

**FORMALLY ANALYZING AND VERIFYING SECURE SYSTEM
DESIGN AND IMPLEMENTATION**

BAI GUANGDONG

NATIONAL UNIVERSITY OF SINGAPORE

2015

**FORMALLY ANALYZING AND VERIFYING SECURE SYSTEM
DESIGN AND IMPLEMENTATION**

BAI GUANGDONG

(B.Sc., Peking University, 2008)

(M.Sc., Peking University, 2011)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

NUS GRADUATE SCHOOL FOR INTEGRATIVE SCIENCES AND
ENGINEERING

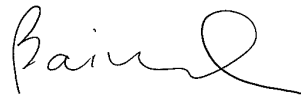
NATIONAL UNIVERSITY OF SINGAPORE

2015

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Bai Guangdong

February 16, 2015

Acknowledgments

Formost, I am greatly indebted to Professor Dong Jin Song. I thank him for his invaluable supervision, unconditional support, and especially, generous tolerance and assistance for my diverse research interests. There is no doubt that what I have learnt from him, in terms of both spirits and academical skills, will benefit my future career. I also thank Professor Liang Zhenkai for his dedicated supervision, encouragement of pursuing a PhD degree in NUS, and continuous guidance in the area of system security.

I would like to thank chair of my thesis advisory committee, Professor Joxan Jaffar, for his active participation and constructive feedback on my research. I would like to thank Professor Andrew Martin for his collaboration and suggestions on my research of trusted computing. I would also like to thank Professor Chen Xiangqun and Professor Guo Yao from Peking University for their continuous encouragement and help.

Furthermore, I would like to thank Dr. Sun Jun and Dr. Liu Yang, who play the roles of both mentors and friends. Without their dedicated guidance, this thesis would never have been completed. My sincere thanks also go to Dr. Prateek Saxena, whose professionalism and enthusiasm is inspiring. He introduced me into the interesting area of web security.

I am indebted to all of my coauthors and collaborators for their ideas, discussions and hard work. I would especially thank those who have participated in the work of this thesis, including but not least to Professor Willem Visser, Heila van der Merwe, Wu Yongzheng, Ye Quanyi, Zhang Qing, Hao Jianan, Wu Jianliang, Lei Jike, Meng Guozhu, Sai Venkatraman, Enrico Budianto. I would specially thank Li Xiaolei for his help, support and valuable comments on my Android research.

I am grateful to everyone in PAT group and Software Engineering Lab, who offers me help, suggestions, entertainment and precious friendship. Meanwhile, I would also thank my friends in System Security Lab, who brighten my life in Singapore.

Finally, I express my thanks to my family. Thank my parents and my younger brother for their

endless and unconditional love. I am so lucky to be living in an environment with their support, encouragement, understanding and confidence. My most heartfelt thanks go to my fiancée, Zhao Jingyu, for her constant love and understanding, which is the light through the darkness.

Contents

List of Tables	viii
List of Figures	x
List of Algorithms	xi
1 Introduction	1
1.1 Insecurity of Building Computer Systems	1
1.2 Use of Formal Methods as an Enhancement of System Security	2
1.2.1 Problems in Building Secure Systems	2
1.2.2 Need of Formal Methods	3
1.2.3 Challenges and Limitations in Practical Use of Formal Methods for Security	5
1.3 Overview of This Thesis	6
1.4 Thesis Structure	8
1.5 Acknowledgement of Published Work	9
2 Background	11
2.1 Security Properties	11
2.2 Formal Analysis of System Design	12
2.3 Formal Analysis of System Implementation	14
2.3.1 Specification Extraction	14
2.3.2 Software Model Checking	15
3 A Formal Foundation for Model Checking Trusted Computing Platforms	18
3.1 Introduction	18
3.2 Motivation & Overview	20
3.2.1 Overview of Key Concepts in Trusted Computing	20
3.2.2 Motivating Example	22
3.2.3 Challenges and TRUSTFOUND Overview	23

3.3	Modeling Trusted Platforms	25
3.3.1	Overview of the Model Library Interface	25
3.3.2	Modeling Security Systems	28
3.3.3	Modeling the TPM	29
3.4	Threat Attacks and Security Goals	33
3.4.1	Attacker's Knowledge and Knowledge Deduction	33
3.4.2	Threat Models	34
3.4.3	Security Goals	35
3.4.4	Uncovering Implicit Assumptions	36
3.5	Implementation and Case Studies	38
3.5.1	Analysis of the Digital Envelope Protocol	38
3.5.2	Analysis of a Trusted Grid Platform	40
3.6	Related Work	41
3.7	Summary	42
4	Automatic Extraction of Web Authentication Protocols from Implementations	43
4.1	Introduction	43
4.2	Challenges & Overview	46
4.2.1	A Running Example	46
4.2.2	Challenges	48
4.2.3	AUTHSCAN Overview	50
4.3	AUTHSCAN System Design	52
4.3.1	Approach Overview	52
4.3.2	Target Model Language	53
4.4	Protocol Extraction Techniques	58
4.4.1	Overview of Protocol Extraction	58
4.4.2	Protocol Refinement Algorithm	58
4.5	Protocol Analysis & Attack Confirmation	65
4.5.1	Security Properties	65
4.5.2	Attacker Models	66
4.5.3	Candidate Attack Confirmation	67
4.6	Evaluation	68
4.6.1	Evaluation Subjects	68
4.6.2	Protocol Analysis and Vulnerabilities	70
4.6.3	Efficiency & Running Time	74
4.7	Related Work	75
4.8	Summary	76

5	Verification of Android Applications against Security Properties Using Targeted Software Model Checking	77
5.1	Introduction	77
5.2	Background	81
5.2.1	Overview of Android	81
5.2.2	An Example	82
5.3	Overview of Our Approach	84
5.3.1	The Model	84
5.3.2	The Property	86
5.3.3	The Model Checking	87
5.4	Static App Reduction	88
5.5	Mocking Up Android OS	91
5.6	Driver Generation	93
5.6.1	Dependency-constrained Event Permutation	93
5.6.2	Driver Generation Algorithm	95
5.6.3	Correctness of Generated Drivers	97
5.7	Implementation and Evaluation	98
5.7.1	Effectiveness of DroidPF	100
5.7.2	Precision of DroidPF	103
5.7.3	Experiments on Non-security Properties	103
5.7.4	Limitation and Discussion	104
5.8	Related Work	105
5.9	Summary	106
6	Conclusion and Future Work	107
6.1	Summary	107
6.2	Future Work	109
	Bibliography	139
A	TML to ProVerif Inputs	140
B	Protocol Extraction	144
B.1	Extracting BrowserID Protocol	144
B.2	Inferred Protocols	145
B.3	Precision of Inferred Protocols	145

Summary

As individuals and enterprises are extensively entrusting their security-sensitive information and core business to computer systems, it is imperative to formally verify the design and implementation of secure systems before they are deployed for real-life use. Unfortunately, existing formal methods are unscalable and often work with only highly abstract models, and thus they are far from the expectations of industrial practitioners. As a result, although formal methods have been proved powerful in some specific areas, we still have not witnessed a wide range of practical use. This thesis aims to enhance the practical use of formal methods for analyzing secure system design and implementation by extending existing formalisms and combining program analysis techniques with formal methods. We focus on three of the most security-critical scenarios: trusted computing, web authentication and mobile computing.

Modern secure systems are too complex to be built from scratch, and they may import many off-the-shelf components which provide particular security features. Focusing on the area of trusted computing, we propose a formal foundation to facilitate model checking of the trusted platforms which are based on Trusted Computing Module (TPM). In particular, we study three problems: the logic and language for formally modeling trusted platforms, modeling of the trusted computing techniques and attack surface, and the so-called *confused responsibility* problem. Further, we show the expressiveness of our formalism in formal modeling and effectiveness in detecting security flaws through applying our formalism and toolkit to two concrete trusted platforms.

Existing formal analysis approaches usually require a precise and complete formal specification of the tested system. In reality, however, establishing such a formal model is not always possible due to reasons like lack of documentation and partial availability of implementation. A typical example is the web authentication systems which involve proprietary server-side implementation. Therefore, we investigate a complementary problem of automatically extracting specifications from implementations under such a constraint. We propose a novel *hybrid inference* approach to combine a blackbox differential fuzzing analysis with the whitebox program analysis. We further apply our approaches to study several real-world web sites, including three popular SSO protocols --- Facebook Connect Protocol, Browser ID and Windows Live Messenger Connect, and demonstrate that extracted fragments of the protocols is of enough precision for finding interesting logic flaws.

Most of previous research, including our aforementioned two studies, analyzes the high-level formal models, which are seldom equivalent to the implementation that the security eventually relies on. Software model checking is a relatively new approach that aims to directly verify im-

plementations. As a meaningful attempt, we apply this approach to verify Android mobile applications, and we extend the approach with a *targeted software model checking* technique, which integrates static analysis to reduce the search space. In addition, we make efforts to address the main challenges in verifying Android apps, such as multiple entry points/event-driven execution, GUI testing and path explosion. Our approach is applied to test both benign and malicious apps and achieves an overwhelming precision and recall rate compared to state-of-the-art analysis approaches.

List of Tables

3.1	Statistics in Experiment of Envelope Protocol	39
4.1	The Action Schema in TML	56
4.2	Statistics in Our Experiments	69
5.1	Main Modules Modified in DROIDPF	92
5.2	Events Supported by DROIDPF	98
5.3	Statistics in Our Experiments	100
5.4	Comparison of Results on DroidBench	102

List of Figures

3.1	Sealing Envelope Process in the Envelope Protocol	21
3.2	Overview of TRUSTFOUND	23
3.3	Interface Used for Modeling Trusted Platforms	27
3.4	Abstraction of a Machine	29
3.5	Models of Loading OS and an Attack Compromising the OS	30
3.6	The State Transition Models of Two TPM Commands	32
3.7	Deduction Rules for Cryptography	34
3.8	A Trusted Grid Platform	41
4.1	An SSO Example	47
4.2	Overview of AUTHSCAN	49
4.3	The TML Model of Running Example in Figure 4.1	54
4.4	AUTHSCAN's Protocol Extraction Process	59
5.1	High-level Overview of Our Approach	80
5.2	Activity Component of the Illustrative App	83
5.3	Part of the xml Layout File	84
5.4	Partial State Space of the Running Example	85
5.5	Dynamic Event Handler Registration	89
5.6	Full L-DFA of Activity Component	94
5.7	DFA of MotivatingAct in Figure 5.2	94

A.1	Applied pi-calculus Model of the Running Example	143
B.1	The HTTP Trace of BrowserID and the Corresponding TML Statements	145
B.2	The Sequence Diagrams Inferred from Implementations of BrowserID and Face- book Connect	147

List of Algorithms

1	Counterexample-guided Assumption Inference Algorithm	37
2	Abstraction Refinement Algorithm	59
3	L-DFA Projection Algorithm	95
4	Dynamic Exploration Algorithm	96

Chapter 1

Introduction

1.1 Insecurity of Building Computer Systems

Individuals and enterprises are extensively entrusting their security-sensitive information and core business to computer systems. Smart phones, online social services, wearable devices, stock trading systems, e-commerce and online banking are merely a few examples. Security of these systems, thus, has become a critical concern. Unfortunately, due to the complexity and the fast-evolving nature of computer systems, building secure systems has been recognized as one of the most difficult problems in computer science. Long-lasting and catastrophic bugs, flaws, errors and vulnerabilities out of the security-critical systems are continuously reported. A notorious example is that the Needham-Schroeder public-key protocol [168] was found vulnerable to a man-in-the-middle attack [147] after it had been published for 17 years. In addition, merely within the single year of 2014, many severe vulnerabilities have been identified from some software systems and protocols which are serving as part of the cornerstone of worldwide computation and communication. For example, the Heartbleed vulnerability, which exposes 66% of web servers on the Internet at the risk of credential leakage [20], is identified from the OpenSSL library, the most popular implementation of the SSL/TLS protocol. As another instance, the recently-revealed Shellshock bug, which leads to arbitrary-command-execution attack, has resided in the widely-used Unix Bash shell since as early as 1992 [136].

1.2. USE OF FORMAL METHODS AS AN ENHANCEMENT OF SYSTEM SECURITY

The rapid growth of openness and interconnectivity of contemporary computer systems has made the security problem even worse, as the systems are exposed to the increasingly hostile Internet. Not surprisingly, anonymity on the Internet and the high revenue from the underground economy marketplaces which facilitate the Internet miscreants to monetize the stolen data (e.g., credit card information and website accounts) and exploited computers, have given rise to numerous worldwide cybercrimes [98, 72, 106]. As an example, it has been reported by IBM that in 2013, there were approximately 1.5 million cyber attacks occurring in the US [127]. These cybercrimes account for large losses, such as finance cost, business disruption, and reputation damage. For example, the annual cost of security breaches to the UK is conservatively estimated in the order of billions of pounds [12].

As a result, before the computer systems are released and used in practice, it is important and necessary to analyze (and ideally, verify) the satisfiability of their *security properties*, including confidentiality, authentication, information flow property, etc., against particular attack models. For example, a trusted computing platform designed for high security assurance must achieve its desired properties (e.g., *attestability*) even with the presence of sophisticated hardware attacks; a mobile application must control its granted privileges and credentials against access from other unauthorized authorities.

1.2 Use of Formal Methods as an Enhancement of System Security

1.2.1 Problems in Building Secure Systems

Security flaws can be introduced throughout the lifecycle of system development, typically design phase and implementation phase.

- Designs of many security systems include flaws even though they are designed by security experts. Besides the aforementioned example of Needham-Schroeder public-key protocol [168], other examples can be frequently seen: the Wired Equivalent Privacy (WEP) protocol that is introduced in the 802.11 standard contains flaws leading to violations of confidentiality due to the misuse of cryptographic functions [66]; because of several design

1.2. USE OF FORMAL METHODS AS AN ENHANCEMENT OF SYSTEM SECURITY

flaws, SSL version 2.0 was substituted by version 3.0 shortly after it was released [163]; the version 1.0 of OAuth protocol, one of the mostly used single sign-on (or SSO) protocols, contains a flaw subject to a session fixation attack [17]; the ISO/IEC 9798 Standard, which specifies a family of authentication protocols, was found subject to various attacks [58] after it had been extensively analyzed [43, 80].

- Even though the design of a system is correct, flaws still can be introduced during the course of implementation. Besides the low-level programming errors like unchecked array bound [20, 38, 25], implementation errors can lie in many other details. For example, as revealed by a recent study [207], the authentication property can be violated in some implementations of online authentication services based on Google ID, such as *Yahoo!* and *zoho*, because a step of correctly verifying a digital signature is neglected.
- Another type of errors, which we term as *confused responsibility*, are caused by the different perspectives of the system designers and the programmers regarding the environments within which the systems are run. At the design phase, the systems are usually designed at a highly abstract level and without the implementation details considered. Therefore they are designed to be resistant to a set of explicitly-defined attacks and under particular assumptions. However, after the systems are implemented and used in reality, the behaviors of the adversary are unpredictable. In addition, the assumptions made on the design phase may not be satisfied in the real-life execution. A typical example is the renegotiation vulnerability of the SSL/TLS [24], which is attributed to the incorrect assumption that the client principal does not change in a single round of execution.

Therefore, in order to building a secure system, security analysis should be conducted throughout the whole lifecycle of system development.

1.2.2 Need of Formal Methods

A rich prior research, including whitebox analysis and blackbox analysis, has been conducted in the literature for the purpose of security analysis. Those approaches based on whitebox analysis

1.2. USE OF FORMAL METHODS AS AN ENHANCEMENT OF SYSTEM SECURITY

mostly use conventional techniques in software engineering, including program analysis [205, 45, 82], reverse engineering [96, 73], taint analysis [95, 44], etc., to extract and vet the behaviors of a system or track the information flow throughout a system. On the other hand, those blackbox analysis approaches often use testing and fuzzing techniques [159, 154, 207, 197, 111]. They infer the system's internal logics from the observed interaction and communication behaviors of the system.

These existing approaches are limited in terms of three aspects. First, they largely rely on the expert knowledge of the security analysts and their insights into the target system, and therefore are error-prone and cannot be generalized. Second, they rarely focus on the design of the systems, whereas the flaws introduced in the design stage are highly likely to be transferred to the implementation. Given that the design-level flaws mostly are logic flows, they are difficult to detect by analysis on the implementation. Third, although these approaches have been shown to be effective in identifying security violations out of the systems, they are only able to cover part of the system behaviors. In other words, they are able to prove insecurity of a system by identifying violations, but rarely prove the opposite.

Therefore, formal methods, which apply mathematical techniques, such as languages, logics and mathematical proofs, to rigorously specify, model, reason and verify computing systems, have been attracting a wide attention of the security community since the 1990s [64]. A typical formal verification method consists of two main phases: *specification* and *verification*. The specification refers to formally specifying and modeling the target systems, and three main types of formalisms are mostly used: *formal logics* (e.g., BAN logic [71] and PCL [182]), *process calculi* (e.g., pi-calculus [35] and CSP [123]) and *diagrammatic formalisms* (e.g., automata). The verification refers to reasoning the desired security properties on the systems, and two categories of techniques are usually used: *theorem proving* and *model checking*. These formal methods have been successfully applied to analyze and verify some commonly-used security protocols [34, 32, 61, 49, 189, 190].

Most of the aforementioned formal methods focus on the design-level models, which are significantly different from the system implementations. Recently, some approaches are proposed

1.2. USE OF FORMAL METHODS AS AN ENHANCEMENT OF SYSTEM SECURITY

to formally analyze the implementation of secure systems, aiming to verify security properties of the system implementation, or on the contrast, to identify vulnerabilities. These studies can be divided into two categories: *specification extraction* [74, 38, 39] and *software model checking* [130]. An introduction on these studies is presented in Chapter 2.

1.2.3 Challenges and Limitations in Practical Use of Formal Methods for Security

Ideally, secure system designs and implementations should be formally verified before they are deployed. In practice, however, formal methods still have not been widely used by industrial practitioners, even in some of the most security-critical scenarios such as *trusted computing*, *web authentication* and *mobile computing*. Through our investigation and experiments, we have experienced challenges as well as limitations of existing formal methods that hinder their practical use, and we have also observed that these problems are general problems for most of other scenarios.

Problems in Formal Analysis of System Design. Instead of being built from scratch, complex systems often import third-party components, for example, Trusted Computing Module (TPM) [108, 109] and ARM TrustZone [23], to reuse their security features. However, for most of the scenarios, there is still a lack of formal foundation including the formal models of those commonly-used components, such that the security analysts have to formally specify the imported components before they can analyze their own systems. In addition, most of the secure systems face a large attack surface, whereas most of designs are analyzed without a comprehensive set of attack models. In particular, most formalisms take into consideration only the Dolev-Yao model which offers a limited behavior model of the adversary [57]. Furthermore, assumptions that are made by system designers and are necessary to achieve the desired security goals are seldom evaluated by previous formal analysis approaches.

Problems in Modeling Process. Previous formal analysis techniques usually rely on manually constructing analyzable models. However, as a system evolves, manually modeling every new version becomes infeasible. Moreover, the manual modeling process is tedious and error-prone. On the other hand, in order to obtain a precise and complete system model, most of these tech-

niques require full implementation of the analyzed system. In practice, however, it is highly unlikely to obtain the complete information, such as source code of some participants. Extremely, several custom systems have never been documented.

Problems in Software Model Checking. Besides the inherent state explosion problem, existing software model checking techniques are still inadequate for general-purpose use. First, since software model checking needs to dynamically execute the analyzed systems, the incompatibility problem may be raised. Furthermore, the asynchronicity and event-driven execution paradigm of some systems (a typical example is the mobile platform) make the traditional exploration strategy which assumes a single entry point ineffective.

1.3 Overview of This Thesis

The main objective of this thesis is *to enhance the practical use of formal methods for secure system design and implementation by extending existing formalisms and combining program analysis with formal methods*. In particular, we have focused on three security-critical areas which have raised challenges as discussed in Section 1.2.3, including *trusted computing*, *web authentication* and *mobile applications*. Our work in this thesis should facilitate the formal analysis of system design and implementation in these areas, and benefit the system designers, developers and security analysts.

A Formal Foundation for Model Checking Trusted Computing Platforms. Trusted computing relies on verified trusted computing platforms to achieve high security assurance. In practice, however, new platforms are often proposed without a comprehensive formal evaluation against a broad spectrum of threat models. In addition, the underlying assumptions that are necessary to achieve the desired security goals are seldom defined explicitly. We thus propose a formal foundation for model checking trusted computing platforms. The foundation includes a model library of formal models of security primitives, a model of trusted computing techniques and a broad spectrum of threat models. It can be used to check the satisfiability of security properties (e.g., confidentiality and attestability) on the design of trusted computing platforms, and uncover the

implicit assumptions (on the trusted computing base, use of the trusted computing techniques and the network infrastructure) which must be satisfied to guarantee the security properties. In our experiments, we formally specify and model check two trusted platforms which have been proposed by previous research.

Automatic Extraction of Web Authentication Protocols from Implementations. Numerous high-profile vulnerabilities have been found in web authentication protocol implementations, especially in single sign-on (SSO) protocols implementations recently. Much of the prior work on authentication protocol verification has focused on theoretical foundations and building scalable verification tools for checking manually-crafted specifications [211, 56, 62].

We address a complementary problem of automatically extracting specifications from implementations. The main challenge we address is the aforementioned problem of partially-available implementation. We propose a novel *hybrid inference* approach to combine a whitebox program analysis with a blackbox differential fuzzing analysis. We aim to automatically recover web authentication protocol specifications from their implementations. In our experiments, we demonstrate that our approach can recover fairly precise specifications, find differences across multiple implementations and uncover security flaws in real-world web authentication protocols.

Verification of Android Applications against Security Properties Using Targeted Software Model Checking. As numerous feature-rich Android applications (*apps* for short) are used in security-sensitive scenarios, methods to verify their security properties are highly desirable. In verification tasks, the precision offered by existing techniques are often not enough. For instance, static analysis often suffers from a high false-positive rate, whereas approaches based on dynamic fuzzing are limited in code coverage.

As a step towards the precise app verification, we propose a *targeted software model checking* technique and implement a general framework. Inheriting the intrinsic merit of traditional model checking in exhaustiveness of exploring finite state spaces, our technique yet verifies the apps on their implementations (Java bytecode), rather than relying on high-level models. With the support of our crafted mock-up Android OS, our technique is capable of exploring the state space in a systematic way using driver programs. The driver programs simulate user interaction and

environmental input, and drive the dynamic execution of the apps. They are generated along with state-space minimization using a slicing-based pruning as well as by exploiting dependency among events. In this work, we focus on checking security properties including sensitive data leakage involving a non-trivial flow- and context-sensitive taint-style analysis. We evaluate our technique with 70 apps, which include real-world apps, malware samples and benchmarks for evaluating app analysis techniques like ours. Our technique precisely identifies nearly all of the previously known security issues and four previously unreported vulnerabilities/bugs.

1.4 Thesis Structure

This section outlines the structure of this thesis.

Chapter 2. Background

This chapter presents a brief overview of formal methods for analysis of secure systems. It introduces the security properties that are commonly targeted by existing formal analysis. In addition, it describes the approaches for formal analyzing system design and implementation in the literature.

Chapter 3. A Formal Foundation for Model Checking Trusted Computing Platforms

This chapter investigates the problem of model checking the designs of trust platforms. This work highlights the need of a broad spectrum of attack models beyond the traditional Dolev-Yao model. In addition, this work also studies the previously-neglected problem of confused responsibility and as a consequence, we recommend that the assumptions regarding implementations, which are made at design stage, should be validated.

Chapter 4. Automatic Extraction of Web Authentication Protocols from Implementations

This chapter presents the work of extracting analyzable models from implementations of web authentication protocols. The main challenge this work address is the incomplete availability of implementations. Our approach detect non-trivial security flows in real-world web sites, in-

1.5. ACKNOWLEDGEMENT OF PUBLISHED WORK

cluding several using important SSO protocols like Facebook Connect Protocol, BrowserID and Windows Live Messenger Connect. The significance of this work is that it demonstrates the advantage and practicality of formal methods in violation detection even under the constraint of imprecise and incomplete models, and therefore, we state that it is worth sacrificing soundness and completeness when formal methods are used for violation detection rather than correctness proving.

Chapter 5. Verification of Android Applications against Security Properties Using Targeted Software Model Checking

This chapter studies the problem of verifying Android applications. The main challenge this work address is the path explosion problem, and we propose a novel targeted software model checking approach to alleviate it. The work highlights the necessity of directly verifying implementations which the security eventually relies on.

Chapter 6. Conclusion and Future Work

This chapter concludes the work conducted in this thesis, and presents possible future work from this thesis.

1.5 Acknowledgement of Published Work

The following publications and work in submission are from this thesis.

- **A Formal Foundation for Model Checking Trusted Computing Platforms** [51]. This work is presented in Chapter 3. It has been published at the 19th International Symposium on Formal Methods (FM'14). The foundation is also employed by our another work ν TRUST [118], which was published at the 15th International Conference on Formal Engineering Methods (ICFEM'13).
- **Automatic Extraction of Web Authentication Protocols from Implementations** [52]. The work is presented in Chapter 4. It has been published at the 20th Annual Network & Distributed System Security Symposium (NDSS'13).

1.5. ACKNOWLEDGEMENT OF PUBLISHED WORK

- **Verification of Android Applications against Security Properties Using Targeted Software Model Checking** [54]. It has been submitted for peer review and the work is presented in Chapter 5.

In addition, we have published three papers which are remotely related to but not presented in this thesis. The research topics covered by these publications are as follows.

- **Confining Android Malware via Resource Virtualization** [141, 142]. We propose resource virtualization as a security mechanism to confine resource-abusing Android apps. The physical resources on a mobile device are virtualized to a different virtual view for selected Android apps. Resource virtualization simulates a partial but consistent virtual view of the Android resources.
- **DroidVault: A Trusted Data Vault for Android Devices** [143]. We introduce the concept of the *trusted data vault*, a small trusted engine that securely manages the storage and usage of sensitive data in an untrusted mobile device. We design and build DroidVault---the first realization of a trusted data vault on the Android platform. DroidVault establishes a secure channel between data owners and data users while allowing data owners to enforce strong control over the sensitive data with a minimal trusted computing base (TCB).

Chapter 2

Background

Formal methods have been assisting on designing and implementing secure systems since decades ago. In view of the effectiveness of formal methods in security analysis, this chapter presents a brief survey on prior studies, including the target security properties, and formal methods on the design and implementation of secure systems.

2.1 Security Properties

Design and implement of secure systems are required to achieve some particular security goals, which are named as *security properties*. In prior studies, two security properties, namely *secrecy* (or *confidentiality*) and *authentication*, are most commonly targeted.

Secrecy property is satisfied if and only if sensitive information and data manipulated by a computing system cannot be obtained by the adversary. Secrecy analysis is defined as querying a term from the adversary Z 's knowledge set. Formal analysis of this property was firstly proposed by Dolev and Yao [93] in 1983. Since then, the majority of formalisms, such as the Woo-Lam model [211], the spi-calculus [35] and the applied pi-calculus [33], have supported secrecy specification and analysis.

Authentication property is achieved if each participant in a system is sure about the identities of other participants whom it is communicating with. This property was firstly formalized and

analyzed by Burrows et al. [71] in 1989, who explained how to analyze authentication protocols with "belief" and identify design flaws from authentication protocols. Besides BAN logic [71, 36], many other formal methods, such as applied pi-calculus [33], have supported authentication specification.

From the literature, it can be observed that formal analysis of secrecy and authentication has been well developed. However, these two properties are the most basic ones required by most computing systems. For a specific system, more complicated and specific properties are required. For example, for a trusted computing platform, the attestability property is desirable; for a mobile application, the privilege property is crucial; and for an e-commerce system, the integrity property of the exchanged messages is important.

2.2 Formal Analysis of System Design

Many prior studies apply formal methods on verifying the satisfiability of the security properties at the system design stage. Generally, these studies consist of two steps: *specification* and *verification*. This section presents a review of these two aspects.

Specification refers to formally specifying or modeling a system and the following three main types of formalisms are mostly used.

- *Finite state machines*. A finite state machine (FSM) is a model to design the sequential logic circuits. It consists of a set of finite number of states, transition and triggering events. A state machine models a system as an abstract machine which is in one of the states at any particular time. The machine can transit its state when a triggering event occurs. Asynchronous product automata (APA) [169] is one of the FSM formalisms and has been used to analyze some cryptographic protocols [112, 113]. A related case study of applying this formalism is the scenario-based formal security analysis of the Trusted Platform Module (TPM) conducted by Gurgens et al.[114]. The main limitation of the FSM formalism is that it lacks the notion of interaction, whereas a system usually has to interact with other

systems during its execution.

- *Formal logics.* A formal logic usually consists of a set of terms, operators and inference rules. The terms stands for the data, keys and participants in the systems. The operators model the operations on the terms, such as cryptographic functions. The inference rules enable to derive the properties of the model. The mostly used logics include BAN logic [71], PCL [182] and LS^2 [89].
- *Process calculi.* Process calculi are mostly used for modeling systems with interaction, communication and synchronization. Popular process calculi used in security analysis includes CSP [123], CCS [160] and pi-calculus [35, 33].

Verification refers to reason and check the desired security properties on the system specification. The verification approaches can be broadly categorized into model checking and theorem proving.

- *Model checking.* Given a system model whose state space is defined and finite, model checking approaches automatically and exhaustively explore the state space to check whether the desired properties are satisfied. If any property is violated, a counterexample containing the path leading to the state where the violation occurs is generated. The advantage is that the state space exploration can be systematic and fully automated, while the disadvantage is that the state spaces of real-world systems can be extremely huge, which may lead to state explosion problem.

A number of model checkers have been proposed for general-purpose analysis. The popular ones that are used in security analysis include PAT [195, 140, 139], USMMC [146, 145], AVISPA [19], SPIN [125], FDR [9], Murphi [162], etc.

- *Theorem proving.* Given a formal specification of the checked system, theorem proving approaches apply inference rules to derive new rules of interest step by step, until the derived rules are sufficient to prove correctness or incorrectness of the desired properties. The main advantage of theorem proving is scalability in large system because it is not based on exhaustive exploration of the state space. Its main disadvantage is the lower automation than model checking. It requires interaction and expertise of the analysts to generated

non-trivial rules.

There are also many theorem provers available for security analysis, including Proverif [62], CryptoVerif [4], Tamarin [157], Isabelle [14], etc.

Formal analysis on system design is effective in detecting security violations at an early stage. However, most of these formalisms and tools require a manual modeling process, which is often error-prone and troublesome. Furthermore, there is still a gap between the verification of the system design and the security of the system implementation, meaning that even if a system design is proved secure, vulnerabilities/bugs may be still introduced during the course of implementing the system. Therefore, recent research attempts to apply formal methods directly on the implementation level, and this is summarized in the next section.

2.3 Formal Analysis of System Implementation

Given the gap between design and implementation, some studies attempt to apply formal methods to verify the security properties of system implementations. These studies use two different approaches. One is to extract specifications from the implementations, and then use the techniques discussed in Section 2.2 to verify the extracted specifications. The other approach attempts to directly check the security properties on the execution of the implementations. This section surveys these two approaches respectively.

2.3.1 Specification Extraction

We define specification extraction by furthering its definition by Avals et al. [46]. A specification is an abstract mapping $\alpha(\cdot)$ which maps the implementation S to an abstract model $\alpha(S)$. Verifying a security property ϕ on $\alpha(S)$ implies that ϕ is satisfied on S , namely

$$\alpha(S) \models \phi \Rightarrow S \models \phi ,$$

if α is an over-approximation.

2.3. FORMAL ANALYSIS OF SYSTEM IMPLEMENTATION

Many studies have attempted to accomplish this "*mapping*" (extracting the specification) by using techniques such as program analysis, compilation and testing. Lie et al. [144] proposed a method to automatically extract specifications from the program implementation of protocols. The model is extracted using program slicing and verified by the tool Murphi. In addition, Aizatulin et al. [38, 39] proposed to use symbolic execution to extract protocol specifications from their C implementations. This work obtains both symbolic and computational models, and checks them using Proverif and CryptoVerif, respectively.

Other studies attempt to translate source code into protocol specification using compiling techniques. For example, the tools FS2PV [10] and FS2CV translate the programs written in F# into the protocol models specified in the typed pi-calculus, which are taken as inputs to Proverif and CryptoVerif. This method has been applied to verify the TLS [61]. Other tools, like Elijah [171], translate Java programs to a process calculus called LySa, and check the model using LySatool [83]. CSur [5] analyzes the secrecy property of the cryptographic protocols implemented in C. It abstracts the program semantics using Horn clauses and uses the H1 prover to prove the property. SPIER [77] compiles the C implementation into a predicate abstraction, using the iterative abstract refinement technique. It then checks the authentication and the secrecy properties. SPIER has been used to analyze the OpenSSL implementation.

