Università
Ca'Foscari
Venezia

CORSO DI DOTTORATO DI RICERCA
IN INFORMATICA

CICLO XXXI

TESI DI RICERCA

Disciplined Techniques for the Analysis and Protection of Security-Critical Systems

SSD: INF/01

COORDINATORE DEL DOTTORATO
Prof. Riccardo FOCARDI

SUPERVISORE
Prof. Riccardo FOCARDI

DOTTORANDO
Mauro TEMPESTA
Matricola 827400

Abstract

In the last years most of our daily activities have moved to the digital world, including sensitive operations related to health data management and financial processes. Security flaws in the systems running these critical operations may seriously impact on our society, ranging from breaches of citizens privacy to severe economical damages.

Important aspects that must be taken into account when reasoning on the security of critical systems comprise the security of the networks where they are hosted and that of the web applications running on these systems. Additional threats are posed by the improper use of cryptography that may allow unintended disclosure of confidential data. In this thesis we introduce a set of disciplined techniques for the analysis and protection of security-critical systems tackling these important aspects.

Regarding network security, we propose a technique to decompile firewall policies into abstract specifications that provide a high-level description of the firewall behaviour. Additionally, we face the problem of cross-compiling policies to different firewall systems. About web security, we survey the techniques proposed in the literature and by web standards to counter the most common attacks against web sessions and we carefully evaluate them in terms of usability, compatibility with existing websites and ease of deployment. Furthermore, we propose a client-side approach to fortify the security of Web protocols by monitoring their execution inside the browser. For what concerns cryptography, we provide a detailed analysis of Java keystores, encrypted files used by Java applications to securely store cryptographic keys. We report novel attacks and weaknesses found in the most widespread keystore implementations and discuss the fixes implemented by Java developers after our responsible disclosure.

Acknowledgements

I want to express my most sincere gratitude to my supervisor, Prof. Riccardo Focardi, for his guidance during this three years-long adventure. I am particularly grateful to Dr. Stefano Calzavara for many fruitful discussions that deeply contributed in improving my scientific background. I also want to thank the reviewers of this thesis, Prof. Joshua D. Guttman and Prof. Andrei Sabelfeld, for accepting this task and for the time spent reading this manuscript.

During these years I had the pleasure to work with wonderful people both from Ca' Foscari and different institutions. I am very proud of the achievements we have accomplished together and I want to thank them for the sleepless nights while struggling to meet a deadline. Among them, I am especially grateful to Prof. Matteo Maffei for offering me the chance to join his research group at TU Wien.

Besides being excellent coauthors, I want to thank Francesco Palmarini, Lorenzo Veronese and Marco Squarcina as part of the *c00kies* hacking team, along with Andrea Baesso, Andrea Possemato, Claudio Bozzato, Francesco Benvenuto, Francesco Cagnin, Leonardo Veronese and Marco Gasparini. Together we managed to achieve outstanding results all around the world, including the qualification to the long-dreamed DEF CON CTF finals in Las Vegas.

I consider myself very lucky for having so many friends sharing funny moments with me during my PhD. In particular, I am grateful to Andrea, Diletta, Gianpietro and Martina for the awesome time spent together and the countless beers during our habitual meetings *"dallo Zio"*.

Finally, I want to thank my parents Armando and Ivana and my sister Claudia for their unconditional support throughout my entire life. I couldn't have managed to complete this journey if it hadn't been for you. Last thing, a huge kiss to my lovely niece Lucrezia!

Mauro Tempesta
Wien, December 2018

Contents

Preface

The work presented in this thesis is based on some research papers written during my Ph.D. studies in Computer Science at Università Ca' Foscari Venezia from September 2015 to August 2018.

Chapter 1 is the result of a long term project conducted with Chiara Bodei, Pierpaolo Degano, Riccardo Focardi, Letterio Galletta and Lorenzo Veronese. It is based on two research papers presented in April 2018 respectively at the *3rd IEEE European Symposium on Security and Privacy* [31] and at the *7th International Conference on Principles of Security and Trust* [33].

Chapter 2 is a joint work with Stefano Calzavara, Riccardo Focardi and Marco Squarcina that has been published in April 2017 in the *ACM Computing Surveys* journal [42]. Chapter 3 is the outcome of a collaboration started with Matteo Maffei and Clara Schneidewind during my visits at CISPA (Saarbrücken, Germany) and TU Wien (Vienna, Austria). This collaboration, which also included Stefano Calzavara, Riccardo Focardi and Marco Squarcina, resulted in a paper that I have presented in August 2018 at the *27th Usenix Security Symposium* [43].

Finally, Chapter 4 is a joint work with Riccardo Focardi, Francesco Palmarini, Marco Squarcina and Graham Steel that was presented in February 2018 at the *25th Network and Distributed Systems Security Symposium* [65].

Introduction

During the past decades our society has experienced a pervasive digitalization process that influenced several aspects of our life such as the way we communicate, work or spend our free time. This phenomenon has also interested delicate topics such as healthcare data management and business processes. Security flaws in the systems running these critical operations may have a serious impact on our society, ranging from breaches of the privacy of citizens to severe economical damages.

There are several important aspects that must be taken into account when we reason on the security of such critical systems. Virtually all systems are connected to computer networks since they need to communicate with other machines to implement their functionalities. These networks are usually connected to the Internet and firewalls are employed to protect internal hosts from the other machines. In this context, it is critical to ensure the correctness of the firewall configuration to guarantee that only the intended services are exposed to the Internet in order to prevent unauthorized accesses that may harm the security of the entire network.

The Web is the primary way in which digital services are made available to their users. The complexity of web applications has grown exponentially during the years and they are now comparable to native desktop applications in terms of functionalities and interactivity. To complicate things further, modern websites interact with each other to implement collaborative scenarios such as e-payments and single sign-on. The rapid evolution of the Web hindered the enforcement of security on web applications given the increasingly wider attack surface and the need to implement defense mechanisms that are compatible with most of the existing websites. Understanding how defenses can be fruitfully adopted and combined to protect web applications and developing new mechanisms to strengthen the aforementioned collaborative scenarios are important challenges that must be faced to improve the security of the Web ecosystem.

Critical systems typically manipulate sensitive information, therefore they employ cryptography to protect the confidentiality or the integrity of such data. The most delicate aspect of cryptography is key management which regards, among other things, how cryptographic keys are securely stored. Leaking a key voids any benefit of using cryptography given that an attacker may use the key to read and modify all the encrypted data or forge cryptographic signatures. Keys are usually stored either in dedicated hardware facilities like smartcards and HSMs or inside keystores, *i.e.*, files where keys are stored in an encrypted form. Differently from hardware solutions, the security of existing keystore implementations has never been assessed in the literature. Filling this gap is of crucial importance to understand the actual level of security provided by systems relying on

keystores as mechanism for key storage.

In this thesis we tackle all the challenges highlighted so far, either by exploiting formal methods or using techniques that are nevertheless inspired by rigorous and disciplined principles.

Summary of the Contributions

Regarding the research field related to firewalls, we contribute by proposing a transcompilation pipeline to assist a network administrator in analysing, maintaining and porting firewall configurations. The peculiarity of our approach is the independence from any specific firewall system which is achieved thanks to `IFCL`, our intermediate language for firewalls configuration. We show how to encode real configurations in `IFCL` and how to transform them into first order logic predicates that determine which packets are accepted by a policy. The model of this predicate is used to construct a table representing the *meaning* of the configuration at an abstract level. This abstract specification can either be analysed by the administrator to reason on the correctness of the underlying specification or provided as input to our algorithm that compiles the configuration for a different system. We have implemented most of our pipeline in FWS and we show how it can be fruitfully used by an administrator to check several properties of interest including reachability, *e.g.*, see which hosts in a subnet can communicate with hosts in other networks, verifying whether two policies are equivalent and eventually spot the differences.

In the context of web security, the contributions are twofold. First we provide a systematic overview of the attacks harming the security of web sessions and we carefully analyse the defense mechanisms proposed in the literature and by web standards along four different axes: protection, usability, compatibility with existing websites and ease of deployment. Additionally, we distill a set of guidelines that should be taken into account when designing new security solutions. The second contribution is a novel client-side security mechanism aimed at strengthening the security of web protocols. First we identify the fundamental challenges in securing web protocols, namely the enforcement of confidentiality and integrity guarantees on the contents of messages as well as the intended protocol flow. Next we introduce WPSE, a browser-side security monitor (implemented as a Google Chrome extension) designed to undertake the challenges we have identified. We prove that WPSE is expressive enough to protect web applications from a wide range of protocol implementation bugs and web attacks and we discuss concrete examples of attacks which can be prevented by WPSE on OAuth 2.0 and SAML 2.0, including a novel attack on the Google implementation of SAML 2.0. Finally, we experimentally assess the effectiveness of our solution by testing WPSE on 90 websites that use OAuth 2.0 to implement single sign-on.

About key storage, we define a general threat model for password-protected keystores and we distill a set of significant security properties and consequent rules that any secure keystore should adhere to. We perform an in-depth analysis of seven keystores from the Oracle JDK and Bouncy Castle, a widespread cryptographic library, highlighting

undocumented details about their implementations. We present new critical attacks and weaknesses in the analysed keystores which have been assigned three CVE IDs [123, 124, 125]. We estimate the speed-up in terms of cracking time due to bad cryptographic implementations with respect to the most resistant keystore and to NIST recommendations. Finally, we discuss the fixes implemented by vendors after our responsible disclosure.

Structure of the Thesis

The thesis is structured as follows:

- Chapter 1 discusses our transcompiling pipeline for the analysis, maintenance and porting of firewall configurations. Proofs of the theorems are in Appendix A;
- Chapter 2 surveys the most widespread attacks against web sessions and the corresponding defenses from the literature and web standards;
- Chapter 3 introduces WPSE, our novel browser-side mechanism for the protection of web protocols;
- Chapter 4 presents our analysis on Java keystore implementations.

Chapter 1

Analysis, Maintenance and Cross-Compilation of Firewall Policies

1.1 Introduction

Firewalls are one of the standard mechanisms for protecting computer networks but, as any other security mechanism, they become useless when incorrectly configured. Configuring and maintaining them is very difficult also for expert system administrators since configurations typically contain hundreds of rules and it is often hard to figure out the overall firewall behaviour. Moreover, firewall rules influence each other, *e.g.*, a rule shadows others making them redundant or preventing them to be triggered. When a network is protected by more than one firewall the situation complicates further since the configurations of the various firewalls need to be kept coherent: enabling or disabling a connection typically requires to update the configurations of all the firewalls that are potentially traversed by the considered connection.

Firewall policy languages are varied and usually rather complex, accounting for low-level system and network details and supporting non trivial control flow constructs, such as jumps and gotos. Configurations enforce policies in a way that typically depends on how packets are processed by the network stack of the operating system running on the firewall machine. Further difficulties for network administrators come from Network Address Translation (NAT), a pervasive component of IPv4 networking that operates while packets traverse the firewall. In IPv4, NAT is indispensable for performing port redirection and translating addresses, *e.g.*, when a single public address is used for a whole private network.

