by the change will be substituted by the content of the namespace, and not the namespace itself. This type of change will request a special semantics, depending on the type of element. For details about this complex substitutions we refer to [126].

3.2 Verification Strategy

As stated in the previous section, evolving a model means that we either *add*, *delete*, or / and *substitute* elements of this model. To distinguish between big-step and small-step evolutions, we will call "atomic" the modifications involving only one model element (or sub-element, e.g. adding a method to an existing class or deleting a dependency). In general there exist evolutions from diagram A to diagram B such that there is no sequence of atomic modifications for which security is preserved when applying them one after another, but such that both A and B are secure. Therefore the goal of our verification is to allow some modifications to happen *simultaneously*.

Since the evolution is defined by additions, deletion and substitutions of model elements, we introduce the sets **Add**, **Del**, and **Subs**, where **Add** and **Del** contain objects representing model elements together with methods id, type, path, parent returning respectively an identifier for the model element, its type, its path within the diagram, and its parent model element. These objects also contain all the relevant information of the model element according to its type (for example, if it represents a class, we can query for its associated stereotypes, methods, and attributes). For example, the class "Customer" in Fig. 6 can be seen as an object with the subsystem "Book a flight" as its parent. It has associated a list of methods (empty in this case), a list of attributes ("Name" of type String, which is in turn a model element object), a list of stereotypes («critical») and a list of dependencies («call» dependency with "Airport Server") attached to it. By recursively comparing

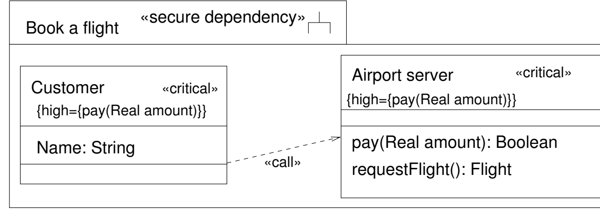


Figure 6: Class Diagram Annotated with «secure dependency»

all the attributes of two objects, we can establish whether they are equal.

The set **Subs** contains pairs of objects as above, where the type, path (and therefore parent) methods of both objects must coincide. We assume that there are no conflicts between the three sets, more specifically, the following condition guarantees that one does not delete and add the same model element:

$$\nexists\, \mathsf{o}, \mathsf{o}'(\mathsf{o} \in \mathbf{Add} \wedge \mathsf{o}' \in \mathbf{Del} \wedge \mathsf{o} = \mathsf{o}')$$

Additionally, the following condition prevents adding/deleting a model element present in a substitution (as target or as substitutive element):

$$\nexists\, \mathsf{o}, \mathsf{o}'(\mathsf{o} \in \mathbf{Add} \vee \mathsf{o} \in \mathbf{Del}) \ \wedge ((\mathsf{o}, \mathsf{o}') \in \mathbf{Subs} \vee (\mathsf{o}', \mathsf{o}) \in \mathbf{Subs})$$

As explained above, in general, an "atomic" modification (that is the action represented by a single model element in any of the sets above) could by itself harm the security of the model. So, one has to take into account other modifications in order to establish the security status of the resulting model. We proceed algorithmically as follows: we iterate over the modification sets starting with an object $\mathsf{o} \in \mathbf{Del}$, and if the relevant simultaneous changes that preserve security are found in the delta, then we perform the operation on the original model (delete o and necessary simultaneous changes) and remove the processed objects until **Del** is empty. We then continue similarly with **Add** and finally with **Subs**. If at any point we establish the security is not preserved by the evolution we conclude the analysis. Given a diagram M and a set Δ of atomic modifications we denote $M[\Delta]$ the diagram resulting after the modifications have taken place. So in general let P be a diagram property. We express the fact that M enforces P by $P(M)$. *Soundness* of the security preserving rules R for a property P on diagram M can be formalized as follows:

$$P(M) \wedge R(M, \Delta) \Rightarrow P(M[\Delta]).$$

To prove that the algorithm described above is sound with respect to a given property P, we show that every set of simultaneous changes accepted by the algorithm preserves P. Then, transitively, if all steps were sound until the delta is empty, we reach the desired $P(M[\Delta])$.

One can obtain these deltas by interpreting the UMLseCh annotations presented in the previous section. Alternatively, one could compute the difference between an original diagram M and the modified M'. This is nevertheless not central to this analysis, which focuses on the verification of evolving systems rather than on model transformation itself.

To define the set of rules R, one can reason inductively by cases given a security requirement on UML models, by considering incremental atomic changes and distinguishing them according to a) their *evolution* type (addition, deletion, substitution) and b) their *UML diagram* type. In the following section we will spell-out a set of possible sufficient rules for the sound and secure evolution of class diagrams annotated with the «secure dependency» stereotype.

3.3 Application examples

In this section we demonstrate the verification strategy explained in the previous section by applying it to the case of the UMLsec stereotype «secure dependency» applied to class diagrams. The associated constraint requires for every communication dependency (i.e. a dependency annotated «send» or «call») between two classes in a class diagram the following condition holds: if a method or attribute is annotated with a security requirement in one of both classes (for example { secrecy = {method()} }), then the other class has the same tag for this method/attribute as well (see Fig. 6 for an example). It follows that the computational cost associated with verifying this property depends on the number of *dependencies*. We analyze the possible changes involving classes, dependencies and security requirements as specified by tags and their consequences to the security properties of the class diagram.

Formally, we can express this property as follows:

$$P(M): \ \forall C, C' \in M.\mathsf{Classes}\ (\exists d \in M.\mathsf{dependencies}(C, C') \Rightarrow C.\mathsf{critical} = C'.\mathsf{critical})$$

where $M.\mathsf{Classes}$ is the set of classes of diagram M, $M.\mathsf{dependencies}(C, C')$ returns the set of dependencies between classes C and C' and $C.\mathsf{critical}$ returns the set of pairs (m, s) where m is a method or an object shared in the dependency and $s \in \{\mathsf{high}, \mathsf{secrecy}, \mathsf{integrity}\}$ as specified in the «critical» stereotype for that class.

We now analyse the set Δ of modifications by distinguishing cases on the evolution type (deletion, addition, substitution) and the UML type.

Deletion

Class: We assume that if a class $\bar{C}$ is deleted then also the dependencies coming in and out of the class are deleted, say by deletions $D = \{o_1, ..., o_n\}$,

and therefore, after the execution of o and D in the model M (expressed $M[o, D]$) property P holds since:

$$P(M[o, D]):$$

$$\forall C, C' \in M.\mathsf{Classes} \setminus \bar{C}\ (\exists d \in M[o, D].\mathsf{dependencies}(C, C') \Rightarrow C.\mathsf{critical} = C'.\mathsf{critical})$$

and this predicate holds given $P(M)$, because the new set of dependencies of $M[o, D]$ does not contain any pair of the type $(x, \bar{C})$, $(\bar{C}, x)$.

Tag in critical: If a security requirement (m, s) associated to in class $\bar{C}$ is deleted then it must also be removed from other methods having dependencies with C (and so on recursively for all classes $C_{\bar{C}}$ associated through dependencies to $\bar{C}$) in order to preserve the secure dependencies requirement. We assume $P(M)$ holds, and since clearly $M.\mathsf{Classes} = (M.\mathsf{Classes} \setminus C_{\bar{C}}) \cup C_{\bar{C}}$ it follows $P(M[o, D])$ because the only modified objects in the diagram are the classes in $C_{\bar{C}}$ and for that set we deleted symmetrically (m, s), thus respecting P.

Dependency: The deletion of a dependency does not alter the property P since by assumption we had a statement quantifying over all dependencies (C, C'), that trivially also holds for a subset.

Addition

Class: The addition of a class, without any dependency, clearly preserves the security of P since this property depends only on the classes with dependencies associated to them.

Tag in critical: To preserve the security of the system, every time a method is tagged within the ⟪critical⟫ stereotype in a class C, the same tag referring to the same method should be added to every class with dependencies to and from C (and recursively to all dependent classes). The execution of these simultaneous additions preserves P since the symmetry of the critical tags is respected through all dependency-connected classes.

Dependency: Whenever a dependency is added between classes C and C', for every security tagged method in C (C') the same method must be tagged (with the same security requirement) in C' (C) to preserve P. So if in the original model this is not the case, we check for simultaneous additions that preserve this symmetry for C and C' and transitively on all their dependent classes.

Substitution

Class: If class C is substituted with class C' and class C' has the same security tagged methods as C then the security of the diagram is preserved.

Tag in critical: If the tag { requirement = method() } is substituted by { requirement' = method()' } in class C, then the same substitution must be made in every class linked to C by a dependency.

Dependency: If a «call» («send») dependency is substituted by «send» («call») then P is clearly preserved.

Example The example in Fig. 7 shows the Client side of a communication channel between two parties. At first (disregarding the evolution stereotypes) the communication is unsecured. In the packages Symmetric and Asymmetric, we have classes providing cryptographic mechanisms to the Client class. Here the stereotype «add» marked with the reference tag {ref} with value add_encryption specifies two possible evolution paths: merging the classes contained in the current package (Channel) with either Symmetric or Asymmetric. There exists also a stereotype «add» associated with the Client class adding either a pre-shared private key k or a public key K_S of the server. To coordinate the intended evolution paths for these two stereotypes, we can use the following first-order logic constraint (associated with add_encryption):

$$[\mathsf{add_encryption(add)} = \mathsf{Symmetric} \Rightarrow \mathsf{add_keys(add)} = \mathsf{k} : \mathsf{Keys} \ \wedge \\ \mathsf{add_encryption(add)} = \mathsf{Asymmetric} \Rightarrow \mathsf{add_keys(add)} = \mathsf{K_S} : \mathsf{Keys}]$$

The two deltas, representing two possible evolution paths induced by this notation, can be then given as input to the decision procedure described for checking «secure dependency». Both evolution paths respect sufficient conditions for this security requirement to be satisfied.

3.4 Tool support

The UMLsec extension [76] together with its formal semantics offers the possibility to verify models against security requirements. Currently, there exists tool support to verify a wide range of diagrams and requirements. Such requirements can be specified in the UML model using the UMLsec extension (created with the ArgoUML editor) or within the source-code (Java or C) as annotations. As explained in this work, the UMLsec extension has been further extended to include evolution stereotypes that precisely define which model elements are to be added, deleted, or substituted in a model (see also the UMLseCh profile in [126]). To support the UMLseCh notation, the UMLsec Tool Suite has been extended to process UML models including annotations for possible future evolutions[134].

Given the sufficient conditions presented in the previous sections, if the transformation does not violate them then the resulting model preserves security. Nevertheless, security preserving evolutions may fail to pass the

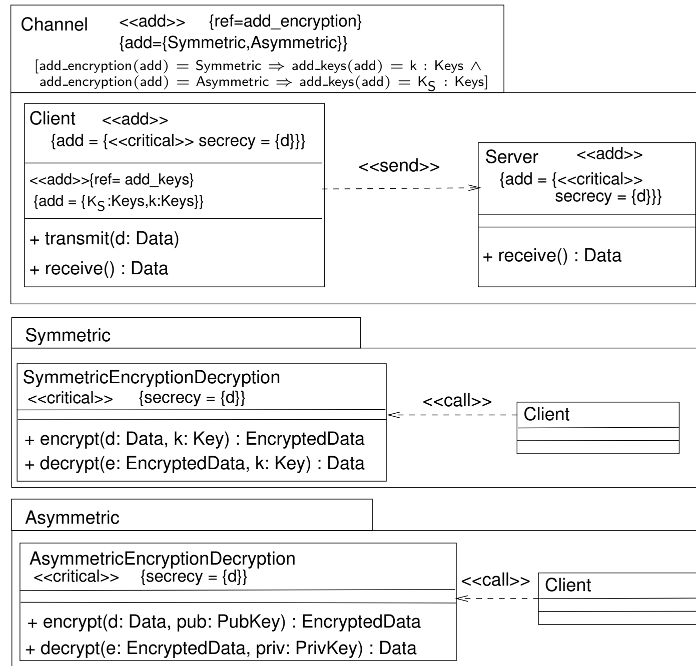


Figure 7: An evolving class diagram with two possible evolution paths

tests discussed, and be however valid: With respect to the security preservation analysis procedures, there is a trade-off between their efficiency and their completeness. Essentially, if one would require a security preservation analysis which is complete in the sense that every specified evolution which preserves security is actually shown to preserve security, the computational difficulty of this analysis could be comparable to a simple re-verification of the evolved model using the UMLsec tools. Therefore if a specified evolution could not be established to preserve security, there is still the option to re-verify the evolved model.

It is of interest that the duration of the check for «secure dependency» implemented in the UMLsec tool behaves in a more than linear way depending on the number of dependencies. In Fig. 8 we present a comparison between the running time of the verification[6] on a class diagram where only 10% of the model elements were modified. One should note that the inefficiency of a simple re-verification would prevent analyzing evolution spaces of significant size, or to support online verification (i.e. verifying security evolution in parallel to the modelling activity), which provides the motivation to profit from the gains provided by the delta-verification presented in this work. Similar gains can be achieved for other UMLsec checks such as «rbac»,

[6] On a 2.26 GhZ dual core processor

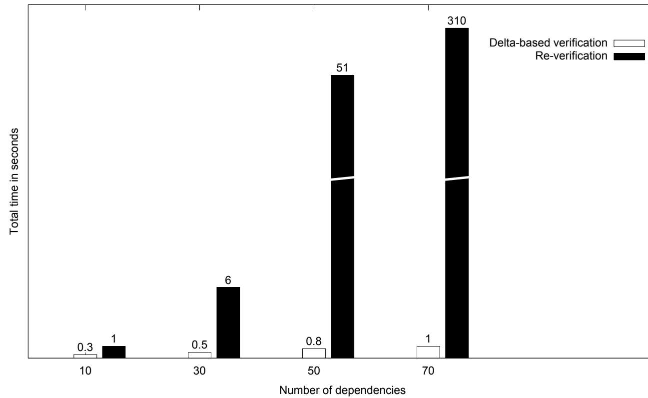


Figure 8: Running time comparison of the verification

«secure links» and other domain-specific security properties for smart-cards, for which sound decision procedures under evolution have been worked out (see [126]).

3.5 Related Work

There are different approaches to deal with evolution that are related to our work. Within Software evolution approaches, [93] derives several laws of software evolution such as "Continuing Change" and "Declining Quality". [103] argue that it is necessary to treat and support evolution throughout all development phases. They extend the UML metamodel by evolution contracts to automatically detect conflicts that may arise when evolving the same UML model in parallel. [127] proposes an approach for transforming non-secure applications into secure applications through requirements and software architecture models using UML. However, the further evolution of the secure applications is not considered, nor verification of the UML models. [74] discussed consistency of models for incremental changes of models. This work is not security-specific and it considers one evolution path only.

Also related is the large body of work on software verification based on assume-guarantee reasoning. A difference is that our approach can reason incrementally without the need for the user to explicitly formulate assume-guarantee conditions. In the context of Requirements Engineering for secure evolution there exists some recent work on requirements engineering for secure systems evolution such as [133]. However, this does not target the security verification of evolving design models. A research topic related to software evolution is software product lines, where different versions of a software are considered. For example, Mellado et al. [101] consider product lines and security requirements engineering. However, their approach does not target the verification of UML models for security properties. Evolving Architectures is a similar context with a different level of abstraction. [58]

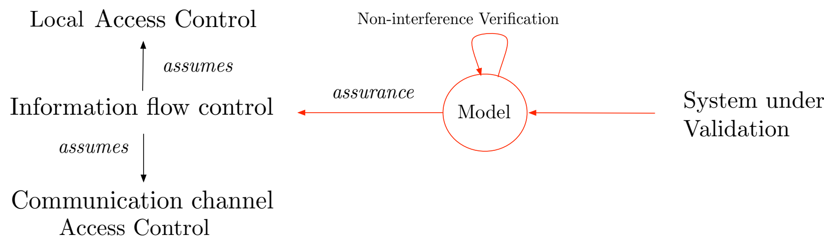
discusses different evolution styles for high-level architectural views of the system. It also discusses the possibility of having more than one evolution path and describes tool support for choosing the "correct" paths with respect to properties described in temporal logic (similar to our constraints in FOL). However, this approach is not security specific. On a similar fashion, but more focused on critical properties, [104] also discusses the evolution of Architectures.

The UMLseCh notation is informally introduced in [81], however no details about verification are given. Both the notation and the verification aspects are treated in more detail in the (unpublished) technical report [126] of the SecureChange Project. Note that UMLseCh does not aim to be an alternative for any existing general-purpose evolution specification or model transformation approaches (such as [69, 24, 33, 120, 86]) or model transformation languages such as QVT [7] or ATL [8]. It will be interesting future work to demonstrate how the results presented in this work can be used in the context of those approaches.

[7]Query/View/Transformation Specification http://www.omg.org/spec/QVT/
[8]The ATLAS Transformation Language http://www.eclipse.org/atl/

Chapter 4

Non-interference on UML state-charts



Secure Information Flow analysis is a fine-grained methodology for studying the confidentiality and integrity of systems. This kind of analysis (first introduced by Goguen and Meseguer [61] in 1982) is mathematically defined over the inputs and outputs visible to groups of users. Its main advantage over other security analysis methods is that it allows one to pin down subtle flows of information, usually known as covert-channels, that are difficult to spot when focusing merely on analysing security mechanisms (such as access control mechanisms). Although information flow properties assume perfect access control to guarantee that different groups of users do not see certain inputs and outputs of other users directly, this is a reasonable assumption: in fact attackers usually exploit the information that is shared by victims through common interfaces instead of trying to break directly the access control mechanisms. In words of Anderson [23] "(...) in practice, security is compromised most often not by breaking dedicated mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being used".

In the past decades different information flow properties have been proposed for coping with different system models such as non-deterministic systems, distributed systems and imperative programming languages. At the

abstract level results about compositionality and refinement have been published for many security properties, for example [95, 96, 113]. In the 'Language Based' realm (i.e. analysis of source code) mature tools for information flow analysis on annotated code exist, like [4, 60, 6]. All of these are indeed promising steps towards the industrial application of the fine-granular analysis offered by the property-centric point of view of Information Flow.

Nevertheless, it seems that production environments are still far from adopting these techniques. Although formal methods can give very precise guarantees about the behaviour of systems , it also should be possible to benefit from their insights by non-experts in the field. In the case of information-flow, many of the verification results at the design-level require a high-level of mathematical sophistication that is unreasonable to expect to be achievable by a regular software developer or security expert.

In this chapter we propose a light-weight, automatic strategy for checking non-interference on a deterministic fragment of UML state-charts. Our aim is to make a formally sound step towards the usability of these techniques based in the so-called *unwinding* theorem, that provides sufficient conditions for non-interference. We have extended previous work on unwinding to cope with the complexity of UML state-charts: the use variables for keeping history of the state, guards for transitions, hierarchical states and actions.

Moreover, we aim at verifying systems where object interaction plays a fundamental role. To achieve this, we discuss sufficient conditions for deciding on the composition of the behaviour of already verified components. This is a key factor in the scalability of our approach, which is also an important criterion for the success of verification in realistic settings.

To validate our theoretical results, we report on a prototypical machine implementation that automatically verifies models where our unwinding theorem is applicable. We apply this implementation to examples motivated by a case study from the Smart Grid domain. Since unwinding conditions are only sufficient conditions, some secure models might be rejected. However it is important to show that non-trivial secure models are actually accepted and verified. The case study allows to discuss and validate the utility of our approach. .

4.1 Preliminaries

In this section we recall some definitions and set the notation for the rest of the chapter. Starting with the original definition of non-interference by [61] many other subtle information flow properties have been proposed (mainly for dealing with non-determinism and distributed systems). In this work we will nevertheless focus on the original definition for deterministic systems, because our focus will be the analysis of deterministic automata.

4.1.1 Non-interference

Assume a system is a deterministic black-box transforming sequences of input events I into sequences of output events O by means of a semantics function:

$$[\;] : I \to O$$

We further assume there are two types of users : *high* users H and *low* users [1] L. The sets of input and output events can be divided into the events a high or low user is allowed to see. Lists of input and output events can then be filtered according to the type of user allowed to see them by the purging functions $\cdot|_H$ and $\cdot|_L$. Non-interference is the property :

$$\forall \; \overrightarrow{i} \;\; [\overrightarrow{i}]|_L = [\overrightarrow{i}|_L]|_L \tag{4.1}$$

In other words, the output seen by the lower users is independent of the input by higher users, up to the point that is not even noticeable whether the high users perform any action on the system.

Some authors (for example [77]) use the equivalent definition (see Appendix A for a proof):

$$\forall \; \overrightarrow{i_1}, \overrightarrow{i_2} \;\; \overrightarrow{i_1}|_L = \overrightarrow{i_2}|_L \Rightarrow [\overrightarrow{i_1}]|_L = [\overrightarrow{i_2}]|_L \tag{4.2}$$

which corresponds to the intuition that two runs where the high user perform different actions are equivalent to low users. A stronger version of this property is usually used in the language-based information flow analysis domain [67, 28, 59].