One limitation of specification extraction is that some security-related semantics, such as memory access, object management and pointers, may be missed during the translation because of the semantics difference between the implementation languages and highly abstracted modeling languages. In addition, these studies require the source code as input, while the source code is not always available in reality.

2.3.2 Software Model Checking

Instead of inferring system models and checking them using off-the-shelf tools, software model checking approach attempts to directly check the security properties on the execution of the implementations.

2.3. FORMAL ANALYSIS OF SYSTEM IMPLEMENTATION

In the context of software model checking, a system is associated with a finite state space. The execution of the system, in turns, can be modeled as a labeled transition system (LTS) $\mathcal{L}_{sys} = (S, init, Block, Tran)$, where S is the set of states, $init$ is the initial state, $Block$ is the set of instruction sequences which lead to the change of states, and $Tran \subseteq S \times Block \times S$ is the set of transition relations. The main difference between software model checking and traditional model checking is that a state is the evaluation of the real variables in the program. Similar to the traditional model checking, software model checking exhaustively explores the state space of the checked system to identify property violations.

Targeting different implementation language, different software model checkers have been proposed. Java Path Finder (JPF) [203] is a Java software model checker. In this thesis, we use it to analyze the Android app implementations for security properties. In addition, many other implementation-level software model checkers have been proposed. For example, VeriSoft [102] systematically searches concurrent reactive software systems to detect errors including deadlocks, divergences, livelocks and assertion violations; CMC [166] and MaceMC [134] check C and C++ implementations.

The software model checking technique has been used to verify many real-world systems. Musuvathi and Engler [165] use CMC to check the Linux TCP/IP implementation and detect errors. Yang et al. use the software model checking technique to successfully detect errors from file system implementations like ext3, JFS, and ReiserFS [215], and storage system implementations like Berkeley DB, NFS and RAID [214]. Brat et al. [67] use JPF to identify subtle errors from NASA's prototype Mars Rover controller.

Similar to traditional model checking, software model checking also suffers from the state-space explosion problem when applied on large-scale systems. In order to alleviate this problem, traditional general-purpose reduction techniques such as partial order reduction [101] and symmetry reduction [129] can be used. On the other hand, reduction based on static analysis and domain knowledge of the checked system are also necessary. With this in mind, we propose a novel *targeted software model checking* (detailed in Chapter 5) for Android app checking.

In addition, software model checking only works on finite-state systems, while real-world

2.3. FORMAL ANALYSIS OF SYSTEM IMPLEMENTATION

software usually has infinite number of states. Abstraction, thus, becomes critical for model checking software. *Predicate abstraction*, which was first proposed by Graf and Saïdi [105], is a promising approach to automatically generate abstractions. Predicate abstraction represents the large set of concrete states with a significantly small set of abstract states, each of which corresponds to an abstraction predicate which is over the variables in the program. It enables a more efficient reasoning than exhaustively exploring the concrete state space. So far, predicate abstraction has been extensively studied [55, 60, 121, 155], and there are mainly two categories of techniques for program verification based on predicate abstraction. The first is the Counterexample-Guided Abstraction Refinement (CEGAR) [85], in which, if the counterexample generated by model checking the abstraction does not exist in the software, the generated counterexample is used to yield new predicate and more precise abstraction. Since constructing the abstraction of the full software in CEGAR's each iteration is too expensive, other techniques use an incremental approach called *lazy abstraction* [121]. In lazy abstraction, the abstraction and refinement are conducted on-the-fly during the course of model checking; the refinement only re-abstracts the relevant parts, instead of the whole program.

Chapter 3

A Formal Foundation for Model Checking Trusted Computing Platforms

3.1 Introduction

The concept of *trusted computing* has been proposed for more than a decade. It introduces hardware-support security, which takes tamper-resistant hardware techniques as the root of trust, such as Trusted Computing Module (TPM) [108, 109], Intel's TXT [13], and ARM TrustZone [23]. These hardware techniques provide a physically isolated storage and computation environment, based on which a *chain of trust* is set up to support the upper layer software.

Benefited from the hardware support, trusted computing achieves an unprecedentedly high security guarantee (i.e., *trust*) in systems involving multi-level trust domains. Therefore, it has been widely embraced by mainstream products. For example, more than 500 million PCs have shipped with TPM [22] so far; Microsoft equips their recent products Windows RT and Windows 8 Pro tablets with built-in TPM technology [152]. In addition, as we have witnessed, it has been significantly influencing the design of contemporary security systems and protocols---many *trusted platforms*¹ have been proposed both in industry [2, 21] and academia [153, 185, 59, 174,

¹In this thesis, *trusted platforms* refer to the systems, infrastructures and protocols built on trusted computing techniques.

110].

Ideally, the design of the trusted platforms must be formally verified before they are implemented. However, there still lacks an analytical foundation to guide the formal analysis. New trusted platforms are often designed and built without comprehensive analysis against common threat models, which often results in vulnerabilities [114, 68, 210, 209, 81, 138].

Formally analyzing trusted platforms is notoriously challenging. First, a trusted platform usually involves more than one component, including hardware, firmware and software, all of which need to be evaluated. In addition, their configurations and communication interfaces also affect the security properties of the platform. Second, a security analyst has to become an expert in the internals of the hardware-support techniques and formally model them before she is able to model her own platform. However, the techniques are subtle and complicated. Taking TPM as an example, the specification of TPM version 1.2 [108] from Trusted Computing Group (TCG) has 800+ pages, and version 2.0 [109] has 1400+ pages. Third, the large attack surface on trusted platforms requires a comprehensive consideration and understanding of the malicious behaviors.

In this work, we propose TRUSTFOUND, a formal foundation for analyzing trusted platforms. TRUSTFOUND is designed for the trusted platforms based on TPM, and it could be further extended to incorporate other platforms, such as mobile platforms equipped with TrustZone. TRUSTFOUND consists of a library of formal models of security primitives, a formal model of the key techniques in trusted computing² (e.g., TPM, static root of trust measurement, late launch and the chain of trust), as well as a broad spectrum of threat models. It provides the platform designers an interface named TrCSP# (CSP# for trusted computing) for modeling the trusted platforms. By invoking TrCSP#, the security analysts can import the models provided by TRUSTFOUND, and this prevents the analysts from stumbling into modeling the complicated internals of the underlying infrastructure that the analyzed platforms depend on. In addition, the threat models cover most of the known attack scenarios, including the *hardware attacker*, the *system attacker* and the *network attacker*. For the security analysis, TRUSTFOUND aims to 1) detect flaws from the designs of trusted platforms using model checking, and 2) uncover the implicit under-

²The rest of this thesis refers this model as the TPM model.

lying assumptions on the trusted computing base (TCB), using of trusted computing techniques and network infrastructure, which must be satisfied for the platform to guarantee the security goals.

We implement TRUSTFOUND as a framework in C# and CSP# [194] based on the model checker PAT [195]. We apply TRUSTFOUND to formally study two trusted platforms --- an envelope protocol [37] and a cloud computing platform [86]. TRUSTFOUND has found that 7 existing attacks may break their security goals, and identified six implicit assumptions for each of them. Besides, it has detected two previous-unknown security flaws in them, which allow the attacker to breach the desired security goals completely by simply rebooting the machine at certain timing.

3.2 Motivation & Overview

This section first introduces the key concepts in trusted computing. It then presents a protocol based on the trusted computing as our motivating example.

3.2.1 Overview of Key Concepts in Trusted Computing

Trusted Platform Model (TPM). TPM [108, 109] is the *root of trust* for secure storage and measurement, which is a tamper-free coprocessor that provides an isolated storage and computation environment. TPM implements the cryptography primitives such as encryption/decryption, signature, hash and key management. TPM provides a set of commands for the external software to implement functionality that cannot be achieved only using software, such as building a *chain of trust* and *remote attestation*. TPM contains 24 internal Platform Configuration Registers (PCRs). The only way to modify their content is through the command `TPM_Extend(s)`: $PCR_i \leftarrow hash(PCR_i, s)$. Therefore, the value of a PCR can be used to indicate the state of the software stack on a platform. A key can be *sealed* to a particular PCR value, such that the key cannot be used (*unsealed*) if the content of the PCR is not in the sealed value. Two important asymmetric key pairs are embedded in a TPM, namely the Endorsement Key (EK) and the Storage Root Key (SRK). These two keys are kept secret from the external software.

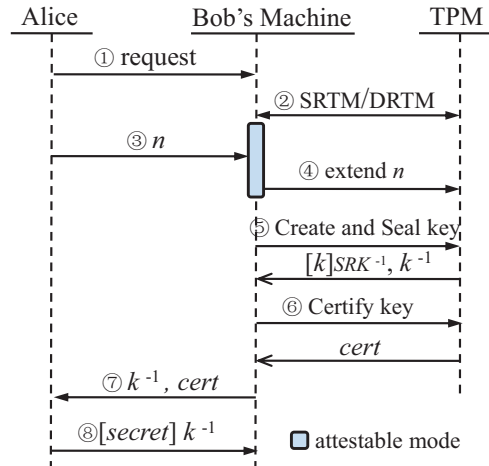


Figure 3.1: Sealing Envelope Process in the Envelope Protocol

Chain of Trust. In trusted computing, the *chain of trust* is a term used to describe the trust relationship among entities in a system. A chain of trust is set up by validating each of the system components from bottom up. There are two ways to build a chain of trust. The first one is the Static Root of Trust Measurement (SRTM) which builds a static chain since the booting of the machine; the other is Dynamic Root of Trust Measurement (DRTM) which dynamically creates a secure execution environment. In SRTM, the first software component is the CRTM (Core Root of Trust for Measurement), while in DRTM, the component is the Authenticated Code Module (ACM).

Remote Attestation. Remote attestation [185] is a process by which a party proves its software/hardware state to another remote party (i.e., the *attester*). The goal of remote attestation is to prove the attester the integrity and trustworthiness of the software/hardware stack. When the attestation is requested by the attester, the software application can ask TPM to sign with the EK the PCR content yielded in the SRTM/DRTM process and present it to the attester. Since no entity except the TPM knows the EK, the signed PCR content can prove the state of the software stack to the attester.

3.2.2 Motivating Example

Ables and Ryan [37] proposed a digital envelope protocol. This protocol has been analyzed and proved correct under certain assumptions by the previous work [91]. Our work attempts to analyze it against a broader range of threat models to uncover the underlying assumptions and if possible, identify security flaws from its design.

Security Goal. The protocol allows Alice to send Bob an enveloped secret, achieving the goal that Bob can either read the secret or revoke his right to unseal the envelope. More importantly, if Bob revokes his right, he is able to prove that he has not accessed the data and will not be able to afterwards.

Protocol Steps. The envelope protocol is designed to work as follows:

1) Sealing Envelope. Shown in Figure 3.1, Alice requests Bob to enter an *attestable mode* (meaning that the state of Bob's machine is known by Alice) which runs a *trusted block*. Bob can achieve this through either SRTM or DRTM. After this step, the PCR is in the state S_0 (①&②). Alice then sends a random nonce n to the trusted block. The value of n is kept secret to Bob (③). The trusted block extends the PCR with n , so its value becomes $hash(S_0, n)$ (④). Bob creates an asymmetric key pair k (private key) and k^{-1} (public key). Bob seals k to the PCR value $hash(hash(S_0, n), accept)$ (⑤) and generates a certificate to prove this (⑥). Bob sends k^{-1} and the certificate to Alice, and Alice sends back the encrypted secret (⑦&⑧).

2) Unsealing Envelope or Revoking Right To unseal the envelope, Bob extends the TPM with *accept*, such that the key can be unsealed for decryption. Alternatively, Bob extends the TPM with *reject* (*accept* and *reject* are two different integers), and requests a TPM quote (a TPM signed PCR value $hash(hash(S_0, n), reject)$) as the evidence.

Possible Property Violations. This protocol is subject to several attacks, which may lead to the violation of the security goal. In the following, we show two examples.

1) Nonce Stealing. The confidentiality of n is critical in this protocol. If it is obtained by Bob, Bob can first choose to extend *reject*, and then reboot the machine and extend *accept* to unseal the key. Here the problem is that it is impossible to set up an encrypted channel directly between

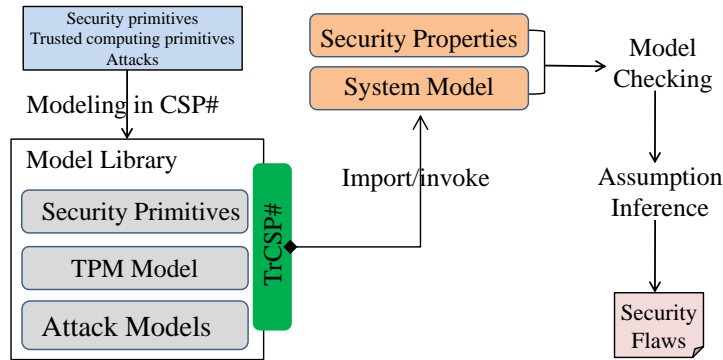


Figure 3.2: Overview of TRUSTFOUND

Alice and Bob's TPM. Therefore, there must be particular software and hardware involved to bridge them, for example, network adapter, LPC bus (where the TPM chip is located), network driver and SSL/TLS library. As a result, there are several existing attacks for malicious Bob to obtain the value of n .

2) **Forging Certificate Attack.** If Bob compromises a CA (Certificate Authority) trusted by Alice, an attack can be conducted at Step ⑦. Bob can forge a certificate for a key pair whose private part is visible to him and deceive Alice into trusting that the key is sealed to the expected PCR state.

Implicit Assumptions. Given the existence of these two possible violations, two underlying assumptions must be satisfied to achieve the security goal.

- **A1:** *a Set of Trusted Components.* The components that the n flows through, such as the SSL library and the LPC bus, must be included in the TCB.
- **A2:** *a Trusted and Uncompromisable CA.* A secure CA is required to validate all certificates.

3.2.3 Challenges and TRUSTFOUND Overview

As shown by this example, the design of a trusted platform must be formally analyzed to reduce possible flaws. However, it also shows that formally analyzing trusted platforms is challenging.

- **Numerous System Details.** Details of the related systems, such as the network and the LPC bus in this example, must be specified to make the analysis precise and practical.

- **Complexity in Trusted Computing.** The semantics of the trusted computing techniques, such as the TPM, SRTM and DRTM in this example, must be precisely and formally modeled before modeling the protocol.
- **Lack of Comprehensive Threat Models.** In order to achieve a comprehensive and convincing evaluation on a trusted platform, we are concerned about the threat models. The key challenge is to figure out a broad spectrum of threat models for analytical use.

To overcome these challenges, we propose TRUSTFOUND, an analytical formal foundation and framework for model checking trusted platforms. Figure 3.2 shows the overall design of TRUSTFOUND. TRUSTFOUND provides a comprehensive model library and an expressive interface named TrCSP# for modeling trusted platforms (Section 3.3). Its model library contains a TPM model such that the security analyst can include the models of trusted computing techniques by simply invoking into the TPM model (Section 3.3.3). The platform model is checked by model checker against a set of attacker models (Section 3.4) provided by the model library. If an attack violates the specified security properties, the model checker generates a counterexample. TRUSTFOUND then infers security flaws and implicit assumptions based on the counterexample (Section 3.4.4).

Scope & Assumptions. The core objective of TRUSTFOUND is to identify whether the design of a trusted platform guarantees the expected properties under a spectrum of attacks. We focus on revealing the flaws and implicit assumptions in the platform designs. We do not target the detection of attacks exploiting implementation vulnerabilities such as the BIOS attack [193], DMA attack [184] and TPM reset attack [15], but we do take them into consideration when identifying the implicit assumptions. We do not consider the DoS attack and side channel attacks such as the timing attack [192]. We also make the following assumptions in TRUSTFOUND:

- **Perfectly Secure Cryptography.** The cryptographic algorithms used by the platforms are perfectly secure.
- **Distinct Secret Keys and Nonces.** The secret keys and nonces are secret and distinct among different sessions.
- **TPM in Compliance with Specifications.** The TPM used by the trusted platforms is

implemented strictly in compliance with the TCG's specifications. TRUSTFOUND does not consider the deviation between the implementation and the specification of the TPM³.

3.3 Modeling Trusted Platforms

This section presents the model library and its interface TrCSP#. The model library consists of models of the security primitives, and these models essentially are the implementation of the existing LS^2 logic [89]. In addition, TrCSP# provides a set predefined keywords to model the trusted platforms.

3.3.1 Overview of the Model Library Interface

We chose to develop TRUSTFOUND based on CSP# because this not only allows us to inherit the expressiveness of extensibility of CSP#, but also enables us to use the well-evolved model checker PAT for automatic reasoning. In this section, we explain the syntax and semantics of CSP# intuitively to ease understanding the rest of this thesis. Then, we introduce how TRUSTFOUND is implemented in CSP#.

Overview of CSP#

The terms defined in CSP# and used in this thesis is underlined. We refer the reader to [194] for the full syntax and semantics of CSP#.

Syntax. The crucial syntax of CSP# is as following.

³This deviation does exist as shown in [184]. Modularity design of TRUSTFOUND allows to substitute the model of a particular TPM implementation for TRUSTFOUND's default one.

<i>Process</i> $P ::= Stop \mid Skip$	termination
$\mid [b]P$	state guard
$\mid e \rightarrow P$	event prefixing
$\mid e\{program\} \rightarrow P$	data operation prefixing
$\mid c?d \rightarrow P(d) \mid c!d \rightarrow P$	channel input/output
$\mid P; Q$	sequence
$\mid P \square Q \mid P \sqcap Q \mid \text{if } b \text{ then } P \text{ else } Q$	choices
$\mid P \parallel Q \mid P \parallel\parallel Q$	concurrency
$\mid P \triangle (e \rightarrow Q);$	interrupt

The core of CSP# is the concurrency and communication. A CSP# model is a 3-tuple $(VS, init, P)$, where VS is a set of variables, $init$ is the initial values of these variables, and P is a process. The e is a simple event; $program$ executes an atomic and sequential program when e is executed; c is a synchronized communication channel. CSP# supports internal choice ($P \square Q$), external choice ($P \sqcap Q$) and conditional branch ($\text{if } b \text{ then } P \text{ else } Q$). Process $P; Q$ behaves as P and after P terminates, behaves as Q . Process $P \parallel\parallel Q$ behaves P and Q simultaneously and only synchronize through the channels, while $P \parallel Q$ requires synchronization over a set of events. Process $P \triangle (e \rightarrow Q)$ behaves as P until e occurs and then behaves as Q .

Semantics. The semantic model of a CSP# model is a Labeled Transition System (LTS), which is a tuple $(S, init, Act, Tran)$ where S is a finite set of states; $init$ is the initial state and $init \in S$; Act is a set of actions; $Tran$ is a set labeled transition relations, each of whose elements is a relation $S \times Act \times S$. We use $s \xrightarrow{e} s'$ to denote $(s, e, s') \in Tran$.

Reachability Checking. Since most of the security properties can be specified in *reachability*, we only use reachability checking in this work (Section 3.4.3), although other properties such as refinement and linear temporal logic can be checked on an LTS. We define a path as a sequence of alternating states and events $\langle s_0, e_0, s_1, e_1, \dots \rangle$. A state s_n is reachable if there exists a path \mathcal{P} such that $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all $i < n$.

TrCSP#: Interface of TRUSTFOUND

Figure 3.3 presents the new keywords that are introduced by TRUSTFOUND, which are twofold. The first is the interface for modeling the secure systems, including cryptographic operations, ma-

3.3. MODELING TRUSTED PLATFORMS

<i>Process</i>	$TP ::= a\{program\} \rightarrow TP$	data operation prefixing
	P	process in CSP#
<i>data/variable</i>	$d ::= n$	number
	c	program
	k	symmetric key
	$sk \mid sk^{-1}$	private/public key
	d, d'	concatenation
	$[d]_k \mid [d]_{sk^{-1}}$	ciphertext of symmetric/asymmetric encryption
	$[d]_{sk}$	signature
	$h(d)$	hash
	TPM_DATA	TPM data
<i>action</i>	$a ::= new\ d$	generate new data object
	$read\ l.d \mid write\ l.d$	read/write d from/to location l
	$asymencrypt\ d.sk^{-1} \mid encrypt\ d.k$	asymmetric/symmetric encryption over d
	$asymdecrypt\ d.sk \mid decrypt\ d.k$	asymmetric/symmetric decryption over d
	$sign\ d.sk \mid verify\ d.sk^{-1}$	sign d with sk & verify signature d
	$hash\ d$	hash over d
	TPM_CMD	TPM commands
<i>location</i>	$l ::= ROM@i \mid RAM@i \mid disk@i$	

Figure 3.3: Interface Used for Modeling Trusted Platforms

chines, network, programs, etc. (Section 3.3.2). The other is on modeling the trusted computing techniques, which are modeled as a set of special data structures in CSP# and used as global variables (Section 3.3.3). Thanks to the expressiveness of CSP#, TRUSTFOUND can be implemented using CSP#. In particular, a process of in TRUSTFOUND is a CSP# process; the *data/variables* and locations are modeled as data structures in the *program* of CSP#; the *actions* are modeled by the *events* in CSP#, with the semantics (e.g., how to perform encryption) implemented in the *program*. By importing TRUSTFOUND and invoking TrCSP#, the analyst is capable of capturing the necessary details of the trusted platforms.

3.3.2 Modeling Security Systems

Machines, Bus and Network. Figure 3.4 shows the abstraction of a machine in TRUSTFOUND. A machine is modeled as a process in CSP#. Each machine contains a CPU, a hard disk, a TPM, a network adapter, ROM and RAM. By default, the firmware such as BIOS and the CRTM are located in the ROM. The hardware drivers and the software, such as the bootloader, OS, network driver and applications, are located in the hard disk. All of them are loaded into the RAM before they can be executed. Figure 3.5(a) demonstrates a simplified model of loading the OS from the disk to the RAM. In this model, the keywords *enum* and *var* stand for enumeration and variable respectively, same as that in other programming languages. By default, the enumeration starts from the integer 0, so both *os* and *Code_{os}* are evaluated 0. The disk and the RAM are modeled by two arrays *DISK* and *RAM*. The execution of the system starts with the process *System()* (line a.8), which first loads the code of the OS from the disk into the RAM (line a.5), and then starts executing the OS (line a.6 and a.7).

We emphasize the communication channel between the CPU and the TPM, namely the LPC bus, because it is more vulnerable than other channels like the north bridge that is between the CPU and the RAM. The LPC bus, actually, has been found vulnerable to an eavesdropping attack [135] and the TPM reset attack [15].

TRUSTFOUND models the communication channels among the components and among the machines with channels. The sender uses *ch!d* to send out data and the receiver listens on the channel using *ch?d*. In the real world, a communication channel can be a private/secure channel or a public/non-secure channel. Therefore, TRUSTFOUND introduces the concept of private channel and public channel accordingly. The private channel is immune to the attacker's eavesdropping, for example, the SSL channel, while public channel leaks all transmitted messages to the network attacker.

Data. Two categories of data are supported in TRUSTFOUND. The first one is primitive data, including the integer, boolean, ciphertext, hash value, signature, encryption/decryption keys, program and concatenated data. Each primitive data is represented symbolically as a 2-tuple

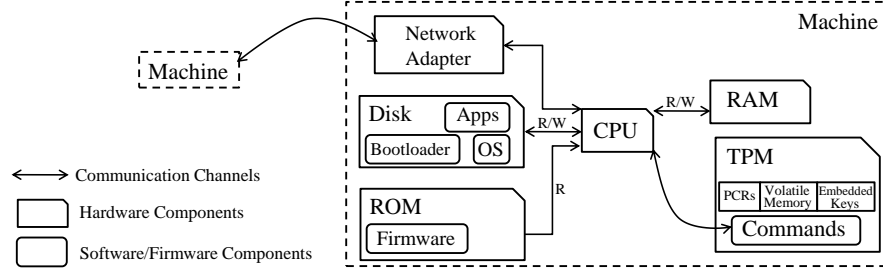


Figure 3.4: Abstraction of a Machine

$d = (type, expression)$, where *type* indicates the type of the data, such as *nonce*, *program* and *public key*; the *expression* may be a number, the identity of a key or a ciphertext $[d]_k$. The other type is the TPM data, which is discussed in Section 3.3.3.

Cryptography Primitives. TRUSTFOUND includes the standard cryptography primitives, such as encryption/decryption, signing/signature verification, hashing, nonce (random number) generation. These primitives take the symbolized data as operands. For example, signing a nonce $n = (nonce, 1)$ with key $sk = (private_key, 001)$ (001 is the identity of sk) generates the signature $sig = (signature, [n]_{sk})$, which can be verified using sk 's inverse key $sk^{-1} = (public_key, 001^{-1})$.

3.3.3 Modeling the TPM

Since all the key concepts, such as the root of trust, chain of trust, SRTM and DRTM, are based on the TPM, we detail the modeling of the TPM in this section.

Abstraction & Simplification. To reduce complexity in modeling and verifying process, a reasonable abstraction and simplification is necessary. The challenge is that the semantics relevant to the security properties cannot be excluded. TRUSTFOUND preserves this semantics in the following three aspects.

- **Functionality.** The functionality of the TPM commands is preserved in a simulation way. For example, the return value of the command `TPM_CreateWrapKey` is a representation of TPM key blob, which contains a symbolic representation of encryption key (discussed

```

a.1  enum {os};
a.2  enum {Codeos};
a.3  var DISK = [Codeos];
a.4  var RAM = [0];
a.5  Config() = load{RAM@os = DISK@os} → OS;
a.6  OS() = OSbenign;
a.7  OSbenign() = /* model of the OS */;
a.8  System() = Config;

```

(a) Loading OS from the Disk

```

b.1  enum {os, osm};
b.2  enum {Codeos, Codeosm};
b.3  var DISK = [Codeos, Codeosm];
b.4  var RAM = [0, 0];
b.5  Config() = load{RAM@os = DISK@os} → SystemAttacker;
b.6  SystemAttacker() = crackMemory{RAM@os = DISK@osm} → OS;
b.7  OS() = [RAM@os == Codeos]OSbenign ∩ [RAM@os == Codeosm]OSmalicious;
b.8  OSbenign() = /* model of the benign OS */;
b.9  OSmalicious() = /* model of the malicious OS */;
b.10 System() = Config;

```

(b) Compromising OS after it is loaded (*code_{os_m}*: the code of compromised OS)

Figure 3.5: Models of Loading OS and an Attack Compromising the OS

soon).

- **Internal Semantics.** The internal security semantics specified by TCG is preserved. For example, in the commands that use a sealed key, such as TPM_Seal, TPM_Unseal, TPM_Unbind, the content of the PCR is checked with the sealed value before the key can be used.
- **Internal State Transition.** The internal state of the TPM changes accordingly when the commands are invoked. For example, when the TPM_Extend (*index, value*) is called, the $PCR[index]$ is extended with *value*; rebooting the machine and activating late launch set the PCRs to a pre-defined value.

We make the following simplifications on the authorization and the key hierarchy.

- 1) *No Authorization Required.* In a real TPM, the authorization protocols such as the OIAP

(Object-Independent Authorization Protocol) and the OSAP (Object-Specific Authorization Protocol) are used to set up a session between the user and the TPM. Since authorization has been well analyzed in previous work [90], we omit it.

2) *No Key Hierarchy*. Based on our assumption that the cryptographic algorithms are perfectly secure, we do not consider the key hierarchy in TPM. Therefore, all the certificates issued by the TPM are signed using its EK, meaning we do not consider the AIK (Attestation Identity Key); similarly, all the encryption operations for secure storage use the SRK.

Abstraction of TPM Data. TRUSTFOUND models the data relevant to the TPM, including the internal data structures (e.g., the PCR value, EK and SRK) and the data generated and consumed by TPM (e.g., TPM certificate, TPM quote, key blob and data blob). Each TPM instance has a unique EK that can be used as its identity.

A TPM data is constructed from the primitive data. A *PCR value* includes the index of the PCR and a hash value to indicate its value. The EK and SRK are asymmetric key pairs. A *TPM certificate* is a certificate issued by a TPM to certify that a key is generated by the TPM and has been sealed on a specific PCR value. A *TPM quote* is a PCR value signed by the TPM. A *key blob*, which is generated by the TPM_CreateWrapKey command, includes the public part and encrypted private part of the generated key. It also indicates the PCR value that the key is sealed to. A *data blob* is returned by the TPM_Seal command. The models of these TPM data can be found in our implementation [51]. Here, we just take the TPM certificate as an example to show how the TPM data is modeled.

Example. A TPM certificate is a 2-tuple (*type, expression*), where the *type* indicates that the tuple is a TPM certificate; the *expression* is a concatenation of a serial of other data: $\langle \text{bool}, sk^{-1}, \text{int}, TPM_PCRValue, ek^{-1}, [\text{bool}, sk^{-1}, \text{int}, TPM_PCRValue, ek^{-1}]_{ek} \rangle$. The first element indicates it is a key generated by the TPM; the second is the public part of the certified key; the third and the fourth indicate the PCR and PCR value the key is sealed to; the fifth is the public part of the EK and the last is a signature by EK.

Formalization of TPM. The TPM is formalized as an LTS $\mathcal{L}^{TPM} = (S^T, \text{init}^T, \text{Cmd}^T, \text{Tran}^T)$, where

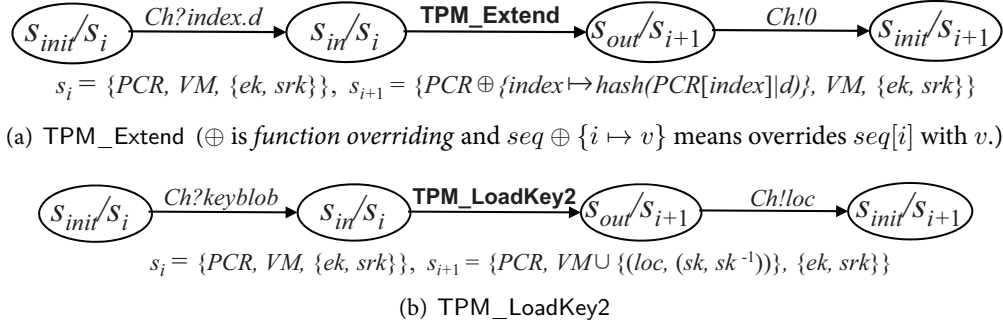


Figure 3.6: The State Transition Models of Two TPM Commands

- $S^{\mathcal{T}} = S_{ctrl}^{\mathcal{T}} \times S_{data}^{\mathcal{T}}$ is a finite set of states, including *control states* and *data states*. The $S_{ctrl}^{\mathcal{T}} = \{s_{init}, s_{in}, s_{out}\}$ models the states regarding the input and output; each of $S_{data}^{\mathcal{T}}$ is a set of variables $V^{\mathcal{T}}$ and their values (detailed later in this Section). An element of $S_{data}^{\mathcal{T}}$ is a set $\{PCR, VM, \{ek, srk\}\}$, where the *PCR* is a sequence which includes the values of 24 PCRs; the *VM* represents the volatile memory and contains indexed key pairs loaded via TPM_LoadKey2, each of which is denoted by $(location, \{sk, sk^{-1}\})$; *ek* and *srk* stand for the EK and the SRK, respectively.
- $init^{\mathcal{T}} \in S^{\mathcal{T}}$ is the initial state.
- $Cmd^{\mathcal{T}}$ is the set of the commands.
- $Tran^{\mathcal{T}}$ is the transition relations, each of which is a relation $S^{\mathcal{T}} \times Cmd^{\mathcal{T}} \times S^{\mathcal{T}}$. $Tran^{\mathcal{T}}$ defines the semantics of the TPM commands, that is, the state transitions upon invoking the TPM commands.

TPM Commands. We use the state transition models of TPM_Extend (Figure 3.6(a)) and TPM_LoadKey2 (Figure 3.6(b)) to demonstrate how we model the TPM commands. The interface TRUSTFOUND provides to the security analyst are the commands same as those specified by TCG. These commands take as input the symbolized TPM data. Thus, from the perspective of the analyst, our TPM model can be regarded as a software-based symbolic and abstract emulator of TPM.

Correctness of TPM Model. One critical issue is that TRUSTFOUND requires a complete and sound TPM model to prevent false positives and negatives. However, verifying the equivalence

of our TPM model and TCG's specification is a much more challenging task, and therefore is not discussed within the scope of this thesis. Nonetheless, we show that this model can be effective in detecting design flaws.

As future work, a reference implementation can be one step towards a specification-equivalent model. For example, Mukhamedov et.al [164] develop a reference implementation part of the TPM (including authorization and encrypted transport session protocols) in F#. The F# model can be automatically translated into Proverif model and verified by Proverif. Afterwards, the F# implementation can be automatically translated into C code.