Over the past few years, there has been a growing interest in high level languages for *programming* the network as a whole. The Software Defined Network (SDN) paradigm decouples network control and forwarding functions, by abstracting the underlying infrastructure from applications and network services [66]. A unified, high level paradigm to configure networks and firewalls is appealing and might, in principle, make firewall configuration simpler and less error-prone. However, SDN requires a suitable infrastructure and, even if it seems to be spreading fast, it will take time before "old" technology

is dismissed in favor of it. In the years to come, we still have to face a variety of firewall configuration languages, including the ones running on a variety of legacy devices. More often than nowadays, network administrators will have to face the porting of legacy firewall configurations to fit this new paradigm.

In this work, we propose a transcompilation pipeline to assist a network administrator in *analysing*, *maintaining* and *porting* firewall configurations. Our proposal is independent of the specific target firewall system, so administrators are not required to have a deep knowledge about the internals of the firewall systems and of their languages, rather they can focus on the policy to enforce. Our transcompiling pipeline is composed of the following stages:

1. decompile the policy in the source language into an intermediate language;
2. extract the meaning of the policy as a set of non overlapping declarative rules describing the accepted packets and their translations in logical terms;
3. compile the declarative rules into the target language.

The first stage relies on `IFCL`, a generic intermediate language that incorporates all the typical features of firewall languages such as NAT, jumps, invocations to rulesets and stateful packet filtering. It has been designed so to make it relatively easy to encode in it real firewall configuration languages. Interestingly, `IFCL` unveils the *bipartite structure* common to real firewall languages: the first component consists of the rulesets determining the destiny of packets, the second one specifies the steps needed to elaborate packets and the order in which rulesets are applied. While the format of the rules and the actions are largely shared by the available firewall languages, apart from minor syntactic differences, the second component is peculiar to each operating system and each firewall tool and, intuitively, summarizes the specific low-level behaviour of a particular system.

The second stage transforms a real firewall configuration encoded in `IFCL` into an abstract specification that represents the set of allowed connections. This version abstracts from low-level details, *e.g.*, the control flow, duplicated or shadowed rules. In this way it exposes the *meaning* of the configuration and makes it easier for system administrators to check whether or not the intended security policy is correctly enforced. Moreover, by comparing two abstract specifications, an administrator can detect the differences between configurations and check that updates have the desired effect on the firewall behaviour.

The last stage supports cross-platform recompilation into different firewall systems. More precisely, we transform the abstract version back into an `IFCL` configuration for the target firewall system and from that we compile the actual configuration.

As a proof of concept, we have developed FWS which is available for download at [32]. Our tool currently implements the first two steps of the transcompiling pipeline, while support for the third stage is currently under active development. We support the most used firewall tools in Linux and Unix [134, 179, 149] and, partially, Cisco IOS routers. New firewall systems can be easily added to our tool by providing a plug-in

for the front-end to our intermediate language. Indeed, once a configuration has been translated into IFCL, the analysis can be performed independently of the initial firewall language and system.

1.1.1 Contributions

Our contributions can be summarized as follows.

1. We present FWS, a language-independent tool that translates a real firewall configuration into an abstract specification. This specification is a table that declaratively represents the set of accepted packets with their possible translations. Roughly, each row corresponds to a configuration rule. However, the resulting table is more readable than the standard rulesets, because its rows are *independent* from each other, while in real firewall configurations the meaning of a rule depends on the others and on the firewall control flow. As a consequence, the table contains no anomalies, *e.g.*, shadowing or redundancies.

2. The query language of FWS allows administrators to analyse the behaviour of a configuration, when translated in its abstract specification. In particular, one can check reachability properties, *e.g.*, which subnets, hosts and ports are reachable from other hosts and subnets, as well as policy equivalence, implication and difference. Furthermore, it helps an administrator in maintaining a configuration by observing the effects of adding, deleting or modifying some rules, *e.g.*, the administrator can compute the overall contribution of a given rule in terms of accepted or dropped packets.

3. We introduce the new language IFCL that decomposes a firewall configuration into rulesets and a *control diagram*. Rules determine the destiny of packets and their form is common to most of the existing languages. The control diagram is specific of each firewall system and it describes the flow of the packet through the network stack of the operating system. Our language is the crucial component that makes FWS language-independent, but it is also of independent interest as a generic firewall configuration language. A relevant aspect of IFCL is its formal semantics (*cf.* Section 1.4.2), while real languages usually have none. The formal semantics enables us to prove the correctness of our transcompiling pipeline, *i.e.*, that all its transformations preserve the meaning of the original firewall policy.

4. We show how FWS transforms a configuration from the intermediate language into a first order logic predicate that determines which are the packets accepted by the configuration in hand, with all the possible NAT translations. The model of this predicate is then used to build the table representing the abstract specification of the configuration. The logical characterization is, by itself, insightful and provides the query engine of FWS with a formal, algorithmic basis.

5. We propose a generic compilation algorithm that distributes the rules of the abstract specification on the relevant points of the firewall where it decides the destiny of packets.

6. We present results of experiments performed with FWS on real firewall configurations. The tool synthesizes whole complex policies, analyses them and answers to queries in a matter of minutes. In some cases for specific queries or simple policies, it works almost in real-time. Policy implication and equivalence are also checked very efficiently.

1.1.2 Structure of the Chapter

In Section 1.2, we briefly survey `iptables`, `ipfw` and `pf`, the most widespread real firewall systems which are used in the examples of this chapter. Section 1.3 illustrates FWS at work on a small yet realistic case study. In particular we focus on how network administrators can exploit FWS to check firewall configurations for host reachability, policy equivalence and difference. Furthermore, we show an example of transcompiling a configuration. In Section 1.4 we present our intermediate language, its formal semantics and the encodings of `iptables`, `ipfw` and `pf`. The logical characterization of all packets accepted by a firewall with their possible translations is in Section 1.5. It also describes the internals of FWS and the algorithm for synthesizing abstract specifications and for policy analysis. Finally, it proves the correctness of first two stages of our pipeline. Section 1.6 describes the last stage, in particular the compilation strategy for generating target configuration. It also proves the source and target configurations equivalent, so establishing the correctness of the whole pipeline. In Section 1.7, we apply our tool on various real firewall policies to assess its effectiveness and scalability. Section 1.8 compares our work with other proposals in the literature.

1.2 Background

Usually, system administrators classify networks into security domains. Through firewalls they monitor the traffic and enforce a predetermined set of access control policies among the various hosts and subnetworks (*packet filtering*). System administrators can also use a firewall to connect a network with private IPs to other (public IP) networks or to the Internet and to perform connection redirections through NAT.

Firewalls are implemented either as proprietary, special devices, or as software tools running on general purpose operating systems. Independently of their actual implementations, they are usually characterized by a set of rules that determine which packets reach the different subnetworks and hosts and how they are modified or translated.

Below, we briefly review the most widespread firewall systems in Linux and Unix: `iptables` [134], `ipfw` [179] and `pf` [149].

1.2.1 `iptables`

It is the default packet filtering tool in Linux distributions and operates on top of Netfilter, the standard framework for packet processing implemented in the Linux kernel [161].

The basic notions of `iptables` are *tables* and *chains*. Intuitively, a table is a collection of ordered lists of policy rules called chains. The most commonly used tables are:

- `filter` for packet filtering;
- `nat` for network address translation;
- `mangle` for packet alteration.

There are five built-in chains that are inspected at specific moments of the packet life cycle [181]:

- `PreRouting` when the packet reaches the host;
- `Forward` when the packet is routed through the host;
- `PostRouting` when the packet is about to leave the host;
- `Input` when the packet is routed to the host;
- `Output` when the packet is generated by the host.

Tables do not necessarily contain all the predefined chains and further user-defined chains can be added.

Each rule specifies a condition and a target. If the packet matches the condition then it is processed according to the specified target, which can be a built-in target or a user-defined chain. The most commonly used targets are:

- `ACCEPT` and `DROP` to accept and discard packets;
- `RETURN` to stop examining the current chain and resume the processing of a previous chain;
- `DNAT` to perform destination NAT, *i.e.*, translating the destination IP address or port of the packet;
- `SNAT` to perform source NAT, *i.e.*, translating the source IP address or port;
- `MARK` to tag a packet with a numeric value that can be used to identify the packet in the condition of the following rules (possibly in different chains).

When the target is a user-defined chain, two "jumping" modes are available: *call* and *goto*. The difference between the two arises when a `RETURN` is executed or the end of the chain is reached: the evaluation resumes from the rule following the last call. Built-in chains have a user-configurable default policy (`ACCEPT` or `DROP`): if the evaluation reaches the end of a built-in chain without matches, its default policy is applied.

1.2.2 `ipfw`

It is the standard firewall for FreeBSD [179]. A configuration consists of a single ruleset that is inspected twice, when the packet enters the firewall and before it exits. It is possible to specify whether a rule should be applied only in one of the two directions using the keywords `in` and `out`.

Similarly to `iptables`, rules are inspected sequentially until the first match occurs and the corresponding action is taken. The packet is dropped if there is no matching rule. The most common actions in `ipfw` are the following:

- `allow` and `deny` are used to accept and reject packets;
- `nat` applies destination NAT to incoming packets and source NAT to outgoing packets;
- `check-state` accepts packets that belong to established connections;
- `skipto`, `call` and `return` allow to alter the sequential order of inspection of the rules in the ruleset.

Packet marking is supported also by `ipfw`: if a rule containing the `tag` keyword is applied, the packet is marked with the specified identifier and then processed according to the rule's action.

1.2.3 `pf`

It is the standard firewall of OpenBSD [149] and is included in macOS since version 10.7. Each rule consists of a predicate which is used to select packets and an action that specifies how to process the packets satisfying the predicate. The most frequently used actions are `pass` and `block` to accept and discard packets, `rdr` and `nat` to perform destination and source NAT. Packet marking works as in `ipfw`.

Differently from other systems, the action taken on a packet is determined by the *last matched rule*, unless otherwise specified by using the `quick` keyword. `pf` has a single ruleset that is inspected both when the packet enters and exits the firewall. When a packet enters the firewall, DNAT rules are examined first and filtering is performed after the address translation. Similarly when a packet leaves the firewall: first its source address is translated by the relevant SNAT rules, and then the resulting packet is possibly filtered. Packets belonging to established connections are accepted by default, thus bypassing the filters.

1.3 The Pipeline at Work

This section introduces a a small yet realistic scenario through which we exemplify the three stages of our pipeline, namely:

1. the decompilation of a firewall configuration into the intermediate language `IFCL`;

2. the synthesis of the declarative specification;

3. the compilation in the target language.

In addition, we illustrate how our tool FWS supports system administrators in reasoning on and managing a firewall configuration, spotting mistakes and modifying the configuration so to fix them. In particular, we check the following behavioural properties:

- *reachability*: verify whether a certain address is reachable from another one, possibly through NAT;
- *policy implication and equivalence*: check if the packets accepted by one configuration are at least/exactly the same accepted by another configuration;
- *policy difference*: see what packets are accepted by a configuration and denied by another one. This feature is particularly useful when maintaining a policy to check how updates affect the firewall behaviour, because one can see which packets are accepted and which are filtered out when a specific rule is added.
- *related rules*: understand which configuration rules affect the processing of the packets identified by a user-provided query.

Finally we show an example of porting a policy produced by a proof-of-concept extension of FWS based on the theory presented in Section 1.6.