4.1.2 State-charts

To model the function $[\cdot]$ of the last subsection, consider *Mealy machines* [100]. Syntactically, a Mealy machine can be represented by a directed graph with annotated transitions of the form α/β, meaning that the input event α triggers the output event β. Formally, a Mealy machine M is defined as a 6-tuple $(S, s_0, \Sigma, \Gamma, T, G)$ where S is a finite set of states, s_0 is an initial state, Σ is finite input alphabet, Γ is a finite output alphabet, $T : S \times \Sigma \to S$ is a transition function defined over states and input symbols and $G : S \times \Sigma \to \Gamma$ is an output function defined over states and input symbols. A Mealy machine induces thus a semantics $[\cdot]$ by:

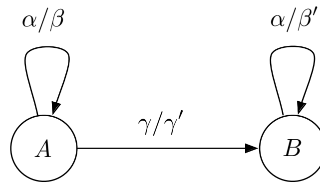
$$[(\sigma_1, \sigma_2, \ldots, \sigma_n)] = G(s_0, \sigma_1) :: \cdots :: G(s_n, \sigma_n)$$

[1] For simplicity of exposition and historical reasons we will discuss about high and low users in the rest of the chapter. However the definition can be extended to an arbitrary partition of groups of users. Also we will restrict to analysing non-interference from high with respect to low (no-down-flows, usually associated to confidentiality), to analyse the converse (i.e. integrity) one can just switch L for H

where $s_n = T(\ldots T(T(s_0, \sigma_1), \sigma_2) \ldots, \sigma_n)$ and :: denotes concatenation. Notice that the functions T and G are naturally induced by the graph representation. In the following we will assume that if an input is not defined in a given state, the machine enters in a state where no further inputs are processed and no outputs are produced.

If we further divide the input and output events into high and low events, we can apply the definition of non-interference to a Mealy machine.

Example 1. *Consider the system defined by the state-machine:*



where $\alpha, \beta, \beta' \in L$ and $\gamma, \gamma' \in H$. Then non-interference does not hold since:

$$[(\gamma, \alpha)]|_L = \beta' \neq [(\gamma, \alpha)|_L]|_L = [\alpha]|_L = \beta$$

To deal with the so-called 'state-explosion' problem, that arises when the number of states and transitions increases due to the specification of complex behaviours, other formalisms have been proposed that include the notion of sub and super-states. More prominently Harel [68] proposed the notion of *statechart* that has been used as the basis for UML. This is basically an extension to Mealy machines that allows the following:

Hierarchical states: Single states can contain sub-states and transitions among the sub-states up to arbitrary depth. Let A be a super state containing finitely many sub-states A_i. Then an external state B can have a transition directly to s or to a sub-state A_i. In the first case the transition is to be interpreted as to go to the initial sub-state of s. In the second case it simply goes to s_i. This allows to modularize certain common behaviours into super-states, improving considerably the presentation of complex state charts.

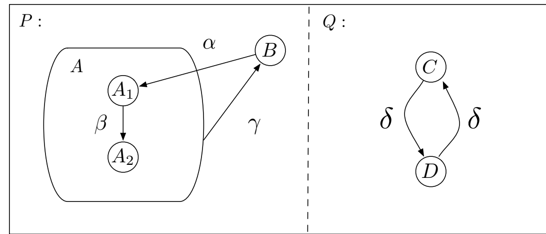
Clustering: To graphically summarize events that trigger a transition to the same state in a group of states, a transition with event γ going out of a super-state A to state B stands for a transition from each sub-state of A with event α to B.

Orthogonality and concurrency: In some cases, processes within the system are orthogonal between each other, in the sense that they could be described with two separate state-charts with disjoint inputs. Thus, Harel state-charts allow multiple sub-state-charts to be modelled as concurrent processes within the same state-chart, compressing notably the notation, since for each two independent Mealy machines with n and m states respectively,

$n \cdot m$ states are needed to represent them in a single machine.

Thus formally, a Harel state-chart can be seen as a set (to represent the concurrent processes) of 6-tuples $(S_i, s_0^i, \Sigma_i, \Gamma_i, T_i, G_i)$ where S_i is as a finite set of super-states and s_0^i is an initial super-state. A super-state is defined as a either a state or a state-chart, such that for every super-state there are only finitely many nested state-charts. T_i and G_i are then similar as in the Mealy machine case, where for a given state T_i depends also on the transitions defined at higher hierarchical states (if any).

Example 2. *The following state-chart:*



contains a (sub) state-chart P containing a superstate with clustering running concurrently with (sub) state-chart Q.

Notice that all these extensions are syntactic sugar for improving the graphical representation: any deterministic Harel state-chart can be represented by a Mealy-Machine with equivalent semantics, and therefore we can use the same definition of non-interference given in the previous subsection for reasoning about the security of Harel State-charts.

4.2 Verification Strategy

Verifying system designs for non-interference is a computationally difficult task, because the definition uses universal quantifiers on inputs and outputs: to verify accurately an arbitrary system implies running and comparing all possible input sequences. Therefore, to achieve a trade-off between security and efficiency, one usually needs to sacrifice some precision on the verification. In this section we discuss how to obtain sufficient conditions for the non-interference analysis on UML state-charts by extending traditional unwinding theorems for finite state machines. We will first introduce the fragment of the UML state-charts considered and discuss briefly the unwinding theorem. Then we report on our extension for UML state-charts.

4.2.1 UML state-charts à la UMLsec

UML has adopted an extension of Harel state-charts to represent the behaviour of classes. It allows a list of *actions* as a consequence of an event,

including calling methods, updating variables and outputting values. In this work we will restrict to a fragment of UML state-charts defined as follows :

- Input events labelling transitions can be either methods of the associated class with concrete parameters or with variables to represent calls with different parameters or global system events (like the tics of a system clock).
- Actions associated to an input event can be either outputting an event (written `return event`) or a variable assignation, where the variables are attributes of the associated class or parameters of the input.
- Guards are decidable conditions on the input parameters or the values of the attributes.

We will restrict to the sub-set of deterministic UML State-charts as defined above. This is similar to the UML state-charts as defined in [77], with the fundamental difference that there state-charts can be non-deterministic, resulting in a more complex semantics. The semantics of the deterministic fragment defined above can be seen as an extension of the Harel state-chart semantics (based on their Mealy machine translation), where the guards and variables are syntactic sugar for describing the history of the state and where parametrized method calls stand for as many transitions as the respective guard allows.

More precisely: a transition labelled with a parametric input stands for multiple transitions, one for each concrete value of the parameter. Transitions where the actions perform variable assignation stand for multiple transitions with distinct targets (one for each possible value of the assignation). Guards with condition C represent the fact that in states where C hold then that particular labelled transition is present, and not present in states where C does not hold. For an example of a UML state-chart and its semantically equivalent Mealy machine see Fig. 1. We will discuss this example in detail in Sect. 4.4.

4.2.2 Unwinding

Unwinding theorems were first proposed by Goguen and Meseguer [62]. They provide sufficient conditions for non-interference that are efficient to verify since they rely basically in local conditions of pairs of states. The basic idea is that if there exist a reflexive relation R of the states in an input/output state machine M for a policy dividing events in high H and low L such that:

R is locally consistent: given a state s and $T(s,h)$ the state resulting after a transition triggered by an event $h \in H$ then $(s,s') \in R$. Formally:

$$\forall_{s\in S,h\in H}\ (s,T(s,h)) \in R$$

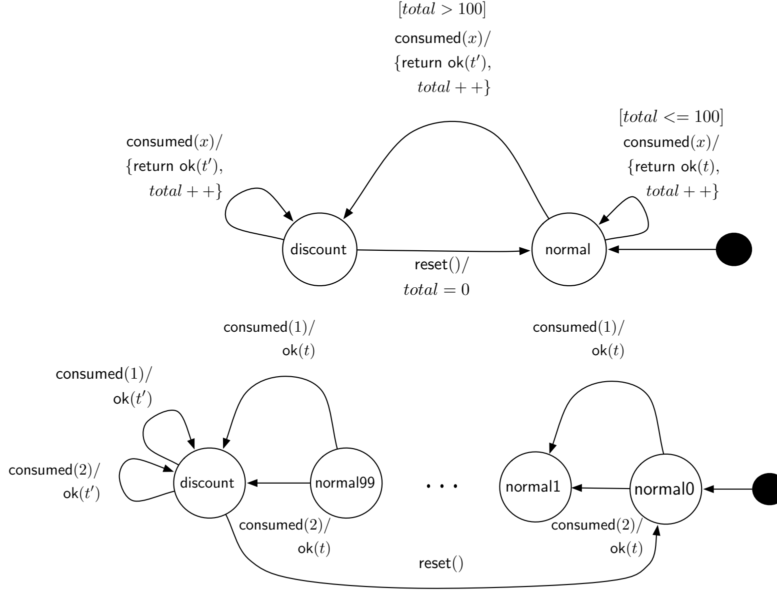


Figure 1: A model of the smart metering payment processing system as an UML state-chart and a semantically equivalent Mealy Machine for x in range [1..2] in the parameters of consumed.

R **is step-consistent:** for all inputs i, if $(s_1, s_2) \in R$ then $(T(s_1, i), T(s_2, i)) \in R$ where $T(s_1, i)$ and $T(s_2, i)$ are the states resulting from the transition triggered by i in s and t:

$$\forall_{s_1, s_2 \in S, i \in \Sigma}\ (s_1, s_2) \in R \Rightarrow (T(s_1, i), T(s_2, i)) \in R$$

R **is output-consistent:** if $(s_1, s_2) \in R$ then the output of an event $l \in L$ in s_1 is equal to the output of l in s_2:

$$\forall_{s_1, s_2 \in S,\ l \in L}\ (s_1, s_2) \in R \Rightarrow G(s_1, l) = G(s_2, l)$$

then non-interference holds on M for the H and L partition. For a proof see for example [62, 122].

Example In example 1, (A, B) must be in R because of locality. However, R is not output consistent and therefore it can not be concluded that the system is secure (in fact we already showed it is not).

4.2.3 Unwinding for UML Statecharts

There are two main difficulties for extending the unwinding theorems from Mealy machines to the subset of UML state-charts as described in Sect. 4.2.1: a) The graphical syntactic sugar of Harel state-charts and b) the use of variables and guards for keeping history of the state and for parametrizing inputs. One possibility to verify UML state-charts would be to remove all syntactic sugar, unfold a semantically equivalent Mealy machine and then find an unwinding relation satisfying the conditions described in the previous subsection. This would be however computationally quite expensive in general: the purpose of the UML state-chart notation is to avoid state and transition explosion. We have extended the unwinding theorem accordingly for coping soundly with these differences in the notation in an efficient way. Intuitively, we statically analyse guarded transitions with actions by simultaneously extending an unwinding relation and a tainted set associated to each state. Tainting keeps track of variables whose value is directly or indirectly dependent on high inputs, in the spirit of language based information flow analysis. This information allows to soundly decide on the output consistency of the relation.

In the following when we refer to a state, we mean a state that does not contain further nested sub-states. When we refer to the transitions going out of a state s, we mean all transitions going out of all the super states containing s. Without loss of generality we will analyse concurrent state-charts separately: by definition (Sec. 4.1) two concurrent Harel state-charts have disjoint inputs, and we further assume they also do not share variables in their UML representation.

Let R' be a relation over the states of a UML state-chart U, H a subset of the inputs of U and $\mathsf{tainted}(s_i)$ a set associated to each state s_i, such that:

Local consistency For a label on the transition t_1 from s_1 to s_2 of the form

$$[C_1]\ \alpha(y_1)\ /\ \{\mathsf{return}\ \beta_1,\ x_1 := E_1\} \tag{4.3}$$

then s_1 is in relation with the initial sub-state of s_2 if there exists a parameter a such that $\alpha(a) \in H$. Moreover $x_1 \in \mathsf{tainted}(s_2)$ and $\mathsf{tainted}(s_2) \supseteq \mathsf{tainted}(s_1)$.

Step consistency If $(s_1, s_2) \in R'$ then for every transition t_1 of the form (4.3) with target s_1' originating from s_1 and every transition t_2:

$$[C_2]\ \alpha(y_2)\ /\ \{\mathsf{return}\ \beta_2,\ x_2 := E_2\} \tag{4.4}$$

with target s_2' originating from s_2 then it follows $(s_1', s_2') \in R'$. Moreover, if $\alpha(a) \in H$ for some a or there is a variable $z_i \in \mathsf{tainted}(s_i)$ such that $z_i \in C_i$

or $z_i \in E_i$ then $x_i \in \mathsf{tainted}(s_i')$ and $\mathsf{tainted}(s_i') \supseteq \mathsf{tainted}(s_i)$.

Output consistency If $(s_1, s_2) \in R'$ with t_1 of form (4.3) with a such that $\alpha(a) \in L$ we distinguish two cases:

- If there exists x such that $x \in \mathsf{tainted}(s_1)$ and $x \in C_1$ then for all t_2 in s_2 of form (4.4) it must follow $\beta_1 = \beta_2$.
- Otherwise: if there exists t_2 of form (4.4) in s_2 such that $C_1 = C_2$ then $\beta_1 = \beta_2$. If no such t_2 exists, then for all other t_2' in s_2 of form (4.4) it holds $\beta_1 = \beta_2$.

Moreover there exists no variable x such that $x \in \mathsf{tainted}(s_i)$ and $x \in \beta_i$.

Theorem 1. *If U admits a relation R' as defined above then it respects non-interference.*

Proof. It suffices to show that the relation R induced by R' on the unfolded Mealy machine M of U is an unwinding relation. A state in M can be seen as a pair $(s, \overrightarrow{v})$ where s is an identifier for a state in U and $\overrightarrow{v}$ is a vector of concrete values $v_1, \ldots, v_n$ for the variables $x_1, \ldots, x_n$ used in U. R is defined thus as $((s_1, \overrightarrow{v}), (s_2, \overrightarrow{w})) \in R \Leftrightarrow (s_1, s_2) \in R'$. Is easy to see that R satisfies local consistency, because R' covers all possible transitions induced by high inputs. Step consistency also holds on R by construction of R'. The extended definition of output consistency is similar to the original one, except for a) it is forbidden to output an expression depending on a tainted variable and b) the output consistency relation is relaxed in case an output is guarded by a condition not depending on tainted variables. It is not hard to see that a) is a necessary condition. Now consider $(s_1, s_2) \in R'$ and w.l.o.g. belonging to the same connected graph of U. Moreover consider a condition C depending on variables $X' = x_1', \cdots, x_n'$ such that $x_j' \notin \mathsf{tainted}(s_i)$. By the definition of R' there exist ancestors p_1 and p_2 (of s_1 and s_2 respectively) such that there is a high transition between p_1 and p_2 and by definition of tainting this transition does not change the value of any variable in X'. For any input η changing the state of p_1 and p_2 to p_1' and p_2' respectively then if $\eta \in H$ then the valuation of X' remains unaltered. If $\eta \in L$ triggers an action changing the value of a variable in X' then the transition was triggered on a condition depending on variables in X'. By hypothesis variables in X' had the same value on p_1 and p_2, and therefore η changes the valuation in both states equivalently. The same reasoning can be done inductively obtaining that the values of X' in s_1 and s_2 depend on the values of X' in p_1 and side-effects triggered by low inputs exclusively. Therefore if C holds in $(s_1, \overrightarrow{v})$ for a given input trace starting on p_1, then it must also hold in $(s_2, \overrightarrow{w})$ for an equivalent trace on the low inputs that reaches s_2.

Notice that it would be also sound to simply compare all outputs in s_1 and s_2, but this would be too coarse for practical uses, where usually a condition

and its negation are defined as guards for the same input on a given state, as we will see in Sect. 4.4. It suffices to compare the outputs guarded by C and not both the outputs of C and those of $\neg C$, because in the unfolded Mealy machine the states where $\neg C$ holds are not necessarily in the minimal unwinding relation.

□

4.3 Object interaction

As discussed in Sect. 4.2.1 UML state-charts are commonly used to represent the behaviour of a class. In the previous section we have discussed unwinding theorems that can be used to decide on the security of single, monolithic state-charts. To reason about the security of a system that is built upon interacting objects, we would need to obtain composed state-charts out of the state-charts defining the single object's behaviour. It would be however desirable to have sufficient conditions that allow to reason on policies on the single components mainly for achieving scalability. In this section we discuss the notion of composition we will use and present sufficient conditions that guarantee that a composition respects non-interference for a given policy.

4.3.1 Composition

We will follow [77] by reasoning at the instance level: we will assume that the behaviour described by a state-chart is that of an instantiated object [2].

The notion of compositionality we will use is based on message passing between state-charts: the output messages generated by a state-chart A can be input messages for a state-chart B but not vice-versa. This corresponds formally to a special case of parallel composition as defined for example in [38, 105] where we restrict the feedback only to occur in one direction. In other words we do not allow call-backs, which are related to recursive method calls. This is indeed a difficult topic on its own, since subtleties on the semantic play a fundamental role (as discussed for example in [130]), and goes outside the scope of this work. Nevertheless, the composition notion defined here is useful to reason about the security of non-trivial object interaction, as we will see in Sect. 4.4, and has nice preservation properties for non-interference.

More precisely, let classes A and B with inputs I_A and I_B respectively and outputs O_A and O_B. A and B are composable if $O_A \cap I_B \neq \emptyset$ and $O_B \cap I_A = \emptyset$. The resulting composed object has inputs $I = I_A \cup (I_B \setminus O_A)$ and outputs $O = O_B \cup (O_A \setminus I_B)$. Semantically, $A \otimes B$ is defined by the product of the states in A and B where the states with at least one output $o \in O_A$ such

[2] Therefore if we want to reason about different instantiations of an object we would need to define as many classes as desired objects.

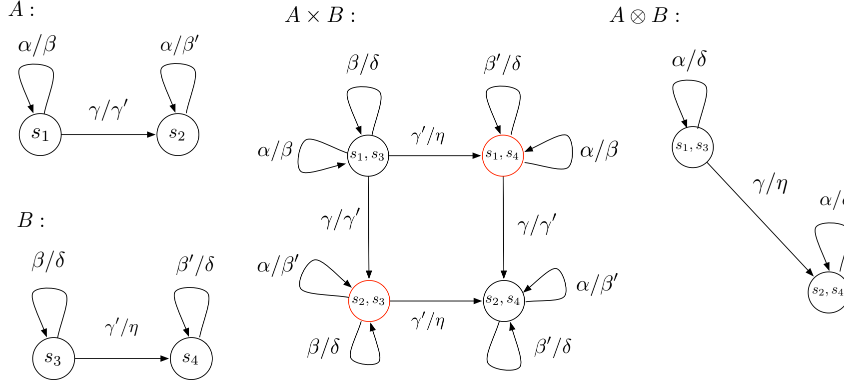


Figure 2: State-chart U_B and its composition with the state-chart U_A

that $o \in I_B$ cannot be processed by B are discarded, because synchronisation cannot take place. This is similar to the notion parallel composition in CCS [105]. The outputs of transitions of A matching inputs of B in every state are then replaced by the induced outputs in B.

Example 3. *Consider the state-charts A and B of Fig. 2. Although in principle there exist four possible states in the product state-chart $A \times B$ we discard the states where an output of A that is in the interface of B cannot be processed by B.*

4.3.2 Compositionality and non-interference

In general, for scalability reasons, it is desirable that verification results on single components can be re-used efficiently for deciding on their composition. In our setting this means, for a given partition of the set of inputs $I = I|_H \cup I|_L$ and outputs $O = O|_H \cup O|_L$ of the composition $A \otimes B$ there exists sufficient conditions on A and B such that this composition respects non-interference. This is notably not the case in general for information flow properties [95]. However, in our case we can derive a positive result in this sense. We first observe that given a policy on a composition $A \otimes B$, the events on $I_B \cap O_A$ remained unspecified since they are not part of the interface of the composition. Although formally possible, it is not sound from a security point of view to mark events from $I_B \cap O_A$ as high in one component and low in the other (or vice-versa), and therefore we will exclude that possibility in the following.

Theorem 2. *Let $I = I|_H \cup I|_L$ and $O = O|_H \cup O|_L$ a partition of the input and output alphabets of $A \otimes B$. If non-interference holds for an extension of the policy in I and O to the unspecified events in $I_B \cap O_A$ in A and B, then non-interference holds on $A \otimes B$.*

Proof. First consider the case where $O_A = I_B$ (sequential composition). If there exists a sequence $\overrightarrow{i}$ of inputs of $A \otimes B$ such that $[\overrightarrow{i}]_{A \otimes B}|_L \neq [\overrightarrow{i}|_L]_{A \otimes B}|_L$. Then, because of sequentiality and subsequent application of non-interference of B followed by non interference of A and of B again:

$$[\overrightarrow{i}]_{A \otimes B}|_L \stackrel{S}{=} [[\overrightarrow{i}]_A]_B|_L \stackrel{B}{=} [[\overrightarrow{i}]_A|_L]_B|_L \stackrel{A}{=} [[\overrightarrow{i}|_L]_A|_L]_B|_L \stackrel{B}{=} [[\overrightarrow{i}|_L]_A]_B|_L \quad (4.5)$$

now observe that:

$$[\overrightarrow{i}|_L]_{A \otimes B}|_L \stackrel{S}{=} [[\overrightarrow{i}|_L]_A]_B|_L \quad (4.6)$$

but by hypothesis (4.5) $\neq$ (4.6), contradiction.