3.4 Threat Attacks and Security Goals

After coming up with the formal specification of a trusted platform, the next step is to evaluate the expected security properties against threats models. This section defines the modeling of the attacks and security properties in TRUSTFOUND.

3.4.1 Attacker's Knowledge and Knowledge Deduction

We define a property called *knowledge set* $\mathcal{AK} \in VS$ for the attacker. The elements of \mathcal{AK} are the data that can be obtained by the attacker. The attacker can enlarge \mathcal{AK} by eavesdropping on the communication channels, generating data using a machine equipped with TPM (discussed in Section 3.4.2) and deducing new knowledge based on the data known to him. We define some rules for the attacker to deduce new knowledge. As an example, Figure 3.7 demonstrates part of the deduction rules for cryptography.

Two events activate the knowledge deduction. First, when a ciphertext is added into \mathcal{AK} , the attacker actively tries to decrypt it using all the keys he possesses. Second, when a data of a particular type is required, for example, outputting a data to a process, the attacker constructs a new data of the required type. The challenge in knowledge deduction is that applying cryptographic functions unboundedly may leads to an infinite \mathcal{AK} . Therefore, we bound the nesting depth of the encryption functions to be less than 3 by default, unless the attacker obtains or a receiver

3.4. THREAT ATTACKS AND SECURITY GOALS

<i>rule 1.1</i> :	$d \in \mathcal{AK} \wedge k \in \mathcal{AK}$	\models	$[d]_k \in \mathcal{AK}$	Symmetric Encryption Rule
<i>rule 1.2</i> :	$[d]_k \in \mathcal{AK} \wedge k \in \mathcal{AK}$	\models	$d \in \mathcal{AK}$	Symmetric Decryption Rule
<i>rule 2.1</i> :	$d \in \mathcal{AK} \wedge sk^{-1} \in \mathcal{AK}$	\models	$[d]_{sk^{-1}} \in \mathcal{AK}$	Asymmetric Encryption Rule
<i>rule 2.2</i> :	$[d]_{sk^{-1}} \in \mathcal{AK} \wedge sk \in \mathcal{AK}$	\models	$d \in \mathcal{AK}$	Asymmetric Decryption Rule
<i>rule 2.3</i> :	$d \in \mathcal{AK} \wedge sk \in \mathcal{AK}$	\models	$[d]_{sk} \in \mathcal{AK}$	Signature Rule
<i>rule 2.4</i> :	$[d]_{sk} \in \mathcal{AK} \wedge sk^{-1} \in \mathcal{AK}$	\models	$d \in \mathcal{AK}$	Signature Verification Rule
<i>rule 3.1</i> :	$d \in \mathcal{AK}$	\models	$hash(d) \in \mathcal{AK}$	Hash Rule

Figure 3.7: Deduction Rules for Cryptography

expects data of deeper nest. In the case that data of deeper nest are required, TRUSTFOUND increases the bound such that the attacker can construct data with the same nest level. This strategy is similar to the backwards reasoning used by other state-of-the-art verifiers like Tamarin [157].

3.4.2 Threat Models

We divide the threat models in trusted computing into three categories, namely the *network attacker*, the *system attacker* and the *hardware attacker*.

Network Attacker. The network attacker is modeled using the Dolev-Yao model [93]. An active network attacker is able to eavesdrop all messages and modify unencrypted messages on network. We assume the SSL channel cannot be compromised; however, if the platform use SSL as the communication channel, TRUSTFOUND reports that the platform relies on two implicit assumptions---the SSL library must be trusted and a trusted CA is required (uncovering the implicit assumptions is discussed in Section 3.4.4).

A novel feature of the network attacker is that the attacker possesses a machine (denoted by \mathcal{M}_A) equipped with TPM. During the knowledge deduction, the attacker can feed TPM with forged data to generate TPM data expected by the victims. Therefore, the attacker can commit the masquerading attack [193], which forges PCR quote with \mathcal{M}_A to convince the attester that the machine is in the expected state, while conducts malicious behaviors on another machine.

System Attacker. The system attacker can compromise all of the legacy software, including the bootloader, the OS and the applications. The attacker can read/write all the locations on hard disk and RAM. Figure 3.5(b) demonstrates the model of an attack which compromises the OS after it

3.4. THREAT ATTACKS AND SECURITY GOALS

has been loaded to the RAM. After the benign OS is loaded into the RAM from the disk (line b.5), the system attacker cracks the memory and replaces the benign OS (line b.6) with a malicious one, such that the malicious OS will take control once the loaded OS is executed (line b.7).

In addition, the system attacker can invoke the TPM's commands with arbitrary parameters. One possible attack is that the attacker invokes `TPM_Extend` with the benign code to convince the attester, but executes a malicious version of the code.

Hardware Attacker. The attacker on hardware level completely controls a machine. The attacker can compromise the add-on hardware and firmware, for example, DMA attack [120], compromising bootloader and BIOS [133], TPM reset attack [15] and eavesdropping on LPC bus [135]. Compromising firmware such as BIOS and bootloader can defeat the SRTM. The DMA attack can modify the program after it has been loaded into the memory, leading to the same consequence as the system attacker. The TPM reset attack can reset the PCRs to the default state without reboot or late launch. The attacker, therefore, becomes capable of setting the PCR to an arbitrary state as what the system attacker can do. The hardware attacker who can access the LPC bus is able to eavesdrop the communication between the TPM and the CPU.

Note that `TRUSTFOUND` also regards rebooting a machine as an attack, given it changes the state of the system and TPM. We name this attack *reboot attack*.

3.4.3 Security Goals

The trusted platforms are designed to satisfy various security goals. This section discusses two most commonly used ones. We also show that these two goals and other properties can be specified as reachability properties.

Confidentiality. Most of the time, a trusted platform needs to introduce some credentials, whose confidentiality needs to be guaranteed, such as the n in our motivating example. To check confidentiality property, `TRUSTFOUND` queries a credential d from the \mathcal{AK} after the execution of the platforms. If $d \in \mathcal{AK}$, the confidentiality is violated.

Attestability. Attestability means if the attester believes the attested machine is in a state

3.4. THREAT ATTACKS AND SECURITY GOALS

S_T , then the machine must be in that state. Violation of attestability may completely violate the design properties of a platform. As shown in Section 3.2.2, the forging certificate attack manages to break the protocol. The security analyst can define this property with reachability, that is, it cannot be reached that the state of the attested machine (in terms of the PCR value) is not equal to the expected state S_T .

Other security properties can also be specified with reachability in TRUSTFOUND. For example, the security goal of our motivating example, can be specified as

```
#define bothCan(isBobGetSec == true && isBobRvk == true);  
#assert Protocol reaches bothCan;
```

where the *isBobGetSec* and *isBobRvk* are two variables in the CSP# model; *isBobGetSec* is set to *true* whenever Bob reads the secret and *isBobRvk* is set to *true* once Alice receives the TPM quote of $hash(hash(S_0, n), reject)$.

3.4.4 Uncovering Implicit Assumptions

Identifying those implicit assumptions is crucial for enhancing the security on the design level, e.g., by decreasing the size of the TCB as much as possible, and guiding the implementation, e.g., correctly using TPM. We propose a counterexample-guided inference algorithm to figure out those assumptions. TRUSTFOUND applies each of the attacks and the combinations on the components in the checked model. It uses the model checker to detect the security property violation and generates counterexamples. In addition, it examines the channels and data used in the model. Taking them as input to Algorithm 1, TRUSTFOUND figures out the assumptions on the following three aspects.

- *TCB*. TRUSTFOUND considers the components of hardware, firmware and software. If an attack targeting a component violates the security goals, the component is added to the assumptions of TCB (Line 3-6).
- *Network Infrastructure*. If TRUSTFOUND finds the platform uses a private channel, it as-

Algorithm 1 Counterexample-guided Assumption Inference Algorithm

Input: CE : counter examples, D : used data set, CH : used channel set**Output:** $ASPT$: assumptions

```

1: for all  $e$  in  $CE$  do
2:    $a \leftarrow ResolveAttacker(e)$ ;
3:   if  $a.isHdwareAtk() \vee a.isSysAtk()$  then
4:      $m \leftarrow a.attackComp()$ ;
5:      $ASPT.add(m)$ ;
6:   end if
7: end for
8: for all  $d$  in  $D$  do
9:   if  $d.type = certificate$  then
10:     $ASPT.add(CA)$ ;
11:  end if
12:  if  $d.type = TPM\_quote$  then
13:     $ASPT.add(usedPCR \neq 16 \text{ or } 23)$ ;
14:  end if
15: end for
16: for all  $ch$  in  $CH$  do
17:  if  $ch.type = private$  then
18:     $ASPT.add(CA)$ ;
19:     $ASPT.add(SSL)$ ;
20:  end if
21: end for
22: return  $ASPT$ ;

```

sumes the SSL is used, and thus the SSL library should be included in the TCB and a trusted CA is needed (Line 16-21). In addition, if the platform uses any certificate, a trusted CA is required (Line 9-11).

- *Use of TPM.* TRUSTFOUND considers the use of the PCRs. One important but likely to be overlooked fact is that two PCRs (16 and 23) are resettable without a system reboot (using the `TPM_PCR_Reset` command), meaning that the system attacker can generate any value for those PCRs and do the same attack as the TPM reset attack. Therefore, they cannot be used for attestation (Line 12-14).

3.5 Implementation and Case Studies

We have implemented TRUSTFOUND in the PAT model checker [195], which is a self-contained model checking framework for modeling and verification. We implement TRUSTFOUND by integrating into the existing CSP# model an external library. This library implements the model of security primitives, TPM models and the threat models in approximately 4k lines of C# code [51]. As case studies, we apply TRUSTFOUND on two existing trusted platforms.

3.5.1 Analysis of the Digital Envelope Protocol

We use TRUSTFOUND to comprehensively analyze the envelope protocol presented in Section 3.2.2. The protocol is modeled in less than 500 lines of CSP# code. This section summarizes our findings; the reader may refer to [51] for the complete models. Since violating either of confidentiality and attestability leads to the violation of *bothCan* (defined in Section 3.4.3), we just check the assertion of *bothCan* in our experiments.

Threat Models. We define the following attack scenarios based on the threat models.

Network Attacker. We define *NA1* as a network attack which can record and replay the transmitted messages, and *NA2* as a compromised CA who issues certificate for a key pair (mk, mk^{-1}) whose private key is known by Bob.

System Attacker. We define *SA1* as a compromised BIOS who extends a benign OS but executes another malicious one, and *SA2* as a buggy software component (e.g., the SSL library) who can be compromised and cause the leakage of n . *SA1* indicates the modules measured in S_0 but can be compromised at runtime, while *SA2* indicates those that are not measured in S_0 but in fact, are sensitive.

Hardware Attacker. We define *HA1a* as the TPM reset attack, *HA1b* as the TPM LPC attack, *HA2* as the DMA attack targeting loaded OS, and *HA3* as the reboot attack.

Note that for all attackers, we model the protocol in a way that Bob can re-execute the protocol and during re-execution, a fake Alice can feed Bob with data included in the attacker's knowledge set.

Table 3.1: Statistics in Experiment of Envelope Protocol

Attacks	Statistics			
	#States	#Transitions	Time(s)	Memory
NA1	3225	8336	2.18	29M
NA2	7023	13528	7.69	220M
SA1	47451	124680	24.35	198M
SA2	16744	43785	7.94	72M
HA1a	4993	11353	1.94	38M
HA1b	2662	6907	1.63	23M
HA2	47451	124680	21.14	186M
HA3	75110	210663	36.66	232M

Experiments. TRUSTFOUND reports that **NA1** can obtain n at Step ①. Bob therefore can first extend *reject* and convince Alice with $hash(hash(S_0, n), reject)$, and then re-executes the protocol with the fake Alice and extends *accept* to get *secret*. After we change the channel to be private, the data leakage is removed and TRUSTFOUND figures out two assumptions: **A1** that SSL library should be included in TCB and **A2** that a trust CA is required. For **NA2**, TRUSTFOUND reports an attack on Step ⑦. Bob forges a certificate to convince Alice that the mk is sealed in TPM. Alice then uses mk^{-1} to encrypt *secret*. Bob is able to decrypt the ciphertext with mk . TRUSTFOUND also figures out **A2** in this case.

For **SA1** and **SA2**, TRUSTFOUND reports the leakage of n . Bob can conduct the same attack as that in **NA1**. We then extend **SA1** to attack all the modules measured by SRTM and DRTM. TRUSTFOUND identifies **A3** that for SRTM, the TCB should include the CRTM, the BIOS, the bootloader, the OS and the trusted block, and **A4** that for DRTM, the ACM and the trusted block should be included in the TCB.

For **HA1a**, TRUSTFOUND reports that Bob does not reboot the machine upon receiving Alice's request at Step ①. Bob then can execute the protocol with Alice and whenever a particular PCR is required, he just resets the PCR and constructs the expected PCR value. **HA1b** eavesdrops all command parameters transferring through the LPC bus, which allows Bob to obtain n . Attack sequence is similar to **NA1**. Since the TPM reset attack and LPC attack are targeting the physical interface, TRUSTFOUND reports **A5** that proper protection on physical interface should

be in TCB. **HA2** only works for SRTM since DRTM disables DMA for measured code by default. TRUSTFOUND reports that the attacker can modify the OS to the malicious one after it has been loaded, as **SA1** does. Therefore, it figures out **A6** that DMA-capable devices must be trusted when SRTM is used.

A Logic Flaw in the Protocol. TRUSTFOUND reports a severe logic flaw which makes the protocol vulnerable to **HA3**. Between the step ② and step ③, malicious Bob can reboot his machine to a malicious state and obtain n at step ③. Then Bob can conduct the same attack as **NA1**. TRUSTFOUND raises it as a logic flaw because the property violation occurs without any component on attack. This flaw happens because remote attestation only guarantees the integrity of the software stack at the attestation time. Afterwards, verifying whether the software that the attester is interacting with is the one that has been remotely attested has to be achieved by the protocol itself. We propose a fixation which protects the confidentiality of n . In particular, after step ②, the TPM on Bob's machine generates a key pair (k_n, k_n^{-1}) and seals it onto the PCR value which has been extended the trusted block, such that only the trusted block can use it for decryption. The key then can be used to protect n during the transmission.

Table 3.1 lists the statistics collected in our experiments. Our experiments were conducted on a PC with Intel Core i7-940 at 2.93 GHz and 12GB RAM. As can be seen, it requires to explore significant numbers of states to detect the security flaws, which is infeasible for manual analysis.

3.5.2 Analysis of a Trusted Grid Platform

We apply TRUSTFOUND to another trusted platform for cloud computing [86], which can be abstracted as the steps shown in Figure 3.8. Basically, Alice locates her encrypted sensitive program in the cloud (①). When the program needs to be executed, Alice attests the software stack in the cloud using a typical remote attestation protocol [193] (②-⑤). If Alice verifies that the cloud is in an expected state, she sends the decryption key to the trusted block (similar to Fig. 3.1).

Experiments and Results. We model this platform in approximately 150 lines of CSP# code. We use the same set of threat models as that in Section 3.5.1. Due to the similarity of these two

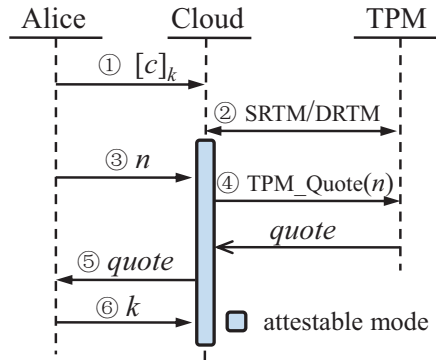


Figure 3.8: A Trusted Grid Platform

platforms, we derive the same set of assumptions as we expected. Furthermore, TRUSTFOUND finds the following logic flaw in this platform when we test the reboot attack (**HA3**).

A Logic Flaw in the Protocol. After Step ⑤, the malicious cloud can reboot to an untrusted mode, and communicate with Alice to obtain k . This flaw occurs because the attestability is violated. Given the cloud is under complete control of the attacker, an authentication between Step ⑤ and ⑥ (as suggested in [86]) cannot defeat this attack. To prevent this, Alice can request a key which is bound to a expected PCR value from the cloud and encrypt k with this key before sending it to the cloud.

3.6 Related Work

A few previous studies have been conducted to formally analyze trusted platforms in the literature. In these studies, trust platforms are manually modeled in different languages. Then, the models are analyzed with various formal analysis tools to check the satisfiability of the properties. During the analysis, the TPM is regarded as trustworthy and secure, and its functionalities are modeled and emulated on a highly abstract level. Delaune et al. [91] present a Horn-clause-based framework for trusted platform analysis, which is featured in sensitiveness of PCR states. Namiluko and Martin [167] propose an abstract framework for TPM-based system based on CSP. In their work, a trusted system is abstracted as composition of the subsystems, including the resources and configurations. The TPM is abstracted as a set of processes. Gürgens et al. [114]

specify the TPM API using FSA (Finite State Automata). All of these works need to model the TPM commands before analyzing the trusted platforms. Therefore, TRUSTFOUND can serve as a foundation for them.

3.7 Summary

We presented TRUSTFOUND, a formal foundation and framework for model checking trusted platforms. TRUSTFOUND provides an expressive formalism, a formal model of the TPM, and three categories of threat models. We successfully detect design-level flaws and a set of implicit assumptions from two existing trusted platforms.

Similar to most previous formal analysis techniques, TRUSTFOUND relies on manually constructing analyzable models. However, the manual modeling process is tedious and error-prone. Even worse, as a system evolves, manually modeling every new version becomes infeasible. To alleviate this problem, we further seek a way to directly extract analyzable models from the implementations in an automatic way. Our study on the model extraction is presented in Chapter 4.

Chapter 4

Automatic Extraction of Web Authentication Protocols from Implementations

4.1 Introduction

Web authentication mechanisms evolve fast. Many web sites implement their own authentication protocols and rely on third-party mechanisms to manage their authentication logic. For example, recent single sign-on (SSO) mechanisms (e.g., Facebook Connect, SAML-based SSO, OpenID and BrowserID) have formed the basis of managing user identities in commercial web sites and mobile applications. For example, OpenID currently manages over one billion user accounts and has been adopted by over 50,000 web sites, including many well-known ones such as Google, Facebook and Microsoft [26]. As another example, Facebook Connect is employed by 2 million web sites and more than 250 million people reportedly use it every month as of 2011 [28]. Ideally, authentication protocols should be formally verified prior to their implementations. However, majority of web sites do not follow this principle. Authentication protocols have historically been hard to design correctly and implementations have been found susceptible to logical flaws [206,

148]. Web authentication protocols are no exception---several of these implementations have been found insecure in post-deployment analysis [207, 131, 42, 196, 69].

There are three key challenges in ensuring that applications authenticate and federate user identities securely. First, most prior protocol verification work has focused on checking the high-level protocol specifications, not their implementations [211, 36, 71]. In practice, however, checking implementations is difficult due to lack of complete information, such as missing source code of some protocol participants. Second, verifying authentication using off-the-shelf tools requires expert knowledge and, in most prior work, conversion of authentication protocol specifications to verification tools has been done manually. However, several custom authentication protocols are undocumented. As new protocols emerge and the implementations of existing protocols evolve, manual translation of every new protocol becomes infeasible. Moreover, manual translation is tedious and can be error-prone. Finally, the authentication of the communication between protocol participants often goes beyond the initial establishment of authentication tokens, which the high-level specifications dictate. In practice, checking the end-to-end authentication of communication involves checking if the authentication tokens are actually used in all subsequent communications and making sure they are not sent on public communication channels or stored in persistent devices from which they can leak. Techniques to address these practical problems of existing implementations are an important area which has received relatively lesser attention.

Our Approach. In this work, we present a framework called AUTHSCAN to automatically extract the formal specifications of authentication protocols from their implementations. Then, these specifications are directly checked for authentication and secrecy properties using off-the-shelf verification tools [62, 19, 87]. AUTHSCAN can automatically confirm the candidate attacks generated by the verification tools and report the true positives (confirmed attacks) in most cases we study. In some cases, AUTHSCAN does not know the attacker's knowledge set enough to generate confirmed attacks --- in such cases, it generates security warnings containing precise communication tokens that need to be manually reviewed by the security analyst.

We design an intermediate language TML to bridge the gap between the detailed implementation of an authentication protocol and its high level semantics that can be used by the verifi-

cation tools. We show that TML is sufficient to capture the communications between protocol participants and their internal actions. AUTHSCAN assumes no knowledge of the protocol being inferred and does *not* require the full source code of the implementation. We propose a refinement method to deal with partial availability of the code implementing the protocol (e.g., if the code located on a web server is not available). It starts with an initial abstraction of the protocol specification, and iteratively refines the abstraction until it reaches a fixpoint. To perform this refinement, we propose a novel *hybrid inference* approach to combine a whitebox program analysis with a blackbox *differential fuzzing* analysis. In particular, the whitebox analysis performs dynamic symbolic analysis on the available code to extract precise data semantics and the internal actions of the protocol participants. The blackbox analysis infers the protocol implementation by probing the protocol participants and comparing the changes in their response. Our final inferred specification in TML can be directly translated into modeling languages used by off-the-shelf verification tools and can be configured to verify against a variety of attacker models [56, 93].

Our techniques focus on recovering as much protocol semantics as possible from dynamic executions of the protocol; we do not aim to find complete specifications. Instead, we aim to recover fragments of the protocol with enough precision to find interesting logic flaws. We apply AUTHSCAN to study several real-world web sites, including three popular SSO protocols --- Facebook Connect Protocol (2 web sites), Browser ID (3 web sites) and Windows Live Messenger Connect (1 web site). We also test several standalone web sites which implement their custom authentication logic and have millions of users sharing personal information. AUTHSCAN successfully recovers precise (but partial) models of their authentication logic, and formally verifies their authentication and secrecy properties against a broad range of attacker models. We have found 7 security flaws in these implementations without their prior knowledge---one of these was found independently by a concurrent work [158] and the remaining are previously unknown. In particular, we find two flaws in Facebook Connect Protocol and one flaw in BrowserID, which arise because the freshness of messages is not guaranteed in the protocol implementations. An attacker is thus able to perpetrate replay attacks to acquire unauthorized authentication credentials.

Several other vulnerabilities are due to unsafe implementation errors in creating and maintaining secrecy of authentication tokens. For example, we find that a web site employing Windows Live Messenger Connect grants the end user a publicly known value as a credential after the user has been authenticated to Windows Live.

4.2 Challenges & Overview

Security analysts often need to guarantee the correctness of authentication protocol implementations without having complete access to the source code. In this section, we explain the problem and its challenges with an example.

4.2.1 A Running Example

Consider one execution of a hypothetical single sign-on (SSO) protocol (similar to Facebook Connect) as shown in Figure 4.1-(a). In our example, Alice wants to authenticate herself to a service provider (SP) web site hosted at `sp.com` by using her login credentials with an identity provider (IDP) hosted at `idp.com`¹. This example shows that much of the communication between the IDP and the SP occurs through the web browser (using `postMessage` between client-side `iframes`), which is similar to real-world protocols [207, 116]. This enables security analysts to analyze protocol behaviors.

The authentication protocol, which the security analyst aims to infer, is as follows:

- **Step ①:** When Alice visits the SP's site and initiates the intent to authenticate, the client-side SP code sends the pre-registered ID and domain of the SP to the IDP's `iframe`. The fact that each SP is pre-registered with the IDP is not known to the security analyst by observing this protocol execution.
- **Step ②:** Assuming that Alice has already logged into the IDP, the IDP generates an HTTP request to its backend server. The request contains a nonce (anti-CSRF) and the session ID of Alice's ongoing web session with the IDP.

¹One sample IDP is `facebook.com` in Facebook Connect and one sample SP is `cnn.com` which uses the Facebook Connect protocol.

4.2. CHALLENGES & OVERVIEW

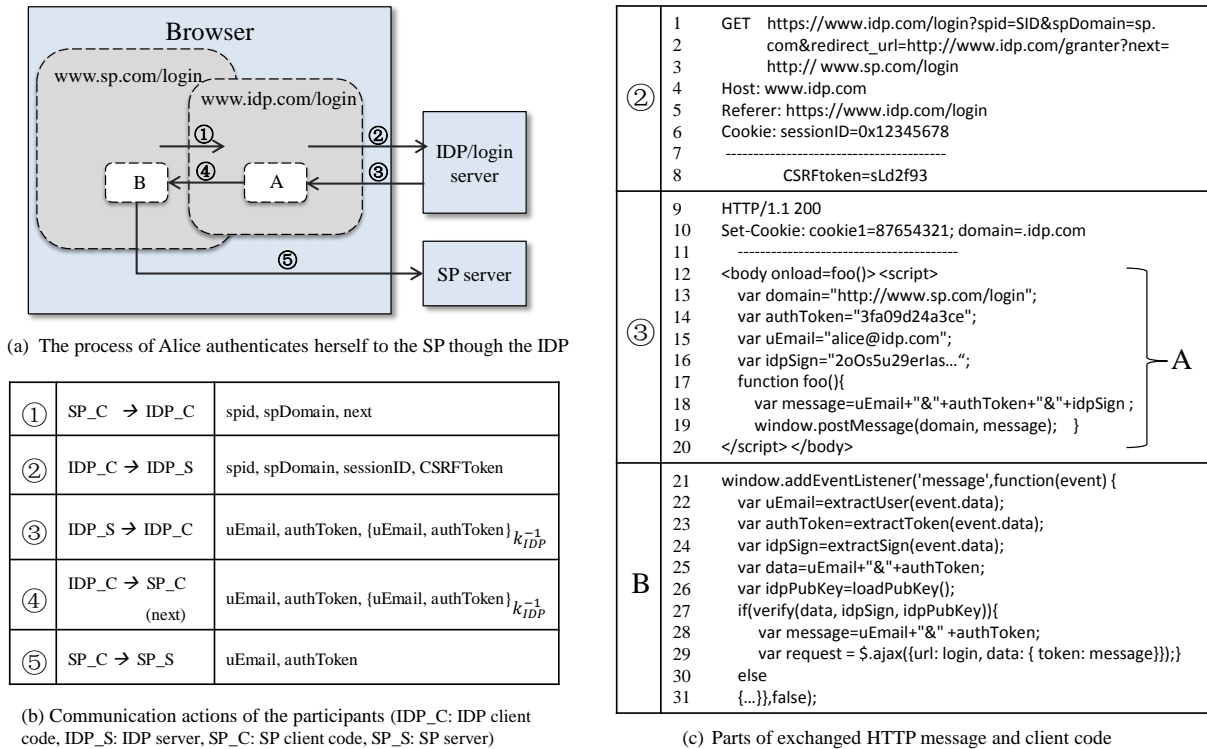


Figure 4.1: An SSO Example: Alice authenticates herself to the SP (*sp.com*) by using her login credentials with the IDP (*idp.com*). The circled numbers indicate the login process, and the capital letters stand for client code.

- **Step ③:** The IDP replies with Alice's registered email identity `uEmail` and an authentication token `authToken`, which authorizes all access to Alice's personal information stored at the IDP. The IDP creates a cryptographic signature over the terms `uEmail` and `authToken` as an authentication credential to be verified by the SP.
- **Step ④:** Client-side IDP code (code A in Figure 4.1-(c)) relays the HTTP data received in step ③ to the SP's `iframe`.
- **Step ⑤:** Client-side SP code (code B in Figure 4.1-(c)) verifies that the signature is valid and extracts the `uEmail` and `authToken`. The SP's `iframe` sends Alice's identity and `authToken` back to the SP's server. This allows the SP's server to access Alice's information stored at the IDP, and allows the IDP to log all SP's actions on Alice's data for audit (not shown).

The security analyst can only observe the network traffic and code execution at the browser end; the server-side logic of the protocol participants is not available for analysis.

Security Flaws. The protocol has several vulnerabilities. We only describe three of them and they can be found automatically if the protocol can be inferred precisely:

- **Man-in-the-middle (MITM) Attack.** The protocol is susceptible to several MITM attacks by a web attacker. For example, consider the target of the `postMessage` call in the client-side code (line 19). This target is derived from an HTTP parameter called `next` (at line 2 of Figure 4.1-(c)). A malicious SP, say Eve, can change the `next` parameter to its own domain, leaving the `spid` parameter as it is. In this attack, the token granted to the `sp.com` is actually sent to Eve by code labeled as **A** in step ④. This attack is similar to a recently reported real-world attack on the site `zoho.com` employing Facebook Connect [207].
- **Replay Attack.** The protocol is susceptible to a replay attack, as the IDP's server does not use any nonce or timestamp to guarantee the freshness of the authentication token `authToken`. If a malicious SP obtains the signed assertion in step ④, it can replay the message to `sp.com` in a new web session and log in as Alice.
- **Guessable Tokens.** Even if the authentication token is kept secret by carefully using only secure (private) communication channels, additional problems can exist. For example, `authToken` remains constant across all of Alice's sessions, which is not apparent from observing a single protocol run. We refer to such tokens as *long-lived* tokens. Long-lived tokens may be used in replay attacks. Similarly, if the IDP uses a weak or guessable scheme to generate authentication tokens, such as a sequentially incrementing counter, an attacker can precisely guess the tokens used in other web sessions.

4.2.2 Challenges

This example shows that implementation-dependent security properties need to be checked in real web applications, where the formal specifications are required. In the following, we list a number of practical challenges in inferring specifications from their implementations.

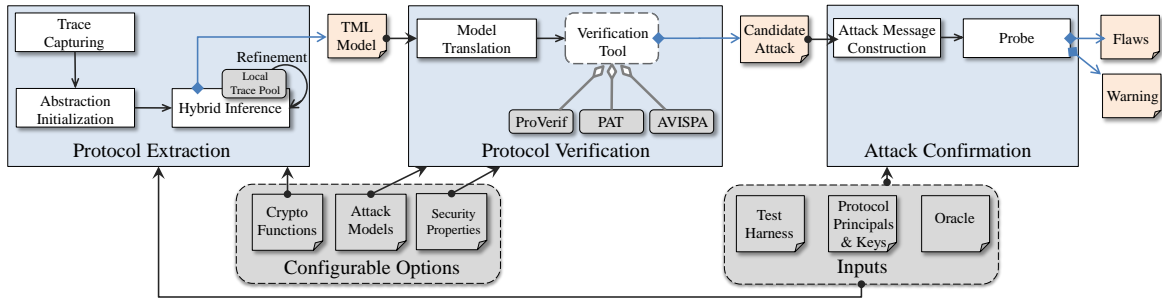


Figure 4.2: Overview of AUTHSCAN

Inferring Semantics. A key challenge is to infer the precise semantics of data elements exchanged in the communication. For example, it is important to know that `authToken` remains constant across all of Alice's sessions with the IDP and does not include a nonce or a timestamp. Inferring this information is critical to discover the replay attack in the protocol. Similarly, identifying that the communication target in ④ is not a fixed domain but instead a variable derived from the HTTP parameter `next` is crucial to find the MITM attack. These semantics are not obvious from the values observed in one message or even in one execution of the protocol.

Partial Code. Only the part of the protocol implementation that executes in the web browser is visible for analysis. For instance, we can infer using whitebox analysis over the client-side code that `idpSign` is a cryptographic signature of `uEmail` and `authToken` under the IDP's private key. This allows us skip generating random guesses about whether it is possible to forge the $(uEmail, authToken)$ pair by the attacker. This can significantly improve the precision, which we discussed in Section 4.6. In other cases, the exact relationship between data elements is not directly available via whitebox analysis. For example, no client-side code reveals whether `authToken` is tied to `sp.com` or is the same for all SPs registered with the IDP. Our analysis needs to infer if there is a one-to-one relation between them.

Redundant Message Elements. Numerous HTTP data elements are contained in the HTTP traces, but most of them are irrelevant to the authentication protocol. The cookie `cookie1` (line 10 in Figure 4.1-(c)) is one of such examples. Including redundant element when using off-the-shelf verification tools can significantly increase the verification time or even lead to a non-

termination. One of the challenges for scalability is to identify and eliminate irrelevant parameters systematically from the traces.

4.2.3 AUTHSCAN Overview

To overcome these challenges, we develop a tool called AUTHSCAN which requires no prior knowledge of the protocol. AUTHSCAN is a system that aids security analysts. It takes the following three inputs.