1.3.1 Network Structure and Policy Requirements

As running example, consider the network shown in Figure 1.1. The internal network consists of two parts:

- network 10.0.1.0/24 contains servers and production machines, including a HTTPS server (10.0.1.15) that runs the company website on port 443;
- network 10.0.2.0/24 contains the machines of the employees, including the computer of the system administrator (10.0.2.15) where a SSH service is running on port 22.

The firewall has three network interfaces: `eth0` connected to 10.0.1.0/24 with IP 10.0.1.1, `eth1` connected to 10.0.2.0/24 with IP 10.0.2.1 and `ext` connected to the Internet with public IP 23.1.8.15.

We want to enforce the following requirements on the traffic:

1. internal networks can freely communicate;

2. connections to the public IP on ports 443 and 22 are translated (DNAT) to 10.0.1.15 and 10.0.2.15, respectively. This condition permits external hosts to access the website by connecting to the public IP address 23.1.8.15 at port 443, that is redirected to the corresponding internal host (similarly for the SSH server);

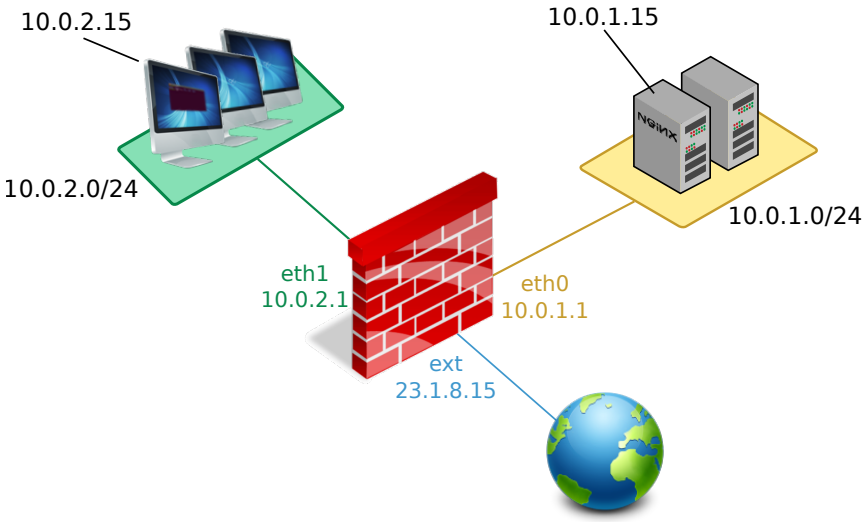


FIGURE 1.1: Network of our case study.

3. connections from the internal hosts to the Internet are allowed only towards HTTP and HTTPS web servers, *i.e.*, with destination ports 80 and 443;

4. source addresses of connections from the internal hosts to the Internet are translated (SNAT) to the external IP address of the firewall. This allows hosts with private IPs to access the Internet;

5. the firewall can connect to any other host.

1.3.2 Compliant Configuration in `iptables`

Here we provide a configuration in `iptables` for the case study of Section 1.3.1 and we use FWS to decompile and analyse it to check whether it complies with the desired requirements.

Configuring the Firewall with `iptables`

Figure 1.2 shows the policy for our example in the standard `iptables-save` format used to store `iptables` rules in a configuration file.

The first sequence of commands delimited by `*nat` and `COMMIT` keywords sets the default policies of all `nat` chains to `ACCEPT`, inserts into the `nat PREROUTING` chain the rules for redirecting the incoming connections to the internal servers (requirement 2) and adds to the `nat POSTROUTING` chain the rule for SNAT (requirement 4).

The subsequent block from lines `*filter` to `COMMIT` specifies a default `DROP` policy for the `INPUT` and `FORWARD` chains and a default `ACCEPT` policy for the `OUTPUT` chain, letting the firewall communicate with any host (requirement 5). The first two filtering rules allow the packets belonging to connections flagged as established to go through and towards the firewall, *i.e.*, whenever a new connection is allowed any further packet belonging to the same connection will also be allowed. This is not explicitly required by the policy

```
### NAT rules ###
*nat
# Default policy ACCEPT in nat chains
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]

# Requirement 2: Redirect incoming SSH and HTTPS connections to hosts
# 10.0.2.15 and 10.0.1.15 (DNAT)
-A PREROUTING -p tcp -d 23.1.8.15 --dport  22 -j DNAT --to 10.0.2.15
-A PREROUTING -p tcp -d 23.1.8.15 --dport 443 -j DNAT --to 10.0.1.15
# Requirement 4: Connections towards the Internet exit with source
# address 23.1.8.15 (SNAT)
-A POSTROUTING -s 10.0.0.0/16 ! -d 10.0.0.0/16 -j SNAT --to 23.1.8.15

COMMIT

### Filtering rules ###
*filter
# Default ACCEPT in output (Requirement 5), DROP in the other chains
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

# Allow established packets
-A FORWARD -m state --state ESTABLISHED -j ACCEPT
-A INPUT   -m state --state ESTABLISHED -j ACCEPT
# Requirement 1: Allow arbitrary traffic between internal networks
-A FORWARD -s 10.0.0.0/16 -d 10.0.0.0/16 -j ACCEPT
# Requirement 3: Allow HTTP/HTTPS outgoing traffic
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 80  -j ACCEPT
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 443 -j ACCEPT
# Requirement 2: Allow SSH/HTTPS incoming traffic to the corresponding
# machines in the internal networks
-A FORWARD -p tcp -d 10.0.2.15 --dport  22 -j ACCEPT
-A FORWARD -p tcp -d 10.0.1.15 --dport 443 -j ACCEPT

COMMIT
```

FIGURE 1.2: Example policy of Section 1.3.1 in `iptables`.

but is necessary to ensure functionality of connection-oriented protocols. Then we have `ACCEPT` rules corresponding to the requirements 1, 3 and 2, respectively. Notice that requirement 2 has also rules in the `nat` table above.

Decompiling and Analyzing the Configuration

We now use FWS to check that the configuration of Figure 1.2 meets the requirements 1–5 of Section 1.3.1. First, we ask the tool the following query:

```
((srcIp == 10.0.1.0/24 && dstIp == 10.0.2.0/24) ||
(srcIp == 10.0.2.0/24 && dstIp == 10.0.1.0/24)) && state == NEW
```

where `srcIp`, `dstIp` represent the fields for source and destination address of the IP packet entering the firewall interfaces, and `state` tells if a connection is new or established. The query checks whether hosts with `srcIp` 10.0.1.0/24 can start new connections

TABLE 1.1: Results of FWS when checking the `iptables` configuration of Figure 1.2.

(A) Requirement 1.

Src IP	Src Port	Dst IP	Dst Port	Protocol	State
10.0.2.0/24	*	10.0.1.0/24	*	*	NEW
10.0.1.0/24	*	10.0.2.0/24	*	*	NEW

(B) Requirement 2.

Src IP	Src Port	DNAT IP	DNAT Port	Dst IP	Dst Port	Protocol	State
*	*	10.0.1.15	-	23.1.8.15	443	tcp	NEW
*	*	10.0.2.15	-	23.1.8.15	22	tcp	NEW

(C) Requirements 3 and 4.

Src IP	Src Port	SNAT IP	SNAT Port	Dst IP	Dst Port	Protocol	State
10.0.0.0/16	*	23.1.8.15	-	* \ {10.0.0.0/16}	80 443	tcp	NEW

(D) Requirement 5.

Src IP	Src Port	Dst IP	Dst Port	Protocol	State
23.1.8.15	*	*	*	*	NEW

towards those with `dstIp` 10.0.2.0/24, or viceversa, as stated by requirement 1. The operator == constrains a variable to be equal to a value or inside a certain interval; the operators `&&` and `||` stand for logical conjunction and disjunction. The output we obtain from the tool is in Table 1.1a, where * denotes any value. The table contains all of the allowed connections matching the query, confirming that requirement 1 is satisfied.

We now check that external hosts can access the web and the SSH servers only by connecting to the firewall IP address 23.1.8.15 at ports 443 and 22 respectively (requirement 2). To do that, we ask which packets can reach the hosts with addresses 10.0.1.15 and 10.0.2.15:

```
(dstIp' == 10.0.1.15 || dstIp' == 10.0.2.15) && state == NEW
```

The variable `dstIp'` represents the destination address of the packet possibly translated by a NAT: in the queries, variables with quotes, *e.g.*, `dstIp'` above, denote constraints applied to packets exiting a firewall interface; variables without primes instead constrain packets entering the firewall. The result in Table 1.1b confirms that requirement 2 is satisfied: indeed, the servers 10.0.1.15 and 10.0.2.15 are reachable from any host connecting to the public IP address of the firewall on ports 443 and 22 only.

The next query checks requirements 3 and 4 together:

```
srcIp == 10.0.0.0/16 && not(dstIp' == 10.0.0.0/16) && state == NEW
```

Intuitively, the query asks for the new connections that are allowed from an internal source to an external destination. The answer in Table 1.1c shows that both the requirements are met. Indeed, the notation `* \ {10.0.0.0/16}` represents all destination addresses except those in the subnet 10.0.0.0/16. Finally, by checking requirement 5 with the query

```
# NAT setup. The first line defines the source NAT for packets leaving the
# firewall through the interface ext (Requirement 4), the other two lines
# specify to perform DNAT on packets arriving to the ports 22 and 443 of
# the firewall (Requirement 2)
ipfw -q nat 1 config if ext unreg_only reset \
                     redirect_port tcp 10.0.1.15:443 443 \
                     redirect_port tcp 10.0.2.15:22  22

# Allow established packets
ipfw -q add 01 check-state
# Requirement 1: Allow arbitrary traffic between internal networks
ipfw -q add 10 allow all from 10.0.0.0/16 to 10.0.0.0/16
# Requirement 2: Apply DNAT on packets arriving to the external interface
# of the firewall
ipfw -q add 20 nat 1 ip from any to 23.1.8.15 in recv ext
# Requirement 2: Allow SSH/HTTPS incoming traffic to the corresponding
# hosts and responses from these services
ipfw -q add 21 allow tcp from any to 10.0.1.15 443
ipfw -q add 22 skipto 70 tcp from 10.0.1.15 443 to any
ipfw -q add 23 allow tcp from any to 10.0.2.15 22
ipfw -q add 24 skipto 70 tcp from 10.0.2.15 22 to any
# Requirements 3 and 4: Allow HTTP/HTTPS outgoing traffic
ipfw -q add 30 skipto 70 tcp from 10.0.0.0/16 to any 80,443 \
  setup keep-state
# Requirement 5: Allow arbitrary outgoing traffic by the firewall
ipfw -q add 50 allow ip from me to any setup keep-state
# Drop all the other packets
ipfw -q add 60 deny all from any to any
# Requirement 4: Apply SNAT to outgoing connnections
ipfw -q add 70 nat 1 ip from any to not 10.0.0.0/16 out
ipfw -q add 71 allow ip from any to any
```

FIGURE 1.3: Policy in `ipfw`.

```
srcIp == 23.1.8.15 && state == NEW
```

we obtain the output of Table 1.1d showing that the firewall can reach any host.

We can thus conclude that the configuration in Figure 1.2 is correct with respect to the requirements.