The other cases follow easily by observing that whenever an input i_B of B is not an output of A then A can be extended by adding a single non connected stated with a transition i'_B/i_B, thus returning to the sequential case and without harming the sufficient conditions (and similarly when output o_A is not an input to B).

□

Notice that the hypothesis of Theorem 2 although sufficient, are not necessary: in fact, the Example 3 has a component A violating non-interference for $H = \{\gamma, \gamma'\}$, but the composition $A \otimes B$ respects it for $H = \{\gamma, \eta\}$

4.4 Validation

In this section we report on experiments made to implement the enhanced unwinding technique and the compositionality theorem and apply them on examples from our case study.

4.4.1 Tool support

There are two basic strategies to construct the relation R' on a given statechart. One possibility is to proceed top down: first put in relation all the states that respect output consistency and then check for local consistency and step consistency. It is however not clear how to proceed from there if the relationship does not respect the unwinding conditions. We have opted to construct it bottom up: first, put every state in relation with itself. Then we compute all relationships due to local consistency, and subsequently for each pair, we enlarge R' by step consistency. When constructing R' we do a preliminary taint analysis, that only holds for forward tainting but could be imprecise in presence of loops. However it was enough to evaluate our examples, where tainting occurs only in one step (more accurate taint analysis are matter of current work). Finally we check for output consistency. If

output consistency does not hold, we know an unwinding relationship cannot be built, because there exists no minimal one.

We have prototypically implemented the algorithm described in Sect. 4.2 in Haskell [7] because of its compact and elegant syntax. A state-chart is represented as a pair of type list of nodes and list of transitions where the nodes contain the tainting set and a list of 5-tuples:

```
type Node = (Label,Tainted)
type Transition = (Condition, Input, Output, Origin, Target)
type StateChart = (Nodes,Transitions)
```

For example, to check for output consistency of a pair in R' with respect to a set of low inputs `low` we have implemented the following code:

```
compareLowOutput :: StateChart -> Low -> (Node,Node) -> Bool
compareLowOutput (nodes,transitions) low (x,y) =
                 ( null[(tran1,tran2) | (tran1,tran2) <-lowTransitions,
                 (getInputMethod tran1 == getInputMethod tran2),
                 (getReturn tran1 /= getReturn tran2)] )
            where lowTransitions = (getLowTransitions transitions low x y)
```

where `getLowTransitions` is defined as the filtering function including the exception based on the taint analysis:

```
getLowTransitions t l x y = [(tran1,tran2) | tran1 <-t , tran2 <-t,
                            (getLabelOrigin tran1 == getLabel x),
                            (getLabelOrigin tran2 == getLabel y),
                            elem (getInputMethod tran1) (map fst l),
                            (or [isInfixOf z (getCondition tran1)| z <-(snd x)]
                            || getCondition tran1 == getCondition tran2 )]
```

4.4.2 Case study

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[107]. 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 [124]. The importance of the smart grid for the society is due to the expectation that it will help optimize the use of renewable energy sources [118] and minimize the collective environmental footprint [49]. Two basic reasons why the attack surface is increasing with the new technologies are:

- The Smart Grid will increase the amount of private sensitive customer data available to the utility and third-party partners.
- Introducing new data interfaces to the grid through meters, collectors, and other smart devices create new entry points for attackers.

Among other requirements, confidentiality and privacy of user data is an important security issue. There are many privacy issues, related to the use of sensitive personally identifiable information (PII) related to the consumption of energy, the location of the electric car, etc. This data must be kept secure from unauthorized access, and the measurement process is subject to strict lawful requirements in terms of accuracy, dependability and security, see in particular the European Union directive "Measuring Instruments Directive 2004/22/EC (MID)" [12], published 2004-05-31. See also [72] for a current version of proposed technologies to solve this power systems management and associated information exchange issues.

In the following we will model two scenarios in this domain (for details see [108]).

Scenario 1 Consider for example the behaviour described in Fig. 1. This models an energy provider that processes the amount energy a user x consumes, described by the event $\mathsf{consumed}(x)$ (x is a positive integer, the user's ID) representing one unit of energy consumed. After one unit is consumed, a confirmation $\mathsf{ok}(t)$ with the price t of the consumption is sent to the user. If all consumers of a given region consume more than 100 units, the price of the unit drops from t to t'. Now, assuming that a given user with id 1 is not supposed to know about the consume of other users, how can we check whether this requirement holds for this system?

By setting the events $\{\mathsf{consumed}(x) \mid x \neq 1\}$ as high, we can check whether the unwinding conditions hold automatically via our Haskell implementation (for example input see Appendix B). In this case, our automatically computed minimal unwinding relation rejects the model because of output inconsistency. Because unwinding only provides an approximation, it would still be possible that the system is secure. In this case the system turns out to be is insecure: by just seeing a difference in the reported price in two consumptions (given that he does not consume himself more than 100 units), the low user could infer bounds on the number of consumptions of his neighbours. In this case, because of a trade between of security and functionality, it is difficult to modify the model in order to obtain a secure one. This can be done formally by providing two input sequences varying on high and showing that the low outputs differ. Unfortunately this has to be done manually.

A possibility to obtain a positive security guarantee is to modify the model as follows: a discount is given if a single user consumes more than 100 units. By modelling then two concurrent machines, one accepting only the

inputs of the low user and the other from high, we can positively prove the security of the model.

Scenario 2 Now consider the following scenario: an electric vehicle buys power to a given provider at an agreed priced. It does so at a public recharging station. For convenience, the car will automatically stop the recharging when the total consumption exceeds a given price (for example 10 €) or when the constant capacity k is reached. The behaviour of the single components and their composed model is depicted in Fig. 3, where for illustrative purposes also the composition is spelled out. To fulfil privacy requirements of both users and companies, it is desirable that the recharging stations do not learn the single unit price of the energy sold to the vehicle. In other words, we want to treat the events $\mathsf{setPrice}(x)$, $\mathsf{readPrice}$ and $\mathsf{getPrice}$ as confidential for the user represented by the charging station. All other events are public. We use then Theorem 2 and proceed to verify the single components. In this case, the behaviour of object V is already violating non-interference, so we cannot positively verify the composition $P \otimes V \otimes C$. However, if we no longer allow the user to have a recharging policy that is dependent on the price by replacing $[t = 10/p]$ with $[t = k]$, then we can verify the composed model automatically as secure, because all components respect the information flow policy.

4.5 Related Work

Starting with the work of Goguen and Meseguer [61], many information-flow properties have appeared for specific system models and to capture different notions of security. Rushby discusses unwinding theorems in a more modern notation [122] along with transitive vs. intransitive information flow policies. General Unwinding theorems for a wide range of information flow properties have also been suggested by Mantel in [62]. Mantel has also unified most of these properties into a common framework, the Modular Assembly Kit for Security MAKS [96], also deriving new unwinding theorems. This work is also probably the best reference for a discussion on the different properties proposed for abstract non-deterministic and distributed system designs.

In the Language-based world, different static approaches have been suggested for verifying information-flow properties, prominently type-based systems like Volpano-Smith [136] or more recently Barthe et al. [28]. Also works based on abstract interpretation and analysis of Program Dependency Graphs [59, 67] give approximations to non-interference for JavaCard-bytecode and Java respectively. Tools for information flow analysis on annotated code using these techniques are for example Jif [4], JOANA [60] and STAN [6]. Works in the language-based domain, in particular program slicing, are related with our analysis, however the non-interference definition at the code

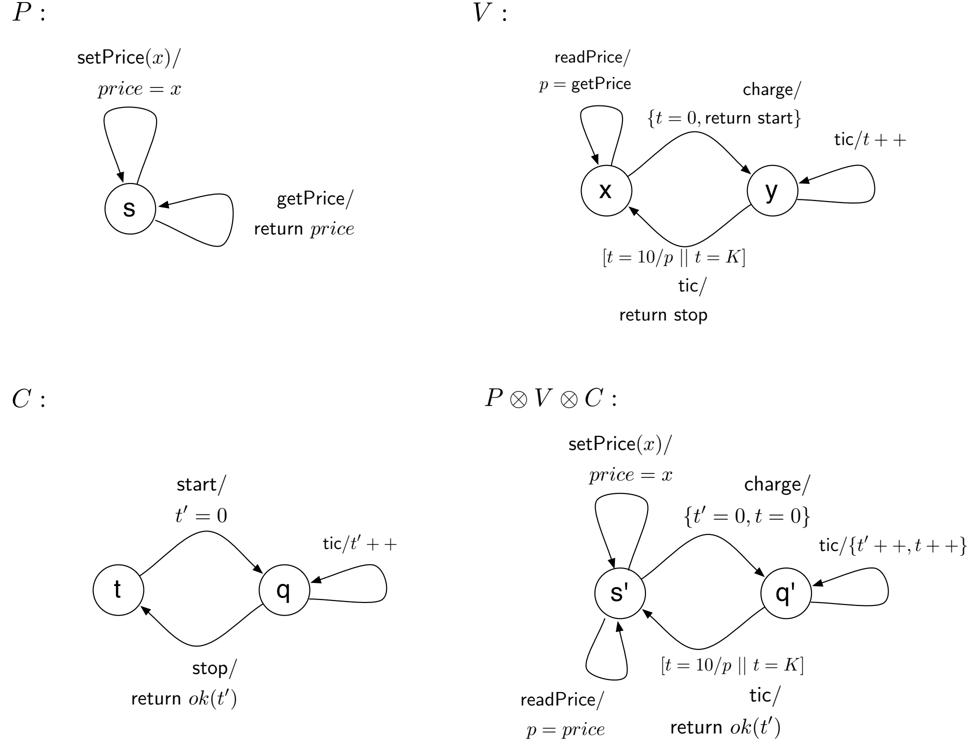


Figure 3: Single components and composition of the Provider P, the Vehicle V and the Charging station C

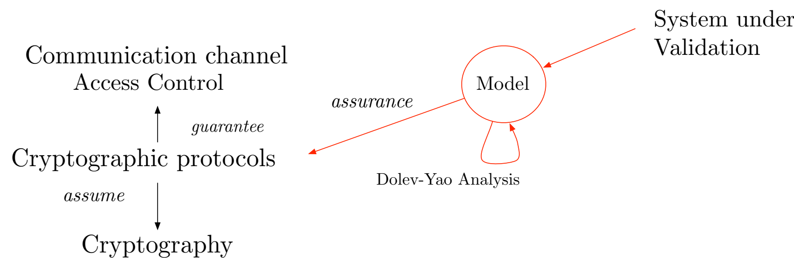
level is generally a stronger property: the language-based definition quantifies over high and low variables, formally seen as inputs at the initial system state and outputs at the final. It is required that for arbitrary fixed low inputs (the valuation of the low variables in the initial state) the final state of the low variables does not change for different initial values of the high variables. Clearly, this implies non-interference, but it is a particular case for a non reactive system model, where inputs at the initial program state determine all outputs in all subsequent states.

Jürjens [77] defined a stereotype for non-interference on state-charts that is equivalent to the notion used in this chapter for the deterministic case, but no verification strategy or compositionality results are discussed. In [20] Alghathbar et al. model flows of information with UML Sequence diagrams and Horn clauses. However their focus is on high-level information flow policies where only actors and the messages their exchange are modelled, and no explicit relation between the information control rules and a semantic property is given. To the best of our knowledge, there exist no works extending unwinding theorems for UML state-charts that consider parametrized events,

guards and actions with side-effects.

Chapter 5

Security protocols



In this chapter we further explore the question of compositional verification, this time for a confidentiality notion that can be defined over composed processes. In general, the composition of protocols can cause unforeseen problems (see for example problems on the SAML based single-sign-on used by Google in [26]). The composition of security properties has been explored substantially for protocols [64, 50], where most of the work focuses on assume/guarantee reasoning. In this chapter we will follow [75] where this question is studied for a stream-based representation of processes exchanging messages for which Dolev-Yao secrecy is required. In particular we will focus in the composition of processes exchanging cryptographic messages (for example a server S composed with a client C) and not in protocol composition in the standard sense of the literature. However our results could be used to reason about standard protocol composition as well, as we will discuss in the following.

We propose a methodology to specify protocols such that given a finite set of session variables (keys, nonces and principals), compositionality of processes exchanging cryptographic messages is decidable in an algorithmical way. This is equivalent to restrict the analysis of processes to finitely many runs. Indeed vulnerabilities in authentication protocols have been shown to be limited to finitely many parallel instantiations [129]. Technically, our analysis generates finite *dependency trees* that can be stored for further deciding

on future compositions. The process of merging such trees can be shown to be empirically more efficient than re-analysing the composition from scratch, and constitutes our central contribution. Moreover, this process is relatively sound and complete with respect to the First Order Logic analysis of [79].

To validate our approach we have implemented our algorithm as an extension to the UMLsec Tool Suite. On the one hand this validates the usability of the approach in a formally sound Software Development process, and extends previous work in the area. On the other hand this has allowed us to measure the efficiency of our approach given the derivation trees for up to 500 small components (amounting to about 1000 messages). Although we do not claim that our approach is more efficient than state of the art protocol verification tools, we believe that the techniques described in this chapter could be also used in other tools/algorithms.

5.1 Preliminaries

This work is based on previous work by Jürjens in [79] for the verification of cryptographic protocols in the context of software engineering. The underlying process model used is based on Broy's stream-processing functions [37]. We recall here briefly the main notions needed for the rest of the paper. A *process* is of the form $P = (I, O, L, (p_c)_{c \in O \cup L})$ where $I \subseteq \mathsf{Channels}$ is called the set of its *input channels* and $O \subseteq \mathsf{Channels}$ the set of its *output channels* and where for each $c \in \tilde{O} \stackrel{\text{def}}{=} O \cup L$, p_c is a closed program with input channels in $\tilde{I} \stackrel{\text{def}}{=} I \cup L$ (where $L \subseteq \mathsf{Channels}$ is called the set of *local channels*). From inputs on the channels in $\tilde{I}$ at a given point in time, p_c computes the output on the channel c. Each channel defines thus a stream processing function based on its input variables allowing for a rigorous notion of sequential composition. The programs in our context will be written in a domain specific language defined by the expressions and the program constructs as in Fig. 1.

A process $P = (I, O, L, (p_c)_{c \in O})$ defines a stream-processing function

$$[\![P]\!] : \mathbf{Stream}_I \rightarrow \mathcal{P}(\mathbf{Stream}_O)$$

from input streams to sets of output streams. We can then compose two processes by using the composition of two stream-processing functions f_1, f_2 with $O_1 \cap O_2 = \emptyset$ as:

$$f_1 \otimes f_2 : \mathbf{Stream}_I \rightarrow \mathcal{P}(\mathbf{Stream}_O)$$

with $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2))$ where $f_1 \otimes f_2(\vec{s}) \stackrel{\text{def}}{=} \{\vec{t}\!\restriction_O : \vec{t}\!\restriction_I = \vec{s}\!\restriction_I \wedge \vec{t}\!\restriction_{O_i} \in f_i(\vec{s}\!\restriction_{I_i})\ (i = 1, 2)\}$ (where $\vec{t}$ ranges over $\mathbf{Stream}_{I \cup O}$). For $\vec{t} \in \mathbf{Stream}_C$ and $C' \subseteq C$, the restriction $\vec{t}\!\restriction_{C'} \in \mathbf{Stream}_{C'}$ is defined by $\vec{t}\!\restriction_{C'}(c) = \vec{t}(c)$ for each $c \in C'$.

$E ::=$	expression
d	data value ($d \in \mathcal{D}$)
N	unguessable value ($N \in \mathbf{Secret}$)
K	key ($K \in \mathbf{Keys}$)
$\mathsf{inp}(c)$	input on channel c ($c \in \mathsf{Channels}$)
x	variable ($x \in \mathbf{Var}$)
$E_1 :: E_2$	concatenation
$\{E\}_e$	encryption ($e \in \mathbf{Enc}$)
$\mathcal{D}ec_e(E)$	decryption ($e \in \mathbf{Enc}$)
$\mathcal{S}ign_e(E)$	signature creation ($e \in \mathbf{Enc}$)
$\mathcal{E}xt_e(E)$	signature extraction ($e \in \mathbf{Enc}$)

$p ::=$	programs
E	output expression ($E \in \mathbf{Exp}$)
either p or p'	nondeterministic branching
if $E = E'$ then p else p'	conditional ($E, E' \in \mathbf{Exp}$)
case E of key do p else p'	determine if E is a key ($E \in \mathbf{Exp}$)
case E of $x :: y$ do p else p'	break up list into head::tail ($E \in \mathbf{Exp}$)

Figure 1: Grammar for simple expressions and programs in the Domain-Specific Language

Example If $f : \mathbf{Stream}_{\{a\}} \to \mathcal{P}(\mathbf{Stream}_{\{b\}}),\ f(\vec{s}) \stackrel{\text{def}}{=} \{0.\vec{s}, 1.\vec{s}\}$, is the stream-processing function with input channel a and output channel b that outputs the input stream prefixed with either 0 or 1, and

$$g : \mathbf{Stream}_{\{b\}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}}),\ g(\vec{s}) \stackrel{\text{def}}{=} \{0.\vec{s}, 1.\vec{s}\}$$

the function with input (resp. output) channel b (resp. c) that does the same, then the composition

$$f \otimes g : \mathbf{Stream}_{\{a\}} \to \mathcal{P}(\mathbf{Stream}_{\{c\}}),\ f \otimes g(\vec{s}) = \{0.0.\vec{s}, 0.1.\vec{s}, 1.0.\vec{s}, 1.1.\vec{s}\}$$

outputs the input stream prefixed with either of the 2-element streams 0.0, 0.1, 1.0 or 1.1.

5.1.1 Secrecy preservation analysis

To proceed with the Dolev-Yao secrecy analysis, one defines rules to translate programs to first-order logic formulas. With the predicate $\mathsf{knows}(E)$ we express the fact that an adversary may know an expression E during the execution of the protocol. To verify the secrecy of data $s \in \mathbf{Secret}$, one then has to check whether the adversary can derive $\mathsf{knows}(s)$, given the formulas that arise from the evaluation ϕ of the single program constructs as in Fig. 3 and the axioms allowing to enlarge its knowledge in Fig. 2. The conjunction of the formulae ϕ for all channel programs of a process is called ψ. This formula encloses all the possible interaction of an adversary with the modelled

$$
\begin{aligned}
&\forall E_1, E_2. \\
&\big[\mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2) \;\Rightarrow\; \mathsf{knows}(E_1 :: E_2) \wedge \;\mathsf{knows}(\{E_1\}_{E_2}) \wedge \;\mathsf{knows}(\mathcal{S}ign_{E_2}(E_1))\big] \\
&\wedge\; \big[\mathsf{knows}(E_1 :: E_2) \;\Rightarrow\; \mathsf{knows}(E_1) \wedge \mathsf{knows}(E_2)\big] \\
&\wedge\; \big[\mathsf{knows}(\{E_1\}_{E_2}) \wedge \mathsf{knows}(E_2^{-1}) \;\Rightarrow\; \mathsf{knows}(E_1)\big] \\
&\wedge\; \big[\mathsf{knows}(\mathcal{S}ign_{E_2^{-1}}(E_1)) \wedge \mathsf{knows}(E_2) \;\Rightarrow\; \mathsf{knows}(E_1)\big]
\end{aligned}
$$

Figure 2: Structural formulas

$$
\begin{aligned}
&\phi(E) = \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \;\Rightarrow \mathsf{knows}(E(i_1,\ldots,i_n))\big] \\
&\phi(\textit{either } p \textit{ or } p') = \phi(p) \wedge \phi(p') \\
&\phi(\textit{if } E = E' \textit{ then } p \textit{ else } p') = \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \;\Rightarrow \\
&\qquad [E(i_1,\ldots,i_n) = E'(i_1,\ldots,i_n) \;\Rightarrow \phi(p)] \\
&\qquad \wedge\, [E(i_1,\ldots,i_n) \neq E'(i_1,\ldots,i_n) \;\Rightarrow \phi(p')]\big] \\
&\phi(\textit{case } E \textit{ of key do } p \textit{ else } p') = \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \;\Rightarrow \\
&\qquad [\mathsf{key}(E(i_1,\ldots,i_n)) \;\Rightarrow \phi(p)] \\
&\qquad \wedge\, [\neg\mathsf{key}(E(i_1,\ldots,i_n)) \;\Rightarrow \phi(p')]\big] \\
&\phi(\textit{case } E \textit{ of } x :: y \textit{ do } p \textit{ else } p') = \forall i_1,\ldots,i_n.\big[\mathsf{knows}(i_1) \wedge \ldots \wedge \mathsf{knows}(i_n) \;\Rightarrow \\
&\qquad \forall h,t.[E(i_1,\ldots,i_n) = h :: t \;\Rightarrow \phi(p[h/x, t/y])] \\
&\qquad \wedge\, [\neg\exists h,t.E(i_1,\ldots,i_n) = h :: t \;\Rightarrow \phi(p')]\big]
\end{aligned}
$$

Figure 3: Definition of $\phi(p)$.

process, and models the ability of a *man in the middle* of manipulating all messages sent and received through an insecure channel.