- **Test Harness.** The security analyst provides AUTHSCAN with at least one implementation of the protocol and provides login credentials (such as username and password) of at least two test accounts. The analyst can optionally provide additional test cases involving many different users and/or different participants (such as different SPs) to utilize AUTHSCAN's full capability---the more test cases, the more precise is the inferred protocol.
- **Protocol Principals & Public keys.** In each test case, the analyst specifies the principals relevant to the protocol, such as the SP, the IDP and the user being authenticated in the running example. In addition, AUTHSCAN takes as inputs the interface APIs (web URIs) that can be queried to obtain public keys of principals involved in the protocol. For instance, JavaScript function `loadPubKey` at line 26 in the running example internally makes an `XmlHttpRequest` (not shown) to retrieve the public key of the IDP; such web interfaces need to be identified by the analyst.
- **Oracle.** AUTHSCAN generates new protocol executions internally during testing. For each internal run generated, AUTHSCAN needs to query a test oracle that indicates whether authentication is successful or not. For AUTHSCAN, this is specified as an HTTP request that AUTHSCAN can make to verify a successful completion. In the running example, AUTHSCAN can generate an HTTP request to access Alice's personal information at the IDP using `authToken` to check if the protocol run succeeds.

Output. AUTHSCAN produces two outputs. First, it produces a specification of the inferred protocol, which can act as a starting point for a variety of manual and automatic analysis [56]. Second,

it produces a vulnerability report for all the attacks that it finds.

Configurable Options. AUTHSCAN is designed to enable checking a variety of security properties under several different attacker models. Additionally, it is designed to incorporate domain knowledge that the security analyst is willing to provide to improve the precision. We next explain these configurable parameters of our system and defaults.

- **Attacker Models.** By default, AUTHSCAN checks for flaws against two standard attacker models: the network attacker [93] and the web attacker [56, 40]. However, it is possible to extend these models with new ones. For example, we can consider a filesystem attacker which steals authorization tokens stored on the client device. Such attacks have been found recently on the Android DropBox application [29].
- **Security Properties.** By default, AUTHSCAN checks for authentication of the inferred protocols. Checking authentication corresponds to two precise, formal definitions provided in previous work: *injective correspondences* [149] and *secrecy* [211]. Additional properties can be added to AUTHSCAN.
- **Cryptographic Functions Names.** AUTHSCAN needs to infer the functions which implement cryptographic primitives such as signature verification, hashes and so on, in the executed client-side JavaScript code (e.g. `verify` at line 27 in Figure 4.1-(c)). By default, AUTHSCAN performs this automatically. It has a built-in list of browser APIs (such as `Window.postMessage()`) and popular JavaScript libraries that provide such functions (such as Node.js [16] and Mozilla `jwtcrypto` [30]). In addition, it has a small set of standardized cryptographic primitives. It can identify functions in the executed client-side code that mimic the behavior of these standardized functions using blackbox testing². Security analysts can improve AUTHSCAN's precision and efficiency by providing additional names of JavaScript source code functions that compute cryptographic function terms.

²Alternative heavy-weight methods (e.g., [208]) to identify cryptographic functions using whitebox analyses are possible.

4.3 AUTHSCAN System Design

In this section, we present an overview of our techniques and introduce an intermediate language called TML to capture the full semantics of the extracted protocol.

4.3.1 Approach Overview

Figure 4.2 shows the internal design steps in our system. AUTHSCAN performs three high-level steps: *protocol extraction*, *protocol verification* and *attack confirmation*.

In the protocol extraction step, AUTHSCAN iteratively processes test cases one-by-one from its input test harness until the test harness is exhausted. For each test case, it records the network HTTP(S) traffic and client-side JavaScript code execution traces through a web browser. Using this information, AUTHSCAN generates an initial abstraction of the protocol specification. It then performs a refinement process to subsequently obtain more precise specifications³. In each refinement step, AUTHSCAN employs a hybrid inference technique which combines both whitebox program analysis on the JavaScript code (if available) and blackbox fuzzing. The refinement process stops if a fixpoint is reached (i.e., no new semantics can be inferred). Our protocol extraction techniques are detailed in Section 4.4.

At the end of the protocol extraction step, AUTHSCAN generates a protocol specification in an intermediate language called TML, which can capture the actions executed by each participant and the semantics of the data exchanged in the protocol execution. AUTHSCAN converts TML to applied pi-calculus, which is a widely-used specification language for security protocols. This protocol specification then can be automatically checked using off-the-shelf verification tools for various security properties, against different attackers. In this work, we use ProVerif [62] and PAT [195] as the verification tools because they can model an unbounded number of parallel sessions⁴. AUTHSCAN models various semantic restrictions, such as the same-origin policy, HTTP headers like `Referrer`, cookies, secure channels (HTTPS, `origin-specified postMessage`),

³By precise, we mean that each refinement contains more expressive semantics about actions performed by protocol participants and more relationships between data terms exchanged in the protocol.

⁴In this thesis, we only use ProVerif to explain our idea. Bounded-state model checkers like AVISPA [19] can also be used but are not implemented as backends yet.

and insecure channels (HTTP, unchecked `postMessages`), before querying off-the-shelf verification tools for precise reasoning, as detailed in [56]. Off-the-shelf verification tools verify these security properties and generate counterexamples which violate the properties. The counterexamples serve as unconfirmed or *candidate* attacks.

The last step of AUTHSCAN is attack confirmation step. In principle, our techniques can generate imprecise protocol specifications; therefore, some of the candidate attacks may not be true security flaws. AUTHSCAN can confirm HTTP attacks by converting counterexamples into HTTP network traffic, relaying them in a live setting and confirming true positives using the analyst-specified oracle. In the cases where AUTHSCAN does not know the attacker's knowledge set enough to generate confirmed attacks, it generates security warnings containing precise communication tokens that need to be manually reviewed by the security analyst.

4.3.2 Target Model Language

The semantics of our inferred authentication protocol is represented in an abstract language called Target Model Language (TML). TML serves as a bridge between protocol implementations and formal models supported by verification tools. It captures enough implementation-level details to check correctness, and at the same time, it can be translated into formal specifications that can be used as inputs to off-the-shelf security protocol verification tools.

We design TML based on the language proposed by Woo and Lam [211], referred as WL model in this work; we add new extensions which are necessary for our protocol inference. We explain the TML semantics in an intuitive way here to ease understanding; the terminology used (underlined) has precise semantics as defined in WL [211]. The TML representation of our running example is shown in Figure 4.3.

TML Syntax. TML represents an authentication protocol as a protocol schema. AUTHSCAN observes several concrete executions of a protocol, each of which is an instantiation of the protocol schema---for instance, our running example is an instantiation of our target protocol with two specific participants namely `idp.com` and `sp.com`. Formally, the protocol schema is a 2-tuple

```

Initial Conditions
(I1)  $\forall x, y : x \text{ has } y$ 
(I2)  $\forall x, y : x \text{ has } key(x, y) \wedge y \text{ has } key(x, y)$ 
(I3)  $\forall x, y : x \text{ has } k_y$ 
(I4)  $r \text{ has } sessionID_r \wedge p \text{ has } sessionID_r$ 
(I5)  $r \text{ has } CSRFToken_r \wedge p \text{ has } CSRFToken_r$ 
(I6)  $Z \text{ has } assoc(i, authToken)$ 
(I7)  $i \text{ has } k_B \wedge r \text{ has } k_B$ 
SP_C(i) Protocol
SC1: BeginInit(j)
SC2: NewAssoc( $\{p, i\}, assoc(j, spid)$ )
SC3: Send( $r, \{assoc(j, spid), next\}_{k_B}$ ) // Step ①
SC4: Receive( $r, \{[M, N, \{[M, N]\}_{k_{IDP\_S}^{-1}}]\}_{k_B}$ ) // Step ④
SC5: Send( $j, [M, N]$ ) // Step ⑤
SC6: EndInit(j)
SP_S(j) Protocol
SS1: BeginRespond(i)
SS2: Receive( $i, [M, assoc(M, N)]$ ) // Step ⑤
SS3: EndRespond(i)
IDP_C(r) Protocol
IC1: Receive( $i, \{X, Y\}_{k_B}$ ) // Step ①
IC2: Send( $p, \{X, sessionID_r, CSRFToken_r\}_{key(r,p)}$ ) // Step ②
IC3: Receive( $p, \{M, N, P\}_{key(r,p)}$ ) // Step ③
IC4: Send( $Y, \{M, N, P\}_{k_B}$ ) // Step ④
IDP_S(p) Protocol
IS1: Receive( $r, \{assoc(T, U), sessionID_r, CSRFToken_r\}_{key(r,p)}$ ) // Step ②
IS2: NewAssoc( $\{p, j\}, assoc(i, authToken)$ )
IS3: Send( $r, \{i, assoc(i, authToken), \{[i, assoc(i, authToken)]\}_{k_{IDP\_S}^{-1}}\}_{key(r,p)}$ ) // Step ③

```

Figure 4.3: The TML Model of Running Example in Figure 4.1. M, N, P, T and U are variables. I2 and the session keys in IC2, IC3, IS1 and IS3 model HTTPS communication. Cross domain restrictions by the browser's SOP are modeled as encryption using the key k_B (initialized in I7). j and p are identities of SP and IDP respectively, i.e., their domains. The behavior of Alice is modeled together on SP client side, thus i stands for Alice's uEmail which is Alice's identity. $sessionID$ and $CSRFToken$ have been inferred to be nonces (I4 and I5). The $authToken$ is inferred to be guessable (I6).

(*Init, ProSet*). The *ProSet* is a set of local protocols $\{P_1(X_1), P_2(X_2), \dots, P_i(X_i)\}$, where each local protocol P_i is executed by a protocol participant X_i . The local protocol specifies a sequence of actions that one participant can perform. The complete specification is characterized by a set of local protocols to be executed by multiple participants. X_i are variables in the schema that may be instantiated by concrete principals (such as `idp.com`) in a protocol instance. The second part of the protocol schema is a set of initial conditions *Init*, such as the initial knowledge

set of each protocol participant prior to the start of the protocol. In the TML of our running example (Figure 4.3), we infer 7 initial conditions (I1–I7); we explain how these are derived during protocol extraction in Section 4.4.

Actions. In executing a local protocol, the participant executes a sequence of actions. Actions can be either communication actions, which send/receive messages with other participants, or internal actions which result in updating local state (or, formally the knowledge set) of that participant. These actions are listed in Table 4.1. The semantics of these actions are fairly intuitive as their names suggest, with the exception of `NewAssoc` which is explained later in this section. For example, `BeginInit(r)` states that an initiator of the protocol begins its role with a responder r . `EndInit(r)` states that the initiator ends the protocol with the responder r ; `BeginRespond(i)` and `EndRespond(i)` are similarly defined with i being the initiator. `Send(p, M)` or `Receive(p, M)` means sending or receiving M to/from p , respectively. `NewNonce(n)` is the action of generating a nonce. `NewKeyPair(k, k^{-1})` is the action of generating an asymmetric key pair, where k is the public key and k^{-1} is the private key. `NewSecret(S, n)` indicates the action of generating a secret, which is intended to be shared with (or distributed to) a set of principals S . Secrets can be data elements such as shared session keys. The secret distribution is only complete when all participants for whom the secret is intended have explicitly executed the `Accept(N)` action. Note that a participant following a local protocol only executes an action after it executes the preceding action state in the schema. As a result of executing certain actions, such as `NewNonce` and `Accept`, participants update their knowledge sets. Intuitively, a participant's knowledge set includes the data terms that it possesses or can compute, which can be used by the participant in communication messages. The attacker, denoted by the principal Z throughout this thesis, is assumed to follow no local protocol and is free to execute any action at any step under the constraints of its knowledge set and the capability of the assumed attacker model.

Terms. We aim to recover as much semantics of the data exchanged and the internal state maintained for each participant as possible. To characterize these semantics, TML provides three kinds of terms: *constant* symbols, *function* symbols and *variable* symbols⁵. Constant symbols

⁵This typesetting is kept consistent with the WL model paper [211]. The constant symbols are typeset in Sans Serif

Table 4.1: The Action Schema in TML

BeginInit(r)	NewNonce(n)
EndInit(r)	NewSecret(\mathbf{S}, n)
BeginRespond(i)	Accept(N)
EndRespond(i)	NewKeyPair(k, k^{-1})
Send(p, M)	NewAssoc($\mathbf{S}, assoc(m_1, \dots, m_n)$)
Receive(p, M)	

include names of principals (web origins), nonces, keys and integer constants. Function symbols include the encryption function $\{\cdot\}_-$, the shared key function $key(\cdot, \cdot)$, the concatenation function $[\cdot, \dots, \cdot]$, the set construction function $\{\cdot, \dots, \cdot\}$ and the arithmetic functions ($+$, $-$, $/$, $*$, and modulo). The public key and private key of a principal P are denoted by k_P and k_P^{-1} , respectively. The symmetric key shared by principles P and Q is denoted by $key(P, Q)$. A term is ground if it only consists of constants and function symbols. Finally, variable symbols represent terms which are not ground.

We aim to recover the precise relationships between terms exchanged in the protocol. For example, our analysis infers that the value of `idpSign` is the signature of `uEmail` concatenated with `authToken`, as can be seen at line 27 of the running example---this translates to the statement labelled IS3 in Figure 4.3. If a participant receives a data element whose precise semantics is not known by the receiver, we represent this data as a variable in TML. For example, consider SC4 in Figure 4.3, we model the messages on the receiver side as variables M and N ; the participant X_i executing local protocol P_i in the schema is a variable; the responder r in the `BeginInit(r)` is also a variable which will be instantiated with concrete values in an execution instance of a protocol schema.

New Extensions in TML. TML extends the WL model with three new extensions. The semantics of other operations are defined in the WL model; we discuss why these extensions are needed. The first extension is arithmetic function symbols. These operations are often utilized in generating sequence numbers from nonces, and, often lead to weak or predictable tokens. Our TML can capture such weak constructions and subject them to testing.

font, the adversary is referred to as the principal Z and the universe of principals is the set SYS . Lower case variables stand for terms that are constant symbols, while upper case variables stand for arbitrary terms.

The second extension is a function symbol called *association relation*, which is written as $assoc(m_1, \dots, m_n)$ to associate n variables, m_1 to m_n . Association relation is necessary because while reconstructing the semantics from implementations, we sometimes cannot infer the exact relation between the terms even though we can infer that they are related. For instance, in the running example, we can infer that `authToken` (line 14, Figure 4.1) does not change during the sessions of the same user, and hence it is related to the user's identity, but the exact semantic relation is unknown. In this scenario, AUTHSCAN generates an association $assoc(i, authToken)$ to indicate that the two terms are related as a key-value pair, but without the exact relation known.

The third extension we introduce in TML is an internal action which is called $NewAssoc(\mathbf{S}, assoc(m_1, \dots, m_n))$. This action means that the association $assoc(m_1, \dots, m_n)$ is known or becomes shared among the principals listed in the set \mathbf{S} . To see why the sharing among \mathbf{S} is needed, consider the following scenario. Principals P and Q possess a mutual shared secret k , that is known prior to the execution. P sends Q a message m in the client browser, both participants send m back to their backend servers, and their servers later respond with entity $\{m\}_k$ in subsequent HTTP messages observed in the browser. AUTHSCAN observes that P and Q compute the same term from m in the code hidden on their servers, but it cannot infer the exact relation between $\{m\}_k$ and m because it does not know that k is a pre-exchanged shared secret. Under such situations, AUTHSCAN introduces a $NewAssoc$ action in the inferred protocol schema to specify that this association is known to both P and Q . The step SC2 in Figure 4.3 shows how this relation is captured at TML.

We define the semantics for these extensions, which extends the original semantic model of the WL model in the following way. We introduce an *association table* for each principal to record the principal's knowledge of associations. When a principal executes $NewAssoc(\mathbf{S}, assoc(m_1, \dots, m_n))$, the $assoc(m_1, \dots, m_n)$ is added into the association table of each principal in \mathbf{S} . Note that the attacker (i.e., Z) is not allowed to update the association table. When a principal receives an association, it checks implicitly if the association is stored in its table.

Assumptions in TML. We make the following assumptions in TML.

- **Correct Cryptographic Algorithms.** TML assumes that the cryptographic algorithms

used in the protocol are ideal. We do not aim to detect vulnerabilities in the implementations of the cryptographic primitives.

- **Distinct Secret Keys and Nonces.** TML assumes the encryption/decryption keys are kept secret prior to the protocol, and are distinct (i.e., cannot be guessed).
- **Knowledge of Principals.** We make the assumption on the knowledge of the principals: Each principal knows the identifiers or names of other principals (represented as (I1) in Figure 4.3). This assumes that the DNS infrastructure has no vulnerability.

4.4 Protocol Extraction Techniques

In this section, we give the details of the hybrid inference approach to address the challenges in Section 4.2.2.

4.4.1 Overview of Protocol Extraction

Our protocol extraction technique operates on the input test harness, one test case at a time. Figure 4.4 shows an overview of the protocol extraction process.

As the first step, the *abstraction initialization* component in our system creates an initial abstraction of the protocol from the first test case in the test harness. It takes HTTP traces (captured by our trace capturing component shown in Figure 4.2) and the initial knowledge provided by the analyst as inputs. The initial abstraction of the inferred protocol is in the form of a TML protocol schema ($Init, ProSet$). By utilizing the test cases from the test harness one-by-one, AUTHSCAN iteratively refines the abstract protocol using our hybrid inference technique discussed in this section. During each iteration of the hybrid inference, AUTHSCAN gradually refines the semantics of terms and actions of the protocol schema until no new semantics can be discovered.

4.4.2 Protocol Refinement Algorithm

The protocol refinement algorithm is shown in Algorithm 2. The inputs of the algorithm are the initial knowledge $InitK$ (i.e., the test harness, protocol participants & public keys of par-

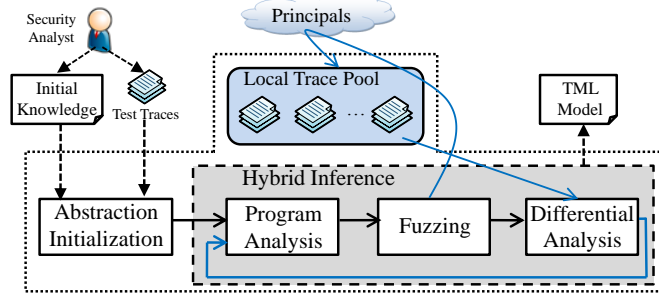


Figure 4.4: AUTHSCAN's Protocol Extraction Process

Algorithm 2 Abstraction Refinement Algorithm

Input: $InitK$: initial knowledge, t : test trace

Output: PS : protocol schema

- 1: $(Init, ProSet) \leftarrow absInit(t, InitK)$;
 - 2: $ProSet_{old} \leftarrow null$;
 - 3: $trPool$: a trace list, initially empty
 - 4: **while** $ProSet \neq ProSet_{old}$ **do**
 - 5: $ProSet_{old} \leftarrow ProSet$;
 - 6: $ProSet \leftarrow JSAnalysis(t, ProSet)$;
 - 7: $(ProSet, T) \leftarrow Blackbox(t, ProSet, InitK, trPool)$;
 - 8: $trPool.add(T)$;
 - 9: **end while**
 - 10: **return** $(Init, ProSet)$;
-

ticipants and oracle, outlined in Section 5.3.3), and a trace t generated from one test case. A trace is a sequence of messages (a_0, a_1, \dots, a_n) , where a_i represents either an HTTP(S) request, response (which may contain JavaScript programs), or a cross-domain communication message over `postMessage`. We refer to all data exchanged in the trace as *HTTP data*, which includes HTTP parameters, cookies, `postMessage` data, HTML form data, JSON data, and so on. AUTHSCAN's trace capturing step identifies the HTTP(S) request/response pairs from the trace. The output of the algorithm is one inferred protocol schema.

Our refinement algorithm (Algorithm 2) has two steps: abstraction initialization (line 1) and refinement process (line 4-9). The *absInit* method (line 1) returns an abstract protocol schema $(Init, ProSet)$. *Init* is a set of predicates, which stands for the initial knowledge of the principals. Some of these are derived from the assumptions of TML (outlined in Section 4.3.2), e.g., $I1 - I3$ shown in Figure 4.3. Other TML terms model the communication channels that are

4.4. PROTOCOL EXTRACTION TECHNIQUES

used in the protocol. For example, to model the HTTPS channels and cross-domain communication channels, we internally introduce symmetric keys ($I6$ in Figure 4.3), as we explain in Section 4.5.2. For every message a in test trace t , if the sender or the receiver of a is not contained in $ProSet$, $absInit$ inserts a new local protocol into the $ProSet$. Then, $absInit$ adds two communication actions (Send and Receive) into the sender's and receiver's protocol, respectively. In addition, $absInit$ can identify some constant terms in the HTTP data, such as the domains of principals, user accounts and public keys of web sites available as the security analyst's inputs to AUTHSCAN. AUTHSCAN identifies them by matching the value of HTTP data with the values in the analyst's inputs. For example, i , r and k_{IDP_S} are identified in this way; they stand for the identity of SP, the identity of IDP and the public key of IDP, respectively. At the end of this step, other HTTP data, which cannot be inferred here, are represented as variable terms whose semantics are inferred in the refinement process explained next. The $Begin^*$ and End^* events are also inserted into the local protocols indicating the SP's client and server.

In the refinement step (line 5-8), AUTHSCAN refines the initial abstraction by utilizing more test cases. This step combines whitebox symbolic analysis ($JSAnalysis$ at line 6) and a blackbox analysis ($Blackbox$ at line 7).

Whitebox Program Analysis. The $JSAnalysis$ procedure uses dynamic symbolic analysis (at line 6) to infer the function terms and the internal actions of the principals. Dynamic symbolic analysis (similar to previous work [186]) is used to obtain symbolic formulae which capture the relations among the HTTP data. These symbolic formulae are over the theory of TML terms, which include arithmetic operations, concatenation function, cryptographic operations and uninterpreted functions. We introduce uninterpreted functions to model semantics unknown function calls, such as calls to browser APIs or JavaScript functions which have many arithmetic and bitwise operations characteristic of cryptographic operations. For the code fragment marked **B** in our running example (Figure 4.1), if the input value for the variable `event.data` is a string "`u&t&s`", the following symbolic formulae are generated by this step:

4.4. PROTOCOL EXTRACTION TECHNIQUES

- (1) $uEmail := u;$ (2) $authToken := t;$
- (3) $idpSign := s;$ (4) $data := [u, t];$
- (5) $idpPubKey := loadPubKey();$
- (6) $verify([u, t], s, idpPubKey);$
- (7) $message := [u, t];$
- (8) $request := $.ajax(login, [u, t]);$

To precisely identify cryptographic function terms in the symbolic formulae, AUTHSCAN needs to identify JavaScript functions implementing cryptographic signature, encryption, random number generation, public key fetching functions and so on. From the above symbolic formulae example, *JSAnalysis* can identify that `idpSign` is the term $\{[uEmail, authToken]\}_{k_{IDP}^{-1}}$, once AUTHSCAN knows that the semantics of the JavaScript procedure `verify(data, sig, key)`. By default, AUTHSCAN identifies these functions based on its built-in list of browser APIs and JavaScript libraries that provide such functions [16]. AUTHSCAN tries to concretely match the semantics of all symbolic terms identified as uninterpreted functions in the symbolic formulae to one of known cryptographic functions in its built-in list. For example, AUTHSCAN can test `verify` with the same inputs as the standard RSA signature verification function from its built-in list and compare the outputs. Security analysts can also provide annotations for source code functions to identify custom implementations of standard cryptographic primitives, in case the default list is not sufficient. In this way, several variables are replaced with newly inferred TML terms in this step. For an uninterpreted function whose semantics cannot be inferred in this step, AUTHSCAN uses an *assoc* to represent it. The *assoc* associates the output of the function with the inputs.

Based on the extracted symbolic formulae, *JSAnalysis* infers the function terms and some internal actions in local protocols. For example, if an HTTP data is identified as a session key, AUTHSCAN treats the principal which first sends it in the communication as generator of this session key. AUTHSCAN infers that this principal has performed a `NewSecret` action and the principals which receive it have performed `Accept` actions. If a principal invokes an asymmetric key pair generation function, AUTHSCAN adds a `NewKeyPair` action to the principal's protocol.

Blackbox Differential Fuzzing Analysis. The blackbox analysis (at line 7) further refines the

4.4. PROTOCOL EXTRACTION TECHNIQUES

output of the whitebox analysis by trying to infer more TML terms and actions while treating the participant implementations as a blackbox. Our blackbox differential fuzzing analysis takes as input the trace t , the refined abstraction after whitebox analysis, and the initial knowledge $InitK$. The first substep in blackbox fuzzing is to remove certain redundant data to make blackbox testing more efficient. Next, the blackbox inference algorithm infers TML terms in two ways: for some terms, it generates "probe" messages and compares the outputs, whereas for other terms, it merely makes the inference based on the observed traces without generating new probes. We describe the redundant data elimination, probe-based inference and non-probe-based inference substep separately. In each iteration of the blackbox fuzzing step, AUTHSCAN internally generates new traces and keeps them in a local trace pool ($trPool$ in Algorithm 1). These traces are not fed back to the initial test harness, and are used only during the blackbox and whitebox steps.

Eliminating Redundant Data. The goal of this step is to identify HTTP data that do not contribute towards the authentication protocol. In this step, we check each HTTP data element by generating a probe message with this element removed. If the probe message results in a successful authentication, we remove the element and all of its occurrences in previous messages. AUTHSCAN performs this operation iteratively for each request/response pairs starting from the last pair and proceeding backwards in t .

Probe-based Inference. The main idea of this fuzzing step is to mutate or remove the HTTP data in the request messages of t , while keeping others unchanged. These modified "probe" messages are sent to the protocol participants and their responses are compared for differences. In addition, to prevent the explosion of number of HTTP traces, we capture at most three traces for each test user account and at most 10 test user accounts for each web site. AUTHSCAN identifies the semantics of several types of HTTP data: URLs, HTTP parameters, web addresses, JSON data, JSON Web tokens, and web cookies. To do this identification, it uses simple pattern matching rules over the values of the data. For instance, a string which has sub-strings separated by "&", with each segment as a key-value pair separated by "=", is treated as an HTTP parameter list. Similar syntactic properties are used for common web objects such as JSONs, JWT, cookies and so on. Once the HTTP data type is inferred, AUTHSCAN makes use of the type in-

4.4. PROTOCOL EXTRACTION TECHNIQUES

formation to speed up the fuzzing process. For example, if `AUTHSCAN` infers that a string is an HTTP parameter-value list, it mutates each key value pair in this string separately. Similarly, if `AUTHSCAN` infers that a string represents a user identity (like usernames) or a web address, it mutates the value of this HTTP data into another user's ID or another web address, instead of trying random modifications. `AUTHSCAN` also incorporates simple pattern-matching rules to identify if values are encoded using common encoding methods such as `URLEncode/URLDecode`, `Base64-encode`, `HexEncode`, `HTMLEncode` and `JavaScript string literal encode`, based on the use of special characters. For an HTTP data with completely unknown semantics, `AUTHSCAN` uses pattern-matching techniques to label it as one of primitive types (`Integer`, `Bool`, or `String`).

Once the basic types are identified, `AUTHSCAN` then infers the TML terms and actions. From the traces in the local trace pool, `AUTHSCAN` attempts to first identify arithmetic function terms, which in turn enables the modeling of weak or guessable tokens. For `Integer`- or `String`-typed value of an HTTP data parameter that change across sessions, `AUTHSCAN` uses the following mechanism to check if it is generated using a predictable arithmetic function. Given such a string value (say *str*), `AUTHSCAN` first conducts a substring matching between its instances across various traces and extracts the parts that are not common between these instances. `AUTHSCAN` then checks if these values form simple arithmetic sequences adding or subtracting a constant. If the function is identified, `AUTHSCAN` treats it as a guessable token, and confirms it by predicating its value and probing the server (discussed in Section 4.5.3). We plan to integrate more powerful off-the-shelf tools, such as Wolfram Alpha, which take such value sequences as inputs and output a closed form arithmetic expression to match it [31]. `AUTHSCAN` also marks any data value which is too short ($L \leq 4$ characters by default and configurable) as guessable *short-length* tokens, as these values may be subject to exhaustive search. For example, in the case where $L = 4$, the search space is less than 2 million $((10 + 26)^4)$, assuming that the term only consists of case-insensitive alpha-numeric characters; `AUTHSCAN` presently does not actually generate these probes but models such values as attacker's knowledge (as detailed in Section 4.5.2), and generates security warnings.

Next, `AUTHSCAN` infers two kinds of associations using techniques similar to those proposed

by Wang et. al. [207]. One kind of association is among HTTP data. AUTHSCAN replaces the value of an HTTP data x in message a_i , while keeping the rest unchanged. Then it sends this "probe" message and compares the response message. If HTTP response \vec{y} changes, AUTHSCAN introduces an $assoc(x, \vec{y})$. Other kinds of association relations are between HTTP data and a web principal or users. Similarly, AUTHSCAN identifies these associations by using differential analysis on multiple traces. The HTTP data which remain constant among the same user's multiple sessions are inferred to be associated to the user; those remaining constant among different users' sessions are inferred to be associated with a web principal (such as the SP or IDP). All remaining HTTP data that change in all such probes are inferred to be nonces (NewNonce), such as session IDs.

Identifying Association Principals. The S in $NewAssoc(S, \dots)$ stands for the principals who share the knowledge of the association terms. AUTHSCAN identifies these principals by observing which terms in an $assoc$ appear in the responses from the protocol participants. Then, it probes these participants by replacing the associated terms with random values. If a principal rejects the fuzzing message, we infer that it knows how to compute the relationship, and add a $NewAssoc$ with these participants in S .

Non-Probe Based Inference. The non-probe based inference infers three kinds of function symbols: cryptographic functions, set functions and concatenation functions. AUTHSCAN employs brute-force search to identify cryptographic functions. It takes every combination of all HTTP data elements and checks if they can be used as inputs to a standard cryptographic primitive to produce another data element. We bound the function nesting depth of terms to be less than 5. In our experiments, we find that this bound is reasonable since all our analyzed protocols do not use no more than 4 levels of nesting cryptographic constructions. This search strategy has been sufficient in practice for our experiments on real-world protocols. For example, as discussed in our BrowserID case study (Section 4.6), AUTHSCAN successfully identifies that one HTTP data element is signed by the IDP, and that the signed elements are the ID and the user's public key. AUTHSCAN identifies the concatenation functions by using a substring search over all combinations of HTTP data elements. For the set construction functions, if a single message contains

multiple data, AUTHSCAN assigns them to a set.

4.5 Protocol Analysis & Attack Confirmation

After extracting a TML model, AUTHSCAN translates it into applied pi-calculus, which is taken as input to ProVerif [62] to check security properties against attack models. We leave the details of this process to Appendix A; and in this section, we discuss the security properties, attacker models and how candidate attacks are checked to confirm security flaws.

4.5.1 Security Properties

By default, AUTHSCAN checks the correctness of two essential security properties in its applied pi-calculus version, *authentication* of an authentication protocol [211] and *secrecy* of credential tokens. A protocol achieves authentication if each principal is sure about the identity of the principal whom it is communicating with. Authentication is checked using *injective correspondence* (\rightsquigarrow , or *injective agreement*) [211, 149, 65, 63], which can check whether two local protocols are executing in "lock-step" fashion, i.e., whether there is an injective mapping between the execution of two participant's protocols. For instance, in our running example, whenever finishing executing $\text{EndRespond}(i)$, SP_S believes that SP_C has executed the protocol with him; thus, to guarantee authentication, SP_C must have executed $\text{BeginInit}(j)$, i.e., $\text{EndRespond}(i) \rightsquigarrow \text{BeginInit}(j)$ ($\text{inj-event}(\text{EndRespond}(i)) \implies \text{inj-event}(\text{BeginInit}(j))$ in applied pi-calculus). Authentication is violated if SP_S believes SP_C has executed the protocol with him, but actually it is Z who has.

Additionally, an authentication protocol may introduce some credentials and thus *secrecy* of them needs to be guaranteed. Secrecy is defined as querying a term from the attacker Z 's knowledge set [211]. The secrecy of a term a is specified as Z has a ($\text{query_attacker}(a)$ in applied pi-calculus), which queries whether a is derivable by Z after the execution of the authentication protocols. If Z has a after the protocol, the protocol fails to guarantee the secrecy of a . By default, AUTHSCAN checks the secrecy of terms used for authentication (such as the `sessionID` in the

running example); the attack analyst can add more queries to check the secrecy of other terms, for example, credentials for resource access (such as OAuth token in OAuth 2.0). For long-lived tokens, AUTHSCAN adds them to Z's knowledge set before querying ProVerif. In general, Z may know a long-lived token's value (through external knowledge) even if it is not sent on a public channel; AUTHSCAN conservatively models this scenario and raises a security warning to alert the analyst. For guessable tokens, AUTHSCAN adds the outputs of the arithmetic operations to Z's knowledge set. In the attack confirmation step, these guessable values are computed and used as we detail in Section 4.5.3.