1.3.3 Non-Compliant Configuration in `ipfw`

Figure 1.3 implements the example policy in `ipfw`. On purpose, we introduce subtle but realistic differences with respect to the one in `iptables` and we show how FWS spots them in a clear and concise way.

Configuring the Firewall with `ipfw`

The first command declares NAT rules, named `nat 1`, that will be activated by the following rules. Notice that the next commands have numbers (after the `add` keyword) that can be used for jumps, as we will see below. We refer to those numbers in the description. Command `01` accepts all the packets that belong to already established connections (`check-state`). As for `iptables` this is important to ensure functionality of connection-oriented protocols. Command `10` enables traffic between internal networks

TABLE 1.2: Results of FWS when checking the `ipfw` configuration of Figure 1.3.

(A) Requirement 1.

Src IP	Src Port	Dst IP	Dst Port	Protocol	State
10.0.2.0/24	*	10.0.1.0/24	*	*	NEW
10.0.1.0/24	*	10.0.2.0/24	*	*	NEW

(B) Requirement 2.

Src IP	Src Port	DNAT IP	DNAT Port	Dst IP	Dst Port	Protocol	State

* \ {
10.0.1.0-10.0.2.255
127.0.0.0/8
}
	*	10.0.2.15	-	23.1.8.15	22	tcp	NEW

* \ {
10.0.1.0-10.0.2.255
127.0.0.0/8
}
	*	10.0.1.15	-	23.1.8.15	443	tcp	NEW

(C) Requirements 3 and 4.

Src IP	Src Port	SNAT IP	SNAT Port	Dst IP	Dst Port	Protocol	State
10.0.2.15	22	23.1.8.15	-	* \ {10.0.0.0/16}	*	tcp	NEW
10.0.1.15	443	23.1.8.15	-	* \ {10.0.0.0/16}	*	tcp	NEW
10.0.0.0/16	*	23.1.8.15	-	* \ {10.0.0.0/16}	80 443	tcp	NEW

(D) Requirement 5.

Src IP	Src Port	Dst IP	Dst Port	Protocol	State
23.1.8.15	*	*	*	*	NEW

(requirement 1). Command 20 applies `nat 1` to the packets received via the interface `ext`, implementing the destination NAT of requirement 2. The actual connections to hosts 10.0.1.15 and 10.0.2.15, respectively on ports 443 and 22, are enabled by the commands 21–24. Notice that packets coming from those hosts are handled by jumping (`skipto 70`) to the last but one line, which applies `nat 1`, translating the source address to 23.1.8.15 (SNAT). Then packets are accepted by command 71. Next line (command 30) implements the requirements 3 and 4 similarly to previous rules, *i.e.*, by jumping to 70 which enforces the SNAT on outgoing connections. Option `keep-state` is the counterpart of `check-state`: the connection is saved in the firewall state so that packets belonging to the same connection will be allowed through the firewall by rule 01. Rule 50 allows the firewall host to communicate to any host. Finally, command 60 rejects any packet that does not match any previous rule, implementing a default deny policy.

Decompiling and Analyzing the Configuration

We now use FWS to check if the configuration of Figure 1.3 meets the requirements 1–5 of Section 1.3.1. We perform exactly the same queries we did for `iptables` in Section 1.3.2. In fact, one of the main advantages of our approach is the independence of the analysis from the particular firewall system in use.

Queries for the requirements 1 and 5 give exactly the same results we got for `iptables` (*cf.* Table 1.2a, 1.2d and 1.1a, 1.1d). For requirement 2, instead, we get an interesting

difference. In the ipfw configuration we obtain that hosts 10.0.1.15 and 10.0.2.15 cannot be reached by the internal network and by the firewall host via DNAT (*cf.* Table 1.2b). This is because, in the ipfw configuration, rule 20 is applied only for packets coming from the interface ext, *i.e.*, packets received from the Internet. In fact, requirement 2 could be interpreted in this stricter way by a system administrator, as hosts 10.0.1.15 and 10.0.2.15 are anyway reachable from internal hosts even without DNAT. FWS is able to spot this subtle difference in the two configurations. To make the ipfw configuration behave as the iptables one (for requirement 2), it is enough to remove recv ext from rule 20.

In checking the requirements 3 and 4, FWS reports that hosts 10.0.1.15 and 10.0.2.15 can start new connections from source ports 443 and 22 (respectively) to any other host. This is due to rules 22 and 24 that enable the two hosts to answer connections done through the DNAT and constitutes an alternative way to make connection-oriented protocols work without exploiting the check-state command. In principle, this should be considered non-compliant with requirement 3 as new connections from 443 and 22 from the two hosts will access any port and not just 80 and 443, as requested. Again, FWS spots this difference in the policy. This error can be rectified by removing rules 22 and 24 from the policy and adding the keep-state keyword to the rules 21 and 23.

Interestingly, FWS can compute the equivalence of configurations written for different firewall systems. In this particular case, FWS outputs that the fixed ipfw configuration and the iptables one are equivalent, relatively to the five requirements.

1.3.4 Maintaining Firewall Configurations

In this section, we show how FWS can be used to perform maintenance of the iptables policy presented in Section 1.3.2.

The company has added a new machine to the subnet 10.0.1.0/24, which has been assigned the IP address 10.0.1.22. Differently from the other hosts of the network, we want to allow Internet access (with SNAT) to this machine only over HTTPS. The other requirements on the traffic should be preserved. For this purpose, we can add the following rule to the FORWARD chain, which drops connections to port 80 from host 10.0.1.22:

```
-A FORWARD -s 10.0.1.22 -p tcp --dport 80 -j DROP
```

However, we must be careful about the position where to place this rule in order to fulfill the desired requirement and avoid to unintentionally block legal traffic.

If we place the new rule at the end of the FORWARD chain, the *policy equivalence* analysis implemented in FWS reports that the new policy is equivalent to the previous version. We can use the *related rules* analysis to understand which rules are relevant for processing HTTP packets. We find out that the output of the analysis includes only the following filtering rule from the FORWARD chain:

```
-A FORWARD -s 10.0.0.0/16 -p tcp --dport 80 -j ACCEPT
```

The above rule accepts all the HTTP traffic from the internal networks and is evaluated before the new DROP rule. Hence, our new rule should be placed before this one.

TABLE 1.3: Maintenance of the `iptables` configuration.

(A) Policy differences after the wrong update.

+/-	Src IP	Src Port	Dst IP	Dst Port	Protocol	State
+	10.0.0.0/16 \ { 10.0.1.22 }	*	10.0.0.0/16	80	tcp	NEW
-	10.0.0.0/16	*	10.0.0.0/16	80	tcp	NEW

+/-	Src IP	Src Port	SNAT IP	SNAT Port	Dst IP	Dst Port	Protocol	State
+	10.0.0.0/16 \ { 10.0.1.22 }	*	23.1.8.15	-	* \ { 10.0.0.0/16 }	80	tcp	NEW
-	10.0.0.0/16	*	23.1.8.15	-	* \ { 10.0.0.0/16 }	80	tcp	NEW

(B) Policy differences after the correct update.

+/-	Src IP	Src Port	SNAT IP	SNAT Port	Dst IP	Dst Port	Protocol	State
+	10.0.0.0/16 \ { 10.0.1.22 }	*	23.1.8.15	-	* \ { 10.0.0.0/16 }	80	tcp	NEW
-	10.0.0.0/16	*	23.1.8.15	-	* \ { 10.0.0.0/16 }	80	tcp	NEW

If we add the new rule before those of the other requirements, *e.g.*, after the rules that allow packets of incoming connections, FWS reports that the policy is not equivalent to the previous one. We can check the impact of our changes by running the *policy difference* analysis projected over the HTTP traffic:

```
protocol == tcp && dstPort == 80
```

The output of the analysis is shown in Table 1.3a. The first column is + or - for lines that appear in the synthesis or disappear after the updates, respectively. We can see that host 10.0.1.22 is now unable to connect to the Internet, as desired (second table of Table 1.3a). However, our update also prevents communications over HTTP with other machines on the internal networks, thus violating requirement 1 (first table of Table 1.3a).

The correct place where to add the new rule is between the rule for requirement 1 and those for requirement 3. In this way we allow HTTP traffic from 10.0.1.22 only to the internal networks. If we repeat the analysis, we see that now the only difference is just in the HTTP traffic towards the Internet, as desired (*cf.* Table 1.3b).

1.3.5 Transcompiling a Configuration

Suppose that the system administrator has to migrate the firewall configuration of Figure 1.2 from `iptables` to `pf`. Performing this porting by hand is complex and error prone because the administrator has to write the `pf` configuration from scratch and test its equivalence with respect to the original one. Furthermore, this requires a deep understanding of the policy meaning, as well as of both `iptables` and `pf` and their configuration languages. We apply below the stages of our pipeline to solve this problem, guaranteeing by construction that the firewall semantics is preserved.

TABLE 1.4: Synthesis of the `iptables` configuration in Figure 1.2.

Src IP	Src Port	SNAT IP	DNAT IP	Dst IP	Dst Port	Protocol	State
10.0.0.0/16	*	-	-	10.0.0.0/16	*	*	NEW
10.0.0.0/16	*	23.1.8.15	-	* \ { 10.0.0.0/16 }	443 80	tcp	NEW
*	*	-	10.0.2.15	23.1.8.15	22	tcp	NEW
*	*	-	10.0.1.15	23.1.8.15	443	tcp	NEW
23.1.8.15	*	-	-	*	*	*	NEW
*	*	-	-	10.0.2.15	22	tcp	NEW
*	*	-	-	10.0.1.15	443	tcp	NEW
*	*	*	*	*	*	*	ESTABLISHED

First we extract the meaning of the `iptables` policy in Table 1.4, corresponding to stage 1 and 2 of our pipeline. The output is almost as expected but for lines 6–7 which are reporting that the HTTPS server and SSH server can be reached bypassing DNAT. Technically this is allowed by the firewall policy but it cannot occur in practice since the two servers have a private IP address that cannot be routed on the Internet.

According to stage 3, we compile the refactored policy in `pf`, in two steps. First, the rows are translated in a sequence of `IFCL` rules, possibly optimized, and then compiled in `pf`. The result is in Figure 1.4 and was computed with a proof-of-concept extension of FWS based on the theory presented in Section 1.6.

```
rdr proto tcp from any to 23.1.8.15 port 443 tag T1 -> 10.0.1.15
rdr proto tcp from any to 23.1.8.15 port 22 tag T2 -> 10.0.2.15
no rdr proto tcp from 10.0.0.0/16 to {!10.0.0.0/16} port {80, 443} tag T3
nat tagged T3 -> 23.1.8.15

block all
pass from 10.0.0.0/16 to 10.0.0.0/16
pass from me to any
pass proto tcp from any to 10.0.2.15 port 22
pass proto tcp from any to 10.0.1.15 port 443
pass tagged T1
pass tagged T2
pass tagged T3
```

FIGURE 1.4: The policy of Section 1.3.2 ported to `pf`.