In the following, we will discuss composition at the level of this First Order Logic translation and not at the underlying stream processing function level because the FOL translation contains implicitly all the possible actions and adversary process could perform (defined by the structural formulas).

5.2 Decision procedure

If we assume that both P and P' preserve the secrecy of the data value s, our goal is to show a procedure so that we can decide if:

$$\psi(P \otimes P') \nvdash \mathsf{knows}(s).$$

In general this does not hold. For example consider a process P which outputs $\{s\}_K$ and a process P' which outputs K^{-1}. Independently both processes preserve the secrecy of s, but when composed an adversary could trivially compute s.

To achieve this, we will construct proof artefacts on each single process called *derivation trees*. Moreover, in order ensure that this trees are finite, we will require that the number of *keys* and *nonces* are also finite and that the conditions in the “if” constructs of the process programs admit only variables

that are of type *key* or *nonce*, or that clearly specify the form of the admitted values for the input channels in terms of keys, nonces or closed expressions. This is not at all restrictive for the definition of practical protocols but is a key element for the decidability of our approach.

Definition 1 (Subterm)**.** *We say that a symbol* x *is a subterm of the symbol* T *and denote it* $\mathsf{x}\,\hat{\in}\, T$ *if one of the following holds:*

1. $\mathsf{x}=T$
2. $T=\{T'\}_{\mathsf{K}}$ *and* $\mathsf{x}\,\hat{\in}\, T'$
3. $T=\mathcal{S}ign_{\mathsf{K}}\{\mathsf{T}'\}$ *and* $\mathsf{x}\,\hat{\in}\, T'$
4. $T= h::k$ *and* $\mathsf{x}\,\hat{\in}\, h$ *or* $\mathsf{x}\,\hat{\in}\, k$

Example $\mathsf{s}\,\hat{\in}\,\{\mathsf{s}\}_{\mathsf{K}}$ but is not true that $\mathsf{K}\,\hat{\in}\,\{\mathsf{s}\}_{\mathsf{K}}$. We denote this by $\mathsf{K}\,\hat{\notin}\,\{\mathsf{s}\}_{\mathsf{K}}$. This means that an adversary could potentially compute s from $\{\mathsf{s}\}_{\mathsf{K}}$ using the structural formulas with the necessary previous knowledge, but he could not compute K.

Definition 2 (Inverse)**.** *Let* $\mathsf{x}\,\hat{\in}\, J$*. We define the cryptographic inverse of a symbol* J *with respect to* x *and denote it* $\mathsf{J}^{-1}(\mathsf{x})$ *in the following way:*

1. $\mathsf{x}^{-1}(\mathsf{x}) = \epsilon$
2. *If* $J=h::k$ *and* $\mathsf{x}\,\hat{\notin}\, h$ *then* $\mathsf{J}^{-1}(\mathsf{x})=\mathsf{k}^{-1}(\mathsf{x})$
3. *If* $J=h::k$ *and* $\mathsf{x}\,\hat{\notin}\, k$ *then* $\mathsf{J}^{-1}(\mathsf{x})=\mathsf{h}^{-1}(\mathsf{x})$
4. *If* $J=h::k$ *and* $\mathsf{x}\,\hat{\in}\, k$*,* $\mathsf{x}\,\hat{\in}\, h$ *then:*

$$\mathsf{J}^{-1}(\mathsf{x}) = \mathsf{and}(\mathsf{h}^{-1}(\mathsf{x}), \mathsf{k}^{-1}(\mathsf{x}))$$

5. *If* $J=\{J'\}_{\mathsf{K}}$ *or* $J=\mathcal{S}ign_{\mathsf{K}}\{\mathsf{J}'\}$ *then:*

$$\mathsf{J}^{-1}(\mathsf{x}) = \mathsf{or}(\mathsf{J}'^{-1}(\mathsf{x}), K^{-1}).$$

Example Let $\mathsf{J} = \{\{\mathsf{s}\}_{\mathsf{K_1}}\}_{\mathsf{K_2}}$. Then $\mathsf{J}^{-1}(\mathsf{s}) = \mathsf{or}(K_1^{-1}, K_2^{-1})$ which we will interpret later as "to preserve the secrecy s we need to preserve either K_1^{-1} or K_2^{-1}".

Let $\psi(P)$ be the first order logic formula associated to P. We define $\bar{\psi}(P)$ to be the set of instantiated formulas of $\psi(P)$ with all possible values satisfying the constraints in $\psi(P)$. Since we require that all constraints only contain variables of type *key* or *nonce*, and that the respective sets are finite,

then $\bar{\psi}(P)$ is also finite. It is possible to show by induction on the program constructs that $\bar{\psi}(P)$ consists of formulas F_i of the form:

$$\mathsf{knows}(\mathsf{E}_i) \Rightarrow \mathsf{knows}(\mathsf{J}_i)$$

for closed expressions E_i and J_i.

Let Pres(x,P) be the following inductively defined predicate:

$$
\begin{aligned}
&[(\forall \mathsf{F}_i \in \bar{\psi}(P)\ \mathsf{x} \hat{\notin} J_i\ ) \Rightarrow \mathsf{Pres(x,P)})\\
&\quad \wedge (\ \forall \mathsf{F}_i \in \bar{\psi}(P)\ (\mathsf{x} \hat{\in} \mathsf{J}_i) \Rightarrow ((\mathsf{Pres}(\mathsf{E}_i,\mathsf{P}) \vee \mathsf{Pres}(\mathsf{J_i}^{-1}(\mathsf{x}),\mathsf{P}))\\
&\quad \wedge (\ (\mathsf{x} = \{\mathsf{x'}\}_\mathsf{K} \vee \mathsf{x} = \mathcal{S}ign_\mathsf{K}\{\mathsf{x'}\}) \Rightarrow (\mathsf{Pres(x',P)} \vee \mathsf{Pres(K,P)})\\
&\quad \wedge ((\mathsf{x} = \mathsf{h::k} \Rightarrow (\mathsf{Pres(h,P)} \vee \mathsf{Pres(k,P)})\\
&\quad \wedge ((\mathsf{x} = \mathsf{and(h,k)} \Rightarrow (\mathsf{Pres(h,P)} \wedge \mathsf{Pres(k,P)})\\
&\quad \wedge ((\mathsf{x} = \mathsf{or(h,k)} \Rightarrow (\mathsf{Pres(h,P)} \vee \mathsf{Pres(k,P)})\ ]\\
&\Rightarrow \mathsf{Pres(x,P)}
\end{aligned}
$$

and $\neg$Pres(ϵ,P).

If we can not derive Pres(x,P) for some x, it follows $\neg$Pres(x,P).

Theorem 3. *If it is possible to derive Pres(x,P) (conversely ¬Pres(x,P)) then* $\psi(P) \nvdash \mathsf{knows}(x)$ *(*$\psi(P) \vdash \mathsf{knows}(x)$*).*

Proof In case $\neg$Pres(ϵ,P) since $\mathsf{knows}(\epsilon) \in \bar{\psi}(P)$ for all P. If $\forall \mathsf{F}_i \in \bar{\psi}(P)$ $\mathsf{x} \hat{\notin} J_i$ that means that there is no formula in $\bar{\psi}(P)$ containing x in a conclusive position, and therefore there is no way to derive knows(x) from the structural formulas. Now assume it is possible to derive Pres(x,P). We have already covered the base cases so we can assume that $\psi(P) \nvdash \mathsf{knows}(y)$ for all the Pres(y,P) y $\neq$ x needed in the precondition. Since in this formulas all the cases where we could apply the Structural Formulas are covered, it is impossible to derive knows(x). The case $\neg$Pres(x,P) is similar. □

Note that the converse does not hold, that is $\psi(P) \nvdash \mathsf{knows}(x)$ does not mean we can derive Pres(x,P), because for some pathological cases we will have an infinite loop, for example for $\bar{\psi}(P) = \mathsf{knows(x)} \Rightarrow \mathsf{knows(x)}$. It is although reasonable to expect this loops not to be present in practical cases. Moreover it is possible to detect this loops in a machine implementation of the preservation predicate by running an initial check on the formulas, and avoid infinite recursion. This makes the verification of the Pres(x,P) predicate sound and complete with respect to the First Order Logic embedding of the process programs.

5.2.1 Composition

As we derive Pres(s,P) for some symbol s and formulas P, we can build a *derivation tree* consisting of the symbols we need to consider to be able to conclude the preservation status of s. If we generate and store the derivation

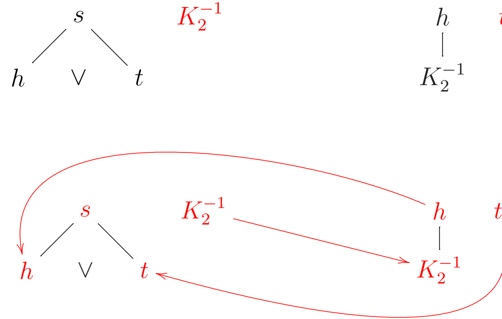


Figure 4: Processes P and P' before and after composition

tree for every symbol x appearing in a process P in a relevant position (that is $x \hat{\in} J_i$ for some i), then we can decide whether the composition with process P' will preserve the secrecy of any given symbol (if we also possess the derivation trees for P'). We will illustrate this in the following example.

Example Consider $P = (\{s\}_{h::t}, K_2^{-1}), P' = (\{\mathsf{h}\}_{\mathsf{K}_2}, t)$. The symbol dependency trees of both process are depicted in Fig. 4 (the symbols in red are the ones which secrecy is compromised). Clearly both processes preserve separately the secrecy of s. To see if the composition also does, we proceed as follows. We update the information on the tree of s by checking whether the truth values of h and t are altered by the composition as depicted in Fig. 4. To do that, we check on the dependency trees of P', that obviously do not preserve the secrecy of t. In the case of h, we have to update that dependency as well, by looking into P. Since the truth value of h is altered due to the fact that P leaks ${K_2}^{-1}$, the new truth value for s is altered as well. Therefore, secrecy is not preserved in this composition.

Although as stated in the introduction we do not focus on the composition of cryptographic protocols on the standard sense, our methodology could be used to reason about the composition of arbitrary processes, thus in particular it can also be used to decide on the secrecy preservation of protocols. For example assume the composition of processes A and B representing a given protocol between two parties is secure, as well as the composition of processes C and D representing a second protocol. We sketch possible applications of our methodology to three standard composition notions:

- *Parallel composition*, that is the composition that arises when two distinct protocols are used over the same insecure channel, perhaps at a different time and for a different purpose, and can be approximated by evaluating the composition $(A \otimes B) \otimes (C \otimes D)$. This constitutes an approximation because in practice it can happen that one protocol

executes before the other and it is never executed again, but by assuming there are arbitrary executions of both protocols we obtain a safe estimate on possible attacks.

- *Sequential composition* is similar to parallel composition, but here two protocols are used one after the other typically with the purpose of re-using information from the first protocol in the second (for example a session key). An approximation to sequential composition can also be obtained by considering $(A \otimes B) \otimes (C \otimes D)$, in the sense that false positive attacks can occur because the sequentiality of the events is lost in the construction of the trees. This can be improved by considering a one way update of the dependency tree of $(C \otimes D)$ (instead of going back and forward as in the normal tree update).

- *Vertical composition* typically uses the key established in a key agreement to build a secure channel and then runs one or more protocols in the secure session (that is encrypted with the session key). This is perhaps the most challenging kind of composition where our approach can be applied because the dependency tree of the second protocol must be updated with the new dependency on the key of the first protocol. In principle this is a feasible task, but requires a deeper study in order to assess its efficiency.

A thorough analysis of standard protocol composition by means of our approach is definitively an interesting but challenging task and it is out of the scope of this chapter.

5.3 An insecure variant of the TLS protocol

As an example we apply our approach to a variant of TLS [25] (not the version of TLS in current use) that does not preserve secrecy as a composition of the client C, the server S and the authority CA . We have that the predicate for C and S after the programs are translated to F.O.L are (for details on the translation see [79]) :

$$\begin{aligned}\psi(\mathsf{C}) = \; & \mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{\mathsf{K}_\mathsf{C}^{-1}}\{\mathsf{C} :: \mathsf{K_C}\}) \\ & \wedge (\mathsf{knows}(s_2) \wedge \mathsf{knows}(s_3) \Rightarrow \mathsf{knows}(\{m\}_y))\end{aligned}$$

$$\begin{aligned}\psi(\mathsf{S}) = \; & \mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(c_3) \\ & \Rightarrow \mathsf{knows}(N_S :: \{ \; \mathcal{S}ign_{\mathsf{K}_\mathsf{S}^{-1}}\{\mathsf{k_{CS}} :: \mathsf{c_1}\} \; \}_{c_2})\end{aligned}$$

where $\{s_3\}_{K_{CA}} = S :: x \wedge \{Dec_{K_C^{-1}}(s_2)\}_x = y :: N_C$ and $\{c_3\}_{c_2} = C :: c_2$ and $\mathsf{key}(c_2), \mathsf{key}(x)$ and $\mathsf{key}(y)$. We assume that the authority CA has already distributed certificates to all parties and that the adversary is in possession

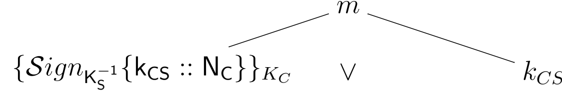


Figure 5: Partial dependency tree for m in C

of this information:

$$\begin{aligned}&\mathsf{knows}(K_{CA})\\ &\wedge\ \mathsf{knows}(\mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}\{\mathsf{S} :: \mathsf{K_S}\})\\ &\wedge\ \mathsf{knows}(\mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}\{\mathsf{A} :: \mathsf{K_A}\}\ )\end{aligned}$$

We can also assume that an adversary knows a key K_A and its inverse $\mathsf{knows}(K_A) \wedge \mathsf{knows}(K_A^{-1})$ and we define the set Keys for this process:

$$\mathsf{Keys} = \{K_A, K_A^{-1}, k_{CS}, k_A, K_C, K_C^{-1}, K_S, K_S^{-1}, K_{CA}, K_{CA}^{-1}\}$$

where k_{CS} and k_A are symmetric keys. The nonces are $\mathsf{Nonces} = \{N_C, N_S, N_A\}$.

Now we show that $\mathsf{C} \otimes \mathsf{S}$ does not preserve the secrecy of m although C and S separately do. First of all, in order to be able to apply our approach and generate the dependency tree, we have to solve the constraints for all the processes involved. So we have:

$$\begin{aligned}\bar{\psi}(\mathsf{C}) =\ &\mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{\mathsf{K_C^{-1}}}\{\mathsf{C} :: \mathsf{K_C}\})\\ &\wedge\ [\ \mathsf{knows}(\{\ \mathcal{S}ign_{\mathsf{x}^{-1}}\{\mathsf{y} :: \mathsf{N_C}\}\ \}_{K_C})\\ &\quad \wedge\ \mathsf{knows}(\ \mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}\{\mathsf{S} :: \mathsf{x}\}\ ) \Rightarrow \mathsf{knows}(\{m\}_y)\ ]\end{aligned}$$

where $x \in \{K_C, K_S, K_A\}$ (the public keys) and $y \in \{k_A, k_{CS}\}$ (the symmetric keys). We do not explicit the whole dependency tree for C but we note that the secrecy of m is preserved because: if $y = k_{CS}$ the adversary does not have knowledge of k_{CS}; if $y = k_A$ the adversary would need knowledge of $\mathcal{S}ign_{\mathsf{x}^{-1}}\{\mathsf{k_A} :: \mathsf{N_C}\}$ and $\mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}\{\mathsf{S} :: \mathsf{x}\}$ for some x. Since he only knows $\mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}\{\mathsf{S} :: \mathsf{K_S}\}$ then $x = K_S$. In that case to gain knowledge of $\mathcal{S}ign_{\mathsf{K_S}^{-1}}\{\mathsf{k_A} :: \mathsf{N_C}\}$ he needs to posses ${K_S}^{-1}$ which he does not. In Figure 5 we depict partially this dependency tree for the case $y = k_{CS}$, $x = K_S$.

Now, the instantiated formulas for S are:

$$\begin{aligned}\bar{\psi}(\mathsf{S}) = &\ \mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(\mathcal{S}ign_{\mathsf{c_2^{-1}}}\{\mathsf{C} :: \mathsf{c_2}\})\\ &\Rightarrow \mathsf{knows}(N_S :: \{\ \mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}} :: \mathsf{c_1}\}\ \}_{c_2})\end{aligned}$$

with $c_1 \in \{N_S, N_C, N_A\}$, $c_2 \in \{K_C, K_S, K_A\}$. The secrecy of m is preserved in S simply because m is not a subterm of any formula in S.

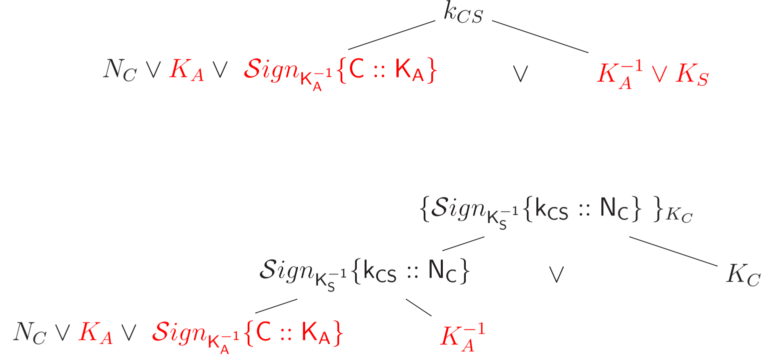


Figure 6: Partial dependency trees for k_{CS} and $\{\ \mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}} :: \mathsf{N_C}\}\ \}_{K_C}$ in S

It is interesting nevertheless to draw (partially) the dependency trees of k_{CS} and $\{\ \mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}} :: \mathsf{c_1}\}\ \}_{c_2}$ in case $c_1 = N_C$ and $c_2 = K_A$, as depicted in Figure 6. In fact, if we analyse the composition of C and S, since C leaks N_C and K_C, k_{CS} turns to be not secret after composition in the tree of S, and the same holds for $\{\ \mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}} :: \mathsf{N_C}\}\ \}_{K_C}$), resulting in a secrecy violation for m after updating the original tree in C.

5.3.1 TLS fixed

We now show that the fixed version of TLS obeys to our rules to preserve secrecy. The predicates for TLS' are similar as before with the following changes on C and S:

$$\begin{aligned}\psi(\mathsf{C}') = \ & \mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{\mathsf{K_C^{-1}}}\{\mathsf{C} :: \mathsf{K_C}\}) \\ & \wedge [\mathsf{knows}(s_2) \wedge \mathsf{knows}(s_3) \Rightarrow \mathsf{knows}(\{m\}_y)]\end{aligned}$$

such that $\{s_3\}_{K_{CA}} = S :: x, \{Dec_{K_C^{-1}}(s_2)\}_x = y :: N_C :: K_C$ where $x \in \{K_C, K_S, K_A\}, y \in \{k_A, \{k_{CS}(z,w) : z, w \in \mathsf{Nonces} \times \mathsf{Keys}\}\}$ and $k_{CS}()$ is such that:

$$\forall\ x, y, z, w\ (x \neq z\ \vee\ y \neq w\ \Rightarrow k_{CS}(x,y) \neq k_{CS}(z,w))$$

and we assume the adversary has no previous knowledge of any such key. The modified server gives:

$$\begin{aligned}\psi(\mathsf{S}') = \ & \forall c_1, c_2, c_3, a_1, x. \\ & [\mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(c_3) \wedge \mathsf{knows}(a_1) \\ & \Rightarrow \mathsf{knows}(N_S :: \{\ \mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}}(\mathsf{c_1}, \mathsf{c_2}) :: \mathsf{c_1} :: \mathsf{c_2}\}\ \}_{c_2} :: a_1)]\end{aligned}$$

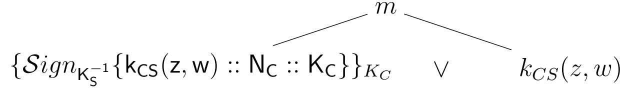


Figure 7: Partial dependency tree for m in case $k_{CS}(z,w)$ in C'

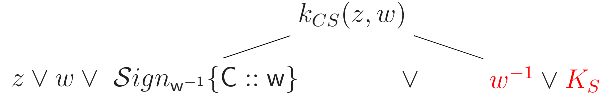


Figure 8: Partial dependency tree for $k_{CS}(z,w)$ in S'

where $\{c_3\}_{c_2} = x :: c_2$, $c_2 \in \mathsf{Keys}$ and $c_1 \in \mathsf{Nonces}$.