4.5.2 Attacker Models

In this work, we consider two different attacker models, namely the network attacker [93] and the web attacker [40]. Previous work (e.g., [56]) has shown that these attackers can be captured in ProVerif. Hence, we ignore the detailed modeling and just give an overview in this section. For example, attacker model in the running example is demonstrated in Appendix A. Note that both the attacker models are checked individually in AUTHSCAN, since ProVerif terminates after finding a counterexample.

Network Attacker. We model the network attacker using the Dolev-Yao model [93], that is, an active network attacker is able to eavesdrop all messages and control the contents of unencrypted messages in the public network under the constraints of cryptographic primitives. In TML, we model HTTPS by assuming that the SSL certificate checking and handshake are complete before the protocol starts; we model the session key between the two communicating principals x and y with a key function $key(x, y)$ (I2 in Figure 4.3). In applied pi-calculus, we model HTTPS using private channels, which are neither readable nor writable by the attacker (shown in Appendix A). Note that modeling the HTTP network attacker is available from ProVerif directly.

Web Attacker. We also reuse web attacker models described in prior work [56, 40]. These models include modeling the same-origin restrictions; for example, the fact that client-side SP code cannot intercept IDP server's messages is implied in the applied pi-calculus semantics that

the local variables of a process are inaccessible by another process. We model HTTP headers like `Referrer` which correspond to the client-side code sending its identity in the messages; of course, if the header is not checked by the server, it will not be inferred in our specification as it is removed as a redundant element. We also model the semantics of `postMessage` by encrypting all messages transmitted through `postMessage` with a key (k_B in IC4 and SC4, Figure 4.3). If AUTHSCAN finds (by whitebox analysis) that the receiver or sender `origin` fields are not checked, it casts k_B to the attacker such that the attacker is able to read and write the `postMessage` channel. The anti-CSRF tokens are not needed to be explicitly modeled in the attacker model as they are observed in the HTTP network messages and are inferred to be nonces if they are relevant to the protocol (I4 and I5 in Figure 4.3). We assume that the attacker has the ability to redirect the user agent to a malicious web site. We do not model web attackers with the ability to perform Cross-Site Scripting (XSS) attacks and complex social-engineering attacks in this work.

4.5.3 Candidate Attack Confirmation

AUTHSCAN confirms candidate attacks generated by ProVerif in this step. If a protocol fails to satisfy the security properties, ProVerif generates a counterexample, which consists of the attacker's actions, the attacker's input/output and details the terms computed by Z at each step using its knowledge set at that step. AUTHSCAN re-constructs the candidate attack probe from this information. For all terms computed at each step, AUTHSCAN substitutes the concrete values for these terms. For guessable tokens that are computed from arithmetic functions, AUTHSCAN evaluates the function to calculate the next concrete value. For short-size guessable tokens, AUTHSCAN only raises a security warning. To map symbols and variables in ProVerif counterexamples to concrete values observed in the HTTP traces, AUTHSCAN maintains the mapping between the original HTTP messages and the protocol statement generated during the protocol extraction. Thus, AUTHSCAN maps back a ProVerif action sequence and terms in the ProVerif counterexample to the ProVerif input, which in turn is mapped to the raw HTTP message. Once the messages are constructed, AUTHSCAN replays the candidate attack probe. During this process, it queries

the oracle provided by the analyst to check whether the attack is successful.

Currently, AUTHSCAN automates confirmation of attacks over HTTP, over `postmessage` and via a web attacker-controlled `iframe`. In cases which AUTHSCAN cannot confirm with concrete attack instances, it reports security warnings containing the communicated data it suspects. Such cases include the use of long-lived token in authentication, secrecy of which is not known in the inferred protocol but conservatively modeled as discussed in Section 4.5.2, and the use of guessable short-length tokens.

4.6 Evaluation

We have built an implementation of AUTHSCAN in approximately 5K lines of C# code, and 3K lines of JavaScript code. The HTTP trace recording and blackbox fuzzing functionalities are implemented in a Firefox add-on. The JavaScript trace extraction is implemented by instrumenting the web browser to generate execution traces in a format similar to JASIL [187]. We developed our own implementation of dynamic symbolic analysis for extracting the TML terms from the execution traces.

4.6.1 Evaluation Subjects

To estimate the effectiveness of AUTHSCAN on real-world protocols, we test several implementations of popular SSO protocols and standalone web sites that implement their custom authentication logic. The inferred protocols are presented in Appendix B.2. Our results are summarized in Table 4.2.

BrowserID. BrowserID [3] is an SSO service proposed by Mozilla, which is used by several Mozilla-based services such as BugZilla and MDN, as well as some other service providers. We test three different SP implementations of BrowserID. Although BrowserID is open-source, most of protocols do not provide the detailed implementation on the server-side. To account for this, we only take into consideration the client-side JavaScript code and HTTP messages to make our analysis approach more general. AUTHSCAN manages to infer the general protocol specification

Table 4.2: Statistics in Our Experiments

Column 2: ratio of messages filtered out by AUTHSCAN w.r.t. the total number of messages occurred in the protocol; Column 3: ratio of parameters filtered out by AUTHSCAN w.r.t. the total number of parameters used in the messages; Column 4: total execution time of AUTHSCAN; Column 5: verification time of running ProVerif *without* and *with* filtering of the messages or HTTP data, under the network attacker, where "-" means nontermination in verification; Column 6: number of rounds; Column 7: number of bugs found in each web site (with repeats); there are 7 distinct (without over-counting) vulnerabilities.

Web Sites	% Redundant Msgs (Total)	% Redundant Elems (Total)	Time(s)	Verification Time (s) WO / W Filter	Fuzzing Rounds	Bugs
myfavoritebeer.com	88% (80)	50% (12)	113	204/3.0	20	2
openphoto.me	82% (93)	75% (24)	72	726/3.0	22	2
developer.mozilla.org	87% (127)	74% (23)	96	-/3.0	28	0
ebayclassifieds.com	72% (58)	57% (152)	127 ^a	-/58.7	107	2
familybuilder.com	97% (290)	51% (144)	110 ^a	-/58.7	77	1
weibo.com	97% (176)	98% (52)	30	0.36/0.03	78	1
iyermatrimony.com	98% (120)	67% (9)	5.33	1.14/0.04	510	1
meetingmillionaires.com	96% (54)	0% (5)	4.72	1.05/0.04	30	1

^a The period that AUTHSCAN halts until Facebook allows to resume fuzzing is not taken into account.

from these three implementations, finding only one crucial difference across the implementations (explained in Section 4.6.2).

Facebook Connect. Facebook Connect [8] is one of the most widely used incarnations of the OAuth 2.0 published by Facebook. We test two SP web sites using this protocol. The experiments are conducted on the basis of client-side JavaScript code and HTTP messages. AUTHSCAN infers the general protocol specification successfully.

Windows Live ID. Windows Live Messenger Connect [27] is another SSO protocol derived from the general OAuth 2.0 specification. We test its implementation using the Sina Weibo service---a China-based web site similar to Twitter and has over 300 million users. AUTHSCAN successfully extracts the protocol from this implementation; we skip the protocol diagram (which is similar to Facebook Connect) for the sake of space.

Standalone Web Sites. We also test two standalone sites, where users share deeply personal information, both of which have from hundreds of thousands to millions of users and utilize custom authorization mechanisms. AUTHSCAN uncovers the custom authentication protocol for both sites.

4.6.2 Protocol Analysis and Vulnerabilities

We test AUTHSCAN on 8 implementations (as shown in Table 4.2). We successfully find 7 security vulnerabilities, all of which we have responsibly disclosed to the developers of the web sites. For the sake of space, we leave the details on how AUTHSCAN extracts protocol specification to Appendix B.1; and in this section, we briefly present the found vulnerabilities in the protocol implementations.

Setup. In our experiment, the input and configuration to AUTHSCAN include:

- *Test harness.* The security analyst is required to input two pre-registered user accounts (for example, email and password in BrowserID), except for the IyerMatrimony case in which five are needed.
- *Protocol principals & public keys.* For the SSO implementation (including BrowserID, Facebook Connect and Windows Live ID), the analyst needs to indicate domains of IDP and SP (for example, in BrowserID case, persona.org and myfavoritebeer.org, respectively). For the standalone web sites, the analyst needs to indicate the domains of the tested sites. In both cases, the public keys of the participants need to be provided if HTTPS is used in the implementation.
- *Oracle.* The analyst needs to provide an indication to represent the successful authentication. In our experiments, we provide unique strings on the response webpage from the server such as "welcome user" to identify if the authentication succeeds.
- *Cryptographic functions.* We manually annotate the cryptographic functions in the Crypto library of Node.js [16], for AUTHSCAN to identify the cryptographic functions. We also annotate the functions in Mozilla jwcrypto [30], which is used in the implementation of BrowserID. AUTHSCAN automatically infers cryptographic operations using its default method in all other case studies.

For all cases, AUTHSCAN checks the authentication of the protocol and secrecy of the terms used for authentication (such as the *assertion* in BrowserID, which is discussed later in this section). These properties are checked against the network attacker as well as the web attacker.

Replay Attack in BrowserID. In two tested implementations of BrowserID, which use the `persona.org` as IDP, AUTHSCAN identifies and generates a confirmed *replay attack* in the network attacker model. AUTHSCAN generates an attack HTTP trace in which a malicious user logs into the SP by replaying the token named *assertion* (message (7) in Figure B.2), without providing login credentials to the IDP. The flaw leading to this attack is that the assertion is sent through an insecure channel (HTTP) and it does not contain any session-specific nonce. We have proposed to add a nonce in the signature to solve this problem and notified Mozilla about our finding. Mozilla acknowledged the security flaw.

CSRF Attack in BrowserID. AUTHSCAN identifies and confirms a replay attack in the web attacker model. AUTHSCAN reports this attack on two of the BrowserID implementations, other than the one from `developer.mozilla.org`. We have responsibly notified the vendors of these vulnerable implementations. After manual analysis of the inferred protocols, we find one crucial difference between the vulnerable implementations from the `developer.mozilla.org` implementation. In the latter, SP client sends two anti-CSRF tokens (`csrfmiddlewaretoken` and `next` which are inferred as nonces) in step 7 (Figure B.2), but these are absent from the protocol schema of the vulnerable SPs implementation, permitting a CSRF attack. AUTHSCAN reports that a malicious web site can send an HTTP POST request to the vulnerable SPs, which do not check the Referrer fields. Using this knowledge, we craft a script which can be used by the attacker to modify the content on the web pages without Alice's approval.

Secret Token Leak in Facebook Connect. By following a similar procedure as illustrated in the case of BrowserID, AUTHSCAN finds one confirmed flaw in the implementation of Facebook Connect, and another one in the usage of Facebook Connect by one out of the two SPs we tested. Both attacks leak secret tokens in the network attacker model. In this case, we report that automatic fuzzing was initially difficult because Facebook blocks login failure for a test username/password after 10 attempts. For this, we manually skipped fuzzing the initial login request to the IDP, but tested the remaining protocol with the SPs.

In the implementation of Facebook Connect, most of the communications are through HTTPS to prevent network attackers from stealing the authorization tokens. However, AUTH-

SCAN reports that the message at step 4 of Figure B.2-(b) is readable to the network attackers because they are transmitted through a non-HTTPS channel, so two credentials `c_user` and `xs` can be obtained by the attacker. Thus, the protocol is subject to a replay attack similar to the one in BrowserID. After our experiments, we discover that a similar attack against the previous version of Facebook Connect has been reported by Miculan *et al.* recently [158]. We conducted our tests in the end of April 2012; Facebook fixed this flaw in early May 2012 before we were able to notify them. In Facebook's latest implementation⁶, the communication in this step is protected with HTTPS. AUTHSCAN finds the other flaw leading to replay attack when an SP called Ebay-Classifieds uses the Facebook Connect. After completing the Facebook Connect, the SP sends the user credentials which can be used to fetch session cookies. However, these credentials are also sent through a non-HTTPS channel.

Non-secret Token in Using Windows Live ID. We tested AUTHSCAN on the authentication mechanism of Sina Weibo, a web site with more than 300 million users. It employs the Windows Live ID to authenticate users. In this experiment, AUTHSCAN initially reported a security warning claiming that a long-lived token (non-nonce value) is used to authenticate the user. We subsequently manually investigated this warning, and found that the long-lived token (named `msn_cid`) reported by AUTHSCAN is known publicly. For example, it can be obtained from various sources such as straight from the MSN user profile page (<https://profile.live.com/cid-xxxx>). When we added this token to the attacker's knowledge set and re-ran the experiment, AUTHSCAN was able to automatically generate an attack trace.

This flow occurs after a user completes the authentication with Windows Live ID, which demonstrates that AUTHSCAN is useful for finding simple, but severe logic flaws beyond the initial SSO authentication token exchange. Note that manually finding these attacks is not easy; AUTHSCAN eliminated 18 redundant cookies with differential fuzzing. The final HTTP packet which is sent from user to Weibo web site for authentication, as constructed by AUTHSCAN, sets the `msn_cid` value to the publicly known value as shown below.

⁶https://s-static.ak.facebook.com/connect/xd_arbiter.php?version=9

```
GET /msn/bind.php HTTP/1.1
Host: www.weibo.com
Connection: keep-alive
Cookie: msn_cid=xxxx
```

This vulnerability impacts all Weibo users who have ever logged in Weibo through Windows Live Messenger. We have reported this security flaw to Sina Weibo. The security department of Sina R&D has confirmed the exploit and posted us a gift for our contribution.

Guessable Token in Standalone Sites. AUTHSCAN detects one severe vulnerability in each of the two standalone web sites: IyerMatrimony and MeetingMillionaries. Both of them have a significant number of registered users, 220,000 and 1,275,000, respectively. The vulnerability shows that both of these two web sites authenticate users by some guessable token. Exploiting these vulnerabilities, the attacker can log into others' accounts and get full privilege of the victim users.

In the case of IyerMatrimony, after eliminating 7 redundant HTTP parameters with differential fuzzing, AUTHSCAN gets the following packet which can be used for a successful authentication.

```
http://www.iyermatrimony.com/login/intermediatelogin.php?sde=
U1ZsU01UZ3d0VE01&sds=QdR.j/ZJEX./A&sdss=Tf/GpQpvtzuEs
```

Through differential fuzzing, AUTHSCAN finds that `sds` and `sdss` keep constant among different accounts' multiple login sessions; for an individual account, the `sde` remains the same in its multiple sessions. Among the test accounts, AUTHSCAN finds that the 14-character prefix of `sde` remains constant and only the 2-character postfix is incremented by one across accounts whose IDs are consecutive numbers. AUTHSCAN confirms this flaw by predicting the value of `sde` for our testing accounts and successfully logging into the account.

In the MeetingMillionaries case study, AUTHSCAN generates a security warning about a short-length token used for authentication. We manually confirmed that this warning is a security flaw and notified the developers. In this site, a user can access his account information (including password stored in plain text) by visiting the following URL.

```
http://app.icontact.com/icp/mmail-mprofile.pl?r=36958596&l=2601&s=21DS&m
=318326&c=752641
```

AUTHSCAN finds l , m and c are constant among different users' sessions and r is associated with the user account. s is the only credential but due to its short length (4 characters), AUTHSCAN raises a warning of guessable token. Upon our manual investigation, we find that s is an alphanumeric string. We believe that automating attack generation for such tokens may be possible in the future; we tested that AUTHSCAN can send about 500 requests to the server within one minute. With such capability, it would take an enhanced implementation of AUTHSCAN at most 56 hours to guess the right s .

4.6.3 Efficiency & Running Time

Running Time. The total analysis time for most cases is less than 2 minutes, and can be as low as 5 seconds. The verification time for ProVerif is within 1 minute in our case studies. It shows that the security-relevant parts of the protocols generated are usually small. We find that additional source code results in the reduced number of iterations in our blackbox fuzzing step. For example, in BrowserID, the client-side code is available, therefore, the number of fuzzing iterations is smaller (20-30 rounds) than other SSO protocols (30-500 rounds as shown in the sixth column, Table 4.2). Our data shows that AUTHSCAN's protocol extraction step is sufficient to find flaws even when much of the protocol implementation is unavailable as shown in the Facebook case.

Redundant Data Reduction. When querying off-the-shelf verification tools like ProVerif, it is important to remove redundant terms for better scalability. As shown in Table 4.2, AUTHSCAN finds that the majority of the messages (more than 80%) and HTTP parameters (more than 50%) are irrelevant to the protocol and AUTHSCAN can successfully filter them out. This shows that an automatic tool is helpful in constructing the models from the complicated implementation details. Furthermore, this reduction helps greatly in reducing the verification time. For BrowserID, ProVerif does not terminate within one hour if we naively retain all terms exchanged in the communication. In summary, we find the AUTHSCAN has promising scalability for real-world security protocol implementations.

4.7 Related Work

Protocol Extraction. Works on automatically extracting models from the protocol implementations are most related to this work. Lie et al. [144] have proposed a method to automatically extract specifications from the protocol code. The model is extracted using program slicing and verified by Murphi tool. Aizatulin et al. [38] have proposed model extraction using symbolic execution. These works extract the protocol specifications from the source code, while our approach does not assume to have the source code and provides blackbox fuzzing to infer the semantics when the source code is not available.

Security Analysis on SSO Protocols. Extensive research has been conducted to manually analyze security of SSO protocols. By reverse engineering the client implementations, Hanna et al. [116] have revealed that some SSO protocols, including Facebook Connect and Google Friend Connect, use the cross-domain communication channel--postMessage insecurely, E.Tsyrklevich and V.Tsyrklevich [199] have demonstrated several attacks such as CSRF against the OpenID protocol. Wang et al.'s work [207] have conducted a field study on the commercially deployed web SSO systems and discovered 8 serious logic flaws in many notable IDPs and SPs. Xing et al. [212] have attempt to protect integrators for their integration of third-party SSO Web services.

Some formal analysis approaches also have been used to analyze the security of SSO protocols. Miculan and Urban [158] manually extract specification of Facebook Connect Protocol from the HTTP messages exchanged. They model the protocol in HLSP and check it using AVISPA. Bansal et al. [56] use applied pi-calculus and ProVerif to analyze the OAuth 2.0 protocol. Their work focuses on constructing concrete attacks from the attack trace reported by ProVerif, and building the operational web attacker model library called WebSpi to map the attack trace to web-site actions. Sun et al. [196] also model the web attacker precisely. Sun et al. manually extract OpenID 2.0 implementation in HLPSL and verify the model using AVISPA and found CSRF attacks. There are also other formal analysis approaches on SSO protocol. Most of them model the protocol manually based on the protocol documentation or specification, and take into consideration only

the network attack model. For example, there have been several formal analysis approaches on SAML SSO protocols [107, 117, 42]. In contrast to these work, AUTHSCAN looks at the security flaws in the implementations.

4.8 Summary

In this work, we present AUTHSCAN, an end-to-end platform to automatically recover authentication protocol specifications from their implementations. AUTHSCAN has successfully detected 7 security vulnerabilities in real-world applications automatically. Our techniques assume no knowledge of the protocol specifications being checked and rely on a small set of practical assumptions.

AUTHSCAN proves that the analyzable specification can be extracted from real-world implementations. However, due to the semantic gap between the implementation and the high-level modeling languages, some security-critical information may be missed during the course of extraction. The extracted models are seldom equivalent to the implementations which the security eventually relies on. Therefore, we aggressively make an attempt of applying formal methods directly on the implementations, which is presented in Chapter 5.

Chapter 5

Verification of Android Applications against Security Properties Using Targeted Software Model Checking

5.1 Introduction

Nowadays, numerous readily accessible Android apps with feature-rich functionalities have been extensively used in security-sensitive scenarios. For example, mobile users (simply *users* hereafter) heavily rely on them to handle personal data (e.g., contacts, financial data and geographic location) and consume premium services (e.g., online banking, online shopping and sending SMS messages). Moreover, Android apps are playing an increasingly important role in enterprise, government and military bureaus. For example, it has been reported recently that the US military is developing an app to enable drone control via Android phones [183]. Nevertheless, various security issues of Android apps are continuously being discovered and discussed, ranging from sensitive data leakage [220, 221, 104], to privilege escalation [150, 177, 70]. To enjoy the benefit of apps while preserving security, verifying them ahead of releasing and installation becomes imperative by market operators and users.

Most of the prior studies rely on static analysis and dynamic testing for security analysis, for instance, detecting sensitive data leakage [180, 44, 100, 99] and analyzing capability leakage [150, 103]. However, these approaches have their limitations. Static analysis may generate false alarms due to its inherent limitations in capturing runtime context (e.g., actual parameters and index of arrays) and tackling late-binding programming paradigms such as polymorphism and reflection. For instance, points-to analysis (which yields an over-approximation) is often used so that all potential violations are identified. While having all potential violations reported and fixed might be desirable for high-assurance government and military scenarios [97], it is nevertheless too restrictive for common usage. In contrary, dynamic testing only executes selected program paths and thus can precisely identify property violations [47, 119], but never proves their absence.

In this work, we seek a practical and credible approach for verifying Android apps against security properties. A potentially promising technique for solving this problem is software model checking [204, 55], which proposes an automatic way to verify properties of a finite-state system. The advantages of software model checking, compared with the above mentioned approaches, include that it does not generate false alarms if the model checker (e.g., Java Pathfinder (JPF) [204] for Java programs) actually executes the program under investigation and that it is capable of proving the absence of violations with an exhaustive search. Inspired by software model checking, we started an effort to build an Android model checker named DROIDPF upon JPF. The core technique of software model checking we use in this work is the dynamic *state-space exploration* which runs through the possible executions of an app. In the exploration engine, we also implement a taint-style system to track sensitive information flow for privacy property checking, and additional reachability checking for the privilege properties (e.g., blocking incoming SMS messages).

Using the state-of-the-art model checker JPF allows us to enjoy well-developed techniques of JPF (e.g., partial order reduction), as well as the various evolving features from JPF's active developer community (e.g., efficient exploration [115, 88]). Nonetheless, the challenge when we apply JPF to verify Android apps is at least threefold. First, unlike ordinary Java programs, Android apps are tightly coupled with the Android OS which consists of a set of libraries containing both

Java and native code, and complex inter-process communications. Although written in Java, the apps are compiled to bytecode that only runs on the Dalvik virtual machine instead of the traditional Java virtual machine (JVM). Therefore, it is hard for JPF to execute apps without using a real device or an emulator, not to mention storing and recovering program states. Second, due to the asynchronicity and event-driven execution paradigm, an app can have many entry points, whereas JPF allows only one entry at a time. One way to solve this problem is to construct a driver program that enumerates all possible event permutations, which would then activate all possible paths within the app. However, this approach could lead to false positives because some of the paths may never appear in practice. Third, same as the traditional model checking, there exists the infamous state space explosion problem. Therefore, verifying real-world apps calls for effective reduction techniques.

In order to tackle the first challenge, DROIDPF includes an extensible mock-up Android OS that abstracts the Android OS using ordinary Java programs so that analysis techniques/tools developed for ordinary Java can be employed. There are techniques available to automatically generate mock-ups of the environment [198, 76], but since the dependencies within the environment are complex, these techniques are not mature enough [200]. Thus, we create the mock-ups manually, which is the same approach used by other tools that require a mock-up of their environment for model-checking purposes [204, 156, 201]. Although mocking up an entire OS manually is a major effort, it is a one-time effort and DROIDPF has supported a range of functionalities of Android OS which allows us to verify a number of real-world apps. Furthermore, DROIDPF provides an extensible framework so that we can incrementally develop the mock-up.

In order to tackle the second challenge, we develop a technique called *dependency-constrained event permutation*. This technique reduces the event permutations by excluding impossible event sequences based on the dependency among the events (The details of this technique is introduced in Section 5.6.1).

Lastly, in order to cope with the problem of state-space explosion, DROIDPF uses a new technique called targeted model checking, which combines the virtue of both static and dynamic analysis. The idea of targeted model checking is to apply static analysis first to reduce the app

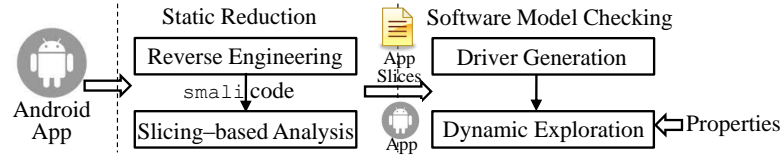


Figure 5.1: High-level Overview of Our Approach

and then model check (i.e., dynamically explore) only the remaining parts of the app. Figure 5.1 shows the high-level work flow of our approach, which consists of two stages. First, we obtain an over-approximation of all program paths which might lead to property violations. The approximation is then used to reduce the program by removing the safe parts of the app. Afterward, we model check the reduced app against the property. As a result, the efficient but imprecise static analysis dramatically reduces the state space, which enhances the scalability of our approach, whereas the precise dynamic exploration pinpoints the actual violations of the properties.

To summarize, our main contributions are threefold. First, we propose DROIDPF, a verification framework which employs the state-of-the-art software model checking techniques to Android apps. It enables JPF-based dynamic state-space exploration, by including an extensible mock-up Android OS that provides the major OS functionalities. Second, we propose targeted model checking and dependency-constrained event permutation for state space reduction. Third, to the best of our knowledge, DROIDPF is the first attempt to apply software model checking to verify security properties of Android apps.

We acknowledge that the effectiveness of DROIDPF is limited by not only the underlying JPF engine, but also the correctness and completeness of the mock-up OS. However, verifying the equivalence of the mock-up and the Android OS, or the correctness of the mock-up is a far more challenging task, and therefore is not discussed within the scope of this thesis. Nonetheless, we empirically show that DROIDPF not only can find bugs effectively but also verify apps in a number of cases. We evaluate DROIDPF using three sets (a total of 70) of apps: four real-world apps downloaded from Google Play and alternative app markets, four known malware samples from recent research [220] and the DroidBench [44] which itself is a comprehensive benchmark (including

62 apps) built to evaluate information leakage analysis. DROIDPF detects nearly all (except those with implicit information flow) of the known security issues from the malware samples, a previously unknown data leakage from a third-party library embedded by benign apps, a *use-after-free* bug from the benchmark, and two software bugs leading to app crash. We compare DROIDPF with state-of-the-art verification tools in terms of precision. Our experiments demonstrate that DROIDPF has low false negative rate and zero false positives.

5.2 Background

In the following, we present an overview of the Android framework and exemplify the difficulties in verifying Android apps.

5.2.1 Overview of Android

Android apps consist of four types of app components: *activity*, *service*, *content provider* and *broadcast receiver*. An activity implements the foreground logic, such as Graphical User Interface (GUI); a service runs in the background; a content provider is devoted to data management, which can use files, SQLite databases or the web as its back-end persistent storage, and a broadcast receiver handles the broadcasts sent by the OS and other components. The app components interact through inter-component communication (or ICC), and the exchanged messages are called *intents*. The components included in an app can be either statically defined in its *manifest* file (`AndroidManifest.xml`) or dynamically created at runtime.

Unlike a traditional Java program with a `main` method as its only entry point, Android adopts an *event-driven* execution mechanism, where an app implements a set of callbacks as event handlers (which can be registered/unregistered either statically or dynamically). Whenever a particular event occurs (e.g., launching an app and clicking a button), the Android OS invokes the corresponding callback methods. Therefore, an app usually has multiple entry points and event handlers (*entry points* and *event handlers* are used interchangeably). The events can be categorized into three types.

- **Lifecycle Events.** A component's *lifecycle* consists of multiple stages. As the user launches, pauses and resumes an app, its components of the app change their stages accordingly. When a component enters a new stage, the corresponding callback that is pre-implemented by the app is invoked by the OS. For instance, an activity has multiple stages such as `launched`, `running`, `paused` and `stopped`. When the user clicks the app's launcher icon, the activity enters the `launched` stage. Accordingly, the OS first calls method `onCreate()` and afterwards `onStart()`.
- **GUI Events.** GUI events occur when the user interacts with the apps, which include two types: data inputs from the user and actions taken by the user.
- **ICC Events.** The ICC events are mostly used for communication among apps. Besides, when some particular events occur, the OS broadcasts a message or directly invokes the callback methods implemented by the app. For example, when there is an incoming SMS message, the OS broadcasts an intent to notify the apps. As another example, when the system is running low on memory, the OS invokes `onLowMemory()` which is pre-implemented by the apps.

5.2.2 An Example

In the following, we present a simple app and illustrate the challenges in verifying Android apps. Figure 5.2 shows a made-up app which combines features of three apps in the DroidBench [44]. In this app, there exists a path through which the IMEI of the device is leaked, which is commonly seen in real-world malware samples [220]. Although we show source code in this section, DROIDPF directly works with the bytecode of the apps.

This app contains three components, i.e., two activity components `MotivatingAct` and `DummyAct`, and a content provider component `CProvider`. The code of `DummyAct` and `CProvider` is omitted in the figure to save space. `DummyAct` does nothing but just displays an empty canvas. Besides, two buttons are defined in the `MotivatingAct`'s xml layout file and their click handlers are set as `leak` and `noleak`, respectively (Figure 5.3). When `MotivatingAct` is


```

1 public class MotivatingAct extends Activity {
2     private String[10] array;
3     private com.droidpf.CProvider cp;
4     private String url = "content://com.droidpf.cp/test1";
5     protected void onCreate(Bundle b) {
6         setContentView(R.layout.DroidPF);
7         cp = new com.test.sqlite.CProvider();
8     }
9     protected void onStart(){
10        TelephonyManager mgr = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
11        String imei = mgr.getDeviceId(); //source
12        ContentValues values = new ContentValues();
13        values.put("theid", 10000);
14        values.put("thename", imei);
15        cp.insert(Uri.parse(url), values); //sink
16    }
17    protected void onRestart(){
18        Cursor cursor = cp.query(Uri.parse(url), null, null, null, null); //source
19        cursor.moveToFirst();
20        array[5] = cursor.getString(1);
21        array[4] = "no privacy";
22    }
23    /*leak() and noleak() are the callback functions of the buttons defined in MotivatingApp/layout/
24       DroidPF.xml*/
25    public void leak(View view){
26        SmsManager smsmanager=SmsManager.getDefault();
27        smsmanager.sendTextMessage("1234", null, array[indexLeak()], null, null); //sink
28    }
29    public void noleak(View view){
30        Intent intent = new Intent(this, DummyAct.class);
31        startActivity(intent); //sink
32    }
33    private int indexLeak(){
34        int index = Calendar.getInstance().get(Calendar.MONTH);
35        index = index/13; //index becomes 0
36        index += 5; // alternatively, index += 4;
37        return index;
38    }
39    private void onStop(){
40        ... // do nothing
41    }
42 }

```

Figure 5.2: Activity Component of the Illustrative App

started, it reads the IMEI by invoking `getDeviceId()` (line 11) and stores it in the database maintained by `CProvider` (line 15). When `MotivatingAct` is restarted (`onRestart()` is invoked), the IMEI is read from `CProvider` (line 18-20). After that, once `button1` is clicked, the IMEI is sent to a particular phone number through an SMS message (line 26).

This example demonstrates several technical challenges that may lead to imprecision in app verification.

- **Dependency on the OS.** The apps heavily depend on the Android OS---even a small app contains multiple Android API calls. These API calls are relevant to the app's behaviors such as the event handlers, and thus cannot be simply ignored during the analysis. For

```

1 <Button android:id="@+id/button1" android:text="@string/send" android:onClick="leak" />
2 <Button android:id="@+id/button2" android:text="@string/start dummy" android:onClick="noleak" />

```

Figure 5.3: Part of the xml Layout File

instance, if an analyzer does not examine the logic in `setContentView` (where the xml layout in Figure 5.3 is registered), the binding between the button and the click handler would be missed. As a result, the data leakage would not be detected because it is activated by the `onClick` handler. An intuitive remedy would be to analyze the part of the OS which becomes relevant given an API call. However, due to the complexity of the OS, even a simple API call would imply that a big portion of the OS must be analyzed.

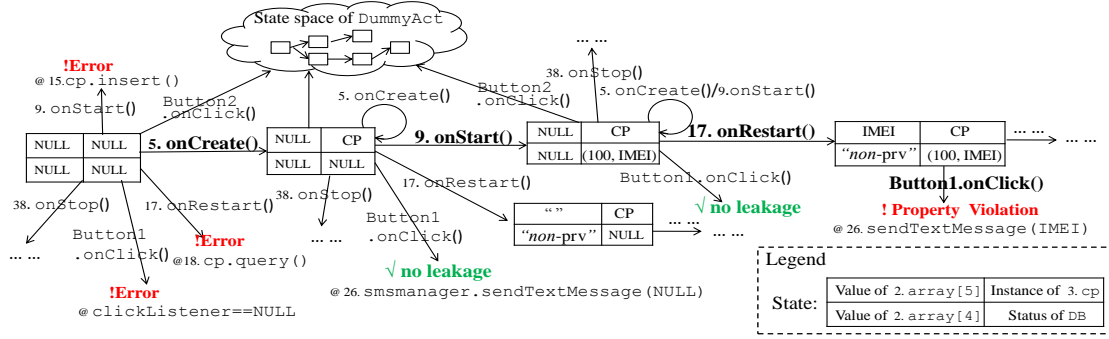
- **Asynchronous Event Occurrences.** Events in Android may asynchronously occur non-deterministically, which may be relevant to the app's behavior. For example, the sequence `onRestart`→`onClick` is safe but `onStart`→`onRestart`→`onClick` causes a leakage.
- **Obfuscation.** To obstruct malware detection, malware developers may employ obfuscation techniques [181, 128]. The example shows a simple obfuscation. Line 26 invokes `sendTextMessage` to send an SMS message. To decide whether this call leaks sensitive information (i.e., the IMEI), the actual values of the parameters have to be examined. However, line 33-36 make it difficult to statically determine the value of `index`, imprecision would be introduced into the approaches based on static analysis.