1.4 IFCL: The Intermediate Firewall Configuration Language

Our intermediate firewall configuration language (`IFCL`) is parametric with respect to the notion of state and the steps performed to elaborate packets. For generality, we do not detail the format of network packets. In the following we only use $sa(p)$ and $da(p)$ to denote the source and destination addresses of a given packet p; additionally, $tag(p)$ returns the tag m associated with p. An address a consists of an IP address $ip(a)$ and possibly a port $port(a)$. An *address range* n is a pair consisting of a set of IP addresses and a set of ports, denoted $ip(n)$:$port(n)$. An address a is in the range n (written $a \in n$) if $ip(a) \in ip(n)$ and $port(a) \in port(n)$ when $port(a)$ is defined, *e.g.*, for ICMP packets we only check if the IP address is in the range.

Firewalls modify packets, *e.g.*, through network address translations. We write $p[da \mapsto a]$ and $p[sa \mapsto a]$ to denote a packet identical to p, except for the destination address da and source address sa, which is equal to a, respectively. Similarly, $p[tag \mapsto m]$ denotes the packet with a modified tag m.

Here we consider *stateful* firewalls that keep track of the state s of network connections and use this information to process a packet. Any existing network connection can be described by several protocol-specific properties, *e.g.*, source and destination addresses or ports, and by the translations to apply. In this way, filtering and translation decisions are not only based on administrator-defined rules, but also on the information built by previous packets belonging to the same connection. We omit a precise definition of a state, but we assume that it tracks at least the source and destination ranges, NAT operations and the state of the connection, *i.e.*, established or not. When receiving a packet p one may check whether it matches the state s or not. We left unspecified the match between a packet and the state because it depends on the actual shape of the state. When the match succeeds, we write $p \vdash_s \alpha$, where α describes the actions to be carried on p; otherwise we write $p \nvdash_s$.

A firewall rule is made of two parts: a predicate ϕ expressing criteria over packets, and an action t, called *target*, defining the "destiny" of matching packets. Here we consider a core set of actions included in most of the real firewalls. These actions not only determine whether or not a packet passes across the firewall, but they also control the flow in which the rules are applied. The list of supported action follows:

`ACCEPT`	a packet passes
`DROP`	a packet is discarded
`CALL(R)`	invoke the ruleset R
`GOTO(R)`	jump to the ruleset R
`RETURN`	exit from the current ruleset
`NAT`(n_d, n_s)	network translation
`MARK`(m)	marking with tag m
`CHECK-STATE`(X)	examine the state

The targets `CALL(_)` and `RETURN` implement a procedure-like behaviour; `GOTO(_)` is similar to unconditional jumps. In the `NAT` action n_d and n_s are address ranges used to translate the destination and source address of a packet, respectively; in the following we use the symbol $\star$ to denote an identity translation, *e.g.*, $n : \star$ means that the address is translated according to n, whereas the port is kept unchanged. The `MARK` action marks a packet with a tag m. The argument $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ of the `CHECK-STATE` action denotes the fields of the packets that are rewritten according to the information from the state. More precisely, $\rightarrow$ rewrites the destination address, $\leftarrow$ the source address and $\leftrightarrow$ both. A rule is formally defined as follows:

Definition 1 (Firewall rule)**.** *A firewall rule r is a pair (ϕ, t) where ϕ is a logical formula over a packet and t is the target action of the rule.*

A packet p matches a rule r with target t whenever ϕ holds.

Definition 2 (Rule match). *Given a rule $r = (\phi, t)$ we say that p matches r with target t, denoted $p \models_r t$, iff $\phi(p)$. We write $p \not\models_r$ when p does not match r.*

We can now define how a packet is processed given a possibly empty list of rules (denoted with ϵ), hereafter called *ruleset*. Similarly to real implementations of firewalls, we inspect the rules in the list, one after the other, until we find a matching one, which establishes the destiny (or target) of the packet. For sanity, we assume that no `GOTO(R)` and `CALL(R)` occur in the ruleset R, so avoiding self-loops. We also assume that rulesets may have a default target denoted by $t_d \in \{\texttt{ACCEPT}, \texttt{DROP}\}$, which accepts or drops according to the will of the system administrator.

Definition 3 (Ruleset match). *Given a ruleset $R = [r_1, \ldots, r_n]$, we say that p matches the i-th rule with target t, denoted $p \models_R (t, i)$, iff*

$$r_i = (\phi, t) \wedge p \models_{r_i} t \wedge \forall j < i \,.\, p \not\models_{r_j}$$

We also write $p \not\models_R$ if p matches no rules in R, formally if $\forall r \in R \,.\, p \not\models_r$. Afterwards, we will omit the index i when immaterial and we simply write $p \models_R t$.

In our model we do not explicitly specify the steps performed by the kernel of the operating system to process a single packet passing through the host. We represent this algorithm through a *control diagram*, *i.e.*, a graph where nodes represent different processing steps and the arcs determine the sequence of steps. The arcs are labeled with a predicate describing the requirements a packet has to meet in order to pass to the next processing phase. We assume that control diagrams are deterministic, *i.e.*, that every pair of arcs leaving the same node has mutually exclusive predicates. For generality, we let these predicates abstract, since they depend on the specific firewall.

Definition 4 (Control diagram). *Let Ψ be a set of predicates over packets. A* control diagram *$\mathcal{C}$ is a tuple (Q, A, q_i, q_f), where*

- *Q is the set of nodes;*
- *$A \subseteq Q \times \Psi \times Q$ is the set of arcs, such that whenever $(q, \psi, q'), (q, \psi', q'') \in A$ and $q' \neq q''$ then $\forall p. \neg(\psi(p) \wedge \psi'(p))$;*
- *$q_i, q_f \in Q$ are special nodes denoting the start and the end of elaboration.*

The firewall filters and possibly translates a given packet by traversing a control diagram accordingly to the following transition function.

Definition 5 (Transition function). *Let (Q, A, q_i, q_f) be a control diagram and let p be a packet. The transition function $\delta \colon Q \times Packet \mapsto Q$ is defined as*

$$\delta(q, p) = q' \quad \textit{iff} \quad \exists (q, \psi, q') \in A.\ \psi(p) \textit{ holds.}$$

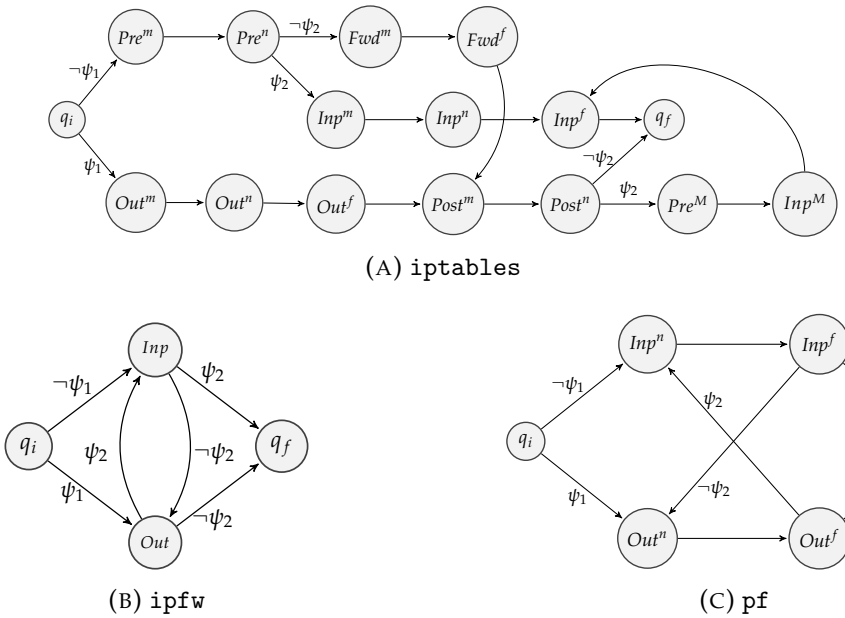


FIGURE 1.5: Control diagrams of the different systems.

We can now define a firewall in IFCL.

Definition 6 (Firewall)**.** *A firewall $\mathcal{F}$ is a triple $(\mathcal{C}, \rho, c)$ where*

- *$\mathcal{C}$ is a control diagram;*
- *ρ is a set of rulesets;*
- *$c : Q \mapsto \rho$ is the mapping from the nodes of $\mathcal{C}$ to the actual rulesets.*

1.4.1 Decompiling Real Systems into IFCL

Here we encode the firewalls system considered so far as triples $(\mathcal{C}, \rho, c)$ of our framework (stage 1). The encoding provides a formal semantics for these systems defined in terms of that of IFCL (see Section 1.4.2).

Modelling iptables

Let $\mathcal{L}$ be the set of local addresses of a host; and let ψ_1 and ψ_2 predicates over packets defined as follows:

$$\psi_1(p) = sa(p) \in \mathcal{L} \qquad \psi_2(p) = da(p) \in \mathcal{L}.$$

Figure 1.5a shows the control diagram $\mathcal{C}$ of iptables, where unlabeled arcs carry the label "*true*". It also implicitly defines the transition function according to Definition 5. In iptables there are twelve built-in chains, each of which corresponds to a single ruleset. So we can define the set $\rho_p \subseteq \rho$ of primitive rulesets as the one made of R^{man}_{INP}, R^{nat}_{INP}, R^{fil}_{INP}, R^{man}_{OUT}, R^{nat}_{OUT}, R^{fil}_{OUT}, R^{man}_{PRE}, R^{nat}_{PRE}, R^{man}_{FOR}, R^{fil}_{FOR}, R^{man}_{POST} and R^{nat}_{POST}, where the superscript and

subscript respectively represent the table name and the chain name. Note that the set $\rho \setminus \rho_p$ contains the user-defined chains.

The mapping function $c : Q \mapsto \rho$ is defined as follows:

$$
\begin{array}{lll}
c(q_i) = R_\epsilon & c(q_f) = R_\epsilon & c(Pre^m) = R^{man}_{\text{PRE}} \\
c(Pre^n) = R^{nat}_{\text{PRE}} & c(Inp^m) = R^{man}_{\text{INP}} & c(Fwd^f) = R^{fil}_{\text{FOR}} \\
c(Inp^n) = R^{nat}_{\text{INP}} & c(Inp^f) = R^{fil}_{\text{INP}} & c(Out^m) = R^{man}_{\text{OUT}} \\
c(Out^n) = R^{nat}_{\text{OUT}} & c(Out^f) = R^{fil}_{\text{OUT}} & c(Fwd^m) = R^{man}_{\text{FOR}} \\
c(Fwd^f) = R^{fil}_{\text{FOR}} & c(Post^m) = R^{man}_{\text{POST}} & c(Post^n) = R^{nat}_{\text{POST}} \\
c(Pre^M) = R^{man}_{\text{PRE}} & c(Inp^M) = R^{man}_{\text{INP}} &
\end{array}
$$

where R_ϵ is an empty ruleset with `ACCEPT` as default policy.

Modelling `ipfw`

The control diagram $\mathcal{C}$ of `ipfw`, displayed in Figure 1.5b, is simpler than the one of `iptables`. The node *Inp* represents the procedure executed when a packet reaches the host from the network. Dually, *Out* is processed when the packet leaves the host. The predicates ψ_1, ψ_2 are defined as for `iptables` and check whether the packet has been generated by the host or is addressed to the host itself, respectively. The transition function δ easily follows from $\mathcal{C}$, according to Definition 5.