Now we show that $\mathsf{C}' \otimes \mathsf{S}'$ preserves the secrecy of m. We have:

$$\begin{aligned} \bar{\psi}(\mathsf{C}') = \; & \mathsf{knows}(N_C :: K_C :: \mathcal{S}ign_{\mathsf{K_C^{-1}}}\{\mathsf{C} :: \mathsf{K_C}\}) \\ & \wedge [\mathsf{knows}(\{\ \mathcal{S}ign_{\mathsf{x^{-1}}}\{\mathsf{y} :: \mathsf{N_C} :: \mathsf{K_C}\}\ \}_{K_C}) \\ & \quad \wedge \mathsf{knows}(\ \mathcal{S}ign_{\mathsf{K_{CA}^{-1}}}\{\mathsf{S} :: \mathsf{x}\}\ ) \\ & \quad \Rightarrow \mathsf{knows}(\{m\}_y)] \end{aligned}$$

where $x \in \{K_C, K_S, K_A\}$ and $y \in \{k_A, \{k_{CS}(z,w) : z, w \in \mathsf{Nonces} \times \mathsf{Keys}\}\}$. As in the insecure variant of TLS discussed before, secrecy of m is guaranteed by the fact that either the adversary does not know any of the $k_{CS}(z,w)$ keys, or in case $y = k_A$ then he can not forge the signature $\mathcal{S}ign_{\mathsf{K_S}^{-1}}\{\mathsf{k_A} :: \mathsf{N_C} :: \mathsf{K_C}\}$.

The dependency tree for m in case $y = k_{CS}(z,w)$ is shown in Figure 7. The instantiated formulas for S' are:

$$\begin{aligned} \bar{\psi}(\mathsf{S}') = \; & \mathsf{knows}(c_1) \wedge \mathsf{knows}(c_2) \wedge \mathsf{knows}(\mathcal{S}ign_{\mathsf{c_2^{-1}}}\{\mathsf{C} :: \mathsf{c_2}\}) \\ & \Rightarrow \mathsf{knows}(N_S :: \{\ \mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}(c_1,c_2)} :: \mathsf{c_1} :: \mathsf{c_2}\}\ \}_{c_2}) \end{aligned}$$

with $c_1 \in \{N_S, N_C, N_A\}$, $c_2 \in \{K_C, K_S, K_A\}$. As in the previous TLS variant, the secrecy of m is preserved in S' simply because m is not a subterm of any formula in S.

Now notice that the dependency tree for $k_{CS}(z,w)$ for any $z, w \in \mathsf{Nonces} \times \mathsf{Keys}$ is the one depicted in Fig. 8. The only choice of w such that the secrecy of $k_{CS}(z,w)$ is not preserved would be then $w = K_A$, since it is the only private/public key pair the adversary knows (no other inverses of public keys are in relevant positions neither in C' or in S').

Therefore, if the composition would harm the secrecy of m, the secrecy

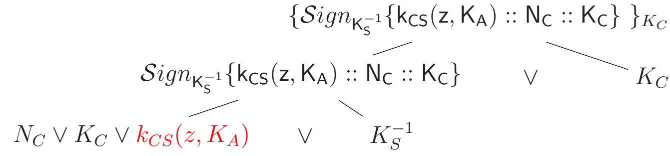


Figure 9: Dependency tree for $\{\mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}(z, K_A)} :: \mathsf{N_C} :: \mathsf{K_C}\}\}_{K_C}$ in S'

of:

$$\{\mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}(z, w)} :: \mathsf{N_C} :: \mathsf{K_C}\}\}_{K_C}$$

must not hold after composition for $w = K_A$, as it results from the tree in Fig. 7. The secrecy value for $\{\mathcal{S}ign_{\mathsf{K_S^{-1}}}\{\mathsf{k_{CS}(z, K_A)} :: \mathsf{N_C} :: \mathsf{K_C}\}\}_{K_C}$ will not be harmed by the composition as we can see from Fig. 9 since in both processes K_S^{-1} is not derivable.

5.4 Validation and Efficiency

We have implemented our approach as an extension to the UMLsec tool support [1]. That is, we can extract the protocol specification from a sequence diagram using the DSL described in Sect. 5.1 and translate it to First Order Logic. Since by construction each guard accepts only finitely many messages (depending on the set of keys and nonces), we can build finite dependency trees for all relevant symbols by means of a properly generated Prolog program.

Since reasoning about composition amounts to join the trees from two processes, it is reasonable to expect that this is computationally faster compared to re-computing the whole tree for the composed processes. Indeed, we can at least avoid to recompute the constraint solving for the single processes. Although we do not have a formal argument to show that the complexity of our approach is lower than re-verification, we have conducted experiments to measure the time of the composition, and compare it to the overall process of constraint-solving and prolog generation.

In Figure 10 we depict this comparison. The first column contains the number of messages for a single session of the composition and the second column corresponds to the number of composed processes. The third column is the time in ms. needed to extract the FOL formulas from the UML diagram and generate the derivation trees. The last column is the time needed for deciding the composition given the single derivation trees. It is thus clear

[1] http://www-jj.cs.tu-dortmund.de/jj/umlsectool

# Messages	# Compositions	Duration (in ms) Generation trees	Duration (in ms) composition
11	5	3660	47
21	10	6214	88
31	15	9323	114
51	25	15406	198
101	50	31730	401
501	250	182771	1948
1001	500	375474	3963

Figure 10: Execution times of our experiment

that the deciding the composition is very efficient if it is possible to reuse the precomputed trees and the generation from scratch is about two orders of magnitude slower. In other words, if we would have a repository of 500 processes that by themselves are secrecy preserving, and we would like to check whether the composition of any 5 of them is also secrecy preserving, it would be highly desirable if we could use the existing results as opposed to re-computing from scratch every time.

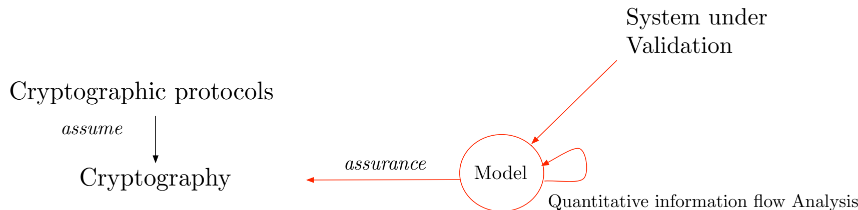
5.5 Related Work

The work of security protocol verification is too large to give a full overview. Overviews of applications of formal methods to security protocols can be found for example in [13, 99, 123, 77], some examples in [84, 94, 14, 116]. More abstractly, compositional model-checking has been explored in general for over two decades starting with Clark [45]. The question of protocol composition has been studied by different authors. More prominently, Datta, Mitchell et al. [50] have defined the PCL (Protocol Composition Logic), aimed at the verification of security protocol by re-using proofs of sub-protocols using a Hoare-like logic. The focus of the PCL is authenticity and it uses an unbounded session model. Automatic proofs are only achieved by a complementary approach using model-checking that only explores a finite-state system [106] that is not based on PCL. Guttmann [64] gives results about protocol composition at a lower level, considering unstructured 'blank slots' and compound keys that result from hashes of other messages. Jürjens [75] has explored the question of composability aiming at given sufficient conditions under which composition holds. Stoller [129] has computed bounds of parallel executions that could compromise the authenticity of protocols. These both last approaches aim at giving at a collection of theorems that if satisfied by two protocols in a composition, ensure a given

property. That means that one must show (by using a theorem prover, or by hand) that some properties are satisfied by both protocols (like *disjointness* in [65]). Our approach differs from this assume/guarantee reasoning in that we efficiently check whether the composition of processes harms secrecy without making any assumptions (besides secrecy preservation) on the composed processes, given pre-computed 'proof artefacts': the dependency trees. In other words, we give accurate results about compositions (that are equivalent to re-verification), by amortizing the cost of verification at an initial phase. The application of the techniques described in this chapter to standard protocol composition is interesting future work.

Chapter 6

Cache side-channel analysis



In this chapter we discuss results linking abstract interpretation and quantitative information flow to measure the leakage of inputs to deterministic programs with respect to realistic adversary models. The efficiency of the methodology proposed (backed up by a tool chain) allows for a quick quantification in case of changes in the cache configuration, which constitutes a useful tool for reasoning about security in evolving micro-architectures.

The motivation of this work is that many modern computer architectures use caches to bridge the latency gap between the CPU and main memory,. Caches are small, fast memory that store the contents of previously accessed main memory locations and can improve the overall performance because typical memory access patterns tend to be "local": they call recently accessed references. On today's architectures, an access to the main memory (a cache miss) may imply an overhead about one order of magnitude compared to an access to the cache (cache hit).

While the use of caches is beneficial for performance, it can have negative effects on security: An observer who can measure the time of memory lookups can see whether a lookup is a cache hit or miss, and thus learn partial information about the state of the cache. This partial information has been used for extracting cryptographic keys from implementations of AES [31, 114, 63], RSA [117], and DSA [15], and can be potentially used for attacking other programs with secret inputs. In particular AES is vulnerable to such cache-attacks, because most high-speed software implementations make heavy use

of look-up tables. Cache attacks are the most effective known attacks against AES and allow to recover keys within minutes [63].

A number of countermeasures have been proposed against cache attacks. Trivially, one could avoid the use of caches for sensitive computations, but this would affect performance, and is thus not generally applicable. Proposal for mitigation strategies include disabling high-resolution timers, hardening of schedulers [63], and preloading [114] of tables to eliminate attack vectors and reduce leakage, respectively. Such strategies are implemented, e.g. in the OpenSSL 1.0 [8] version of AES, however, as observed by Bernstein [31] their effectiveness is highly dependent on the OS and the CPU. Without considering/modelling all implementation details, such mitigation strategies necessarily remain heuristic. In general, there is no general-purpose countermeasure against cache attacks that is backed-up by mathematical proof.

In this chapter we propose a novel method for establishing formal security guarantees against cache-attacks. The guarantees we obtain are upper bounds on the amount of information about the input that an adversary can extract by observing the CPU's cache state after execution of the program. They are based on the actual program binary and a concrete processor model and can be derived entirely automatically. At the heart of our approach is a novel technique for counting that enables us to connect state-of-the-art techniques for static cache analysis and quantitative information-flow analysis.

Technically, we build on work on static cache analysis [21] that was primarily used for the estimation of worst-case execution time by abstract interpretation [47]. There, two abstract domains for cache-states are introduced; one of them captures a superset of the memory locations that *may* be in the cache, the other captures a subset of the memory locations that *must* be in the cache. Soundness of these abstractions means that each of them computes a superset of the set of reachable cache-states. We combine this technique with results from quantitative-information-flow analysis that enable establishing bounds for the amount of information that a program leaks about its input. Our approach relies on the fact that (an upper bound on) the number of reachable states of a program corresponds to (an upper bound on) the number of leaked bits [128, 90].

We develop a novel technique for counting the number of cache states represented by the abstract states of the static cache analyses described above and give a concise implementation of our counting procedures in Haskell [7]. We connect this counting engine to the AbsInt a^3 [1], the state-of-the-art tool for static cache analysis. a^3 efficiently analyses binary code based on accurate models of several modern embedded processors with a wide range of cache types (e.g. data caches, instruction caches, or mixed) and replacement strategies. Using this tool-chain, we perform an analysis of a binary implementation of 128-bit AES from the PolarSSL library [5], based on a 32-bit ARM processor with a 4-way set associative data cache with LRU replacement strategy and different cache sizes. We analyse this implementation

with and without the preloading countermeasure applied and for two different adversary models, giving formal bounds for the leakage with different cache configurations.

6.1 Preliminaries

In this section we summarize useful concepts from quantitative information-flow analysis. In particular, we introduce measures of confidentiality based on information theory in Section 6.1.1, and we present techniques for their approximation in Section 6.1.2.

6.1.1 Quantifying Information Leaks

A (deterministic) *channel* is a function $C\colon S \to O$ mapping a finite set of secrets S to a finite set of observations O. We characterize the security of a channel in terms of the difficulty of guessing the secret input from the observation. This difficulty can be captured using information-theoretic entropy, where different notions of entropy correspond to different kinds of guessing [41]. In this work, we focus on min-entropy as a measure, because it is associated with strong security guarantees [128].

Formally, we model the choice of a secret input by a random variable X with $ran(X) = S$ and the corresponding observation by a random variable Y with $ran(Y) = O$. The dependency between X and Y is formalized as a conditional probability distribution $P_{Y|X}$ with $P_{Y|X}(o, s) = 1$ if $C(s) = o$, and 0 otherwise. We consider an adversary that wants to determine the value of X from the value of Y, where we assume that X is distributed according to P_X. The adversary's a priori uncertainty about X is given by the *min-entropy* [121]

$$H_\infty(X) = -\log_2 \max_s P_X(s)$$

of X, which captures the probability of correctly guessing the secret in one shot. The adversary's a posteriori uncertainty is given by the *conditional min-entropy* $H_\infty(X|Y)$, which is defined by

$$H_\infty(X|Y) = -\log_2 \sum_o P_Y(o) \max_s P_{X|Y}(s, o)$$

and captures the probability of guessing the value of X in one shot when the value of Y is known.

The *(min-entropy) leakage* L of a channel with respect to the input distribution P_X is the reduction in uncertainty about X when Y is observed,

$$L = H_\infty(X) - H_\infty(X|Y) \ ,$$

and is the logarithm of the factor by which the probability of guessing the secret is reduced by the observation. Note that L is not a property of the

channel alone as it also depends on P_X. We eliminate this dependency as follows.

Definition 3 (Maximal Leakage)**.** *The* maximal leakage *ML of a channel C is the maximal reduction in uncertainty about X when Y is observed*

$$ML(C) = \max_{P_X}(H_\infty(X) - H_\infty(X|Y)) \ ,$$

where the maximum is taken over all possible input distributions.

For computing an upper bound for the maximal leakage of a deterministic channel, it suffices to compute the size of the range of C. While these bounds can be coarse in general, they are tight for uniformly distributed input.

Lemma 1.

$$ML(C) \leq \log_2 |C(S)| \ ,$$

where equality holds for uniformly distributed P_X.

Proof. The maximal leakage of a (probabilistic) channel specified by the distribution $P_{Y|X}$ can be computed by $ML(P_{Y|X}) = \log_2 \sum_o \max_s P_{Y|X}(o, s)$, where the maximum is assumed (e.g.) for uniformly distributed input [36, 91]. For deterministic channels, the number of non-zero (hence 1) summands matches $|C(S)|$. □

6.1.2 Static Analysis of Channels

We consider channels *of programs*, which are channels that are given by the semantics of (deterministic, terminating) programs. In this setting, the set of secrets is a part of the initial state of the program, and the set of observables is a part of the final state of the program. Due to Lemma 1, computing upper bounds on the maximal leakage of a program can be done by determining the set of final states of the program. Computing this set from the program code requires computation of a fixed-point and is not guaranteed to terminate for programs over unbounded state-spaces. Abstract interpretation [47] overcomes this fundamental problem by resorting to an approximation of the state-space and the transition relation. By choosing an adequate approximation one can enforce termination of the fixed-point computation after a finite number of steps. The soundness of the analysis follows from the soundness of the abstract domain, which is expressed in terms of a *concretization function* (denoted γ) relating elements of the abstract domain to concrete properties of the program, ordered by implication.

For the purpose of our work, we define soundness with respect to a channel, i.e., we will use a concretization function mapping to sets of observables (where implication corresponds to set inclusion).

Definition 4. *An abstract element* $t^\sharp$ *is* sound *for a concretization function* γ with respect to a channel $C : S \to O$ *if and only if* $C(S) \subseteq \gamma(t^\sharp)$*.*

The following theorem is an immediate consequence from Lemma 1; it states that a counting procedure for $\gamma(t^\sharp)$ can be used for deriving upper bounds on the amount of information leaked by C.

Theorem 4. *Let* $t^\sharp$ *be sound for* γ *with respect to* C*. Then*

$$ML(C) \leq \log_2 \left|\gamma(t^\sharp)\right| \ .$$

For a more detailed account of the connection between abstract interpretation and quantitative information-flow, see [90].

6.2 Cache Channels

In this section, we define channels corresponding to two adversary models that can only observe cache properties. We also revisit two abstract domains for reasoning about cache-states and show how they relate to those channels. We begin by briefly explaining the technical functioning of CPU caches.

6.2.1 Caches

Typical caches work as follows. The main memory is partitioned into *blocks* of size β that are referenced using locations *loc*. A cache consists of a number of *sets*, each containing a fixed number of *lines* that can each store one memory block. The size A of the cache sets is called the *associativity* of the cache. Each memory block can reside in exactly one cache set, which is determined by the block's location. We can formally define a single *cache set* as a mapping

$$t\colon \{1, \dots, A\} \to \mathit{loc} \cup \{\bot\} \ ,$$

from line numbers to locations, where $\bot$ represents an empty line. A *cache* is a tuple of independent cache sets. For simplicity of presentation, we focus on single cache sets throughout the chapter, except for the case study in Section 6.5.

There exists different *replacement strategies* to deal with the state of the cache upon a memory request. Here we focus on the LRU (Least Recently Used) strategy, which is used e.g. in the Pentium I processor. With LRU, each cache set forms a queue. When a memory block is requested, it is appended to the head of the queue. If the block was already stored in the cache (cache hit), it is removed from its original position; if not (cache miss), it is fetched from main memory. Due to the queue structure of sets, memory blocks *age* when other blocks are looked up, i.e. they move towards the tail of the queue and (due to the fixed length of the queue) are eventually removed.

For a formalization of the LRU set update function see [21] or Appendix A.2. For a formalization of alternative update functions, such as FIFO (First In First Out) see [119]. Depending on the concrete processor model, data and instructions are processed using dedicated caches or a common one [21]. Unless mentioned otherwise (e.g. in the experiments for AES), our results hold for any cache analysis that is sound.

6.2.2 Two Adversary Models Observing the Cache

We consider a scenario where multiple processes share a common CPU. We assume that one of these processes is adversarial and tries to infer information about the computations of a victim process by inspecting the cache after termination. We distinguish between two adversaries Adv_{prec} and Adv_{prob}. Both adversaries can modify the initial state of the cache with memories in their virtual memory space, which we assume is not shared between processes, but they differ in their ability of observing the final cache state:

Adv_{prec} : This adversary can observe the precise content of the cache at the end of the victim's computation.

Adv_{prob} : This adversary can observe which blocks of his virtual memory space are in the cache after the victim's computation.

The channel corresponding to the adversary Adv_{prec} simply maps the victim's input to the corresponding final cache state. The channel corresponding to Adv_{prob} can be seen as an abstraction of the channel corresponding to Adv_{prec}, as it can be described as the composition of the channel of Adv_{prec} with a function *blur* that maps all memory blocks not belonging to the adversary's virtual memory space to one undistinguishable element. Adv_{prob} corresponds to the adversaries encountered in synchronous "prime and probe" attacks [114], which observe the cache-state by performing accesses to different locations and use timing measurements to distinguish whether they are contained in the cache or not.

Considering that our adversary models allow some choice of the initial state, they formally define families of channels that are indexed by the adversarially chosen part of the initial cache. To give an upper bound on the leakage of all channels in those families we would need relational information, which is not supported by the existing cache analysis tools. One possible solution is to consider an abstract initial state approximating all possible adversary choices, which leads to imprecision in the analysis. In the particular case of a LRU replacement strategy, we can use the following property:

Proposition 1. *For caches with LRU strategy, the leakage to* Adv_{prec} *(*Adv_{prob}*) w.r.t. any initial cache state containing only memory locations from the adversary's memory space corresponds to the leakage to* Adv_{prec} *(*Adv_{prob}*) w.r.t. an empty initial cache state.*

This result follows from the following observation: for each initial cache state containing locations disjoint from the victim's memory space, the first i lines of the final cache state will contain the locations accessed by the victim, and the remaining lines will contain the first $A-i$ locations of the initial state shifted to the right, where i depends on that particular run of the victim. That is, modulo the adversarial locations, the number of possible final cache states corresponding to an empty initial state matches the number of final cache states corresponding to an initial state that does not contain locations from the victim's memory space. The assertion then follows immediately from Theorem 4. Proposition 1 will be useful in our case study, since the analysis we use provides a more accurate final state when run with an initial empty cache.

6.2.3 Abstract Domains for Cache Analysis

Alt et al. [21] propose abstract interpretation techniques for cache analysis and prove their soundness with respect to reachability of cache states, which corresponds to soundness w.r.t the channel of Adv_{prec} according to Definition 4. In particular, they present two abstract domains for cache-states: The first domain corresponds to a *may*-analysis and represents the set of memory locations that possibly reside in the cache. The second domain corresponds to a *must*-analysis and represents the set of memory locations that are definitely in the cache. In both cases, an abstract cache set is represented as a function

$$t^\sharp\colon \{1,\dots,A\} \to 2^{loc}$$

mapping set positions to sets of memory locations. In the following we will use ${t_1}^\sharp$ and ${t_2}^\sharp$ for abstract sets corresponding to the *may* and *must* analysis respectively. For the *may* analysis, the concretization function $\gamma^\cup$ is defined by

$$\gamma^\cup({t_1}^\sharp) = \{t \mid \forall j \in \{1,\dots,A\}\colon t(j) = \bot \;\vee\; \exists i \le j : t(j) \in {t_1}^\sharp(i)\}\ .$$

This definition implies that each location that appears in the concrete state appears also in the abstract state, and the position in the abstract state is a lower bound for the position in the concrete. For the *must* analysis, the concretization function $\gamma^\cap$ is defined by

$$\gamma^\cap({t_2}^\sharp) = \{t \mid \forall i \in \{1,\dots,A\}\colon \forall a \in {t_2}^\sharp(i)\colon \exists\, j \le i : t(j) = a\}\ .$$

This definition implies that each location that appears in the abstract state is required to appear in the concrete, and its position in the abstract is an upper bound for its position in the concrete.