5.3 Overview of Our Approach

In this section, we define the problem of app verification and present an overview on how DROIDPF solves the problem.

5.3.1 The Model

According to our study of the vulnerabilities impacting Android which are listed in CVE [6], most (86.8% of 387) security threats on the Android platform reside in the apps. Therefore, we focus on application-level threats in this work (and exclude those in the OS) and verify one app



(The numbers before the instructions/variables stands for the line numbers in Figure 5.2; CP stands for a valid instance of the content provider; the emphasized path (in bold) shows the shortest path leading to a violation of privacy property.)

Figure 5.4: Partial State Space of the Running Example

at a time. In order to model check an app, we formalize its behaviors as a labeled transition system (LTS) $\mathcal{L}_{app} = (S, init, \Sigma, Tran)$, where S is the set of states, $init \in S$ is an initial state, Σ is an alphabet, and $Tran \subseteq S \times \Sigma \times S$ is the set of labeled transition relations.

States. The state that we consider includes the program state of an app. The program state is the same as the state in JPF, which is a snapshot of the execution status. It consists of the status of the heap (e.g., the values of the program variables), threads states and the program counter. We refer the readers to [204] for details on how program states are represented and compared in JPF.

Transitions. A transition is a state change by executing an atomic sequence of instructions (i.e., a block). We assume that the transition label is a string presentation of the instructions. For instance, if the transition is triggered by a method call `Button2.onClick()`, the label is simply `Button2.onClick()`.

Given the above LTS interpretation of an app, we then define relevant terms in the standard way. For instance, the execution of an app is formalized as runs, which are sequences of alternating states and blocks $\tau = \langle s_0, b_0, s_1, b_1, \dots \rangle$; a slice $\xi = \langle b_0, b_1, b_2, \dots \rangle$ is a sequence of blocks occurring in a run; and a state s_n is reachable if there exists a run τ such that $s_0 = init$ and $(s_i, b_i, s_{i+1}) \in Tran$ for all $i < n$. Furthermore, given a property (for model checking), the truth value of the property is defined based on the above-defined LTS in the standard way.

Figure 5.4 shows partly the state space of our running example. Each state contains the program state, i.e., values of the global variables (other information such as the program counter is

not shown), and the system state, i.e., status of the database. At each state, there are multiple choices for different behaviors. The different choices correspond to different thread scheduling and different events. In order to systematically explore all the states, we need an environment which systematically generates not only all different scheduling and but also all events. The former is solved by relying on JPF, which implements a backtrackable JVM that provides non-deterministic choices and control over thread scheduling. The latter (specific to the event-driven nature of Android apps) is solved by constructing driver programs which generates all relevant event sequences (detailed in Section 5.6). Figure 5.4 also suggests that the state space may quickly grow if no reduction/abstraction is performed.

5.3.2 The Property

DROIDPF focuses on checking the so-called privilege properties and privacy properties, which, according to the research of Android Malware Genome Project [220], are the main concerns of app verification. A privilege property is related to the use of the sensitive permissions, e.g., stealthily sending SMS messages, installing/uninstalling apps, deleting contacts and blocking incoming SMS messages. A privilege property is violated if there is a finite path in \mathcal{L}_{app} which leads to an invocation of the high-privileged APIs. This property can be checked through *reachability checking*, i.e., whether a state where a high-privileged API is invoked is reachable or not.

A privacy property is related to the actions of disclosing the private information, e.g., IMEI, GPS location and contacts. Privacy properties are often checked with the so-called taint analysis [44, 95]. A private property is violated if there is a finite path in \mathcal{L}_{app} from a private information *source* (e.g., contacts) to an information *sink* (e.g., the invocation of messaging sending API), and furthermore, the data at the sink must be tainted by the data at the source, i.e., there is a data dependency between them. To model check privacy properties, DROIDPF introduces an auxiliary Boolean variables *tainted* (for each data variable in the app) whose truth value tells whether a state in \mathcal{L}_{app} is tainted. Initially, *tainted* is false. For each transition (s_1, b, s_2) in \mathcal{L}_{app} , *tainted* is true at s_2 if and only if b is an information source, or there is a data-dependency between the

data variable at s_2 to a tainted variable at s_1 . As a result, checking private property is effectively reduced to reachability analysis, i.e., whether a sink is reachable with *tainted* being true.

By focusing on the above-mentioned properties, DROIDPF is effectively designed to cope with the following attacker model. That is, the adversary is able to develop and release malicious apps, or embed his malicious bytecode into benign apps. The adversary can also use advanced obfuscation techniques, e.g., complicating code flow, inserting extraneous code blocks and using reflection.

5.3.3 The Model Checking

Based on the above discussion, the problem of verifying an Android app is reduced to reachability analysis and thus can be solved using model checkers like JPF. In the following, we briefly introduce how JPF works. In order to systematically enumerate all states, at a program point where two or more choices become possible, e.g., scheduling points, human interaction and random system events, JPF records the current *state* and executes all choices separately, i.e., after completing one choice, it *backtracks* to the previous point and restores the state. In order to check whether a target state (i.e., a state at which the property is violated) is reached, DROIDPF uses the call-back mechanism provided in JPF (i.e., by introducing *listeners*). For privilege properties, DROIDPF listens on the `methodEntered` callback to check if over-privileged APIs are invoked¹. Checking privacy properties is slightly more complicated. First, DROIDPF listens on the `methodExited` callback to detect the invocations to the information sources and then labels the return values of those calls as *tainted*. Second, it listens on the `instructionExecuted` callback to track the taint flow. Third, it listens on the `methodEntered` to check whether the parameters of the invocations to the sinks contain *tainted* values. In order to precisely track the data flowing through the composite data types, we adopt a fine-grained taint tracking. When sensitive data are transmitted into a field of an object, DROIDPF taints the specific field instead of the whole object.

However, as discussed previously, there are three challenges that we need to solve before JPF

¹To minimize false positives, we use *context* to determine whether a behavior is initiated by the user, similar to prior research [78, 217].

can be applied. In the following, we present an overview of how DROIDPF solves the challenges and leave the details in the following sections. The high-level workflow of DROIDPF is shown in Figure 5.1. Firstly, given a privilege property or privacy property, DROIDPF performs a static analysis (i.e., program slicing) to identify all potential program paths which would lead to property violation. For instance, in Figure 5.4, the state space contains some parts which are obviously irrelevant, such as the state space of `DummyAct`, which can be identified statically. Once the irrelevant parts are identified, DROIDPF modifies the app accordingly so that the irrelevant parts are pruned and thus avoided during dynamic exploration later. The details of this step is presented in Section 5.4. Secondly, DROIDPF provides a mock-up of the Android OS so that JPF can dynamically execute the app. The details on how the mock-up OS in DROIDPF is constructed is in Section 5.5. Lastly, DROIDPF incrementally constructs a driver program which would drive the explosion of the app using JPF. On this step, we exclude the event sequences that are infeasible in reality, e.g., `onStart` \rightarrow `onCreate` in the app shown in Figure 5.4. Notice that these infeasible event sequences cannot be reduced by those general reduction techniques used in JPF. The details on how the driver program is constructed is presented in Section 5.6.

5.4 Static App Reduction

The goal is to statically prune part of the app which is irrelevant to a given property. Recall that the problem of verifying a privilege property or a privacy property is reduced to the problem of checking whether any *bad* state is reachable or not. Intuitively, static reduction in DROIDPF works by firstly over-approximating the set of bad states and then using backward program slicing to identify all potential program paths which would lead to any bad state. Since program slicing is a relatively mature technique (see [124] for details), in the following we focus on how DROIDPF over-approximates the set of bad paths and how DROIDPF makes sure all relevant program paths are included. The slicing implemented in DROIDPF is based a static slicing analysis framework called SAAF [124].

Given an app, the potential bad states include the following.

```

Button btn = new Button(this);
btn.setText("send");
btn.setOnClickListener(new View.OnClickListener(){
    public void onClick(View v){
        leak(v);
    }
});

```

Figure 5.5: Dynamic Event Handler Registration

- **Taint Sinks.** Reaching a taint sink (e.g., an invocation of the system function to send an SMS) would potentially violate a privacy property and thus all taint sinks are included. In addition, we include taint sources in the set of bad states so that we can tell whether a path reaching a sink goes through a source. Identifying all sources and sinks has been well researched in the literature [124, 44, 179] and thus DROIDPF simply uses those defined in SAAF, which include file/network/database I/O APIs (e.g., line 15&18 in Figure 5.2) and ICC APIs (e.g., line 30).
- **High-privileged APIs.** All invocations of high-privileged APIs (e.g., `abortBroadcast()` which may block incoming SMS) are included since reaching them may violate a privilege property.
- **APIs for Reflection.** Since the exact class/method/field accessed through reflection is hard to decide statically, reaching an API for reflection, such as `java.lang.reflect.Method.invoke()`, might lead to violation of the property and thus they are included.
- **APIs for Dynamic Registration.** Android allows an app to register components and event handlers both statically and dynamically. For example, Figure 5.3 shows how to register the button and its `onClick` listener statically through a layout file. Equivalently, the code snippet in Figure 5.5 registers a button at runtime. Due to this flexibility, it is hard to predicate statically whether the dynamic registration would lead to a property violation and thus they are included.

We remark that the given property may be violated only if a bad state, as defined above, is reached through a path in the app. Thus, next the set of bad states are used as slicing criteria so that program slicing techniques are used to identify all paths in the app which would lead to a bad state.

In particular, DROIDPF starts the backward slicing from each bad state in the set. It tracks both intra- and inter-component data and flow dependency. For intra-component backward slicing, DROIDPF searches for *use-def chains* and *call chains*. The former captures data flow relations while the latter captures call relations. First, based on the use-def chains, DROIDPF identifies 1) all of the registers and fields that have data dependency with the criteria, and 2) all of the methods (we call them relevant methods) in which the depended data are modified. For instance, tracking the parameter array in line 26 of Figure 5.2 leads to identification of the method `onRestart`. Second, based on the call chains, DROIDPF identifies the program slices which start from an event handler and lead to the invocation of any relevant method.

The more precise the slicing is, the more we can prune statically from the app. Nonetheless, because DROIDPF relies on dynamic execution afterwards to find actual problems, we can afford to over-approximate (without worrying about false alarms) when a precise points-to analysis is expensive. In particular, the following strategies are adopted in DROIDPF for efficiency. First, once an element of an array/string becomes relevant, the array/string becomes relevant. Second, once a field of an object becomes relevant, the whole object becomes relevant. Third, for polymorphism and overriding, whenever we cannot decide precisely, all methods with the same signature are included.

DROIDPF tracks inter-component flow as well. After identifying relevant methods within components, the next step is to collect the relevant components. The approach is to over-approximate the receivers of each ICC. Although it is sometimes feasible to statically identify the exact receivers of each ICC (e.g., using the approach in [170] which requires string analysis), for the same reason above, we (soundly) approximate the ICC by treating all the components which send/receive intents to be relevant.

We extract the components and event handlers from the slices and only execute them in the dynamic exploration step. The irrelevant parts are replaced with noops so that they are skipped by JPF. Taking the running example for instance, after static reduction, the paths starting from `onStop` and the state space of `DummyAct` can be excluded. Based on the above discussion, the following is established.

Proposition 5.4.1 *None of the potential property violating behaviors is reduced during the reduction.*

5.5 Mocking Up Android OS

In this section, we address the challenge of executing Android apps in a traditional JVM, which is necessary for JPF. A straightforward way might be to use an Android emulator as a *stub* and interact with DROIDPF through JPF's *native peer* mechanism. However, DROIDPF would not be able to track the execution in the stub since from its perspective, the stub is a black box. As a result, the logic in the OS which might be relevant to the checked property would be lost. One such example, in the content provider (CProvider) of our running example, is the sensitive data flow through the database file. Without tracking the behaviors of the OS, the data flow would become untraceable for DROIDPF.

DROIDPF's solution is to develop a set of mock-ups of OS functionalities in Java. The main challenge of OS abstraction is to balance scalability and correctness. For instance, while it might be feasible to mock-up all native code in the Android OS, it may not be easy to ensure that the mock-up is correct, due to lack of specification of the native code. Rather, we choose to develop the mock-up at such a level that the semantics relevant to the property is easy to understand. For instance, instead of mocking up only the native code in the database component in Android (which contains both native code and Java code), we mock up the entire database component instead (by implementing a HashMap essentially). In addition, instead of building the mock-up OS from scratch, we reuse the source code of the Android OS as much as possible.

Table 5.1 lists the main modules of the Android OS that we have mocked up. DROIDPF intercepts the invocations to the native code and mocks-up their expected behavior in Java. Some mock-ups are straightforward. For example, the resource managers (e.g., location manager and SMS manager), can be substituted with dummy ones which simply return faked values without invoking the real managers in native code. Some are rather complicated, including the GUI and the I/O management. We have thus devoted most of our efforts to those.

Table 5.1: Main Modules Modified in DROIDPF

Types	Modules
Components	android.app.(Activity Service ContextImpl Dialog), android.content.(BroadcastReceiver ContextWrapper), android.content.pm.PackageManager, android.os.(AsyncTask Bundle Looper)
GUI	android.widget.(TextView Button CheckBox EditText ImageView LinearLayout Toast), android.opengl.GLSurfaceView, android.view.(View Surface WindowManagerImpl)
ICC	android.content.Context.(startActivity sendBroadcast registerReceiver unregisterReceiver startService stopService bindService unbindService), android.content.Intent
Resource Manager	android.location.LocationManager, android.net.ConnectivityManager, android.telephony.(TelephonyManager SmsManager)
I/O	java.io.*, android.database.sqlite.*, android.content.SharedPreferences

GUI. GUI relies on enormous native code to render images and handle the user's interaction. Its events include action events (e.g., clicking a button) and data inputs (for example, typing texts through a `TextEdit`). DROIDPF mocks them up in different ways. For an action event, it directly invokes the event handler from the drivers. For the data inputs, DROIDPF relies on the analyst to configure the range of the input values with a *choice generator* for high precision. Since most of the GUI events are related to the actions in the mobile application domain, we consider this approach practical.

I/O Management. As shown in the running example, the sensitive information can flow through the database and files, which may become untraceable. To address this challenge, we simulate the database and files I/O with in-memory data structures. When the app performs a write operation to store a data item into an external file, the item is written into an in-memory buffer (under control of DROIDPF). Similarly, file-based SQL databases are simulated using in-memory tables. With the interception and proper implementation of a set of I/O APIs, such as `read`, `cursor`, `uri` and `SQLiteOpenHelper`, the mock-up I/O is completely transparent to the apps.

Limitation. Mocking up the Android OS indeed requires a lot of engineering effort, as we have experienced. However, this step is a one-time effort and as far as as we can tell, this is the only way

to verify apps. Having said that, mocking up the whole OS before analyzing any app is perhaps not smart. Thus, in our work, besides mocking up common used components like GUI and I/O management, we always start with static app reduction and then focus on mocking up only the relevant components, which based on our experience, are often much reduced.

5.6 Driver Generation

The goal of driver generation is to construct a driver program which allows JPF to systematically explore all states in the app. Furthermore, in order to tackle the state-space explosion problem, DROIDPF is designed to generate only feasible event sequences relevant to the app.

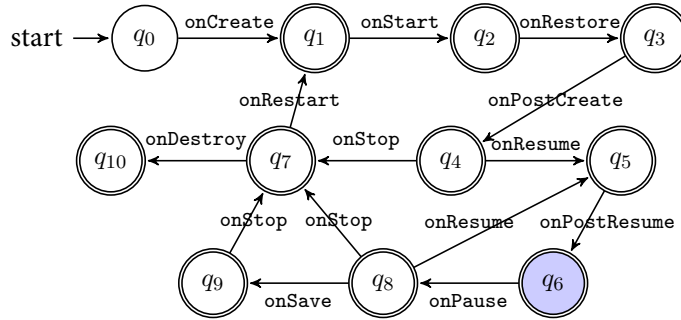
5.6.1 Dependency-constrained Event Permutation

One challenge for the driver generation is that events can occur asynchronously. An intuitive but costly strategy is to enumerate all of the permutations of the events, which guarantees completeness but definitely generates invalid sequences and leads to the notorious path explosion problem. Therefore, we introduce a dependency-constrained event permutation, which exploits the dependency relations among the events to prune the exploration paths.

We use deterministic finite automata (DFA) to specify the dependency relations among the events. An event sequence is valid if and only if it is accepted by the DFA. A DFA is a 5-tuple: $(Q, \Sigma, \Delta, q_0, S_a)$, where Q is a finite set of states; Σ is a finite set of events (alphabet); Δ is the transition set; q_0 is the start state and S_a is the set of final/accepting state.

We start with defining a *full* lifecycle DFA (L-DFA) for each of the four types of app components, which is built based on the official documentations of Android [18]. The alphabet of the full L-DFA includes all of the lifecycle event handlers. As an example, Figure 5.6 shows the full L-DFA of the activity component.

Most of the time, not all of the lifecycle events are relevant to the checked properties. Therefore, at the second step, DROIDPF projects the full L-DFA with respect to the app's alphabet using algorithm 3. We call the obtained DFA a concrete L-DFA. After obtaining the concrete L-DFA,



q_0 : start state; q_1 : activity is launched; q_2 : activity is restoring saved state; q_3 : saved state is restored; q_4 : activity is becoming visible; q_5 : activity is resuming; q_6 : activity is active in the foreground; q_7 : activity is finishing; q_8 : activity is not visible; q_9 : system is saving activity state; q_{10} : activity is being destroyed by the system. `onSave` stands for `onSaveInstanceState` and `onRestore` for `onRestoreInstanceState`

Figure 5.6: Full L-DFA of Activity Component

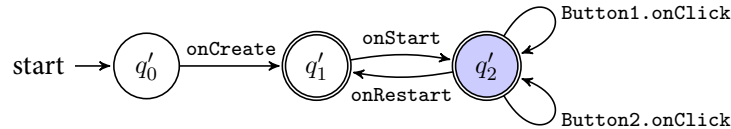


Figure 5.7: DFA of `MotivatingAct` in Figure 5.2

DROIDPF extends it to incorporate those relevant GUI and ICC events identified in static reduction. Our key insight is an invariant that an activity only handles GUI events and ICC events when it is in *active* state (q_6 in Figure 5.6). Therefore, we add the permutations of these events to the active state (availability of a GUI item is checked before invoking its handlers, which ensures that it is not disabled). As an example, Figure 5.7 shows the final DFA specifying the dependency relations of events in `MotivatingAct`.

Based on the DFA, DROIDPF generates legitimate event sequences. The number of the sequences may be infinite due to the loops in the DFA (e.g., $q'_1 q'_2 q'_1$ in Figure 5.7). However, we do not have to restrict the number of loops as the state space exploration would terminate when all states (not event sequences) are explored.

Algorithm 3 L-DFA Projection Algorithm

Input: dfa —full L-DFA, $EvntSet$ —lifecycle event set
Output: $cdfa$ —concrete L-DFA

- 1: **for all** $t = q_i \xrightarrow{e} q_j$ in $dfa.\Delta$ **do**
- 2: **if** $e \notin EvntSet$ **then**
- 3: $q_i \leftarrow combine(q_i, q_j)$
- 4: $dfa.Q \leftarrow dfa.Q - q_j$;
- 5: $dfa.\Delta \leftarrow dfa.\Delta - t$
- 6: **for all** $t' = q_j \xrightarrow{e'} q_k$ **do**
- 7: $dfa.\Delta \leftarrow dfa.\Delta \cup \{q_i \xrightarrow{e'} q_k\}$
- 8: **end for**
- 9: **for all** $t' = q_k \xrightarrow{e'} q_j$ **do**
- 10: $dfa.\Delta \leftarrow dfa.\Delta \cup \{q_k \xrightarrow{e'} q_i\}$
- 11: **end for**
- 12: **end if**
- 13: **end for**
- 14: $dfa.\Sigma \leftarrow EventSet$
- 15: **return** dfa ;

5.6.2 Driver Generation Algorithm

Overview. Given the relevant components and events, DROIDPF creates an initial driver which only includes a set of relevant lifecycle event sequences of the app's main activity component (i.e., the first component invoked when the user launches the app). By dynamically executing the created driver, DROIDPF is able to reach more relevant components and events. When a relevant event is reached, DROIDPF constructs new event sequences which are accepted by the DFA, and generates new drivers to drive the next-round execution. When a relevant component is reached, DROIDPF includes its lifecycle event sequences. By iteratively executing each of the drivers, DROIDPF gradually expands the driver set until no more relevant components and events can be reached (meaning a complete set of drivers are generated).

Example. We take our running example (Figure 5.2) for instance to show the generation process. Initially, DROIDPF generates a driver which only includes the three lifecycle events `onCreate`, `onStart` and `onRestart` of `MotivatingAct`. It then dynamically executes the driver. When executing the `setContentView` (line 6), it reaches two relevant buttons which are registered

Algorithm 4 Dynamic Exploration Algorithm

Input: App —app, $RCmpSet$ —relevant components, $REvntSet$ —relevant events, ED —dependency DFA

Output: A driver set

- 1: $EvntSet \leftarrow drvInit(App, RCmpSet, REvntSet, ED)$
- 2: $CmpSet \leftarrow EMPTY$
- 3: $DrvSet \leftarrow drvGen(EvntSet, CmpSet, ED)$
- 4: $DrvSet_{old} \leftarrow EMPTY$
- 5: **while** $DrvSet \neq DrvSet_{old}$ **do**
- 6: $DrvSet_{old} \leftarrow DrvSet$
- 7: **for all** drv **in** $DrvSet$ **do**
- 8: $(EvntSet, CmpSet) \leftarrow dynExplore(drv)$
- 9: **end for**
- 10: $(EvntSet, CmpSet) \leftarrow filter(EvntSet, CmpSet, RCmpSet, REvntSet)$
- 11: $DrvSet \leftarrow drvGen(EvntSet, CmpSet, ED)$
- 12: **end while**
- 13: **return** $(Init, ProSet)$;

in the layout xml file. It thus includes the `onClick` events of the buttons into the new driver. Later, it reaches the content provider `CProvider` which is dynamically registered in line 7. It then includes the lifecycle events of `CProvider` and generates another driver which contains all present components and events. When executing line 30, it parses the intent and identifies the invoked component as `DummyAct`. Because `DummyAct` has been found irrelevant in the static analysis step, `DROIDPF` does not include it into the driver.

Algorithm. Algorithm 4 details our driver generation algorithm. The inputs of the algorithm include the app, the relevant components/events and a DFA. The outputs of the algorithm are a set of drivers. The algorithm consists of two steps: driver initialization (line 1-3) and expansion process (line 4-10). The `drvInit` method (line 1) first identifies the main component by parsing the app's `AndroidManifest.xml` file. After obtaining the main component, `DROIDPF` searches its lifecycle event handlers from the relevant event set. The identified events are taken as inputs to the `drvGen` which permutes the events and generates a set of event sequences. Each of the sequences (denoted by seq) is then used to generate a driver, which invokes the event handlers in the order of seq .

In the expansion step (line 4-10), `DROIDPF` expands the initial driver set. It starts from a dynamic exploration (line 7-8). The `dynExplore` method executes each of the drivers in the

driver set *DrvSet*. During the execution, it identifies dynamically-registered components and GUI elements.

- *Components*. DROIDPF considers two types of registration: *instantiation* and *invocation by ICC*. First, once a component is instantiated dynamically, such as *CProvider* in our running example (line 7, Figure 5.2), DROIDPF adds it into *CmpSet*. Second, when the program invokes a new component using ICC, such as `startActivity()` and `startService()`, the invoked component is included.
- *GUI Elements*. For each GUI element that is registered dynamically, DROIDPF adds its event handlers into *EvntSet*.

After identifying new components and events, DROIDPF only selects those relevant ones for driver generation (line 9-10).

5.6.3 Correctness of Generated Drivers

Our driver generation algorithm preserves both soundness and completeness. Here soundness means that all the event sequences generated can occur in the real-world execution, while completeness means all the sequences leading to property violations are included. Intuitively, the soundness and completeness are based on proposition 5.4.1 and the following three rules.

Proposition 5.6.1 *DFA-based sequence generation is sound and complete.*

Proposition 5.6.2 *The code executed by dynamic exploration is reachable in the real-world execution.*

Proposition 5.6.3 *Parsing manifest in *drvInit* (line 1) is able to precisely identify lifecycle events, since they cannot be hidden or obfuscated by the adversary in order to be invoked by OS.*

For soundness, initially, *drvInit* (line 1) is sound (rule 5.6.3) and the initial sequences generated in *drvGen* are sound (rule 5.6.1). In each iteration, expansion of relevant components and events is sound (rule 5.6.2). For completeness, initially, *drvInit* (line 1) is able to identify all the relevant lifecycle callbacks (rule 5.6.3 and 5.4.1) and the initial sequence generation

Table 5.2: Events Supported by DROIDPF

	Class Name	Handlers
lifecycle	android.app.Activity	onCreate, onDestroy, onPause, onStop, onPostResume, onRestart, onResume, onRestoreInstanceState, onPostCreate, onSaveInstanceState, onStart
	android.app.Service	onBind, onCreate, onStart, onDestroy
	android.app.ContentProvider	onCreate
	android.app.Application	onCreate, onTerminate, ActivityLifecycleCallbacks
	android.preference.PreferenceActivity	onCreate, onDestroy, onStop, onPreferenceTreeClick
GUI	android.app.Activity	onCreateOptionsMenu, onPrepareOptionsMenu
	android.widget.EditText	getText, setText
	android.widget.Button	performClick
	android.view.View.OnClickListener	onClick
	android.view.View.OnTouchListener	onTouch
	android.preference.Preference	onPreferenceChange
	android.preference.Preference.OnPreferenceChangeListener	onPreferenceChange
	android.preference.Preference.OnPreferenceClickListener	onPreferenceClick
ICC	android.app.Activity Service	onLowMemory
	android.content.BroadcastReceiver	onReceive
	android.app.Application	onConfigurationChanged, onLowMemory, onTrimMemory
	android.location.LocationListener	onLocationChanged, onProviderEnabled, onProviderDisabled, onStatusChanged

in *drvGen* is complete (rule 5.6.1). In each iteration, since generation of event sequences are complete (rule 5.6.1), all the dynamically-/statically-registered relevant components and events can be reached (assumption, rule 5.6.2 and 5.4.1).

5.7 Implementation and Evaluation

DROIDPF has been implemented with approximately 20K lines of Java code, in addition to various libraries that we employ. Most of our engineering efforts on implementing DROIDPF are spent on driver generation and mocking up the methods in Android OS. The static app reduction is implemented based on SAAF [124]. We use apktool [1] to translate the DEX code of apps into *smali* format, on which the static slicing is performed. Directly analyzing the bytecode instead of source code allows us to avoid the imprecision of decompiling apps. The events that are taken into consideration by driver generation are listed in Table 5.2. The mock-up Android OS is developed based on the Framework of Android 4.0 (API Level: 14). We use dex2jar [7] to translate DEX code of apps into Java code compatible with JPF.

In the following, we evaluate DROIDPF in terms of its effectiveness and accuracy. In particular, we investigate the following four research questions.

5.7. IMPLEMENTATION AND EVALUATION

- **RQ1** Effectiveness: can DROIDPF detect security and privacy property violations in real-world apps, or prove their absence, and what is its efficiency?
- **RQ2** Precision, can DROIDPF achieve a more correct and precise analysis than the advanced static and dynamic analysis tools?
- **RQ3** Is static app reduction effective for improving the scalability of state space exploration?
- **RQ4** Given that DROIDPF has the mock-up OS to enable the execution of Android apps on JPF's JVM, is it possible to support JPF's diverse set of property checkers for non-security properties?

Our evaluation subjects include the following three sets. The first set consists of four real-world apps downloaded from Google Play and an alternative market in China, including an ebook app (denoted by ebook), a scientific calculator app and a location sharing app named GPS Share [11] which allows users to share locations through SMS. These apps represent real-world applications which utilize a broad variety of core functionalities supported by Android, such as accessing locations, accessing the IMEI and sending SMS messages. We select the ebook app and the calculator app because we find that they request privileges that are not necessary for their functionality; for example, the ebook app requests the permissions of `INTERNET` and `READ_PHONE_STATE`. We use them to investigate whether DROIDPF can identify malicious behaviors abusing these permissions. Our subject set also includes a complicated and realistic open source app called InsecureBank [173]. It is embedded with harmful API calls and various behaviors of leaking information.

The second set consists of malware samples. We test four known malware samples released by recent research [220], namely ZitMo, Geinimi, Spitmo and Zsone. These samples violate the privilege property by sending premium SMS messages (Geinimi and Zsone) and blocking incoming SMS messages (ZitMo and Spitmo), and privacy property by stealing the incoming SMS messages (ZitMo and Spitmo) and IMEI (ZitMo).

The third set of apps is a comprehensive benchmark called DroidBench [44], which has been created to evaluate information flow analysis. It includes 62 open source apps, some of which lead to violations of privacy properties. It contains a suite of challenges for analysis tools to check

5.7. IMPLEMENTATION AND EVALUATION

Table 5.3: Statistics in Our Experiments

LOC: lines of code in *smali*, including third-party libraries; #Components: number of relevant/overall components; #Events: number of relevant/overall events; OM: out of memory; AC: app crashed. Those numbers in the brackets stand for the statistics in the exploration without static reduction.

Subjects		Static Reduction			Dynamic Exploration		
App name	LOC	#Components	#Events	Time(S)	Exploration Times(S)	Memory (MBytes)	#Backtracked States
Ebook	7.0K	2/2	7/9	10.6	2 (4)	188 (321)	1428 (3900)
Calculator (main Activity)	6.5K	3/4	3/26	3.4	1 (AC)	119 (AC)	8 (AC)
GPSShare	1.2K	1/3	8/17	2.4	3 (172)	366 (409)	2792 (49556)
InsecureBank [†]	2.3K	4/5	11/12	3.0	183 (184)	596 (596)	132307 (133809)
Youmi (main view)	15.6K	1/1	1/5	10.6	2 (13)	185 (590)	73 (5257)
ZitMo	576	1/3	3/4	2.3	6 (6)	111 (117)	393 (394)
Geinimi	13.0K	4/6	6/21	5.9	OM (OM)	OM (OM)	2.5K (OM)
Spitmo	704	1/1	1/1	2.6	4 (4)	78 (78)	66 (66)
Zsone [‡]	29.4K	2/4	3/10	30.5	3 (AC)	61 (AC)	113 (AC)
PrivateDataLeak1	50.8K	1/1	3/3	8.6	2 (2)	78 (78)	38 (38)
Button2	50.7K	1/1	4/4	9.0	2 (2)	78 (78)	41 (41)
AnonymousClass1	50.7	1/1	3/6	8.6	2 (2)	78 (78)	20 (177)
LocationLeak2	50.7K	1/1	3/6	8.7	2 (2)	78 (78)	20 (177)

[†] When fed with infinite event sequence, the exploration of InsecureBank did not terminate. The statistics in this row was obtained by setting the length of sequence as 16.

[‡] In the experiment of full permutations, Zsone crashed due to a null reference when its `onResume()` is called.

both false negatives and false positives, such as locations in arrays and lists, callbacks, field and object sensitivity, ICC, obfuscation, reflection and implicit flows. This benchmark is challenging for the state-of-the-art static verifiers such as FlowDroid [44], which partly motivates this work.

5.7.1 Effectiveness of DroidPF

In our experiments, we check both privilege and privacy properties. For privilege properties, we focus on the sensitive behaviors that are not initiated by the GUI events (i.e., stealthy behaviors), such as blocking incoming SMS messages and sending SMS messages. For privacy properties, we check whether the sensitive data, including device ID, location and contacts are leaked through the network, SMS and logs. Table 5.3 lists the statistics of our experiments (for the sake of brevity, we omit most statistics of the DroidBench experiments). Our experiments were conducted on a PC with Intel Core 2 DUO CPU E6550 at 2.33 GHz and 4GB RAM. The verification results using DROIDPF (i.e., the counterexamples or the correctness claim) have been confirmed by manually analyzing the *smali* code.