We present the construction of the rulesets associated to the node *Inp*. Let $R = [r_{id_1}, \ldots, r_{id_k}]$ be the unique ruleset of `ipfw`, where the id_i's are the numeric identifiers associated to the rules and r_{id_k} is the rule encoding the default policy set by the user. The idea is to generate k different rulesets R^I_i, one for each rule in R. If the rule r_{id_i} contains the keyword `out`, *i.e.*, the rule is not considered when the packet enters the firewall, we let $R^I_i = [(true, \texttt{GOTO}(R^I_{i+1}))]$. Otherwise, we define $R^I_i = [trs(r_{id_i}), (true, \texttt{GOTO}(R^I_{i+1}))]$, where the translation *trs* is defined by cases below:

$$
trs(r) = \begin{cases}
(\phi, \texttt{GOTO}(R^I_n)) & \text{if } r \text{ is } \texttt{skipto } id_n\ \phi \\
(\phi, \texttt{CALL}(R^I_n)) & \text{if } r \text{ is } \texttt{call } id_n\ \phi \\
(\phi, \texttt{t}) & \text{if } r \text{ is } \texttt{t}\ \phi
\end{cases}
$$

The construction of the rulesets R^O_i for the node *Out* is similar, but in this case the rules containing the keyword `in` should be ignored. The mapping function c returns R^I_1 for *Inp*, R^O_1 for *Out*, and an empty ruleset with `ACCEPT` as default policy for q_i and q_f. These rulesets form the component ρ.

Modelling `pf`

Differently from `iptables`, `pf` has a single ruleset and the rule applied to a packet is the last matching one, apart from the case of the so-called `quick` rules: as soon as one of these

rules matches the packet, its action is applied and the rest of the ruleset is ignored.

Figure 1.5c shows the control diagram $\mathcal{C}$ for `pf` that also defines the transition function. The nodes Inp^n and Inp^f represent the procedure executed when a packet reaches the host from the network. Dually, Out^n and Out^f are for when the packet leaves the host. The predicates ψ_1 and ψ_2 are those defined for `iptables`. Given the ruleset R representing the firewall policy, we include the following rulesets in ρ:

- R_{dnat} contains the rule $(state == \texttt{ESTABLISHED}, \texttt{CHECK-STATE}(\rightarrow))$ as the first one, followed by all the rules `rdr` of R;
- R_{snat} contains the rule $(state == \texttt{ESTABLISHED}, \texttt{CHECK-STATE}(\leftarrow))$ as the first one, followed by all the rules `nat` of R;
- R_{finp} contains the rule $(state == \texttt{ESTABLISHED}, \texttt{ACCEPT})$ followed by all the `quick` filtering rules of $R_{\texttt{pf}}$ without modifier `out`, and finally the rule $(true, \texttt{GOTO}(R_{finpr}))$;
- R_{finpr} contains all the no `quick` filtering rules of R without modifier `out`, in reverse order;
- R_{fout} contains the rule $(state == \texttt{ESTABLISHED}, \texttt{ACCEPT})$ followed by all the `quick` filtering rules of R without modifier `in`, and $(true, \texttt{GOTO}(R_{foutr}))$ as last rule;
- R_{foutr} includes all the no `quick` filtering rules of R without modifier `in` in reverse order.

Let R_ϵ be an empty ruleset with `ACCEPT` as default policy. The mapping function c is defined as follows:

$$\begin{array}{lll} c(q_i) = R_\epsilon & c(Inp^n) = R_{dnat} & c(Out^n) = R_{snat} \\ c(q_f) = R_\epsilon & c(Inp^f) = R_{finp} & c(Out^f) = R_{fout} \end{array}$$

1.4.2 Semantics

Now, we formally define the semantics of `ipfw` through two transition systems operating in a master-slave fashion. The master has a labeled transition relation of the form $s \xrightarrow{p,p'} s'$. The intuition is that the state s of a firewall changes to s' when a new packet p reaches the host and becomes p'. The configurations of the slave transition system are triples (q, s, p) where:

- $q \in Q$ is a control diagram node;
- s is the state of the firewall;
- p is the packet.

A transition $(q, s, p) \rightarrow (q', s, p')$ describes how a firewall in a state s deals with a packet p and possibly transforms it in p', according to the control diagram $\mathcal{C}$. Recall that the state records established connections and other kinds of information that are updated

after the transition. In the slave transition relation, we rely on the following predicate that describes an algorithm running a ruleset R on a packet p in the state s:

$$p, s \models^{S}_{R} (t, p')$$

This predicate searches for a rule in R matching the packet p through $p \models_R (t, i)$. If it finds a matching rule, the target t is applied to p to obtain a new packet p'.

Recall that actions `CALL(R)`, `RETURN` and `GOTO(R)` are similar to procedure calls, returns and jumps in imperative programming languages. To correctly deal with them, our predicate $p, s \models^{S}_{R} (t, p')$ uses a stack S to implement a behaviour similar to the one of procedure calls. We will denote with ϵ the empty stack and with $\cdot$ the concatenation of elements on the stack. This stack is also used to detect and prevent loops in ruleset invocation, as it is the case in real firewalls.

In the stack S we overline a ruleset R to indicate that it was pushed by a `GOTO(_)` action and it has to be skipped when returning. Indeed, we use the following $pop^\star$ function in the semantics of the `RETURN` action:

$$pop^*(\epsilon) = \epsilon \qquad pop^*(R \cdot S) = (R, S) \qquad pop^*(\overline{R} \cdot S) = pop^*(S)$$

In case there is a non-overlined ruleset on the top of S, it behaves as a standard pop operation; otherwise it extracts the first non-overlined ruleset. When S is empty, we assume that pop^* returns ϵ to signal the error.

Furthermore, in the definition of $p, s \models^{S}_{R} (t, p')$ the notation R_k indicates the ruleset $[r_k, ..., r_n]$ resulting from dropping the first $k-1$ rules from the ruleset $R = [r_1, ..., r_n]$. We also assume the function *establ* that, taken an action α from the state, a packet p and the fields $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ to rewrite, returns a possibly changed packet p', *e.g.*, in case of an established connection. This function depends on the specific firewall we are modeling and so it is left unspecified.

Finally, we assume as given a function $nat(p, s, d_n, s_n)$ that returns the packet p translated under the corresponding NAT operation in the state s. The arguments d_n and s_n are used to modify the destination range and the source range of p, *i.e.*, to perform destination NAT and source SNAT. Also this function is left abstract.

Figure 1.6 shows the rules defining $p, s \models^{S}_{R} (t, p')$. The first inference rule deals with the case when the packet p matches a rule that says `ACCEPT` or `DROP`; in this case the ruleset execution stops returning the found action and leaving p unmodified. When a packet p matches a rule with action `CHECK-STATE`, we query the state s: if p belongs to an established connection, we return `ACCEPT` and a p' obtained by rewriting p. If p belongs to a new connection, the packet is matched against the remaining rules in the ruleset. When a packet p matches a `NAT` rule, we return `ACCEPT` and the packet resulting by the invocation of the function *nat*. There are two cases if a packet p matches a `GOTO(_)`. If the ruleset R' is not already in the stack, we push the current ruleset R onto the stack overlined to record that this ruleset dictated a `GOTO(_)`. Otherwise, if R' is in the stack, we detect the loop and discard p. The case when a packet p matches a rule with action

$$(1)\ \frac{p \models_R (t, i) \qquad t \in \{\texttt{ACCEPT}, \texttt{DROP}\}}{p, s \models^S_R (t, p)}$$

$$(2)\ \frac{p \models_R (\texttt{CHECK-STATE}(X), i) \qquad p \vdash_s \alpha \qquad p' = \mathit{establ}(\alpha, X, p)}{p, s \models^S_R (\texttt{ACCEPT}, p')}$$

$$(3)\ \frac{p \models_R (\texttt{CHECK-STATE}(X), i) \qquad p \nvdash_s \qquad p, s \models^S_{R_{i+1}} (t, p')}{p, s \models^S_R (t, p')}$$

$$(4)\ \frac{p \models_R (\texttt{NAT}(d_n, s_n), i)}{p, s \models^S_R (\texttt{ACCEPT}, \mathit{nat}(p, s, d_n, s_n))} \qquad (5)\ \frac{p \models_R (\texttt{GOTO(R')}, i) \qquad R' \notin S \qquad p, s \models^{\overline{R}\cdot S}_{R'} (t, p')}{p, s \models^S_R (t, p')}$$

$$(6)\ \frac{p \models_R (\texttt{GOTO(R')}, i) \qquad R' \in S}{p, s \models^S_R (\texttt{DROP}, p)} \qquad (7)\ \frac{p \models_R (\texttt{CALL(R')}, i) \qquad R' \notin S \qquad p, s \models^{R_{i+1}\cdot S}_{R'} (t, p')}{p, s \models^S_R (t, p')}$$

$$(8)\ \frac{p \models_R (\texttt{CALL(R')}, i) \qquad R' \in S}{p, s \models^S_R (\texttt{DROP}, p)}$$

$$(9)\ \frac{p \models_R (\texttt{RETURN}, i) \qquad \mathit{pop}^*(S) = (R', S') \qquad p, s \models^{S'}_{R'} (t, p')}{p, s \models^S_R (t, p')}$$

$$(10)\ \frac{p \models_R (\texttt{RETURN}, i) \quad \mathit{pop}^*(S) = \epsilon}{p, s \models^S_R (t_d, p)} \qquad (11)\ \frac{p \not\models_R \qquad \mathit{pop}^*(S) = (R', S') \qquad p, s \models^{S'}_{R'} (t, p')}{p, s \models^S_R (t, p')}$$

$$(12)\ \frac{p \not\models_R \qquad (S = \epsilon \vee \mathit{pop}^*(S) = \epsilon)}{p, s \models^S_R (t_d, p)} \qquad (13)\ \frac{p \models_R (\texttt{MARK}(m), i) \qquad p[\mathit{tag} \mapsto m], s \models^S_{R_{i+1}} (t, p')}{p, s \models^S_R (t, p')}$$

FIGURE 1.6: The predicate $p, s \models^S_R (t, p')$.

`CALL(_)` is similar, except that the ruleset pushed on the stack is not overlined. When a packet p matches a rule with action `RETURN`, we pop the stack and match p against the top of the stack. When no rule matches, an implicit return occurs: we continue from the top of the stack, if non empty. Finally, the `MARK` rule simply changes the tag of the matching packet to the value m. If none of the above applies, we return the default action t_d of the current ruleset.

We can now define the slave transition relation as follows.

$$\frac{c(q) = R \qquad p, s \models^{\epsilon}_R (\texttt{ACCEPT}, p') \qquad \delta(q, p') = q'}{(q, s, p) \to (q', s, p')}$$

The rule describes how we process the packet p when the firewall is in state s and performs the step represented by the node q. We match p against the ruleset R associated with q and if p is accepted as p', we continue considering the next step of the firewall execution represented by the node q'. Finally, we define the master transition relation

that transforms states and packets as follows, where $\rightarrow^+$ is the transitive closure of $\rightarrow$:

$$\frac{(q_i, s, p) \rightarrow^+ (q_f, s, p')}{s \xrightarrow{p,p'} s \uplus (p, p')}$$

This rule says that when the firewall is in the state s and receives a packet p, it elaborates p starting from the initial node q_i of its control diagram. If this elaboration succeeds, *i.e.*, it reaches the node q_f that accepts p as p', we update the state s by storing information about p, its translation p' and the connection they belong to through the function $\uplus$, left unspecified for the sake of generality.