Example 4. *Consider the following program running on a 4-way fully associative (i.e. only one set) data cache where* $\dots x \dots$ *stands for an instruction*

that references location x, and let e, a, b are pairwise distinct locations.

$$\text{if} \ldots e \ldots \text{then} \ldots a \ldots \text{else} \ldots b \ldots$$

With an empty initial abstract cache before execution, the abstract may- and must-analyses return

$$t_1{}^\sharp = [\{a,b\},\{e\},\{\},\{\}] \quad \textit{and} \quad t_2{}^\sharp = [\{\},\{e\},\{\},\{\}]$$

as final states, respectively. The following caches states are contained in their respective concretizations:

$$[a,\bot,\bot,\bot],[a,b,e,\bot],[\bot,e,a,b] \in \gamma^\cup(t_1{}^\sharp)$$
$$[a,e,\bot,\bot],[e,\bot,\bot,\bot],[\bot,e,a,b] \in \gamma^\cap(t_2{}^\sharp)$$

Notice that both concretizations include the two possible states $[a,e,\bot,\bot]$ and $[b,e,\bot,\bot]$ (which is due to the soundness of the analyses) but also impossible states (which is due to the imprecision of the analysis). In particular, states in which empty cache lines are followed by non-empty cache lines are artifacts of the abstraction (i.e. they cannot occur according to the concrete cache semantics from [21], as we prove in the Appendix). More precisely, we have

$$\forall i,j \in \{1,\ldots,A\} \colon t(i) = \bot \wedge j > i \implies t(j) = \emptyset \ . \tag{6.1}$$

It is hence sufficient to consider only the concrete states that also satisfy (6.1), which enables us to derive tighter bounds in Section 6.3. For simplicity of notation we will implicitly assume that (6.1) is part of the definition of $\gamma^\cup$ and $\gamma^\cap$.

To obtain the channel corresponding to the adversary model Adv_{prob}, we just need to apply *blur* to the concretization of the must and may cache analysis, which is equivalent to first applying *blur* to the sets appearing in the abstract elements and then concretizing.

6.3 Counting Cache States

We have introduced channels corresponding to two adversaries, together with sound abstract interpretations. The final step needed for obtaining an automatic quantitative information-flow analysis from Theorem 4 are algorithms for counting the concretizations of the abstract cache states presented in Section 6.2.3, which we present next. As before, we restrict our presentation to single cache sets. Counting concretizations of caches with multiple sets can be done by taking the product of the number of concretizations of each set.

6.3.1 Concrete states respecting *may*

We begin by deriving a formula for counting the concretizations of an abstract may-state ${t_1}^\sharp$. To this end, let $n_i = \left|{t_1}^\sharp(i)\right|$, $n^*_i = \sum_{j=1}^{i} n_j$, for all $i \in \{1,\ldots,A\}$ and $n^* = n^*_A$. The definition of $\gamma^\cup({t_1}^\sharp)$ informally states that when reading the content of $t^\sharp$ and $t \in \gamma^\cup({t_1}^\sharp)$ from head to tail in lockstep, each non-empty line in t has appeared in the same or a previous line of ${t_1}^\sharp$. That is, for filling line k of t there are n^*_k possibilities, of which $k-1$ are already used for filling lines $1,\ldots,k-1$. The number of concrete states with a fixed number i of non-empty lines is hence given by

$$\prod_{k=1}^{i} \left(n^*_k - (k-1)\right) \tag{6.2}$$

As the definition of $\gamma^\cup$ does not put a lower bound on the number i of nonempty lines, we need to consider all $i \in \{1,\ldots,A\}$. We obtain the following explicit formula for the number of concretizations of ${t_1}^\sharp$.

Proposition 2 (Counting May)**.**

$$\left|\gamma^\cup({t_1}^\sharp)\right| = \sum_{i=0}^{A} \prod_{k=1}^{i} \left(n^*_k - (k-1)\right)$$

Example 5. *When applied to the abstract may-state* ${t_1}^\sharp = [\{a,b\},\{e\},\{\},\{\}]$ *obtained from the analysis of the program in Example 4 we obtain* $\left|\gamma^\cup({t_1}^\sharp)\right| = 11$*, which illustrates that the bounds obtained by Proposition 2 can be coarse.*

6.3.2 Concrete states respecting *must*

For counting the concretizations of an abstract must-state ${t_2}^\sharp$, let $m_i = \left|{t_2}^\sharp(i)\right|$, $m^*_i = \sum_{j=1}^{i} m_j$, for all $i \in \{1,\ldots,A\}$ and $m^* = m^*_A$. The definition of $\gamma^\cap$ informally states that when reading the lines of an abstract state ${t_2}^\sharp$ and a concrete state $t \in \gamma^\cap({t_2}^\sharp)$ from head to tail in lockstep, each element of ${t_2}^\sharp$ has already appeared in the same or a previous line of t. More precisely, the m_j elements contained in line j of ${t_2}^\sharp$ appear in lines $1,\ldots,j$ of t, of which m^*_{j-1} are already occupied by the must-constraints of lines $1,\ldots,j-1$. This leaves $\binom{j-m^*_{j-1}}{m_j} m_j!$ possibilities for placing the elements of ${t_2}^\sharp(j)$, which amounts to a total of

$$\prod_{j=1}^{A} \binom{j - m^*_{j-1}}{m_j} m_j! \tag{6.3}$$

possibilities for placing all elements in ${t_2}^\sharp$. However, notice that $m^* \leq A$ is possible, i.e. must-constraints can leave cache lines unspecified. The number

of possibilities for filling those unspecified lines is

$$\prod_{k=m^*+1}^{A} (\ell - (k-1)) \ , \tag{6.4}$$

where $\ell = |loc|$ is the number of possible memory locations.

Finally, observe that (6.3) and (6.4) count concrete states in which each line is filled. However, the definition $\gamma^{\cap}$ only mandates that at least m^* lines of each concrete state be filled. We account for this by introducing a variable i that ranges from m^* to A. We modify (6.3) by choosing from $\min(i, j)$ instead of j positions[1] and we modify (6.4) by replacing the upper bound by i. This yields the following for explicit formula for the number of concretizations of ${t_2}^\sharp$.

Proposition 3 (Counting Must)**.**

$$\left|\gamma^{\cap}({t_2}^\sharp)\right| = \sum_{i=m^*}^{A} \left(\prod_{j=1}^{A} \binom{\min(i,j) - m^*_{j-1}}{m_j} m_j! \prod_{k=m^*+1}^{i} (\ell - (k-1)) \right)$$

Example 6. *When applied to the must-state* ${t_2}^\sharp = [\{\}, \{e\}, \{\}, \{\}]$ *and a set of locations* $loc = \{a, b, c, d, e\}$*, Proposition 3 yields a number of* 81 *concretizations of* ${t_2}^\sharp$*. This over-approximation stems from the fact that the abstract state requires only the containment of* e *and that the rest of the lines can be chosen from loc.*

We next tackle this imprecision by considering the intersection of may and must.

6.3.3 Concrete states respecting *must* and *may*

For computing the number of concrete states respecting both ${t_2}^\sharp$ and ${t_1}^\sharp$ we reuse the notation introduced in Sections 6.3.1 and 6.3.2. As in Section 6.3.2 we use (6.3) for counting the cache lines constrained by the must-information. However, instead of filling the unconstrained lines with *all* possible memory locations, we now choose only from the lines specified by the may-information. The counting is similar to equation (6.2), the difference being that, as in (6.4), the product starts with $k = m^* + 1$ because the content of m^* lines is already fixed by the must-constraints. The key difference to (6.4) is that now we pick only from at most n^*_k lines instead of ℓ lines. We obtain the following proposition.

Proposition 4 (Counting May and Must)**.**

$$\left|\gamma^{\cup}({t_1}^\sharp) \cap \gamma^{\cap}({t_2}^\sharp)\right| \leq \sum_{i=m^*}^{A} \left(\prod_{j=1}^{A} \binom{\min(i,j) - m^*_{j-1}}{m_j} m_j! \prod_{k=m^*+1}^{i} (n^*_k - (k-1)) \right)$$

[1]The index j still needs to go up to A in order to collect all constraints

Two comments are in order. First, notice that the inequality Proposition 4 stems from the fact that the lines unconstrained by the must-information may be located at positions $j < k$. Using the constraint n_j^* instead of n_k^* would lead to tighter bounds, however, an explicit formula for this case remains elusive. Second, observe that the rightmost product is always non-negative. For this it is sufficient to prove that the first factor $n_{m^*+1}^* - m^*$ is non-negative, because the value of subsequent factors decreases by at most 1. Assume that $n_{m^*+1}^* - m^* < 0$ (and hence $n_{m^*}^* < m^*$). By (6.1), $n_j^* < j$ implies that line j is empty for all concrete states, which for $j = m^*$ contradicts the requirement that all states contain at least m^* lines.

Example 7. *When applied to the abstract cache states* ${t_1}^\sharp = [\{a,b\},\{e\},\{\},\{\}]$ *and* ${t_2}^\sharp = [\{\},\{e\},\{\},\{\}]$ *from Example 4, Proposition 4 delivers a total of* 9 *concrete states.*

It is easy to see that the expression in Proposition 4 can be evaluated in time $O(A^3)$ because both the factorial and and n_i^* can be computed in linear time and they are nested in two loops of length at most A. Although efficient, an approximation using Proposition 4 can be coarse: In Example 7 we computed a bound of 9 states, although (as is easily verified manually) there are only 4 concrete states respecting the constraints of both abstract states. We have developed more accurate (but more complex) variants of Proposition 4 that yield the exact bounds for this example, however, they are also not tight in general.

In the absence of a closed expression for the exact number of concrete states, one can proceed by enumerating the set of all concrete states respecting may, and filtering out those not respecting must. We present an implementation of the exact counting by enumeration in Section 6.4.2 The price to pay for this brute-force approach is a worst-case time complexity of $O(A!)$, e.g. if there are no must-constraints and the first location of the abstract may-state contains A or more locations. This is not a limitation for the small associativities often encountered in practice ($A = 2$ or $A = 4$), however, for fully associative caches in which A equals the total number of lines of the cache, the approximation given by Proposition 4 is the more adequate tool.

6.3.4 Counting for Probing Adversaries

For counting the possible observations of Adv_{prob} for arbitrary replacement strategies, we can apply the techniques presented above to previously blurred abstract states. For the case of a LRU strategy, we obtain the following better bounds.

Proposition 5. *The number of observations* Adv_{prob} *can make is bounded by*

$$min(n^*, A) - m^* + 1$$

The assertion follows from the fact that, after the computation, each cache set will first contain the victim's locations (which Adv_{prob} cannot distinguish), and then a fixed sequence of locations from the adversary's virtual memory whose length only depends on the number of the victim's blocks. I.e., when starting from an empty cache set, the adversary can only observe the length of the final cache set. This size is at least m^* (because at least that number of lines must be filled), and at most $\min(n^*, A)$. The additional 1 accounts for the empty state.

6.4 Implementation

In this section we report on the implementation of a tool for quantifying cache leaks. Its building blocks are the AbsInt a^3 tool for static cache analysis, and a novel counting engine for cache-states based on the results presented in Section 6.3.

6.4.1 Abstract Interpreter

The AbsInt a^3 [1] is a suite of industral-strength tools for the static analysis of embedded systems. In particular, a^3 comprises tools (called aiT and TimingExplorer) for the estimation of worst-case execution times based on the static cache analysis by Alt et al. [21]. The tools cover a wide range of CPUs, such as ERC32, TriCore, M68020, LEON3 and several PowerPC models (aiT), as well as CPU models with freely configurable LRU cache (TimingExplorer). We base our implementation on the TimingExplorer for reasons of flexibility.

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

6.4.2 Counting Engine

We implemented an engine for counting the concretizations of abstract cache states according to the development in Section 6.3. Our language of choice is Haskell [7], because it allows for a concise representation of sums, products, and enumerations using list comprehensions. We have implemented both the approximate counting described in the previous section, and an exact counting (which as discussed above can be more than exponential in worst case).

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

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

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

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

As described in Section 6.3.3, this is achieved by enumerating all concrete states that satisfy a given set of may-constraints (done by `genAllMay`), and keeping only those that also satisfy the must-constraints (done by filtering with `checkMust`). At the core of the function `genAllMay` is the following function `genMay` that returns all concretizations of the same length as the given abstract set,

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

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

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

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

For the approximate counting we have a `countMay` function:

```
countMay :: AbstractSet ->  Int
countMay xs = sum [ productMay xs i 1 | i <- [0..n]]
                              where n = length xs
```

that uses an auxiliary `productMay`:

```
productMay :: AbstractSet -> Int -> Int -> Int
productMay xs up low = product [ (star xs j) - (j-1) | j <- [low..up]]
```

This modularization is useful for the `countMayAndMust` procedure (which also uses a similar abstraction for the internal product of the `countMust` function):

```
countMayAndMust :: AbstractSet -> AbstractSet -> Int
countMayAndMust xs ys = sum [ (productMay xs i ((star ys n) + 1))
                         *(productMust ys i) | i <-[(star ys n)..n] ]
                         where n = length xs
```

6.5 Case Study

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

6.5.1 Target Implementations

Code. We analyze the implementation of 128 bit AES encryption from the PolarSSL library [5], a lightweight crypto suite for embedded platforms. As is standard for software implementations of AES, the code consists of single loop (corresponding to the rounds of AES) in which heavy table lookups are performed to indices computed using bit-shifting and masking, see Appendix B.2 for details. We also analyze a modified version of this implementation, where we add a loop that loads the entire lookup table into the cache before encryption. This preloading has been suggested as countermeasure against cache attacks because, intuitively, all lookups during encryption will hit the cache.

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

6.5.2 Improving Precision by Partitioning

The TimingExplorer can be very precise for simple expressions, but loses precision when analyzing array lookups to non-constant indexes. This source of imprecision is well-known in static analysis, and abstract interpretation offers techniques to regain precision, such as abstract domains specialized for arrays, or automatic refinement of transfer functions. For our analysis, we use results on *trace partitioning* [98], which consists in performing the analysis on a partition of all possible runs of a program, each partition yielding more precise results.

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

change the cache state for an LRU cache strategy, since the indices have to be fetched before accessing the table anyway. An example of the AES code with trace partitioning can be found in Appendix B.2.

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

6.5.3 Results and Security Interpretation

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

The results of our analysis with respect to the (less powerful, but more realistic) adversary Adv_{prob} are depicted in Figure 2. As for Adv_{prec}, the bounds obtained without preloading exceed the size of the secret key. With preloading, however, they remain below 6 bits and even drop to 0 bits for caches of 128KB, giving a formal proof of noninterference for this implementation and platform.

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

6.6 Related work

Timing attacks against cryptosystems date back to [85]. They can be divided into those exploiting timing variations due to control-flow [85, 39] and those exploiting timing variations of the execution platform, e.g. due to caches [115, 31, 117, 114, 16, 18], or branch prediction units [17]. In this work we focus solely on caching.

The literature on cache attacks is stratified according to a variety of different adversary models: In *time-driven attacks*[31, 18] the adversary can observe the overall execution time of the victim process and estimate the overall number of cache hits and misses. In *trace-driven attacks* [16] the adversary can observe whether a cache hit or miss occurs, for every single memory access of the victim process. In *access-driven attacks* [117, 114] the

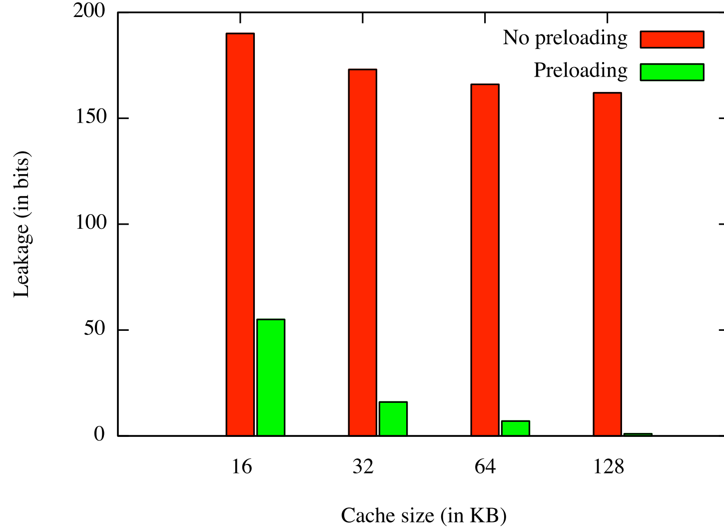


Figure 1: Upper bounds for the maximal leakage w.r.t. the adversary Adv_{prec} and a 4-way set associative cache with 32B lines of sizes 16KB-128KB

adversary can probe the cache either during computation (*asynchronous* attacks) or after completion (*synchronous* attacks) of the victim's computation, giving him partial information about the memory locations accessed by the victim. Finally, some attacks assume that the adversary can choose the cache state before execution of the victim process [114], whereas others require that the cache does not contain the locations that looked-up by the victim during execution [18]. The information-theoretic bounds we derive hold for single executions of synchronous access-driven adversaries, where we consider initial states that are either empty or do not contain the victim's data. The derivation of bounds for alternative adversary models is left future work.

A number of mitigation techniques have been proposed to counter cache attacks. Examples include coding guidelines [46] for thwarting cache attacks on x86 CPUs, or novel cache-architectures that are more resistant to cache attacks [137]. One commony proposed technique is preloading of tables [114, 31]. However, as observed by [114],

> [...], it should be ensured that the table elements are not evicted by the encryption itself, by accesses to the stack, inputs or outputs. Ensuring this is a delicate architecture-dependent affair [...]."

The methods developed in this chapter enable us to perform a formal analysis of the preloading heuristic. 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 [131]. The goal of our work is different in that

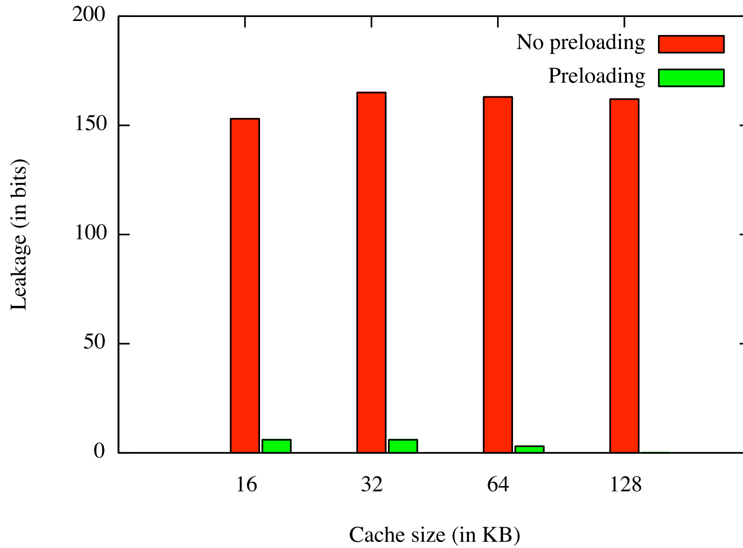


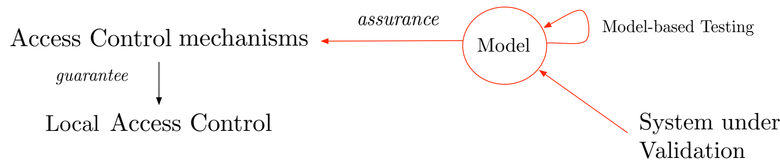
Figure 2: Upper bounds for the maximal leakage w.r.t. the adversary Adv_{prob} and a 4-way set associative cache with 32B lines of sizes 16KB-128KB

we aim for provable security guarantees based on accurate processor models and the actual code.

Technically, our work builds on methods from quantitative information-flow analysis (QIF) [43], where the automation by reduction to counting problems appears in [27, 109, 71, 102], and the connection to abstract interpretation in [90]. Prior applications of QIF to side-channels in cryptosystems [87, 88, 91] are limited to stateless systems. For the analysis of caches, we rely on the abstract domains from [21] and their implementation in the AbsInt TimingExplorer [1]. Finally, our work goes beyond language-based approaches that consider caching [19, 70] in that we rely on more realistic models of caches and aim for more permissive, quantitative guarantees.

Chapter 7

Model-Based Testing



In this chapter we discuss preliminary results on *availability* properties, that although not the main focus of this thesis, constitute an interesting and promising research direction. Moreover, some of the technical framework to reason about change discussed in Chapter 3 can be instantiated for the properties and notation under consideration. This is possible because the models used for the test generation are UML state-charts and the security consistency properties are of a structural nature.

Typically, UML models verified against security properties are explicit models of the *system* design as we have seen in the previous chapters. In conformance Model-Based Testing (MBT) the expected behaviour of an application is described, seen thus as a *blackbox*. In this chapter we propose an extension to UMLsec that allows to verify the consistence of models used for black-box MBT with respect to the security properties that are to be tested. We will show results on two security properties, but our approach can be generalized to other security properties related with access control and availability. These two properties assume an operating system with a finite number of states, that represent stages on the system's life-cycle (i.e. 'initialized', 'operation ready', 'blocked' and 'terminated'). The requirement is that only authorized applications may set the system into the 'terminated' status, and that whenever in that status, it is impossible by any other operation to revert it. Although inspired on actual requirements on smart-cards, these two properties are of interest to a wide range of devices: for example, it would be convenient to be able to remotely set stolen mobile phones

or laptops into a non-revertible non-operating status, while having strong guarantees that this can not be done by malicious applications.