DROIDPF successfully verifies one of the real-world app samples and detects property viola-

5.7. IMPLEMENTATION AND EVALUATION

tions or bugs from the remaining three. DROIDPF detects in GPS Share that the location information is taken as a parameter to invoke Android's SMS activity, we do not regard it as a "sink" since SMS activity prompts the user and will not send out the SMS messages without the user's consent. Second, DROIDPF detects that a library in the calculator app connects the server located at `http://58.221.57.115:81` to download app packages. Third, DROIDPF reports leakage of the device ID in the ebook app, which was previously unknown. Our investigation reveals that its main components do not leak the device ID. Instead, the leakage occurs in an embedded advertising SDK named Youmi. Lastly, from the InsecureBank, DROIDPF identifies the leakage of sensitive data (phone number and input data) through the channels of HTTP, system logs and SD card.

For the malware samples, DROIDPF identifies data leakage from three of them. First, from Zsone, DROIDPF detects a trace which is initiated by the `onCreate` lifecycle event of its main activity and leads to the invocation of `sendTextMessage`, which sends seemingly meaningless SMS messages (e.g., `aAHD`) to four numbers (e.g., `10626213`). After further investigation of the phone numbers, we find that those messages are used to register premium services from the Chinese mobile networks. DROIDPF also identifies that one of Spitmo's broadcast receivers (i.e., `SMSReceiver`) blocks incoming SMS messages by invoking `abortBroadcast`. It also forwards the messages to a phone number stored in the file `asset/settings.xml`. In ZitMo, DROIDPF detects a similar trace. However, DROIDPF does not terminate on Geinimi and we stop the exploration when the machine is out of memory. After our manual investigation, we find that Geinimi blocks on contacting the C&C server for commands, while the server is no longer active.

To investigate the effectiveness of the static path reduction, we test the main components of the apps without applying the reduction (i.e., the numbers are in brackets in Table 5.3). As shown in the table, when we take the full permutations of the events as input to the apps, the time and space efficiencies decrease significantly. In addition, in the experiment on the calculator app, DROIDPF detected a software bug. The app does not validate the input before parsing a value of type `double`, which causes the crash of the app when DROIDPF clicks button "=" after clicking button ".".

5.7. IMPLEMENTATION AND EVALUATION

Table 5.4: Comparison of Results on DroidBench

✓ = correct alarm, ✗ = false alarm, ○ = missed leak
 multiple circles in one cell: multiple leaks expected
 empty cell: no leaks expected and none reported

App Name	FlowDroid	TaintDroid	DROIDPF
Arrays, Lists and HashMaps			
ArrayAccess1	✗	✗	
ArrayAccess2	✗	✗	
ListAccess1	✗	✗	
HashMapAccess1	✗	✗	
Callbacks			
AnonymousClass1	✓✓	✓✓	✓✓
Button1-3	✓✓✓✓	✓✓✓✓	✓✓✓✓
LocationLeak1&2	✓✓✓✓	✓✓✓✓	✓✓✓✓
LocationLeak3	✓✓	✓✓	✓✓
MethodOverride1	✓	✓	✓
MultiHandlers1	✗		
Unregister1	✗		
Field and Object Sensitivity			
FieldSensitivity1&2&4		✗✗✗	
FieldSensitivity3	✓	✓	✓
InheritedObjects1	✓	✓	✓
ObjectSensitivity1&2		✗✗	
Inter-Component Communication			
IntentSink1	○	✓	✓
IntentSink2	✓	✓	✓
ActivityCommunication1	✓	✓	✓
Lifecycle			
BroadcastReceiverLifecycle1	✓	✓	✓
ActivityLifecycle1-4	✓✓✓✓	○✓○✓	✓✓✓✓
ServiceLifecycle1	✓	✓	✓
ApplicationLifecycle1-3	✓✓✓	✓✓✓	✓✓✓
Ordering1	✗✗		
General Java			
Loop1&2	✓✓	✓✓	✓✓
SourceCodeSpecific1	✓	✓	✓
StaticInitialization1	○	✓	✓
UnreachableCode			
Exceptions1&2&4	✓✓✓	✓✓✓	✓✓✓
Exceptions3	✗		
Obfuscation			
Obfuscation1	✓	✓	✓
Reflection			
Reflection1-4	✓○○○	✓✓✓✓	✓✓✓✓
Miscellaneous Android-Specific			
PrivateDataLeak1-3	✓✓✓	✓✓✓	✓✓✓
DirectLeak1	✓	✓	✓
InactiveActivity			
LogNoLeak			
Library1&2	✓	✓	✓
Implicit Flow			
ImplicitFlow1-4	○○○○ ○○○○	✓✓○○ ○○○○	○○○○ ○○○○
Sum, Precision and Recall			
✓, higher is better	39	44	44
✗, lower is better	9	9	0
○, lower is better	13	8	8
Precision $p = \checkmark / (\checkmark + \text{✗})$	81.3%	83.0%	100%
Recall $r = \checkmark / (\checkmark + \text{○})$	75.0%	84.6%	84.6%
F-measure $2pr / (p + r)$	0.78	0.84	0.92

5.7.2 Precision of DroidPF

We use the DroidBench to evaluate DROIDPF in terms of precision, which is the most critical criteria for a verification tool. We compare DROIDPF with state-of-the-art static information-flow analysis tool FlowDroid [44] and dynamic taint analysis tool TaintDroid [95].

Table 5.4 summarizes the test results. Most of the statistics of FlowDroid is collected from [44]. As DroidBench evolved, 25 new apps were newly added. We run FlowDroid to test these new apps (results are in red font in the table). Overall, DROIDPF achieves more precise results than both FlowDroid and TaintDroid. There are two factors why DROIDPF outperforms FlowDroid. The first reason is that some program paths leading to leakage is very hard to identify statically, for example, data flow through HashMaps, lists and arrays. The second reason is that the dynamic feature of DROIDPF makes it more capable in addressing the Android-specific event-driven execution feature. In particular, DROIDPF can precisely bind event handlers, track dynamic (un)registrations and order the occurrence of events. Compared with TaintDroid, DROIDPF adopts a fine-grained taint tracking on composite data types. Therefore, it becomes more precise in detecting taint flowing through these data types.

During the exploration of DroidBench, one event sequence causes one of the apps called `FieldAndObjectSensitivity_FieldSensitivity1` to crash. Our examination confirms that the crash is resulted by a *use-after-free* vulnerability. We have report this vulnerability to the author of DroidBench, who acknowledged our finding and will rectify it in the upcoming DroidBench 2.0.

5.7.3 Experiments on Non-security Properties

Given that DROIDPF enables the execution of Android apps on JPF, we can compare it to JPF-Android [201], another tool that verifies Android applications using JPF. JPF-Android makes use of JPF's class modeling and native method modeling features to bound the environment of the application created by the Android core libraries. To drive the execution of the application, users script nondeterministic event sequences and can also set the state of the environment. This is

a more manual approach to limit the number of input events than DroidPF where the tool can slice away unimportant or uninteresting parts of the application to reduce the number of available events. JPF-Android supports detection of deadlock, race conditions and runtime errors using the listeners provided by JPF. It does not, however, target security errors in Android application but allows users to specify properties in the form of *Checklists* to verify that the application executes specific event sequences [202].

We apply DROIDPF on two apps tested by JPF-Android in the paper [201], an app containing deadlock and a calculator app. Since this experiment does not target the detection of security errors, we directly apply driver generation and exploration without static reduction of the application code. DROIDPF successfully detects the deadlock which has been reported by JPF-Android. It also detects an unreported bug in the calculator leading to an app crash. The bug is caused by an unchecked array index, which occurs when the input box is empty and DROIDPF "presses" the backspace button. JPF-Android does not detect this bug because the triggering sequence was not part of the script used to analyze the calculator.

5.7.4 Limitation and Discussion

We acknowledge that verifying Android apps is an extremely challenging task. DROIDPF is only a step towards that goal and it has its limitations.

Firstly, DROIDPF is based on model checking and thus it has the limitation of model checking, i.e., the app (after static reduction) must have only finitely-many states and the number of states is not beyond the capability of current model checking techniques. Since DROIDPF leaves the task of state exploration and state comparison (to avoid exploring the same state more than once) to the underlying JPF, DROIDPF can verify an app only if the app, after reduction performed in DROIDPF, can be verified by JPF.

Secondly, similar to most information flow analysis [191, 95], DROIDPF cannot identify the implicit leakage through control flow dependency; for example, the value of x is leaked implicitly in this statement: `if (x==1) y=1; else y=0;`. A straightforward solution is to propagate the

taint to y , but this may lead to over-tainting and false-propagation problems. For future work, DROIDPF can employ more advanced solution such as DTA++ [132].

Thirdly, DROIDPF is perhaps not effective if the app contains behavior which is triggered by data inputs and which cannot be pruned through static reduction, such as the attack behaviors enabled on a specific date and our Geinimi case study which launches attacks based on received commands. In theory, we could always enumerate all possible values for the data input. In practice, a data input of type string or float would often result in state space explosion. For future work, techniques like symbolic execution [41, 217, 75, 48, 178] can be considered to alleviate this problem.

Lastly, similar to related work in the literature, DROIDPF cannot handle apps that include native code (in the app itself not from the Android OS). In this work, we also do not consider hardware attacks (e.g., cold-boot attack [172]) and side channels (e.g., Soundcomber [188]).

5.8 Related Work

Android Application Verification. DROIDPF is not the first study on verifying Android apps. Merwe et al. propose JPF-Android [201, 202] which verifies the Android apps also using JPF. DROIDPF is distinctly different from JPF-Android in terms of many aspects. First, JPF-Android uses simplified model of the Android Framework while DROIDPF mostly reuses the source code of the Framework to preserve the semantics. Second, JPF-Android requires the analysts to script the input events while DROIDPF attempts to generate the drivers automatically. Third, JPF-Android currently checks common properties like deadlock or runtime errors while DROIDPF incorporates non-trivial security property checking.

Android Application Analysis. There have been several approaches using program analysis to analyze the security and privacy properties of Android apps [150, 44, 99, 78, 217, 126]. FlowDroid [44] is one of the most advanced approaches, which proposes a taint analysis featuring in addressing Android's ubiquitous callbacks. Pegasus [78] checks apps for the properties that can be specified in Linear Temporal Logic (LTL). Similar to Pegasus, AppIntent [217] aims to

detect malicious behaviors by identifying whether the suspicious behaviors are initiated by the user. CHEX [150], SCandroid [99] and IccTA [137] are mainly devoted to inter-component flow analysis. Although program analysis is mature and has been proven powerful in detecting vulnerabilities, its precision is limited by point-to analysis [122]. There are several dynamic analysis approaches to test the apps, which is either by instrumenting the Android OS [95, 218, 50] or based on virtualization [213, 161]. TaintDroid [95] and VetDroid [218] dynamically track the sensitive data flow through the OS and apps.

The precision of DROIDPF depends on activating all application behaviors. Its dependency-constrained event permutation approach aims to cover possible valid event sequences. There are a few studies also attempting to solve this challenge. SmartDroid [219] and ORBIT [216] combines static and dynamic analysis to trigger UI events. *SwiftHand* [84] uses an *abstract-refinement* approach to generate sequences of test inputs. AppIntent [217] and [161] use symbolic analysis to identify inputs to drive the analysis further. Dynodroid [151] uses an *observe-select-execute* approach which selects event inputs based on observed states to improve the coverage. A³E [47] explores app components and mimics user actions based on the strategy learned from the control flow graphs.

5.9 Summary

We present DROIDPF, which provides a framework for verifying Android apps against security properties based on targeted software model checking. We have made efforts to address the main problems in verifying Android apps, such as multiple entry points/event-driven execution, GUI testing and path explosion. DROIDPF shows that it is feasible to model check the software implemented in high-level language like Java and running on a complicated OS. We hope DROIDPF can inspire future research that brings the cutting-edge model checking techniques from the specifications to the implementations.

Chapter 6

Conclusion and Future Work

The work conducted in this thesis has made some scientific-technical contributions. This chapter briefly summarizes these contributions and presents possible future work.

6.1 Summary

This thesis aims to enhance the practical use of formal methods for analyzing secure system design and implementation by extending existing formalisms and combining program analysis techniques with formal methods. We have investigated three formal methods, i.e., symbolic model checking, specification extraction and software model checking, and focused on three security-critical areas, i.e., trusted computing, web authentication and mobile applications. In addition, we have proposed formalisms and techniques that assist designers, developers and security analysts on automating the formal analysis of the systems in these areas.

Observing that designing trusted platforms can be error-prone, we applied model checking to verify the security properties of trusted platforms on the design level. In particular, we proposed TRUSTFOUND, a formal foundation for model checking trusted platforms. It includes an interface TrCSP# for modeling the trusted platforms, a formal model of the key techniques used in trusted computing, as well as a broad spectrum of threat models. We implemented it as a framework in C# and CSP# based on the model checker PAT. Our analysis on a digital envelope protocol and a

cloud computing platform has detected two previous-unknown security flaws, thus demonstrating the power of model checking in detecting property violations. The digital envelope protocol, which has been proved correct by a previous work [91], is found by TRUSTFOUND to be subject to a reboot attack. This is because TRUSTFOUND embeds more comprehensive attack models. Due to the expressiveness of TRUSTFOUND and its comprehensive attack models, TRUSTFOUND can serve as a formal foundation for research on formal verification of trusted platforms.

In reality, establishing precise and complete formal models (that TRUSTFOUND and other similar studies require) of many secure systems is impossible due to factors such as lack of documentation and partial availability of implementation. In order to extract analyzable models for the use of formal methods in the area of web authentication, we proposed AUTHSCAN, an end-to-end platform to recover authentication protocol specifications from their implementations. We applied AUTHSCAN to several real-world web sites, including several using important SSO protocols like Facebook Connect Protocol, BrowserID and Windows Live Messenger Connect, and we successfully detected 7 security flows in their implementations. Our work is significant because it demonstrates the advantage and practicality of formal methods in violation detection even under the constraints of imprecise and incomplete models. Consequently, it is worth sacrificing soundness and completeness when formal methods are used for violation detection rather than for correctness proving.

Both TRUSTFOUND and AUTHSCAN analyzed the formal models translated from the checked systems. However, there still exists gap between these formal models and the implementations which the security eventually relies on. Therefore, our work also made an attempt of applying the formal methods directly on the system implementations. In particular, we targeted Android mobile platform, one of the security-critical areas, and presented DROIDPF which is a security verification framework for Android apps based on targeted software model checking. We have made efforts to address the main problems in verifying Android apps, such as multiple entry points/event-driven execution, GUI testing and path explosion. We evaluated DROIDPF using 70 apps including both benign and malicious ones, and DROIDPF detected nearly all (except those with implicit information flow) of the known vulnerabilities from the malware samples,

a previously unknown data leakage vulnerability from a third-party library embedded by benign apps, and two software bugs. DROIDPF achieves overwhelming precision and recall rate than static analysis approaches. This can be attributed to the particular features of our software model checking technique, such as the dynamic inference which enables DROIDPF to precisely detect property violations that can only be determined at runtime and the event permutation method which enables DROIDPF to address the Android-specific event-driven execution feature. Our work on DROIDPF suggests that it is feasible to model check the software that is implemented in advanced programming languages like Java and is running on a complicated OS.

6.2 Future Work

As it can be observed, there is yet a wide gap to be filled before formal methods are widely and extensively used in the real world. This section outlines possible future directions for research related to this thesis.

As trusted computing techniques are increasingly employed in newly emerging areas (e.g., Smart Grid [176, 175] and Web of Things [53]), TRUSTFOUND could be used as a foundation to analyzing the security properties of these platforms. Currently, our work of TRUSTFOUND mainly targets TPM version 1.2, whereas TCG recently released the specification of TPM 2.0 for public review. A necessary extension of TRUSTFOUND, therefore, is to embed models of TPM 2.0, such that TPM 2.0 based platforms such as the Direct Anonymous Attestation (DAA) protocol [79] can be supported. In addition, TPM is not the only hardware-based security feature used in trusted computing. Another possible avenue of future research is to consider other features which are significantly different from TPM, such as Intel's TXT and ARM TrustZone.

Although AUTHSCAN has been shown to be effective in identifying flaws in implementations of authentication systems, proving their correctness is impossible under the circumstance of partially-available implementations. Given the wide employment and security sensitivity of web authentication systems, future research on the verification of full implementations should be conducted.

6.2. FUTURE WORK

The work of DROIDPF targets the problem of exploring execution paths activated by events. However, as discussed in Section 5.7.4, in order to completely verify the apps, another critical problem is to explore the behaviors triggered by data inputs, such as the attack behaviors enabled on a specific date and our Geinimi case study which launches attacks based on received commands. As future work, we would introduce some advanced techniques, for example, symbolic execution which identifies and solves the conditions to trigger those behaviors [41, 217], to alleviate this problem.

Bibliography

- [1] apktool. <http://code.google.com/p/android-apktool/>. (Cited on page 98.)
- [2] BitLocker. <http://technet.microsoft.com/en-us/library/ee449438%28v=ws.10%29.aspx>. (Cited on page 18.)
- [3] BrowserID. <https://wiki.mozilla.org/Identity/BrowserID>. (Cited on pages 68 and 145.)
- [4] Cryptoverif: Cryptographic protocol verifier in the computational model. <http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/>. (Cited on page 14.)
- [5] CSur. <http://www.lsv.ens-cachan.fr/Software/csur/>. (Cited on page 15.)
- [6] CVE. <http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android+>. (Cited on page 84.)
- [7] dex2jar. <http://code.google.com/p/dex2jar/>. (Cited on page 98.)
- [8] Facebook Connect Authentication. <http://developers.facebook.com/docs/authentication/>. (Cited on page 69.)
- [9] FDR. <http://www.fsel.com/>. (Cited on page 13.)

BIBLIOGRAPHY

- [10] FS2PV: A Cryptographic-Protocol Verifier for F#. <http://research.microsoft.com/en-us/downloads/d54de3ef-085e-47f0-b7dc-8d56c858aba2/>. (Cited on page 15.)
- [11] GPS Share. <https://play.google.com/store/apps/details?id=com.kkinder.sharelocation&hl=en>. (Cited on page 99.)
- [12] Information Security Breaches Survey 2013. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/200455/bis-13-p184-2013-information-security-breaches-survey-technical-report.pdf. (Cited on page 2.)
- [13] Intel Trusted Execution Technology: White Paper. <http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html>. (Cited on page 18.)
- [14] Isabelle. <http://www.cl.cam.ac.uk/research/hvg/Isabelle/>. (Cited on page 14.)
- [15] TPM Reset Attack. <http://www.cs.dartmouth.edu/~pkilab/sparks/>. (Cited on pages 24, 28, and 35.)
- [16] Node.js v0.8.14 Manual & Documentation. <http://nodejs.org/api/crypto.html>. (Cited on pages 51, 61, and 70.)
- [17] OAuth Security Advisory: 2009.1. <http://oauth.net/advisories/2009-1/>. (Cited on page 3.)
- [18] Process lifecycle. <http://developer.android.com/reference/android/app/Activity.html#ProcessLifecycle>. (Cited on page 93.)
- [19] The AVISPA project homepage. <http://www.avispa-project.org/>. (Cited on pages 13, 44, and 52.)

BIBLIOGRAPHY

- [20] The Heartbleed Bug. <http://heartbleed.com/>. [Online; accessed 01-10-2014]. (Cited on pages 1 and 3.)
- [21] Trusted Boot. <http://sourceforge.net/projects/tboot/>. (Cited on page 18.)
- [22] Trusted Platform Module (TPM): Built-in Authentication. <http://www.trustedcomputinggroup.org/solutions/authentication>. (Cited on page 18.)
- [23] TrustZone. <http://www.arm.com/products/processors/technologies/trustzone.php>. (Cited on pages 5 and 18.)
- [24] Vulnerability Summary for CVE-2009-3555. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3555>. (Cited on page 3.)
- [25] Vulnerability Summary for CVE-2012-2110. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-2110>. (Cited on page 3.)
- [26] What is OpenID. <http://openid.net/get-an-openid/what-is-openid/>. (Cited on page 43.)
- [27] Windows Live Messenger Connect, Version 4.1. <http://msdn.microsoft.com/en-us/library/ff749458.aspx>. (Cited on page 69.)
- [28] Facebook Connect Used By 250 Million People Per Month. http://allfacebook.com/facebook-connect-used-by-250-million-people-per-month_b25501, December 8, 2010. (Cited on page 43.)
- [29] Security Vulnerability Allegedly Discovered in Dropbox Client. <http://news.softpedia.com/news/Design-Security-Flaw-Allegedly-Discovered-in-Dropbox-Client-194427.shtml>, April 11, 2011. (Cited on page 51.)
- [30] Mozilla jwcrypto. <https://github.com/mozilla/jwcrypto>, May 13, 2012. (Cited on pages 51 and 70.)
- [31] Wolfram alpha. <http://www.wolframalpha.com/>, May 13, 2012. (Cited on page 63.)

- [32] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):1--59, July 2007. (Cited on page 4.)
- [33] Martín Abadi and Cédric Fournet. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 104--115, 2001. (Cited on pages 11, 12, and 13.)
- [34] Martín Abadi and Neal Glew. Certified Email with a Light On-line Trusted Third Party: Design and Implementation. In *Proceedings of the 11th International Conference on World Wide Web (WWW)*, pages 387--395, 2002. (Cited on page 4.)
- [35] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The spi Calculus. *Information and Computation*, 148(1):1--70, 1999. (Cited on pages 4, 11, and 13.)
- [36] Martín Abadi and Mark R. Tuttle. A Semantics for A Logic of Authentication (Extended Abstract). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 201--216, 1991. (Cited on pages 12 and 44.)
- [37] King Ables and Mark D. Ryan. Escrowed Data and the Digital Envelope. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST)*, pages 246--256, 2010. (Cited on pages 20 and 22.)
- [38] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 331--340, 2011. (Cited on pages 3, 5, 15, and 75.)
- [39] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational Verification of C Protocol Implementations by Symbolic Execution. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 712--723, 2012. (Cited on pages 5 and 15.)

BIBLIOGRAPHY

- [40] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, pages 290--304, 2010. (Cited on pages 51 and 66.)
- [41] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012. (Cited on pages 105 and 110.)
- [42] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuellar, and Llanos Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 1--10, 2008. (Cited on pages 44 and 76.)
- [43] Alessandro Armando and Luca Compagna. SAT-based model-checking for security protocols analysis. *International Journal of Information Security*, 7(1):3--32, 2008. (Cited on page 3.)
- [44] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014. (Cited on pages 4, 78, 80, 82, 86, 89, 99, 100, 103, and 105.)
- [45] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (Se&P)*, 2002. (Cited on page 4.)
- [46] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal Verification of Security Protocol Implementations: a Survey. *Formal Aspects of Computing*, 26(1):99--123, 2014. (Cited on page 14.)

- [47] Tanzirul Azim and Iulian Neamtii. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 641--660, 2013. (Cited on pages 78 and 106.)
- [48] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic Search-Based Testing. In *International Conference on Automated Software Engineering (ASE)*, pages 53--62, 2011. (Cited on page 105.)
- [49] Michael Backes, Catalin Hritcu, and Matteo Maffei. Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-Calculus. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, pages 195--209, 2008. (Cited on page 4.)
- [50] Guangdong Bai, Tao Feng, Liang Gu, Yao Guo, and Xiangqun Chen. Context-Aware Usage Control for Android. In *6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 326--343, 2010. (Cited on page 106.)
- [51] Guangdong Bai, Jianan Hao, Jianliang Wu, Yang Liu, Zhenkai Liang, and Andrew Martin. TrustFound: Towards a Formal Foundation for Model Checking Trusted Computing Platforms. In *19th International Symposium on Formal Methods (FM)*, pages 110--126. 2014. (Cited on pages 9, 31, and 38.)
- [52] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, February 2013. (Cited on page 9.)
- [53] Guangdong Bai, Lin Yan, Liang Gu, Yao Guo, and Xiangqun Chen. Context-aware Usage Control for Web of Things. *Security and Communication Networks*, 7(12):2696--2712, 2014. (Cited on page 109.)

- [54] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Jun Sun, Yang Liu, Jin Song Dong, Heila van der Merwe, and Willem Visser. Verification of Android Applications against Security Properties Using Targeted Software Model Checking. *Under review*, 2015. (Cited on page 10.)
- [55] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203--213, 2001. (Cited on pages 17 and 78.)
- [56] Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffei. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 247--262, 2012. (Cited on pages 7, 45, 50, 51, 53, 66, and 75.)
- [57] David Basin and Cas Cremers. Know Your Enemy: Compromising Adversaries in Protocol Analysis. *ACM Transactions on Information and System Security (TISSEC)*, 17(2):7:1--7:31, November 2014. (Cited on page 5.)
- [58] David Basin, Cas Cremers, and Simon Meier. Provably Repairing the ISO/IEC 9798 Standard for Entity Authentication. *Journal of Computer Security*, 21(6):817--846, November 2013. (Cited on page 3.)
- [59] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium (USENIX Security)*, 2006. (Cited on page 19.)
- [60] Dirk Beyer and M. Erkan Keremoglu. CPACHECKER: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 184--190, 2011. (Cited on page 17.)

- [61] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically Verified Implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 459--468, 2008. (Cited on pages 4 and 15.)
- [62] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW)*, pages 82--96, 2001. (Cited on pages 7, 14, 44, 52, and 65.)
- [63] Bruno Blanchet. Computationally Sound Mechanized Proofs of Correspondence Assertions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, pages 97--111, 2007. (Cited on page 65.)
- [64] Bruno Blanchet. Security Protocol Verification: Symbolic and Computational Models. In *Principles of Security and Trust (POST)*, pages 3--29, 2012. (Cited on page 4.)
- [65] Bruno Blanchet and Avik Chaudhuri. Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, pages 417--431, 2008. (Cited on page 65.)
- [66] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 180--189, 2001. (Cited on page 2.)
- [67] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25(2-3):167--198, September 2004. (Cited on page 16.)

BIBLIOGRAPHY

- [68] Danilo Bruschi, Lorenzo Cavallaro, Andrea Lanzi, and Mattia Monga. Replay Attack in TCG Specification and Solution. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 127--137, 2005. (Cited on page 19.)
- [69] Enrico Budianto, Yaoqi Jia, Xinshu Dong, Prateek Saxena, and Zhenkai Liang. You Can't Be Me: Enabling Trusted Paths and User Sub-origins in Web Browsers. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 150--171. 2014. (Cited on page 44.)
- [70] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012. (Cited on page 77.)
- [71] Michael Burrows, Martín Abadi, and Roger M. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18--36, 1990. (Cited on pages 4, 12, 13, and 44.)
- [72] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring Pay-per-install: The Commoditization of Malware Distribution. In *Proceedings of the 20th USENIX Conference on Security (Usenix Security)*, 2011. (Cited on page 2.)
- [73] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 621--634, 2009. (Cited on page 4.)
- [74] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 317--329, 2007. (Cited on page 5.)

BIBLIOGRAPHY

- [75] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 1066--1071, 2011. (Cited on page 105.)
- [76] Matteo Ceccarello and Oksana Tkachuk. Automated Generation of Model Classes for Java PathFinder. *SIGSOFT Software Engineering Notes*, 39(1), 2014. (Cited on page 79.)
- [77] Sagar Chaki and Anupam Datta. ASPIER: An Automated Framework for Verifying Security Protocol Implementations. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF*, pages 172--185, 2009. (Cited on page 15.)
- [78] Kevin Zhijie Chen, Noah Johnson, Vijay D'Silva, ShuaiFu Dai, Kyle MacNamara, Tom Margrino, Edward XueJun Wu, Martin Rinard, and Dawn Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013. (Cited on pages 87 and 105.)
- [79] Liqun Chen and Jiangtao Li. Flexible and Scalable Digital Signatures in TPM 2.0. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS)*, pages 37--48, 2013. (Cited on page 109.)
- [80] Liqun Chen and Chris J. Mitchell. Parsing Ambiguities in Authentication and Key Establishment Protocols. *International Journal of Electronic Security and Digital Forensics*, 3(1):82--94, March 2010. (Cited on page 3.)
- [81] Liqun Chen and Mark Ryan. Offline Dictionary Attack on TCG TPM Weak Authorisation Data, and Solution. In *Future of Trust in Computing*, 2008. (Cited on page 19.)
- [82] Brian Chess. Improving Computer Security Using Extended Static Checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P)*, 2002. (Cited on page 4.)

- [83] Bodei Chiara, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and H. Riis Nielson. Automatic Validation of Protocol Narration. In *Proceedings. 16th IEEE Computer Security Foundations Workshop*, pages 126--140, 2003. (Cited on page 15.)
- [84] Wontae Choi, George Necula, and Koushik Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013. (Cited on page 106.)
- [85] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12nd International Conference on Computer Aided Verification (CAV)*, pages 154--169. 2000. (Cited on page 17.)
- [86] Andrew Cooper and Andrew Martin. Towards a Secure, Tamper-Proof Grid Platform. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2006. (Cited on pages 20, 40, and 41.)
- [87] Cas J. Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 414--418, 2008. (Cited on page 44.)
- [88] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta Execution for Efficient State-space Exploration of Object-oriented Programs. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 50--60, 2007. (Cited on page 78.)
- [89] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kaynar. A Logic of Secure Systems and Its Application to Trusted Computing. In *IEEE Symposium on Security and Privacy (S&P)*, 2009. (Cited on pages 13 and 25.)

BIBLIOGRAPHY

- [90] Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. A Formal Analysis of Authentication in the TPM. In *International Conference on Formal Aspects of Security and Trust (FAST)*, 2010. (Cited on page 31.)
- [91] Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. Formal Analysis of Protocols Based on TPM State Registers. In *IEEE Computer Security Foundations Symposium (CSF)*, 2011. (Cited on pages 22, 41, and 108.)
- [92] Giorgio Delzanno and Pierre Ganty. Automatic Verification of Time Sensitive Cryptographic Protocols. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 342--356, 2004. (Cited on page 146.)
- [93] Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198--208, 1983. (Cited on pages 11, 34, 45, 51, and 66.)
- [94] D.Recordon E.Hammer-Lahav and D.Hardt. The OAuth2.0 Authorization Protocol. 2011. IETF Internet Draft. (Cited on page 146.)
- [95] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010. (Cited on pages 4, 86, 103, 104, and 106.)
- [96] William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium (Usenix Security)*, August 2011. (Cited on page 4.)
- [97] Michael D. Ernst, René Just, Suzanne Millstein, Werner M. Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhorkar, Seungyeop Han, Paul

- Vines, and Edward X. Wu. Collaborative Verification of Information Flow for a High-assurance App Store. In *21st ACM Conference on Computer and Communications Security (CCS)*, 2014. (Cited on page 78.)
- [98] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 375--388, 2007. (Cited on page 2.)
- [99] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CS-TR-4991, November 2009. (Cited on pages 78, 105, and 106.)
- [100] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing (TRUST)*, 2012. (Cited on page 78.)
- [101] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, 1996. (Cited on page 16.)
- [102] Patrice Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174--186, 1997. (Cited on page 16.)
- [103] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012. (Cited on page 78.)

BIBLIOGRAPHY

- [104] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012. (Cited on page 77.)
- [105] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, pages 72--83, 1997. (Cited on page 17.)
- [106] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, Niels Provos, M. Zubair Rafique, Moheeb Abu Rajab, Christian Rossow, Kurt Thomas, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Manufacturing Compromise: The Emergence of Exploit-as-a-service. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 821--832, 2012. (Cited on page 2.)
- [107] Thomas Groß. Security Analysis of the SAML Single Sign-on Browser/Artifact Profile. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, 2003. (Cited on page 76.)
- [108] Trusted Computing Group. TPM Specification 1.2. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. (Cited on pages 5, 18, 19, and 20.)
- [109] Trusted Computing Group. TPM Specification 2.0. https://www.trustedcomputinggroup.org/resources/tpm_library_specification. (Cited on pages 5, 18, 19, and 20.)
- [110] Liang Gu, Guangdong Bai, Yao Guo, Xiangqun Chen, and Hong Mei. Security Model Oriented Attestation on Dynamically Reconfigurable Component-based Systems. *Journal of Network Computer Applications*, 35(3):974--981, May 2012. (Cited on page 19.)