Example 1. *Suppose to have the following chains*

Chain C_B	*Chain* u_1	*Chain* u_2
$(\phi_1, \texttt{DROP})$	$(\phi_{11}, \texttt{ACCEPT})$	$(\phi_{21}, \texttt{ACCEPT})$
$(\phi_2, \texttt{CALL}(u_1))$	$(\phi_{12}, \texttt{CALL}(u_2))$	$(\phi_{22}, \texttt{RETURN})$
$(\phi_3, \texttt{ACCEPT})$	$(\phi_{13}, \texttt{DROP})$	$(\phi_{23}, \texttt{DROP})$

and that the condition $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ *holds for a packet* p*. Then, the semantic rules (a), (b) and (c) are applied in order:*

$$\text{(A)}\ \frac{p \models_{C_B} (\texttt{CALL}(u_1), 2) \qquad u_1 \notin S \qquad p, s \models_{u_1}^{C_{B_3} \cdot \epsilon} (\texttt{ACCEPT}, p)}{p, s \models_{C_B}^{\epsilon} (\texttt{ACCEPT}, p)}$$

$$\text{(B)}\ \frac{p \models_{u_1} (\texttt{ACCEPT}, 1)}{p, s \models_{u_1}^{C_{B_3} \cdot \epsilon} (\texttt{ACCEPT}, p)} \qquad \text{(C)}\ \frac{c(q) = C_B \qquad p, s \models_{C_B}^{\epsilon} (\texttt{ACCEPT}, p) \qquad \delta(q, p) = q'}{(q, s, p) \rightarrow (q', s, p)}$$

1.5 Synthesizing Configurations

We now extract the meaning of a firewall written in our intermediate language by transforming it into a declarative, logical presentation that preserves the semantics (stage 2). This transformation is independent of the real firewall language in hand, and consists of the following steps:

1. generate an unfolded firewall with a single ruleset for each node of the control diagram;
2. construct a logical formula that characterizes the packets accepted by the firewall;
3. determine the model for the formula through a SAT solver.

The correctness of stage 2 follows from Theorem 1, which guarantees that the unfolded firewall is semantically equivalent to the original one, and from Theorem 2, which ensures that the derived formula characterizes exactly the accepted packets and their translations.

1.5.1 Unfolding Rulesets

Our intermediate language can deal with involved control flows, by using the targets `GOTO(_)`, `CALL(_)` and `RETURN` (see Example 1). The following unfolding operation $[\![\_]\!]$ rewrites a ruleset into an equivalent one with no control flow rules.

Hereafter, let $r; R$ be a non empty ruleset consisting of a rule r followed by a possibly empty ruleset R; and let $R_1@R_2$ be the concatenation of R_1 and R_2. The unfolding of a ruleset R is defined as follows:

$$
\begin{aligned}
[\![R]\!] &= [\![R]\!]^{true}_{\{R\}} \\
[\![\epsilon]\!]^f_I &= \epsilon \\
[\![(\phi, t); R]\!]^f_I &= (f \wedge \phi, t); [\![R]\!]^f_I \quad \text{if } t \notin \{\texttt{GOTO(R')}, \texttt{CALL(R')}, \texttt{RETURN}\} \\
[\![(\phi, \texttt{RETURN}); R]\!]^f_I &= [\![R]\!]^{f \wedge \neg\phi}_I \\
[\![(\phi, \texttt{CALL(R')}); R]\!]^f_I &= \begin{cases} [\![R']\!]^{f\wedge\phi}_{I\cup\{R'\}} @ [\![R]\!]^f_I & \text{if } R' \notin I \\ (f \wedge \phi, \texttt{DROP}); [\![R]\!]^f_I & \text{otherwise} \end{cases} \\
[\![(\phi, \texttt{GOTO(R')}); R]\!]^f_I &= \begin{cases} [\![R']\!]^{f\wedge\phi}_{I\cup\{R'\}} @ [\![R]\!]^{f\wedge\neg\phi}_I & \text{if } R' \notin I \\ (f \wedge \phi, \texttt{DROP}); [\![R]\!]^{f\wedge\neg\phi}_I & \text{otherwise} \end{cases}
\end{aligned}
$$

The auxiliary procedure $[\![R]\!]^f_I$ recursively inspects the ruleset R. The formula f accumulates conjuncts of the predicate ϕ; the set I records the rulesets traversed by the procedure to detect loops. If a rule does not affect control flow, we just substitute the conjunction $f \wedge \phi$ for ϕ, and continue to analyse the rest of the ruleset with the recursive call $[\![R]\!]^f_I$.

In the case of a return rule $(\phi, \texttt{RETURN})$ we continue to recursively analyse the rest of the ruleset, by updating f with the negation of ϕ. For the rule $(\phi, \texttt{CALL(R')})$ we have two cases: if the callee ruleset R' is not in I, we replace the rule with the unfolding of R' with $f \wedge \phi$ as predicate, and add R' to the traversed rulesets. If R' is already in I, *i.e.*, we have a loop, we replace the rule with a `DROP`, with $f \wedge \phi$ as predicate. In both cases, we continue unfolding the rest of the ruleset. We deal with the rule $(\phi, \texttt{GOTO(R')})$ as the previous one, except that the rest of the ruleset has $f \wedge \neg\phi$ as predicate.

Example 2. *Back to Example 1, unfolding the chain C_B gives the following rules:*

$$
\begin{aligned}
[\![C_B]\!] = \; & (\phi_1, \texttt{DROP}); \\
& (\phi_2 \wedge \phi_{11}, \texttt{ACCEPT}); \\
& (\phi_2 \wedge \phi_{12} \wedge \phi_{21}, \texttt{ACCEPT}); \\
& (\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}, \texttt{DROP}); \\
& (\phi_2 \wedge \phi_{13}, \texttt{DROP}); \\
& (\phi_3, \texttt{ACCEPT}); \\
& \epsilon
\end{aligned}
$$

We just illustrate the first three steps:

$$\begin{aligned}[\![C_B]\!] =& [\![(\phi_1, \texttt{DROP}); C_{B2}]\!]^{true}_{\{C_B\}} = (\phi_1, \texttt{DROP}); [\![(\phi_2, \texttt{CALL}(u_1)); C_{B3}]\!]^{true}_{\{C_B\}} \\ =& [\![u_1]\!]^{true \wedge \phi_2}_{\{C_B\} \cup \{u_1\}} @ [\![C_{B3}]\!]^{true}_{\{C_B\}}\end{aligned}$$

Note that our transformation does not change the set of acccepted packets: all the packets satisfying $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ are still accepted by the unfolded ruleset.

An unfolded firewall is obtained by repeatedly rewriting the rulesets associated with the nodes of its control diagram, using the procedure above. Formally,

Definition 7 (Unfolded firewall)**.** *Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$, its unfolded version $[\![\mathcal{F}]\!]$ is $(\mathcal{C}, \rho', c')$ where $\forall q \in \mathcal{C}.\, c'(q) = [\![c(q)]\!]$ and $\rho' = \{[\![c(q)]\!] \mid q \in \mathcal{C}\}$.*

We now prove that a firewall $\mathcal{F}$ and its unfolded version $[\![\mathcal{F}]\!]$ are semantically equivalent, *i.e.*, they perform the same action over a given packet p in a state s, and reach the same state s'. Formally, the following theorem holds:

Theorem 1 (Correctness of unfolding)**.** *Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $[\![\mathcal{F}]\!]$ its unfolding. Let $s \xrightarrow{p,p'}_X s'$ be a step of the master transition system performed by the firewall $X \in \{\mathcal{F}, [\![\mathcal{F}]\!]\}$. Then, it holds*

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{[\![\mathcal{F}]\!]} s'.$$

1.5.2 Logical Characterization of Firewalls

We construct a logical predicate that characterizes the packets accepted by a ruleset, together with the relevant translations. Because of Theorem 1, hereafter we will only consider unfolded firewalls.

To deal with NAT, we define the function tr that computes the set of packets resulting from all possible translations of a given packet p. The parameter $X \in \{\leftarrow, \rightarrow, \leftrightarrow\}$ specifies if the translation applies to source, destination or both addresses, respectively, similarly to $\texttt{CHECK-STATE}(X)$.

$$\begin{aligned}tr(p, d_n, s_n, \leftrightarrow) &\triangleq \{p[da \mapsto a_d, sa \mapsto a_s] \mid a_d \in d_n, a_s \in s_n\} \\ tr(p, d_n, s_n, \rightarrow) &\triangleq \{p[da \mapsto a_d] \mid a_d \in d_n\} \\ tr(p, d_n, s_n, \leftarrow) &\triangleq \{p[sa \mapsto a_s] \mid a_s \in s_n\}\end{aligned}$$

Furthermore, we model the default policy of a ruleset R with the predicate dp, true when the policy is `ACCEPT`, false otherwise.

Given an unfolded ruleset R, we build a predicate $P_R(p, \tilde{p})$ that holds when the packet p can be accepted as $\tilde{p}$ by R. The predicate is defined as follows:

$$
\begin{aligned}
P_\epsilon(p,\tilde{p}) &= dp(R) \wedge p = \tilde{p} \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge p = \tilde{p}) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{ACCEPT}) \\
P_{r;R}(p,\tilde{p}) &= \neg\phi(p) \wedge P_R(p,\tilde{p}) && \text{if } r = (\phi, \texttt{DROP}) \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge \tilde{p} \in tr(p, d_n, s_n, \leftrightarrow)) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{NAT}(d_n, s_n)) \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge \tilde{p} \in tr(p, *{:}*, *{:}*, X) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{CHECK-STATE}(X)) \\
P_{r;R}(p,\tilde{p}) &= (\phi(p) \wedge P_R(p[tag \mapsto m],\tilde{p})) \vee (\neg\phi(p) \wedge P_R(p,\tilde{p})) && \text{if } r = (\phi, \texttt{MARK}(m))
\end{aligned}
$$

The empty ruleset ϵ applies the default policy $dp(R)$ and does not transform the packet, encoded by the constraint $p = \tilde{p}$. The rule $(\phi, \texttt{ACCEPT})$ considers two cases: when $\phi(p)$ holds the packet is accepted as it is; when instead $\neg\phi(p)$ holds, p is accepted as $\tilde{p}$ only if the continuation R accepts it. The rule $(\phi, \texttt{DROP})$ accepts p only if the continuation does and $\phi(p)$ does not hold. The rule $(\phi, \texttt{NAT}(d_n, s_n))$ is like an $(\phi, \texttt{ACCEPT})$: the difference is when $\phi(p)$ holds, and it gives $\tilde{p}$ by applying to p the NAT translations $tr(p, d_n, s_n, \leftrightarrow)$. Finally, $(\phi, \texttt{CHECK-STATE}(X))$ is like a NAT that applies all possible translations of kind X (written as $tr(p, *:*, *:*, X)$). The idea is that, since we abstract away from the actual established connections, we over-approximate the state by considering any possible translations. At run-time, only the connections corresponding to the actual state will be possible. The rule $(\phi, \texttt{MARK}(m))$ is like a NAT, but when $\phi(p)$ holds it requires that the continuation accepts p tagged by m as $\tilde{p}$.