To validate our approach, we demonstrate our results by experimental semi-automatic generation of test schemas using the language introduced in [53] on the Global Platform Specification for smart cards [9] and its evolution from V 2.1.1 to V2.2. We also report on prototypical tool support for our approach.

7.1 Consistency verification with UMLsec

In this section we describe our approach for well-founded security MBT and we explain how to profit from the UMLseCh approach for security annotated models presented in Chapter 3. Our main objective is that the model that is used for test generation is verified for consistency with respect to the considered security properties, and this consistency should hold also after the model has evolved. If not, the model may authorize an incorrect behaviour and the produced tests will expect from the System Under Test (SUT) to present the same erroneous behaviour as the model.

7.1.1 Security properties

We will illustrate our approach using the following two properties (originally defined for the Global Platform [9]) that are critical for a device issuer/owner in order to have control over compromised running devices.

Security Property 1, *locked-status*: *For any execution, whenever the system is set to the state TERMINATED by means of an operation performed by a privileged application, then it should not be possible to revert to another state.*

This property ensures that whenever an application with enough privileges terminates the system, the system cannot be put back in operation. This is an important feature to control devices running malicious applications or that have been compromised in some way (for example stolen or lost).

Security Property 2, *authorized-status*: *It should not be possible for an application that does not have the given privilege to set the system into a given state TERMINATED.*

Conversely, to avoid the Denial of Service (DoS) attacks on the system, only applications with sufficient privileges should be able to terminate it.

7.1.2 Extending UMLsec for these security properties

Assuming the SUT has a variable state representing the card status, we model the expected behaviour of the SUT as a statechart where its states represent the status of the card's life-cycle. We further assume there is a command set_status, only executable by privileged applications to change the card's status from one to another, and this is the event triggering all transitions in the model. To model failed attempts i.e change the card's status by a non privileged application, we allow internal transitions in a given state to represent them, for which the consequence is that a variable statusWord is affected with an error message.

Under these assumptions, a statechart in which from one state having the value {status} there are not only incoming but also *outcoming* transitions (with satisfiable guards, otherwise they would be superfluous) would be trivially violating the Security Property 1. This would contradict the property to test and could be the source of misinterpretations of the testing results. Potentially, it could also mean that the system specification is contradictory with respect to the wished security properties. To avoid this, we can extend UMLsec with a stereotype «locked-status» together with a tag {status} where a specific status can be defined. Semantically, a statechart annotated with this stereotype would require that there are not outgoing transitions from the state specified in {status}.

Similarly, we can define a stereotype «authorized» with two tags {status} and {permission}. This stereotype enforces that there exists no incoming transition to the status specified in {status} with a guard NOT containing {permission}. Under the assumption that in each transition from state to state we check for given application privileges, this stereotype would avoid having a model trivially violating Security Property 2.

These properties can be checked statically on UML statecharts since we are not aiming at verifying behavioural properties, but at ensuring a structural property as a precondition to the testing process. For example, the check performed by «authorized» on a statechart could be summarized by the following algorithm, where Status is the state corresponding to the value of {status} and the auxiliary function IncomingTransitions() returns all the incoming transitions relatively to that status.

```
Transitions := Status.IncomingTransitions();
for T in Transitions do
if Permission not in  T.Guard
return false;
```

In a similar way define the algorithm for «locked-status», where we check whether Status.OutgoingTransistions() is empty.

7.1.3 Transformation Rules of UMLsec stereotypes to Schemas

We model the SUT expected behaviour with a statechart where its states represent the status of the card's life-cycle. We further assume there is a command set_status, only executable by privileged applications to change the card's status from one to another, and this is the event triggering all transitions in the model. To model failed attempts i.e change the card's status by an non privileged application, we allow internal transitions in a given state to represent them, for which the consequence is that a variable statusWord is affected with an error message.

Under these assumptions, property 1 requires (written as a Hoare triple):

$$\{\mathsf{state} = \mathsf{TERMINATED}\}\ \mathsf{set_status}^{+}\ \{\mathsf{state} = \mathsf{TERMINATED}\}$$

that is, if we reach the state *TERMINATED*, then after an arbitrary number of calls to any operation, the resulting status should be the same. In other words, there should be no *outgoing transitions* from that status in the state-chart to states different than *TERMINATED*. To verify this, we have extended UMLsec with the «locked-status» stereotype with a {status} tag, that tries to find a counterexample to property 1 by negating it, that is we look for violating outgoing transitions in the model.

The second property can be written in a triple as:

$$\{\mathsf{state} \neq \mathsf{TERMINATED}\}\ \mathsf{set_status}^{+}\ \{\mathsf{state} \neq \mathsf{TERMINATED}\}$$

assuming that the application executing the operations has not enough privileges. For this, we have defined a stereotype «authorized» with two tags {status} and {permission}. This stereotype enforces that there exists no incoming transition to the status specified in {status} with a guard NOT containing {permission}. Under the assumption that in each transition from state to state we check for given application privileges, this stereotype would avoid having a model trivially violating Security Property 2. These properties can be checked statically on UML statecharts since we are not aiming at verifying behavioural properties, but at ensuring a structural property as a precondition to the testing process.

At the end of the verification process our goal is to export test schemas based on the properties specified before, encapsulating the expected behaviour of the system after executing particular instructions that could potentially violate the property and make the system not to behave as expected. This generation represents thus the link from UMLseCh to testing, since we can automatically generate test sequences from schemas. Moreover, we export also the delta of changes as specified with UMLseCh (see Section 4.4), from which we will trigger the automatic test generation for the evolved parts of the model.

7.2 Handling change

To reason about model evolution, as discussed in Chapter 3 we have defined a set of sufficient conditions that if respected by the delta specified in with UMLseCh will guarantee the preservation of the security properties on the evolved model. That is, we define rules for each *evolution type* (addition, deletion, substitution) and for each *UML type* (that is relevant for the property, in this case, transitions, guards and states). In other words, by parsing the evolution stereotypes («add», «del»,«substitute») we obtain a set of atomic elements $\Delta = o_1, \ldots, o_n$ that we can classify by its evolution type and UML type. We can then evaluate Δ element by element with the sufficient conditions, and in each step obtain a model that satisfies the desired property (otherwise we stop and report an error).

The definition of the stereotype «locked-status» with tag { status = Status } requires that no outgoing transition from a state with label *Status* exists. On the other hand, the stereotype «authorized-status» with tag { status = Status } and { permission = Permission } requires that every incoming transition to a state with label *Status* has within its guard the substring *Permission*.

Sufficient conditions for the preservation of both properties are given in the following. Note that most of the soundness proofs follow directly from the definition of the properties, and therefore are not explicitly given.

Deletion

State, Transition Deleting a state including all incoming and outgoing transitions (otherwise the resulting state chart would not be syntactically valid) does not alter neither of the two security properties. The same holds for deletion of a transition in both cases. The proofs follow directly from the definition.

Guard with respect to «authorized-status»: Let $D = \{o_1, ..., o_n\}$ the set of objects to be deleted . Let P be the security property with respect to stereotype «authorized-status», P is satisfied in the modified model if the following condition is fulfilled. If the deleted guard contains the condition { permission = Permission } and the transition corresponding to this guard has target state which is labeled as *Status*, the transition itself should be deleted. Formally, this is described by the rule:

$$\forall_{o \in D}(o.\mathsf{type} = guard \wedge Permission \in o.\mathsf{permission} \quad \wedge$$
$$o.\mathsf{parentTransition.targetState.label} = Status) \Rightarrow$$
$$\exists_{o' \in D}(o' = o.\mathsf{parentTransition})$$

Addition

State From the definition of both stereotypes, it follows that addition of a new state without any transition does not violate the security property from a state machine.

Transition with respect to «locked-status»: Let $A = \{o_1, ..., o_n\}$ the set of objects to be added. Let P be the security property defined by the stereotype «locked-status». P will be satisfied in the model after the additions in A if the source state from new transition o is labeled as $Status$, then the target state from the transition should be labeled as $Status$ too, i.e. the new transition should have the same state as source and target. We can express this rule as follows:

$$\forall_{o \in A}(o.\mathsf{type} = transition \wedge o.\mathsf{sourceState.label} = Status) \Rightarrow$$
$$(o.\mathsf{targetState} = o.\mathsf{sourceState})$$

The proof follows directly from the definition of P.

In the case of «authorized-status»: Let $A = \{o_1, ..., o_n\}$ the set of objects to be added. P is satisfied after committing each o in the set A if the following condition is fulfilled;

$$\forall_{o \in A}(o.\mathsf{type} = transition \wedge o.\mathsf{targetState.label} = Status) \Rightarrow$$
$$\exists_{o' \in A}(o' = o.\mathsf{guard} \wedge Permission \in o'.\mathsf{permission})$$

Again, this follows easily from the definition.

Guard Addition of a new guard does not alter the security properties.

Substitution

Let $S = \{((o_1, o'_1), ..., (o_n, o'_n))\}$ the set of objects pairs to be substituted.

State with respect to «locked-status»: In case that a state o , which is not labelled as $Status$, is substituted with a State o' , which is labelled as $Status$, o' should have no outgoing transition to preserve the secure requirement of the system. In the case of «authorized-status»: If a state o , which is not labelled as $Status$, is substituted with a state o' , which is labelled as $Status$, all incoming transitions to o' should have a guard, which contains the condition $\{\,\mathsf{permission} = \mathsf{Permission}\,\}$ to preserve the secure requirement of the system.

Transition Both cases are similar to their respective rules for the addition of a transition.

Guard with respect to «locked-status»: Substitution of a guard does not alter the secure properties of the system. In the case of «authorized-status»: a guard o which contains { permission = Permission } can be substituted with a guard o' containing this condition as well.

7.3 Validation

We have applied our methodology to a real case study: the Global Platform [9] in the context of the SecureChange project. The Global Platform is a non-profit organization involving over 60 industry members (including American Express, MasterCard, Visa, Nokia, Sun and Gemalto) that defines a publicly available smart card application management specification. The goal of this specification is to be a hardware and operating neutral system and to cover a wide range of security critical industrial applications and therefore focuses on many security aspects. For example precise protocols for the communication of the card with an application provider or central server are defined aiming at guaranteeing confidentiality, integrity and authenticity aspects of both over-the-air and terminal connections. Moreover, Global Platform supports external software updates. Implementations of the specification with tailored applications include Financial, Mobile telecommunications, Government initiatives, Healthcare, Retail merchants and Transit domains.

The scope of our work is the management of the card life cycle, from the card's production until its destruction. We have created test models for the version on the Card Life Cycle Scope of Global Platform 2.1.1 respecting the assumptions mentioned in the previous section: each status of the statechart correspond to a state of the card and each transition's guard from state to state checks for certain application privileges.

7.3.1 Correctness Verification with UMLsec

We have verified a life-cycle testing model representing the expected behaviour of the card according to the Global Platform 2.1.1 Specification with respect to the stereotypes «locked-status» and «authorized-status» using the UMLsec verification tool, which we have extended for these new stereotypes. The UMLsec tool takes a statechart diagram in XMI format as an input and runs the proposed verification process on it. For illustrative purposes a fragment of a violating statechart w.r.t «locked-status» for the Global Platform 2.1.1 life cycle is shown on Fig. 1. In this statechart, there is a transition coming out from TERMINATED to SECURED, which is a contradiction to the desired property. This is reported to the user, who can correct the model accordingly and re-verify it.

Figure 1: Example of a violating fragment of the GP 2.1.1 Life-Cycle modelled with ArgoUML

7.3.2 Schemas for the Security Properties

Although the formalization used (Hoare triples) to express the intention of the test can be generally applicable to any testing generation language, we have explored the automatic generation of schemas using the SmartTesting language [53]. Using transformation rules based on Hoare triples automatically generated with the instantiated permission and relevant state information, we have obtained the following test schema that reflects the security property w.r.t. to the test intention we have defined. We needed only to define manually the existing model instance, for which we want to generate tests:

```
for each literal $X from TERMINATED
for each literal $Y from TERMINATED
for each $Z from any operation
use any operation at least once
to reach state respecting (self.selectedApp.cardTermPriv = true)
on instance 'card' then
use  APDU_Set_Status any number of times
to reach state respecting (self.state=$X)
on instance 'card'' then
use $Z at least once
to reach state respecting (self.state = $Y)
on instance 'card'
```

Informally the *test intention* associated to this schema is:

- set the status of the card to TERMINATED;
- try all operations (to see if they behave as predicted by the model, i.e. by returning a status word of error).

The test intention for the authorized-status security property that we exhibit, is defined informally as a scenario to test the nominal case of failure of this security property:

- select any application without the Card Terminate Privilege
- set the card to a state different than terminated
- try to use the set status command to reach the TERMINATED with an unprivileged application and check that the system behaves as predicted.

Then from the verified stereotype and the exported Hoare triple, we generate the corresponding *test schema*:

```
for each litteral $STATE from LIST
use any operation any number of times
to reach self.selectedApp.cardTermPriv = false
and self.state =$STATE
on instance 'card'
then use APDU_Set_Status( , , CARD, TERMINATED, )
```

where LIST = OP_READY, INITIALIZED, SECURED, CARD_LOCKED.

7.4 Related work

Recently, the use of models to describe the expected behaviour of a system have been proposed in the context of conformance testing for security properties [35, 78, 135]. There exist different approaches to model-based testing. One possibility is the test generation based on model-checkers by generating execution sequences that contradict the properties (see [57, 22] for example). This properties as proposed by M. Dwyer [51] can be defined by a temporal pattern. Other possibility is the translation of security properties into test target to cover test needs associated to the properties. This approach uses the same set of actions of the model. Some elements to drive the test generation can be added. For example, in input/Output Symbolic or Labelled Transition Systems [73, 56] two trap states named *Accept* and *Refuse* are added. The *Accept* states are used as end states for the test generation while the *Refuse* states allow for cutting the traces not wanted in the generated tests. These formalisms are for example used in tools such as TGV [73], STG [44], TorX [132], Agatha [34]. Some approaches are based on the definition of scenarios for the test in the model. In [32][29], test cases are issued from UML diagrams as a set of trees. The scenarios are extracted by a breadth-first search on the trees. A similar approach is implemented in the tool *Telling TestStories* [52], based on defining a test model from elementary test sequences made of an initial state, a *test story* and test data. In [83] the authors propose an approach for systematically generating test sequences for security properties in a model-based way that can be used to test the implementation for vulnerabilities.

In [40] authors use scenarios based on regular expressions, to enrich the test generation test suite produced by the Smartesting Test Designer Tool, which cannot generate tests for dynamic system properties. Their work, is an adaptation for UML base on the work done in [97]. This language was designed during an industrial project dedicated to testing the conformance of a system to a security policy. In the smart-card application domain, in [42] Chetali has pointed out the need to have an automated and formally sound approach allowing to prove security properties in the context of security certification, and the requirements such an approach should fulfil. The results discussed in this chapter are a step towards this direction.

Chapter 8

Conclusions

This thesis discusses the preservation of exemplary security properties of models in different evolution scenarios and at different levels of abstraction, focusing on confidentiality properties. For structural properties, we considered selected classes of changes in models such as addition, deletion, and substitution of model elements based on UMLsec diagrams. Assuming that the starting UMLsec diagrams are secure, which one can verify using the UMLsec tool framework, our goal was to re-use these existing verification results to minimize the effort for the security verification of the evolved UMLsec diagrams. In case of model consistency properties that are local enough, we have discussed a fine-grained incremental analysis. This technique can be applied to discuss the preservation of UMLsec security properties on structural diagrams such as class diagrams, deployment diagrams and for some properties even on state-charts. For example, the two model consistency properties considered as a basis for a sound model-based-testing analysis are defined on UML state-charts and because of their local nature we have been able to spell out sufficient conditions for the preservation of security after atomic changes. A prototypical implementation of this verification techniques has been implemented that has also been used as a basis for evaluating the practical efficiency of our methodology, allowing us to preliminary demonstrate the feasibility of this methodology.

On the other hand we have presented a light-weight verification strategy for state-charts that is sound with respect to classical non-interference. Our technique is fully automatic and can help to bridge the gap between theory and practice for information-flow in secure-software development in industrial context, by applying our results to a non-trivial subset of UML Statecharts, extending previous work in the area. On a technical level, we have shown how to link unwinding theorems defined in input/output state-machines with verification techniques related to the imperative programming language domain. With respect to model evolution, we have shown that non-interference

is compositional in this setting for the composition notion used in UMLsec. This is a useful result for reasoning about changes in the model, since in the software life-span components might be added, deleted or substituted for which it will be desirable to reuse existing verification results. To validate our approach we have modelled interesting aspects of a Smart Metering scenario where subtle information flows related to confidentiality can be captured by non-interference. These examples show that although approximate, our unwinding theorems are fine-grained enough to verify non-trivial state-charts. We have also prototypically implemented the construction and the verification of the newly defined unwinding relation and unwinding conditions. This implementation allowed us to verify the examples of the case study and discuss about the practical efficiency of the procedure.

Because the non-interference analysis performed on UML State-charts assumes some kind of access control to separate the inputs and outputs of different groups of users, we have also analysed cryptographic protocols, which are an important building block for network access control. We have shown a decision procedure for the compositionality of secrecy in the setting of processes exchanging symbolic cryptographic messages that is sound and complete with respect to previous work on First Order Logic protocol verification. We illustrate our approach by applying it to an insecure variant of TLS and its fix. To reason empirically about the efficiency of our approach, we compare the running time of the composition of multiple processes. Tool support, based on Prolog, supports the validation of protocols specified using UML Sequence diagrams.

To be able to justify perfect cryptography as it is assumed by the Dolev-Yao analysis one should use cryptographic implementations that are secure against side-channel attacks. We have shown that cache side-channels can be automatically quantified using static cache analysis and quantitative information-flow analysis. We have demonstrated the practicality of this approach by deriving information-theoretic security guarantees for an off-the-shelf implementation of 128-bit AES (with and without a commonly suggested countermeasure) on a realistic model of an embedded CPU.

In summary, we have contributed incremental and compositional techniques to reason about confidentiality at different levels of abstraction and highlighted their informal relation. These reasoning tools allow to handle changes to models and their impact to the overall security of the system. The feasibility of our approach is backed up by proof-of-concept implementations that have allowed us also to discuss heuristically about the computational time required by our algorithms.

As it is the case with most doctoral works, there are many directions in which our work could be extended. In general, a formal soundness of the way we link different security properties and levels of abstraction constitutes an interesting and challenging topic for future work. Also, extensions to other information flow properties for non-deterministic state-machines are

interesting to study in the context of UML. The study of more complex subset of the state-charts, allowing for example call-backs and recursion is a challenging and interesting subject. Moreover, studying the preservation of non-interference on code generated from secure UML specifications (refinement) constitutes also a necessary step towards industrial acceptance of these verification techniques. Improvements on the precision of our approximations through unwinding theorems is an important topic as well. With respect to our work in CPU caches and side-channel analysis, it would be interesting to develop abstract cache domains that enable the derivation of bounds that hold for an arbitrary number of executions of the victim process, and to extend our quantification to account for alternative adversary models, such as asynchronous, trace-based, and timing-based.

Finally, a thorough analysis of the usability and efficiency of the proposed approach would certainly be necessary to determine the precise applicability in industrial domains. We have discussed some preliminary results in this direction, but it would be interesting future work to systematically validate them with different groups of users and in a realistic production environment. This however goes outside of the scope of our research.

Bibliography

[1] AbsInt aiT Worst-Case Execution Time Analyzers. http://www.absint.com/a3/.

[2] Arm7tdmi datasheet. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf.

[3] GNU ARM. http://www.gnuarm.com/.

[4] Jif: Java + Information Flow. http://www.cs.cornell.edu/jif/.

[5] PolarSSL. http://polarssl.org/.

[6] STAN: Information flow analysis for small embedded systems. http://stan-project.gforge.inria.fr/.

[7] The Haskell Programming Language. http://www.haskell.org/.

[8] The Open Source toolkit for SSL/TSL. http://www.openssl.org/.

[9] Global platform specification. http://www.globalplatform.org/specificationscard.asp, May 2011.

[10] LulzSec hackers claim CIA website shutdown. BBC news, 2011. http://www.bbc.co.uk/news/technology-13787229.

[11] LulzSec takes down Brazil government sites. Cnet news, 2011. http://news.cnet.com/8301-1009_3-20073219-83/lulzsec-takes-down-brazil-government-sites/.

[12] The European Parliament and Council. Measuring instruments directive (2004/22/ec). Published in the Official Journal of the EU, http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2004:135:0001:0080:EN:PDF, April 2004.

[13] M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, Amsterdam, 2000. 20th International Summer School, Marktoberdorf, Germany.

[14] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

[15] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Proc. 12th International Workshop of Cryptographic Hardware and Embedded Systems (CHES 2010)*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.