- [111] Lin Gui, Jun Sun, Yang Liu, Yuan Jie Si, Jin Song Dong, and Xin Yu Wang. Combining Model Checking and Testing with an Application to Reliability Prediction and Distribution. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 101--111, 2013. (Cited on page 4.)
- [112] Sigrid Gürgens, Peter Ochsenschlager, and Carsten Rudolph. Role Based Specification and Security Analysis of Cryptographic Protocols using Asynchronous Product Automata. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, pages 473--479, 2002. (Cited on page 12.)
- [113] Sigrid Gürgens and Carsten Rudolph. Security Analysis of Efficient (Un-)Fair Non-repudiation Protocols. *Formal Aspects of Computing*, 17(3):260--276, 2005. (Cited on page 12.)
- [114] Sigrid Gürgens, Carsten Rudolph, Dirk Scheuermann, Marion Atts, and Rainer Plaga. Security Evaluation of Scenarios Based on the TCG's TPM Specification. In *Proceedings of the 12th European Conference on Research in Computer Security (ESORICS)*, pages 438--453, 2007. (Cited on pages 12, 19, and 41.)
- [115] Tihomir Gvero, Milos Gligoric, Steven Lauterburg, Marcelo d'Amorim, Darko Marinov, and Sarfraz Khurshid. State Extensions for Java Pathfinder. In *ACM/IEEE 30th International Conference on Software Engineering (ICSE)*, pages 863--866, 2008. (Cited on page 78.)
- [116] Steve Hanna, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The Emperor's New API: On the (In)Secure Usage of New Client Side Primitives. In *Proceedings of the 4th Web 2.0 Security and Privacy Workshop (W2SP)*, 2010. (Cited on pages 46, 75, and 146.)
- [117] Steffen M. Hansen, Jakob Skriver, and Hanne Riis Nielson. Using Static Analysis to Validate the SAML Single Sign-On Protocol. In *Workshop on Issues in the Theory of Security (WITS)*, pages 27--40, 2005. (Cited on page 76.)

BIBLIOGRAPHY

- [118] Jianan Hao, Yang Liu, Wentong Cai, Guangdong Bai, and Jun Sun. vTRUST: A Formal Modeling and Verification Framework for Virtualization Systems. In *15th International Conference on Formal Engineering Methods (ICFEM)*, pages 329--346. 2013. (Cited on page 9.)
- [119] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014. (Cited on page 78.)
- [120] James Hendricks and Leendert van Doorn. Secure Bootstrap is Not Enough: Shoring Up the Trusted Computing Base. In *ACM SIGOPS European Workshop*, 2004. (Cited on page 35.)
- [121] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 58--70, 2002. (Cited on page 17.)
- [122] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001. (Cited on page 106.)
- [123] C. A. R. Hoare. Communicating Sequential Processes. *Communications of ACM*, 21:666--677, August 1978. (Cited on pages 4 and 13.)
- [124] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 2013. (Cited on pages 88, 89, and 98.)
- [125] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23:279--295, May 1997. (Cited on page 13.)

BIBLIOGRAPHY

- [126] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014. (Cited on page 105.)
- [127] IBM. 1.5 million monitored cyber attacks in the United States in 2013. <http://www-935.ibm.com/services/us/en/it-services/security-services/data-breach/>. (Cited on page 2.)
- [128] Costin Ionescu. Obfuscating embedded malware on android. <http://www.symantec.com/connect/blogs/obfuscating-embedded-malware-android>. (Cited on page 84.)
- [129] C. Norris Ip and David L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, pages 41--75, 1996. (Cited on page 16.)
- [130] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4):21:1--21:54, October 2009. (Cited on page 5.)
- [131] Somorovsky Juraj, Mayer Andreas, Schwenk Jörg, Kampmann Marco, and Jensen Meiko. On Breaking SAML: Be Whoever You Want to Be. In *Proceedings of the 21st USENIX Conference on Security Symposium (Usenix Security)*, 2012. (Cited on page 44.)
- [132] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011. (Cited on page 105.)
- [133] Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *Proceedings of 16th USENIX Security Symposium (Usenix Security)*, 2007. (Cited on page 35.)
- [134] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th*

- USENIX Conference on Networked Systems Design Implementation (NSDI)*, 2007. (Cited on page 16.)
- [135] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing Trusted Platform Communication. In *ECRYPT Workshop on CRASH-Cryptographic Advances in Secure Hardware*, 2005. (Cited on pages 28 and 35.)
- [136] John Leyden. Patch Bash NOW: 'Shellshock' bug blasts OS X, Linux systems wide open. http://www.theregister.co.uk/2014/09/24/bash_shell_vuln/. (Cited on page 1.)
- [137] Li Li, Alexandre Bartel, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 International Conference on Software Engineering (ICSE)*, 2015. (Cited on page 106.)
- [138] Li Li, Hong Hu, Jun Sun, Yang Liu, and Jin Song Dong. Practical Analysis Framework for Software-Based Attestation Scheme. In *16th International Conference on Formal Engineering Methods (ICFEM)*, pages 284--299, 2014. (Cited on page 19.)
- [139] Li Li, Jun Sun, Yang Liu, and Jin Song Dong. TAuth: Verifying Timed Security Protocols. In *6th International Conference on Formal Engineering Methods (ICFEM)*, pages 300--315, 2014. (Cited on page 13.)
- [140] Li Li, Jun Sun, Yang Liu, and Jin Song Dong. Verifying Parameterized Timed Security Protocols. In *20th International Symposium on Formal Methods (FM)*, page 342--359, 2015. (Cited on page 13.)
- [141] Xiaolei Li, Guangdong Bai, Zhenkai Liang, and Heng Yin. A Software Environment for Confining Malicious Android Applications via Resource Virtualization. In *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 208--211, 2013. (Cited on page 10.)

- [142] Xiaolei Li, Guangdong Bai, Benjamin Thian, Zhenkai Liang, and Heng Yin. A Light-Weight Software Environment for Confining Android Malware. In *the 8th IEEE International Conference on Software Security and Reliability-Companion (SERE-C)*, pages 158-167, 2014. (Cited on page 10.)
- [143] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. Droid-Vault: A Trusted Data Vault for Android Devices. In *Proceedings of the 19th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2014. (Cited on page 10.)
- [144] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A Simple Method for Extracting Models for Protocol Code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 192--203, 2001. (Cited on pages 15 and 75.)
- [145] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and JinSong Dong. A Formal Semantics for Complete UML State Machines with Communications. In *Integrated Formal Methods (iFM)*, volume 7940, pages 331--346. 2013. (Cited on page 13.)
- [146] Shuang Liu, Yang Liu, Jun Sun, Manchun Zheng, Bimlesh Wadhwa, and Jin Song Dong. USMMC: A Self-contained Model Checker for UML State Machines. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 623--626, 2013. (Cited on page 13.)
- [147] Gavin Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Information Processing Letters*, 56:131--133, 1995. (Cited on page 1.)
- [148] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 147--166, 1996. (Cited on page 44.)

BIBLIOGRAPHY

- [149] Gavin Lowe. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations (CSFW)*, pages 31--43, 1997. (Cited on pages 51 and 65.)
- [150] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *19th ACM Conference on Computer and Communications Security (CCS)*, 2012. (Cited on pages 77, 78, 105, and 106.)
- [151] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013. (Cited on page 106.)
- [152] Kurt Mackie. Wave Outlines Windows 8 Mobile Device Management Alternative. <http://redmondmag.com/articles/2013/02/14/wave-outlines-windows-8-mobile-device-management-alternative.aspx>. (Cited on page 18.)
- [153] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an Execution Infrastructure for TCB Minimization. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys)*, 2008. (Cited on page 19.)
- [154] Gary McGraw and Bruce Potter. Software Security Testing. *IEEE Security and Privacy Journal*, 2(5):81--85, September 2004. (Cited on page 4.)
- [155] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, pages 123--136, 2006. (Cited on page 17.)
- [156] Peter Mehlitz, Oksana Tkachuk, and Mateusz Ujma. JPF-AWT: Model checking GUI applications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 584--587, 2011. (Cited on page 79.)

- [157] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, pages 696--701, 2013. (Cited on pages 14 and 34.)
- [158] M. Miculan and C. Urban. Formal Analysis of Facebook Connect Single Sign-On Authentication Protocol. In *37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 99--116, 2011. (Cited on pages 45, 72, 75, and 146.)
- [159] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32--44, December 1990. (Cited on page 4.)
- [160] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. (Cited on page 13.)
- [161] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. *ACM SIGSOFT Software Engineering Notes*, 37, 2012. (Cited on page 106.)
- [162] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated Analysis of Cryptographic Protocols Using Murphi. pages 141--151. IEEE Computer Society Press, 1997. (Cited on page 13.)
- [163] John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state Analysis of SSL 3.0. In *Proceedings of the 7th USENIX Security Symposium (Usenix Security)*, 1998. (Cited on page 3.)
- [164] Aybek Mukhamedov, AndrewD. Gordon, and Mark Ryan. Towards a Verified Reference Implementation of a Trusted Platform Module. In *Security Protocols XVII*, volume 7028 of *Lecture Notes in Computer Science*, pages 69--81. 2013. (Cited on page 33.)

BIBLIOGRAPHY

- [165] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI)*, 2004. (Cited on page 16.)
- [166] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, pages 75--88, 2002. (Cited on page 16.)
- [167] Cornelius Namiluko and Andrew Martin. An Abstract Model of a Trusted Platform. In *Proceedings of the Second International Conference on Trusted Systems (INTRUST)*, pages 47--66, 2011. (Cited on page 41.)
- [168] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993--999, December 1978. (Cited on pages 1 and 2.)
- [169] Peter Ochsenschläger, Jürgen Repp, Roland Rieke, and Ulrich Nitsche. The SH-Verification Tool --- Abstraction-Based Verification of Co-operating Systems. *Formal Aspects of Computing*, 10(4):381--404, 1998. (Cited on page 12.)
- [170] Damien Ocateau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013. (Cited on page 90.)
- [171] Nicholas O'Shea. Using Elyjah to Analyse Java Implementations of Cryptographic Protocols. In *Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, pages 221--226, 2008. (Cited on page 15.)
- [172] Nancy Owano. Danger on ice: Android info thaws in cold boot attack. <http://phys.org/news/2013-02-danger-ice-android-info-cold.html>. (Cited on page 105.)

BIBLIOGRAPHY

- [173] Paladion. Insecurebank. <http://www.paladion.net/downloadapp.html>. (Cited on page 99.)
- [174] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical State Continuity for Protected Modules. In *IEEE Symposium on Security and Privacy (S&P)*, 2011. (Cited on page 19.)
- [175] Andrew Paverd and Andrew Martin. Hardware Security for Device Authentication in the Smart Grid. In *1st Open EIT ICT Labs Workshop on Smart Grid Security (SmartGridSec)*, 2012. (Cited on page 109.)
- [176] Andrew J Paverd, Andrew P Martin, and Ian Brown. Privacy-Enhanced Bi-Directional Communication in the Smart Grid using Trusted Computing. In *5th IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2014. (Cited on page 109.)
- [177] Paul Pearce, Adrienne P. Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012. (Cited on page 77.)
- [178] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 179--180, 2010. (Cited on page 105.)
- [179] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *21st Network and Distributed System Security Symposium (NDSS)*, 2014. (Cited on page 89.)
- [180] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013. (Cited on page 78.)

- [181] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2013. (Cited on page 84.)
- [182] Arnab Roy, Anupam Datta, Ante Derek, JohnC. Mitchell, and Jean-Pierre Seifert. Secrecy analysis in protocol composition logic. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 197--213. Springer Berlin Heidelberg, 2007. (Cited on pages 4 and 13.)
- [183] RTNews. Bomb here, please: US troops may start using Android app to order airstrikes. <http://rt.com/news/us-troops-android-strikes-282>. (Cited on page 77.)
- [184] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stübke, Christian Wachsmann, and Marcel Winandy. TCG Inside?: A Note on TPM Specification Compliance. In *ACM workshop on Scalable Trusted Computing (STC)*, 2006. (Cited on pages 24 and 25.)
- [185] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium (Usenix Security)*, 2004. (Cited on pages 19 and 21.)
- [186] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, pages 513--528, 2010. (Cited on page 60.)
- [187] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010. (Cited on page 68.)

BIBLIOGRAPHY

- [188] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and Xiaofeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011. (Cited on page 105.)
- [189] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *2012 IEEE Computer Security Foundations Symposium (CSF)*, pages 78--94, 2012. (Cited on page 4.)
- [190] Benedikt Schmidt, Ralf Sasse, Cas Cremers, and David Basin. Automated Verification of Group Key Agreement Protocols. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*, pages 179--194, 2014. (Cited on page 4.)
- [191] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, 2010. (Cited on page 104.)
- [192] Evan R. Sparks. A Security Assessment of Trusted Platform Modules. Technical Report TR2007-597, Dartmouth College, Computer Science, 2007. (Cited on page 24.)
- [193] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A Robust Integrity Reporting Protocol for Remote Attestation. In *Workshop on Advances in Trusted Computing (WATC)*, 2006. (Cited on pages 24, 34, and 40.)
- [194] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. Integrating Specification and Programs for System Modeling and Verification. In *International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2009. (Cited on pages 20 and 25.)
- [195] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards Flexible Verification under Fairness. In *International Conference on Computer Aided Verification (CAV)*, 2009. (Cited on pages 13, 20, 38, and 52.)

BIBLIOGRAPHY

- [196] San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov. Systematically Breaking and Fixing OpenID Security: Formal Analysis, Semi-Automated Empirical Evaluation, and Practical Countermeasures. *Computers & Security*, 31:465--483, 2012. (Cited on pages 44 and 75.)
- [197] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008. (Cited on page 4.)
- [198] Oksana Tkachuk, Matthew Dwyer, and Corina Pasareanu. Automated Environment Generation for Software Model Checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 116--127, 2003. (Cited on page 79.)
- [199] E Tsyrlkevich and V Tsyrlkevich. Single Sign-On for the Internet: A Security Story. In *BlackHat*, July 2007. (Cited on page 75.)
- [200] Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. Generation of Library Models for Verification of Android Applications. In *Java Pathfinder Workshop*, 2014. (Cited on page 79.)
- [201] Heila van der Merwe, Brink van der Merwe, and Willem Visse. Verifying Android Applications Using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 37, 2012. (Cited on pages 79, 103, 104, and 105.)
- [202] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Execution and Property Specifications for JPF-Android. *ACM SIGSOFT Software Engineering Notes*, 39(1):1--5, 2014. (Cited on pages 104 and 105.)
- [203] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203--232, April 2003. (Cited on page 16.)

BIBLIOGRAPHY

- [204] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203--232, 2003. (Cited on pages 78, 79, and 85.)
- [205] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, pages 3--17, 2000. (Cited on page 4.)
- [206] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce (WOEC)*, pages 29--40, 1996. (Cited on page 44.)
- [207] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, pages 365--379, 2012. (Cited on pages 3, 4, 44, 46, 48, 64, 75, and 146.)
- [208] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, pages 497--512, May 2010. (Cited on page 51.)
- [209] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel Trusted Execution Technology. *Black Hat DC*, 2009. (Cited on page 19.)
- [210] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology. *Invisible Things Lab*, 2009. (Cited on page 19.)
- [211] Thomas Y. C. Woo and Simon S. Lam. A Semantic Model for Authentication Protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy (S&P)*, 1993. (Cited on pages 7, 11, 44, 51, 53, 55, and 65.)

BIBLIOGRAPHY

- [212] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013. (Cited on page 75.)
- [213] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX conference on Security symposium (USENIX Security)*, 2012. (Cited on page 106.)
- [214] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006. (Cited on page 16.)
- [215] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. (Cited on page 16.)
- [216] Wei Yang, Mukul R. Prasad, and Tao Xie. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 250--265, 2013. (Cited on page 106.)
- [217] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *20th ACM Conference on Computer and Communications Security (CCS)*, pages 1043--1054, 2013. (Cited on pages 87, 105, 106, and 110.)
- [218] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *20th ACM Conference on Computer and Communications Security (CCS)*, pages 611--622, 2013. (Cited on page 106.)

BIBLIOGRAPHY

- [219] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012. (Cited on page 106.)

- [220] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012. (Cited on pages 77, 80, 82, 86, and 99.)

- [221] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-stealing Smartphone Applications (on Android). In *Proceedings of the 4th international conference on Trust and Trustworthy Computing (TRUST)*, 2011. (Cited on page 77.)

Appendix A

TML to ProVerif Inputs

TML is an high-level abstract model language, which can be directly translated into applied pi-calculus. We do not present the formal semantics translation between these two languages, but intuitively explain the mapping between them. The applied pi-calculus model of the running example (Figure 4.1 and 4.3) is shown in Figure A.1.

Conversion. Most syntax and semantics can be directly mapped to applied pi-calculus. The initial conditions (initial knowledge of the participants) are represented with a set of global variables (line 17-21), where the terms initially unknown to Z is labeled as `private`, such as `k_IDP_s` (line 18), the private key of `IDP_S`. The cryptographic functions are translated into constructor (`fun`) and destructor (`reduc`) (line 6-15). The local protocols are represented with the processes (line 33-82), whose identifiers are represented with `i, j, r, p` (line 17) of `Host` type (line 1). For the action schema, the `Begin*` and `End*` are mapped to `event` (line 67 and 57); the `Send` and `Receive` are mapped to `out` and `in`; the `assoc` is represented with the `table` (line 22), and `NewAssoc` is mapped to `insert` a tuple into the `table` (line 34). However, one problem is that ProVerif does not scale as the number of tables increases. To solve this problem, we also can model the `assoc` using functions. In particular, `AUTHSCAN` uses the same modeling method as modeling symmetric cryptographic primitives. For example, the `assoc(i, authtoken)` in Figure 4.3 is modeled as `mysenc` at line 13-15. Specially, if this `assoc` happens to be a long-lived or

guessable token which needs to be added into Z's knowledge set, AUTHSCAN just casts the encryption key to the attacker (addattackerknow at line 77-78). The checking action is mapped to the matching action, for example, let(=M, =N) = checksign(P, spk(k_IDP_s)) (line 42) checks whether P is a signature over (M, N) using the private key K_IDP_s. The channel is slightly different from TML because ProVerif supports both public and private channels. AUTHSCAN translates HTTP into public channel (ch at line 23, 38 and 46) which is readable and writable to the attacker; HTTPS and cross-domain communication is translated as private channels (https at line 25 and 48, and browser at line 24 and 40).

For the syntax or semantics not supported by ProVerif, AUTHSCAN models them in alternative ways. For example, ProVerif does not support a writable but non-readable (for the attacker) or a readable but non-writable channel. When AUTHSCAN finds that the sender origin of postMessage is not checked (such as Step ② in Figure 4.1), which means this channel becomes an attacker-writable channel (but remains unreadable), it turns the browser channel writable by adding an input before out messages to browser, as shown at line 38-40. Conversely, if it finds that the channel is readable, it adds an out after in message from the channel. Finally, after we fixing all the vulnerabilities, ProVerif reports that the protocol is verified.

Detected vulnerabilities. ProVerif detects three attacks in this model. First, it reports that the attacker can derive the token using the key k_i_j_com cast to his knowledge set (line 77-78). After "fixing" this flaw (Here fixing means correcting the flaw in the model instead of in the implementation) as shown at line 74-78, it reports a replay attack where the attacker can obtain the token from line 46, and then replay it to line 54. After "fixing" this flaw using HTTPS to replace HTTP as shown at line 48 and 55, ProVerif reports the MITM attack shown in Section 4.2.1. The attacker replaces mynext at line 38 and finally gets the token from line 63.

```

1 type Host.
2 type key. (*symetric key*)
3 type spkey. (*public key*)
4 type sskey. (*private key*)
5
6 (* Shared key encryption *)
7 fun senc(bitstring, key):bitstring.
8 reduc forall x:bitstring,y:key;sdec(senc(x,y),y)=x.
```

```

9 (* Signatures *)
10 fun spk(sskey):spkey.
11 fun sign(bitstring, sskey):bitstring.
12 reduc forall x:bitstring,y:sskey; checksign(sign(x,y), spk(y)) = x.
13 (*fun*)
14 fun mysenc(Host, key):bitstring.
15 reduc forall x:Host,y:key;mysdec(mysenc(x,y),y) = x.
16
17 free i, j, r, p:Host.
18 free k_IDP_s:sskey [private].
19 free k_i_j_com:key [private].
20 free sp:bitstring.
21 free sessionID, CSRFToken:bitstring[private].
22 table sp_table(Host, bitstring).
23 channel ch.
24 free browser:channel [private].
25 free https:channel [private].
26
27 event BeginInit(Host).
28 event EndResponse(Host).
29
30 query x:Host, y:Host; inj-event(EndResponse(x)) ==> inj-event(BeginInit(y)).
31 query attacker(mysenc(i, k_i_j_com)).
32
33 let SP_C = (*i*)
34   insert sp_table(j, sp);
35   (*****
36     3. Fix postmessage flaw
37     *****
38     (*in(ch, (j:Host,sp:bitstring,mynext:channel)); *)
39     new mynext:channel;
40     out(browser, ((j,sp),mynext));(*Step 1*)
41     in(mynext, (M:Host,N:bitstring,P:bitstring)); (*Step 4*)
42     let(=M, =N) = checksign(P, spk(k_IDP_s)) in
43     (*****
44       2. Fix HTTP replay attack
45       *****
46       (*out(ch, (M,N))*
47       in(ch, (M:bitstring, N:bitstring));
48       out(https, (M,N))(*step 5*).
49
50 let SP_S = (*j*)
51   (*****
52     2. Fix HTTP replay attack
53     *****
54     (*in(ch, (M:Host,token:bitstring))*
55     in(https, (M:Host,token:bitstring));(*step5*)
56     let(=M) = mysdec(token, k_i_j_com) in
57     event EndResponse(i).
58
59 let IDP_C = (*r*)
60 in(browser, (X:bitstring,Y:channel));(*step 1*)
61 out(https, (X,sessionID,CSRFToken));(*step2*)
62 in(https, (M:Host,N:bitstring,P:bitstring));(*step 3*)
63 out(Y, (M,N,P)). (*step 4*)
64

```

```

65 let IDP_S = (*p*)
66 in(https, (X:bitstring, =sessionID, =CSRFToken)); (*step 2*)
67 event BeginInit(j);
68 let (M:Host, Mdomain:bitstring) = X in
69 get sp_table(=M, =Mdomain) in
70 let token = mysenc(i, k_i_j_com) in
71 let idpsign = sign((i, token), k_IDP_s) in
72 out(https, (i, token, idpsign)).(*step 3*)
73
74 (*****
75     1. Fix guessable token
76     *****)
77 let addattackerknow =
78     (*out(ch, k_i_j_com)*)
79     new padding:bitstring.
80
81 process
82     (!SP_C|!SP_S|!IDP_C|!IDP_S|!addattackerknow)

```

Figure A.1: Applied pi-calculus Model of the Running Example

Appendix B

Protocol Extraction

B.1 Extracting BrowserID Protocol

In this section, we detail the process on analyzing `myfavoritebeer.org` to demonstrate how AUTHSCAN extracts model from the implementation. As shown in Figure B.1, the traces captured by AUTHSCAN are listed in the first two columns, and the corresponding TML statements inferred are placed in the third column. From message (2), AUTHSCAN infers the HTTP parameter `csrf` as a nonce. AUTHSCAN also associates user name (`USER`) and password (`PWD`) to represent that they should be matching. From message (4), through white box analysis, AUTHSCAN infers that `spkUser` and `spkUser-1` are an asymmetric key pair generated by function `generateKeypair()`. In message (5), AUTHSCAN figures out that the HTTP parameter `cert` is encoded as a JSON Web Token (JWT) with each segment separated with "." and encoded with Base64 encoding (as described in Section 4.4.2). When applying the signature verification algorithm RSA over one of the segment (the brute-force search as discussed in Section 4.4.2), AUTHSCAN finds that it is a signature by `IDP_S` over four data elements occurring previously: $\{USER, spkUser, p, expire\}_{k_{IDP_S}^{-1}}$. Similarly, in message (6), AUTHSCAN identifies that function `sign()` is used to generate signature $\{j, expire1\}_{spkUser^{-1}}$ and this signature is concatenated with IDP's signature (i.e., `cert`) with function `bundle()`. Afterwards, this concatenation is sent by invoking function `Window.postMessage()`.

#	Input		TML
	HTTP Messages	Javascript code snippet	
(2)	POST https://login.persona.org/wsapi/authenticate_user Host: login.persona.org "email":"alicesstotester@gmail.com", "pass":"alice", "csrf":"UaZWfqrQmYwemitM1U8nUw=="	NONE	Initial Conditions r has $csrf \wedge p$ has $csrf$ IDP_C(r) NewAssoc($\{r,p\}$, $assoc(USER, PWD)$) Send(p , $\{assoc(USER, PWD), csrf\}$) IDP_S(p) Receive(r , $\{assoc(M, N), csrf\}$)
(4)	POST https://login.persona.org/wsapi/cert_key Host: login.persona.org "email":"alicesstotester@gmail.com", "pubkey":{"algorithm":"DS",".....6233397a"}", "csrf":"UaZWfqrQmYwemitM1U8nUw=="	syncEmailKeyPair:function(...){..., d.withContext(function(){ a.generateKeyPair({ algorithm:"DS", keySize:c.KEY_LENGTH, ...})})	IDP_C(r) NewKeyPair($spkUser$, $spkUser^{-1}$) Send(p , $USER$, $spkUser$, $csrf$) IDP_S(p) Receive(r , M , Y , $csrf$)
(5)	GET https://login.persona.org/wsapi/cert_key Host: login.persona.org "cert":"eyJhbGciOiJSUzU1Ni9.eyJwdW....SfqAt5..."	NONE	IDP_C(r) Receive(p , X) IDP_S(p) NewNonce($expire$) Send(r , $\{M, Y, p, expire\}_{k-1}^{IDP_S}$)
(6)	NONE	assertion.sign($\{$,{audience:c,expiresAt: j ,g, function(d,g){ k=a.cert.bundle([f.cert],g,...)}) b.window.postMessage(JSON.stringify(a), b.origin)	IDP_C(i) NewNonce($expireI$) Send(j , X , $\{j, expireI\}_{spkUser^{-1}}$) SP_C(j) Receive(i , R)

Figure B.1: The HTTP Trace of BrowserID and the Corresponding TML Statements

B.2 Inferred Protocols

Figure B.2 demonstrates the protocols inferred using AUTHSCAN; the inferred models are simplified for readability.

B.3 Precision of Inferred Protocols

We investigate the precision of our inferred protocol, which is possible for two of our case studies, to available documentation and manually-crafted specifications. We find that our protocols are fairly precise, subject to our qualitative analysis.

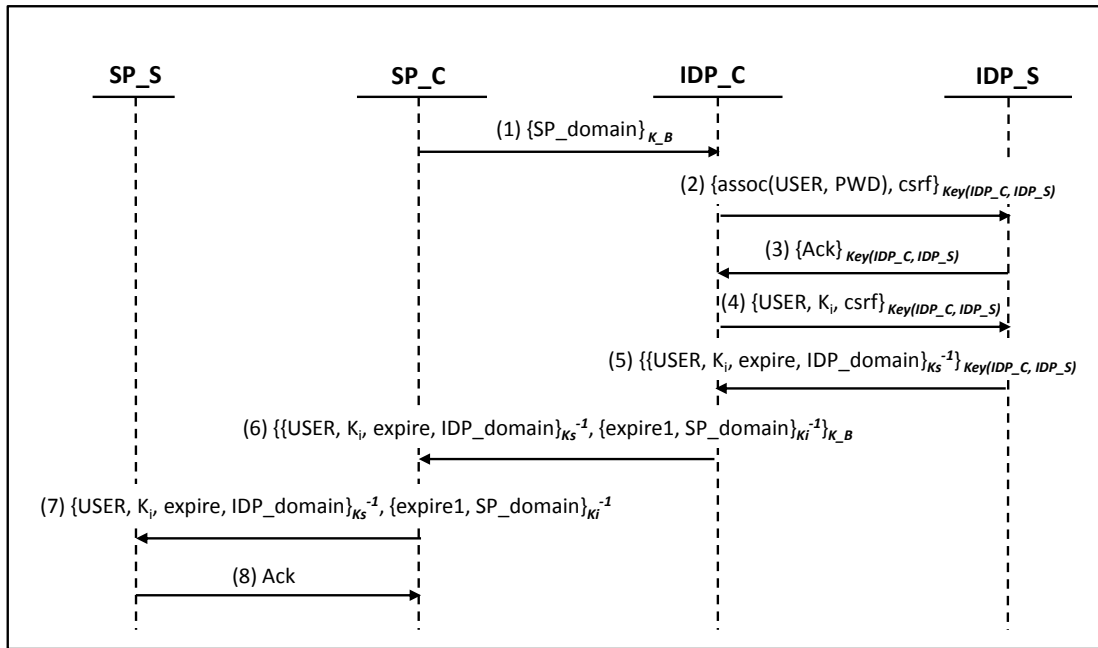
BrowserID Precision. We compare our inferred specification to the documented description of the protocol online [3]. Our inferred protocol matches closely to the description in the documentation. In some cases, it reveals useful information that is unspecified in the documentation. For instance, the documentation says that, the IDP returns a signed structure containing expiration time in the Step 5 of Figure B.2-(a)), but documentation does not precisely specify the duration of the "expiration time". AUTHSCAN finds that the duration is large enough to permit replay attacks

that are longer than 726 seconds. This intermediate result is useful for further analysis, such as verification on time sensitive protocols [92].

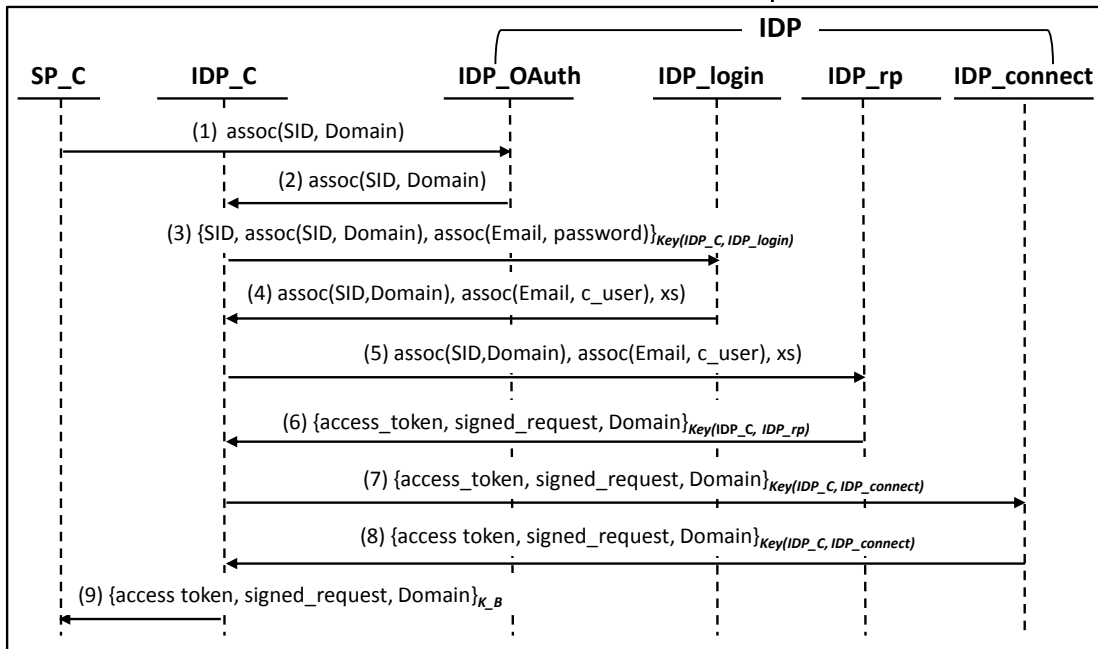
We find the protocol to match the documentation exactly (subject to our manual interpretation), except for one additional difference. The document states that the SPs are allowed to send the signed data to BrowserID for verification in the specification rather than local verification. Since this message is sent between SP and IDP servers rather than been relayed in the browser, it is not represented in our inferred specification.

Facebook Connect Precision. Facebook Connect originates from OAuth 2.0 authorization protocol [94]. In EbayClassified case, our inferred protocol consists of 11 rounds and 65 parameters (including cookies and GET/POST parameters), comparing to 7 rounds and 11 parameters in the specification. The extra rounds and parameters, which shows our inferred protocol is more precise, may be vulnerable to the protocol and have been analyzed by AUTHSCAN. Furthermore, compared to recent work which manually extracts the Facebook Connect protocol, our model has defined more precisely the terms exchanged in the protocol [158]. Our inferred specification is also more detailed than the prior work of Hanna et al. [116]. Finally, we find that our Facebook Connect model is different from the description in Wang et al.'s recent work [207]--- this is because their work considers the Flash implementation whereas we analyze the JavaScript-based implementation which works in today's web browsers by default.

B.3. PRECISION OF INFERRED PROTOCOLS



(a) The Sequence Diagram of BrowserID



(b) The Sequence Diagram of Facebook Connect

Figure B.2: The Sequence Diagrams Inferred from Implementations of BrowserID and Facebook Connect