[16] O. Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (Short Paper). In *8th International Conference on Information and Communications Security (ICICS 2006)*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2006.

[17] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference (CT-RSA '07)*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.

[18] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the aes. In *The Cryptographers' Track at the RSA Conference 2007 (CT-RSA 2007)*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2007.

[19] J. Agat. Transforming out Timing Leaks. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL 2000)*, pages 40–53. ACM, 2000.

[20] K. Alghathbar, C. Farkas, and D. Wijesekera. Securing UML information flow using flowUML. In *Journal of Research and Practice in Information Technology*, pages 229–238. INSTICC Press, 2006.

[21] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *3rd International Symposium on Static Analysis (SAS 1996)*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[22] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. *Formal Engineering Methods, International Conference on*, page 46, 1998.

[23] R. J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.)*. Wiley, 2008.

[24] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[25] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *IN PROCEEDINGS OF THE IEEE INFOCOM*, pages 717–725, 1999.

[26] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In V. Shmatikov, editor, *FMSE*, pages 1–10. ACM, 2008.

[27] M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. 30th IEEE Symposium on Security and Privacy (S& P '09)*, pages 141–153. IEEE, 2009.

[28] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 125–140. Springer LNCS, 2007.

[29] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite approach to planning and deriving test suites in UML projects. In *UML'02, 5-th int. conf. on the UML language*, volume 2460 of *LNCS*, pages 383–397, London, UK, 2002.

[30] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[31] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[32] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electron. Notes Theor. Comput. Sci.*, 116:85–97, Jan. 2005.

[33] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 440–453. Springer, 2006.

[34] C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin. Automatic test generation with AGATHA. In H. Garavel and J. Hatcliff, editors, *TACAS 2003, Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference*, volume 2619 of *LNCS*, pages 591–596. Springer, 2003.

[35] M. Blackburn, R. Busser, and A. Nauman. Model-based approach to security test automation. In *International Software Quality Week*, Nov. 2002.

[36] C. Braun, K. Chatzikokolakis, and C. Palamidessi. Quantitative notions of leakage for one-try attacks. In *Proc. of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009)*, volume 249 of *ENTCS*, pages 75–91. Elsevier, 2009.

[37] M. Broy. A logical basis for component-based systems engineering. In *Calculational System Design. IOS*. Press, 1999.

[38] M. Broy. A logical basis for component-oriented software and systems engineering. *Comput. J.*, 53:1758–1782, December 2010.

[39] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[40] K. Cabrera Castillos and J. Botella. Scenario based test generation using test designer. In *SCENARIOS'11, 1st Int. Workshop on Scenario Based Testing – co-located with ICST'2011*, Berlin, Germany, Mar. 2011. IEEE Computer Society Press. To appear.

[41] C. Cachin. Entropy Measures and Unconditional Security in Cryptography. Dissertation No. 12187. ETH Zürich, 1997.

[42] B. Chetali. Security testing and formal methods for high levels certification of smart cards. In *Proceedings of the 3rd International Conference on Tests and Proofs*, TAP '09, pages 1–5, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[44] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS'02, Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 151–173. Springer, 2002.

[45] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Annual Symposium on Logic in Computer Science (LICS)*, pages 353–362, June 1989.

[46] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proc. 2009 IEEE Symposium on Security and Privacy (Oakland 2009)*, pages 45–60. IEEE Computer Society, 2009.

[47] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Pinciples of Programming Languages (POPL '77)*, pages 238–252.

[48] J. Cúellar, M. Ochoa, and R. Rios. Indistinguishable regions in geographic location privacy. In *Proc. of the 2012 ACM Symposium on Applied Computing (SAC), Security Track*. ACM, 2012. To appear.

[49] D. Das, F. Kreikebaum, D. Divan, and F. Lambert. Reducing transmission investment to meet renewable portfolio standards using smart wires. In *2010 IEEE PES Transmission and Distribution Conference and Exposition: Smart Solutions for a Changing World*, 2010.

[50] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172(0):311 – 358, 2007. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

[51] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE'99, 21st international conference on Software engineering*, pages 411–420, LA, California, United States, 1999.

[52] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for Model-based Requirements Testing of Service Oriented Systems. In *Proceedings of the IASTED International Conference*, volume 642, page 018, 2009.

[53] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon. Selective test generation method for evolving critical systems. In *REGRESSION'11, 1st Int. Workshop on Regression Testing - co-located with ICST'2011*, Berlin, Germany, Mar. 2011. IEEE Computer Society Press. To appear.

[54] E. Fourneret, F. Bouquet, M. Ochoa, J. Jürjens, and S. Wenzel. Vérification et test pour des systèmes évolutifs. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2012.

[55] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jürjens, and P. Yousefi. Model-based security verification and testing for smartcards. In *ARES 2011, 6-th Int. Conf. on Availability, Reliability and Security*, Vienna, Austria, Aug. 2011.

[56] L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004, Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 1–15. Springer, 2005.

[57] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, 1999.

[58] D. Garlan, J. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *WICSA/ECSA 2009*, pages 131 –140, sept. 2009.

[59] D. Ghindici, G. Grimaud, and I. Simplot-Ryl. Embedding verifiable information flow analysis. In *Proc. Annual Conference on Privacy, Security and Trust*, pages 343–352, Toronto, Canada, nov 2006.

[60] D. Giffhorn and C. Hammer. Precise analysis of java programs using joana (tool demonstration). In *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 267–268, September 2008.

[61] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[62] J. A. Goguen and J. Meseguer. Unwinding and inference control. *Security and Privacy, IEEE Symposium on*, 0:75, 1984.

[63] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on aes to practice. In *Proc. 2011 IEEE Symposium on Security and Privacy (Oakland 2011)*, pages 490–505. IEEE Computer Society, 2011.

[64] J. D. Guttman. Cryptographic protocol composition via the authentication tests. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 303–317, Berlin, Heidelberg, 2009. Springer-Verlag.

[65] J. D. Guttman, F. Javier, and F. J. T. FÃ¡brega. Protocol independence through disjoint encryption. In *In Proceedings, 13th Computer Security Foundations Workshop. IEEE Computer*, pages 24–34. Society Press, 2000.

[66] A. Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, Sept. 1990.

[67] C. Hammer. Information flow control for java based on path conditions in dependence graphs. In *In IEEE International Symposium on Secure Software Engineering*, 2006.

[68] D. Harel. Statecharts: A visual formalism for complex systems, 1987.

[69] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Proceedings of international conference on Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.

[70] D. Hedin and D. Sands. Timing Aware Information Flow Security for a JavaCard-like Bytecode. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 141(1):163–182, 2005.

[71] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *26th Annual Computer Security Applications Conference, (ACSAC '10)*, pages 261–269. ACM, 2010.

[72] International Electrotechnical Commission (IEC). Iec 62351 parts 1-8, information security for power system control operations. power system control and associated communications - data and communication security.

[73] C. Jard and T. Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[74] S. Johann and A. Egyed. Instant and incremental transformation of models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 362–365, Washington, DC, USA, 2004. IEEE Computer Society.

[75] J. Jürjens. Composability of secrecy. In *Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security*, MMM-ACNS '01, pages 28–38, London, UK, 2001. Springer-Verlag.

[76] J. Jürjens. *Principles for Secure Systems Design.* PhD thesis, Oxford University Computing Laboratory, 2002.

[77] J. Jürjens. *Secure Systems Development with UML.* Springer, 2005.

[78] J. Jürjens. Model-based security testing using umlsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93 – 104, 2008.

[79] J. Jürjens. A domain-specific language for cryptographic protocols based on streams. *J. Log. Algebr. Program.*, 78(2):54–73, 2009.

[80] J. Jürjens, L. Marchal, M. Ochoa, and H. Schmidt. Incremental Security Verification for Evolving UMLsec models. In *Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA)*, LNCS, pages 52–68. Springer, 2011.

[81] J. Jürjens, M. Ochoa, H. Schmidt, L. Marchal, S. Houmb, and S. Islam. Modelling secure systems evolution: Abstract and concrete change specifications (invited lecture). In I. Bernardo, editor, *11th School on Formal Methods (SFM 2011), Bertinoro (Italy) 13-18 June 2011*, LNCS. Springer, 2011.

[82] J. Jürjens and P. Shabalin. Tools for secure systems development with UML. *Intern. Journal on Software Tools for Technology Transfer*, 9(5–6):527–544, Oct. 2007. Invited submission to the special issue for FASE 2004/05.

[83] J. Jürjens and G. Wimmel. Formally testing fail-safety of electronic purse protocols. In *16th International Conference on Automated Software Engineering (ASE 2001)*, pages 408–411. IEEE Computer Society, 2001.

[84] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[85] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. Advances in Cryptology (CRYPTO 1996)*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[86] D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update transformations in the small with the epsilon wizard language. *Journal of Object Technology*, 6(9):53–69, 2007.

[87] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proc. 14th ACM Conference on Computer and Communication Security (CCS 2007)*, pages 286–296. ACM, 2007.

[88] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 324–335. IEEE Computer Society, 2009.

[89] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, 2012. To appear.

[90] B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proc. 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 3–14. IEEE Computer Society, 2010.

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

[92] D. Kuhn, R. Chandramouli, and R. Butler. Cost Effective Uses of Formal Methods in V&V. In *Foundations '02 Workshop, US Dept of Defense, Laurel MD*, 2002.

[93] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution – The Nineties View. In *METRICS'97*, pages 20–32, Washington, DC, USA, 1997. IEEE Computer Society.

[94] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17(3):93–102, 1996.

[95] H. Mantel. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 88–101, Oakland, CA, USA, 2002. IEEE Computer Society.

[96] H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, Juli 2003.

[97] P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, and A. Haddad. An access control model based testing approach for smart card applications: Results of the POSÉ project. *JIAS, Journal of Information Assurance and Security*, 5(1):335–351, 2010.

[98] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[99] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 237–250. IEEE Computer Society, 2000.

[100] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[101] D. Mellado, J. Rodriguez, E. Fernandez-Medina, and M. Piattini. Automated Support for Security Requirements Engineering in Software Product Line Domain Engineering. In *AReS'09*, pages 224–231, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[102] Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *6th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '11)*. ACM, 2011.

[103] T. Mens and T. D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering Journal*, 7(1):39–59, February 2000.

[104] T. Mens, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *Computer*, 43(5):42 –48, May 2010.

[105] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[106] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murphi. pages 141–151. IEEE Computer Society Press, 1997.

[107] National Energy Technology Laboratory. A vision for the smart grid. Report, June 2009. Available via: http://www.netl.doe.gov/moderngrid/.

[108] Network of Excellence on Engineering Secure Future Internet Software Services and Systems (Nessos). Deliverable 11.2, 2011.

[109] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proc. Workshop on Programming Languages and Analysis for Security (PLAS '09)*, pages 73–85. ACM, 2009.

[110] M. Ochoa. Security guarantees and evolution: From models to reality. In *Electronic Proc. of the 1st ESSoS Doctoral Symposium, ESSoS-DS 2012*. CEUR-WS.org, 2012.

[111] M. Ochoa, J. Jürjens, and J. Cuéllar. Non-interference on UML Statecharts. In *50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe 2012)*, LNCS. Springer, 2012. To appear.

[112] M. Ochoa, J. Jürjens, and D. Warzecha. A sound decision procedure for the compositionality of secrecy. In *Proc. of the 4th International Symposium on Engineering Secure Software and Systems, ESSoS 2012*, volume 7159 of *LNCS*, pages 97–105. Springer, 2012.

[113] D. v. Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004.

[114] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Proc. RSA Conference 2006, Cryptographers' Track*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[115] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel, 2002.

[116] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[117] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[118] C. W. Potter, A. Archambault, and K. Westrick. Building a smarter smart grid through better renewable energy information. In *2009 IEEE/PES Power Systems Conference and Exposition, PSCE 2009*, 2009.

[119] J. Reineke. Caches in WCET Analysis. PhD thesis, Saarland University, 2008.

[120] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proceedings of the International Conference in Graph Transformation (ICGT)*, pages 226–241. Springer, 2004.

[121] A. Rényi. On measures of entropy and information. In *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability 1960*, pages 547–561, 1961.

[122] J. Rushby. Noninterference, transitivity and channel-control security policies. Technical report, 1992.

[123] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, Reading, MA, 2001.

[124] R. Schneiderman. Smart grid represents a potentially huge market for the electronics industry. *IEEE Signal Processing Magazine*, 27(5):8–15, 2010.

[125] M. J. Schwartz. Sony sued over playstation network hack. Information Week, 2011. `http://www.informationweek.com/news/security/attacks/229402362`.

[126] Secure Change Project. Deliverable 4.2. Available as `http://www-jj.cs.tu-dortmund.de/jj/deliverable_4_2.pdf`.

[127] M. E. Shin and H. Gomaa. Software requirements and architecture modeling for evolving non-secure applications into secure applications. *Science of Computer Programming*, 66(1):60–70, 2007.

[128] G. Smith. On the Foundations of Quantitative Information Flow. In *Proc. 13th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2009)*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2009.

[129] S. D. Stoller. A bound on attacks on authentication protocols. Technical report, Proc. of the 2nd IFIP International Conference on Theoretical Computer Science: Foundations of Information Technology in the Era of Network and Mobile Computing, 2001.

[130] J. Tenzer and P. Stevens. On modelling recursive calls and callbacks with two variants of unified modelling language state diagrams. *Form. Asp. Comput.*, 18:397–420, November 2006.

[131] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. In *14th International Workshop on Fast Software Encryption (FSE '07)*, volume 4593 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2007.

[132] G. J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In *1st Europ. Conf. on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, Dec. 2003.

[133] T. T. Tun, Y. Yu, C. B. Haley, and B. Nuseibeh. Model-based argument analysis for evolving security requirements. In *SSIRI'10*, pages 88–97. IEEE Computer Society, 2010.

[134] UMLsec group. UMLsec Tool Suite, 2001-2011. `http://www.umlsec.de`.

[135] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[136] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[137] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA 2007)*, pages 494–505. ACM, 2007.

[138] S. Zimmerman. *The Inferno of Dante Alighieri.* iUniverse, 2003.

Appendix A

Omitted proofs

A.1 Chapter 4

Although not difficult, the following argument is normally not explicitly given when referring to non-interference.

Theorem 5. *The property:*

$$\forall \overrightarrow{i} \ [\overrightarrow{i}]|_L = [\overrightarrow{i}|_L]|_L \tag{A.1}$$

is equivalent to:

$$\forall \overrightarrow{i_1}, \overrightarrow{i_2} \quad \overrightarrow{i_1}|_L = \overrightarrow{i_2}|_L \Rightarrow [\overrightarrow{i_1}]|_L = [\overrightarrow{i_2}]|_L \tag{A.2}$$

Proof. $(A.1) \Rightarrow (A.2)$: Since (A.1) holds for any input sequence $\overrightarrow{i}$ it holds in particular for two arbitrary $\overrightarrow{i_1}, \overrightarrow{i_2}$ such that $\overrightarrow{i_1}|_L = \overrightarrow{i_2}|_L$. Then it follows:

$$\forall \overrightarrow{i_1}, \overrightarrow{i_2} \ [\overrightarrow{i_1}]|_L = [\overrightarrow{i_1}|_L]|_L = [\overrightarrow{i_2}|_L]|_L = [\overrightarrow{i_2}]|_L$$

$(A.2) \Rightarrow (A.1)$: Follows directly from the definition of (A.2) and (A.1) for $\overrightarrow{i_1} = \overrightarrow{i}$ and $\overrightarrow{i_2} = \overrightarrow{i}|_L$ since $\overrightarrow{i_2}|_L = (\overrightarrow{i}|_L)|_L = \overrightarrow{i}|_L$.

□

A.2 Chapter 6

The concrete cache updating function for a set t for access of a location m as defined in [21] defines a new set t' as follows

$$\mathcal{U}(t,m) = \begin{cases} t'(1) = m, & \\ t'(i) = t(i-1) \mid i = 2 \dots h, & \\ t'(i) = t(i) \mid i = h+1 \dots A; & \text{if } \exists h : t(h) = m \\ & \\ t'(1) = m, & \\ t'(i) = t(i-1) \text{ for } i = 2 \dots A; & \text{otherwise} \end{cases}$$

This can be generalized for a list $\langle m_1, \dots, m_k \rangle$ of locations:

$$\mathcal{U}(t, \langle m_1, \dots, m_k \rangle) = \mathcal{U}(\mathcal{U}(\dots \mathcal{U}(t, m_1) \dots), m_k)$$

Lemma 2. *Let Φ be the following property over cache sets t:*

$$\Phi(t) : t(k) = \emptyset \Rightarrow \forall\ k' > k\ t(k') = \emptyset.$$

Then if an initial concrete state t satisfies Φ, then the final concrete state after the update of a sequence of memory references by means of the cache update function $\mathcal{U}$ also respects Φ.

Proof. We consider the case where the cache consists of only one cache set (the generalization is similar). We apply induction on the list of memory references for:

$$t' = \mathcal{U}(t, \langle m_1, \dots, m_k \rangle)$$

If the list is empty, the claim follows trivially. Otherwise, there are two cases:

$\exists h : t(h) = m$: As m is in the cache, it is shifted to the first position and all other locations are placed afterwards. Therefore Φ holds on the resulting cache.

otherwise: As m is not in the cache, it is inserted in the first positions and the existing locations are placed afterwards. Therefore Φ holds on the resulting cache.

□

Appendix B

Code snippets

B.1 Chapter 4

In this appendix we spell out the complete set of types used for the implementation:

```
type Label = String
type Variable = String
type Condition = String
type Parameter = String
type Method = String
type Return = String
type High = [Input]
type Low = [Input]
type Tainted = [Variable]
type Node = (Label,Tainted)
type MethodCall = (Method,Parameter)
type Input = MethodCall
type Action = (Variable,String)
type Output = (Action,Return)
type Origin = Label
type Target= Label
type Transition = (Condition, Input, Output, Origin, Target)
type Nodes = [Node]
type Transitions = [Transition]
type StateChart = (Nodes,Transitions)
```

Thus, in the example of the vehicle component V in Sect 4.4 we have that the nodes are (after the tainting analysis):

```
[("x",["p","price"]),("y",["p","price"])]
```

and the transition list is:

```
[("",("readPrice",""),(("p","getPrice"),""),"x","x"),
  ("",("charge",""),(("",""),"start"),"x","y"),
  ("",("tic",""),(("t","t+1"),""),"y","y"),
  ("t=10/p || t = k",("tic",""),(("",""),"stop"),"y","x")]
```

The security policy can be written as:

```
h = [("readPrice",""),("setPrice",""),("getPrice","")]
l = [("tic",""),("start",""),("stop",""),("charge","")]
```

The computed transition relation is a list of type `[(Node,Node)]`:

```
[(("x",["p","price"]),("x",["p","price"])), (("y",["p","price"]),("y",["p","price"]))]
```

and output consistency fails when the pair (y, y) is evaluated. By replacing the transition:

```
  ("t=10/p || t = k",("tic",""),(("",""),"stop"),"y","x")
```

with

```
   ("t = k",("tic",""),(("",""),"stop"),"y","x")
```

the component V can be verified as secure.

B.2 Chapter 6

In the encrypting process of the AES code as implemented in [5], access to the look-up tables are performed for example in the various AES forward rounds. For encryption for example, the forward rounds are initially contained in a loop,

```
for(j = (nr >> 1) - 1; j > 0; j--) {
        AES_FROUND(Y0, Y1, Y2, Y3, X0, X1, X2, X3);
        AES_FROUND(X0, X1, X2, X3, Y0, Y1, Y2, Y3);
}
```

where for a 128 bit key `nr=10`. The AES forward round is defined as:

```
#define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3)     \
{                                               \
    X0 = *RK++ ^ FT0[ ( Y0       ) & 0xFF ] ^   \
                 FT1[ ( Y1 >>  8 ) & 0xFF ] ^   \
                 FT2[ ( Y2 >> 16 ) & 0xFF ] ^   \
                 FT3[ ( Y3 >> 24 ) & 0xFF ];

                ...
}
```

For partitioning (see 6.5.2) we have defined a function `TEST_INDEX` as follows:

```
#define TEST_INDEX(i,a,v) { \
                 if(i<4) {  \
                   v = a[i]; \
                 } \
                 else{ \
                       if(i<12){ \
                                    v = a[i]; \
                        } \
...
                              if(i<252){ \
                                         v = a[i];        \
                              }\
                              else {\
                                      v = a[i];\
                                      }
...
}
```

This code checks the range of the index i within a partition of the possible index range of table a. In this case i ranges from 0 to 255, partitioned in 33 sub-ranges of length at most 8 corresponding to the sets associated to the table blocks within a given sub-range. Then we have changed the original look-up by the following semantically equivalent code.

```
#define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3) \
{ \
/* temporary variable */ \
unsigned long t; \
\
TEST_INDEX(Y0,FT0,t); \
X0 = *RK++ ^ t; \
TEST_INDEX(Y1 >> 8,FT1,t); \
X0 ^= t; \
TEST_INDEX(Y2 >> 16,FT2,t); \
X0 ^= t; \
TEST_INDEX(Y3 >> 24,FT3,t); \
X0 ^= t;
...}
```