Example 3. *Consider again the unfolded chain of Example 2. If $dp(C_B) = F$, the predicate is defined as:*

$$
\begin{aligned}
P_{[\![C_B]\!]}\,(p,\tilde{p}) = \neg\phi_1 \wedge (\\
&(\phi_2 \wedge \phi_{11} \wedge p = \tilde{p}) \vee (\neg(\phi_2 \wedge \phi_{11}) \wedge (\\
&(\phi_2 \wedge \phi_{12} \wedge \phi_{21} \wedge p = \tilde{p}) \vee (\neg(\phi_2 \wedge \phi_{12} \wedge \phi_{21}) \wedge (\\
&\neg(\phi_2 \wedge \phi_{12} \wedge \neg\phi_{22} \wedge \phi_{23}) \wedge (\\
&\neg(\phi_2 \wedge \phi_{13}) \wedge (\\
&(\phi_3 \wedge p = \tilde{p}) \vee (\neg\phi_3 \wedge (\\
&F \wedge p = \tilde{p}))))))))
\end{aligned}
$$

Note that if $\neg\phi_1 \wedge \phi_2 \wedge \phi_{11}$ holds then the formula trivially holds and therefore the formula accepts the packet as the semantics does.

As a further example, consider the case in which $\phi_2, \phi_{12}, \phi_{22}, \phi_{23}, \phi_3$ hold for a packet p, while all the other ϕ's does not. Then, p is accepted as it is: the rule $(\phi_{23}, \texttt{DROP})$ is not evaluated since ϕ_{22} holds and the `RETURN` *is performed (*cf. *Example 1). Indeed, the predicate $P_{[\![C_B]\!]}(p, p)$ evaluates to:*

$$T \wedge (F \vee (T \wedge (F \vee (T \wedge (T \wedge (T \wedge (T \vee (F \wedge F)))))))) = T$$

Instead, if ϕ_{13} holds as well, the packet is rejected as expected:

$$T \wedge (F \vee (T \wedge (F \vee (T \wedge (T \wedge (F \wedge (T \vee (F \wedge F))))))))) = F$$

The predicate $P_R(p, p')$ is semantically correct, because if a packet p is accepted by a ruleset R as p', then $P_R(p, p')$ holds, and viceversa.

Lemma 1. *Given a ruleset R we have that*

1. $\forall p, s.\ p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p') \implies P_R(p, p')$; *and*
2. $\forall p, p'.\ P_R(p, p') \implies \exists s.\ p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p')$

We define the predicate associated with a whole firewall as follows.

Definition 8. *Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall with control diagram $\mathcal{C} = (Q, A, q_i, q_f)$. The predicate associated with $\mathcal{F}$ is defined as*

$$\mathcal{P}_{\mathcal{F}}(p, \tilde{p}) \triangleq \mathcal{P}^{\emptyset}_{q_i}(p, \tilde{p}) \quad \textit{where}$$

$$\mathcal{P}^{I}_{q_f}(p, \tilde{p}) \triangleq p = \tilde{p} \qquad \mathcal{P}^{I}_{q}(p, \tilde{p}) \triangleq \exists p'. P_{c(q)}(p, p') \wedge \left(\bigvee_{\substack{(q, \psi, q') \in A \\ q' \notin I}} \psi(p') \wedge \mathcal{P}^{I \cup \{q\}}_{q'}(p', \tilde{p}) \right)$$

for all $q \in Q$ such that $q \neq q_f$, and where $P_{c(q)}$ is the predicate constructed from the ruleset associated with the node q of the control diagram.

Intuitively, in the final node q_f we accept p as it is. In all the other nodes, p is accepted as $\tilde{p}$ if and only if there is a path in the control diagram starting from the current node that obtains $\tilde{p}$ from p through intermediate transformations. More precisely, we look for an intermediate packet p', provided that:

1. p is accepted as p' by the ruleset $c(q)$ of node q;
2. p' satisfies one of the predicates ψ labeling the branches of the control diagram;
3. p' is accepted as $\tilde{p}$ in the reached node q'.

Note that we ignore paths with loops because firewalls have mechanisms to detect and discard a packet when its elaboration loops. To this aim, our predicate uses the set I for recording the nodes already traversed.

We conclude this section by establishing the correspondence between the logical formulation and the operational semantics of a firewall. Formally, $\mathcal{F}$ accepts the packet p as $\tilde{p}$ if the predicate $\mathcal{P}_{\mathcal{F}}(p, \tilde{p})$ is satisfied, and viceversa:

Theorem 2 (Correctness of the logical characterization)**.** *Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and its corresponding predicate $\mathcal{P}_{\mathcal{F}}$ we have:*

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_{\mathcal{F}}(p, p')$

2. $\forall p, p'.\, \mathcal{P}_{\mathcal{F}}(p, p') \implies \exists s.s \xrightarrow{p,p'} s \uplus (p, p')$

Recall that the logical characterization abstracts away the notion of state, and thus $\mathcal{P}_{\mathcal{F}}(p, p')$ holds if and only if there exists a state s in which p is accepted as p'. In particular, if the predicate holds for a packet p that belongs to an established connection, p will be accepted only if the relevant state is reached at runtime. This is the usual interpretation of firewall rules for established connections.

1.5.3 Synthesis Algorithm

Here we present the module of FWS [32] that syntesizes the declarative specification of a firewall configuration $\mathcal{F}$ expressed in `IFCL`. In particular, the core of this module constructs a set of logical predicates $\mathcal{P} = \{\mathcal{P}_1(p, \tilde{p}), \ldots, \mathcal{P}_n(p, \tilde{p})\}$ defined over pairs of packets that characterize the firewall behaviour. In particular, $\mathcal{P}_i(p, \tilde{p})$ evaluates to true if the input packet p is accepted as $\tilde{p}$ by the firewall. The set of pairs that satisfy one of the predicates in $\mathcal{P}$ is exactly the set of pairs satisfying $\mathcal{P}_{\mathcal{F}}$ presented in Section 1.5.2.

We use the *multi-cubes* representation introduced in [88] to succinctly enumerate all the packets accepted by the firewall. From these multi-cubes our tool builds a tabular representation of the firewall containing only `ACCEPT` and `NAT` rules, with no overlapping rows.

Encoding in Z3

We model packets as tuples of Z3 bit-vector variables of appropriate size

```
(srcIP, srcPort, dstIP, dstPort, protocol, state)
```

that represent source and destination IPs and ports, the protocol and the packet state. Firewall predicates are expressed as logical formulas on those packet variables. For example, the constraint

$$\texttt{dstIp} \equiv 10.0.2.15 \wedge \texttt{dstPort} = 22$$

selects packets with destination 10.0.2.15 and port 22. We write $\texttt{dstIp} \equiv 10.0.2.15$ as a shortcut for equating `dstIp` with the numerical representation of the IP address 10.0.2.15. Intervals are encoded with two $\leq$ constraints.

In order to succinctly enumerate packets, a multi-cube maps each packet variable v to a union of disjoint intervals I_v to which the value of v belongs. For instance, the solutions of the formula

$$(\texttt{dstIp} \equiv 10.0.2.15 \ \vee \ \texttt{dstIp} \equiv 10.0.1.0/24) \ \wedge \ (\texttt{dstPort} = 22 \vee \texttt{dstPort} = 443)$$

can be expressed by the following multi-cube:

$$\texttt{dstIp} = \{10.0.2.15\} \cup [10.0.1.0,\ 10.0.1.255] \quad \texttt{dstPort} = \{22\} \cup \{443\}$$

Algorithm 1 All-BVSAT*

Require: Formula φ over bit-vectors with free variables $\vec{x}$
Ensure: Set of multi-cubes $\mathcal{M}$ that are models of φ

```
1:  B ← φ
2:  M ← ∅
3:  while B is satisfiable do
4:      v⃗ ← a satisfiable assignment to B
5:      for each multi-cube M⃗ ∈ M do
6:          Extend M⃗ with v⃗ if possible
7:          B ← B ∧ (x⃗ ∉ M⃗)
8:      if B ∧ x⃗ = v⃗ is still satisfiable then
9:          C⃗ ← {v_1} × ... × {v_n}
10:         for each i in 1..n do
11:             Expand interval C_i
12:         M ← M ∪ {C⃗}
13:         B ← B ∧ (x⃗ ∉ M⃗)
14: return M
```

For each formula over bit-vector variables, we compute the satisfying multi-cubes using Algorithm 1. Intuitively, each iteration of the while loop selects an assignment of variables $\vec{v}$ that is not covered by any of the existing multi-cubes. First the algorithm tries to extend the existing multi-cubes with the values in $\vec{v}$; next, if the formula is still satisfiable, a new multi-cube is created.

During the extension/creation of multi-cubes, the algorithm performs an expansion step that extends as much as possible the intervals both downwards and upwards. This step uses a variant of the binary search algorithm to find the bounds of the maximal interval that satisfies the given formula. The complexity of this step is linear in the size in bits of the variable under consideration. We refer the interested reader to [88] for additional details about the algorithm.

Dealing with NAT

Synthesis gets complicated with NAT because it can introduce many variables in the formulas, representing intermediate address values for the packet during different processing phases. Some variables, however, are not touched by NAT and this needs to be represented in the predicates, as discussed in the following.

A natural way is to impose equality constraints on variables that are not affected by NAT. Despite intuitive, this approach has a severe drawback: equality constraints do not work well with Algorithm 1. For instance, consider the formula $1 \leq v_1 \leq 5 \wedge v_1 = v_2$: the algorithm uses the SMT solver to find a solution, *e.g.*, $v_1 = v_2 = \{3\}$, and tries to expand the intervals associated to v_1 and v_2, one after the other. However, increasing the interval of v_1 violates the equality constraint with v_2. The result of the algorithm are thus 5 distinct multi-cubes, *i.e.*, $v_1 = v_2 = \{i\}$ for $i \in [1..5]\}$.

A careful treatement of equality introduces new variables only for the packet features that are modified by NAT rules and implicitly models equality constraints by sharing the same variable in the input and in the output packet. For instance, if a NAT rule modifies the destination address of the input packet p, the output packet $\tilde{p}$ is represented with the same variables as p with the exception of the destination address that uses a fresh variable. Since the introduction of these fresh variables is only required for the packets that are subject to NAT, we consider separate predicates covering the different cases: DNAT, SNAT and filtering. In DNAT and SNAT all variables will be the same except for the destination and source addresses/ports, respectively. In filtering, all variables will coincide, as the input and the output packets are the same.

In principle, this separation could lead to an explosion of the number of predicates. However, when studying existing firewall systems, we found that the maximum number of packets to be considered is three: in fact, in real systems NAT is applied at most twice during packet processing (one SNAT and one DNAT). For this reason, the proposed approach works very well in practice.

1.5.4 Supported Analyses

Besides synthesizing high-level specifications, once we have a firewall expressed as logical constraints in Z3, FWS can perform various interesting fully automated analyses:

- *Reachability*: is a certain address is reachable or not from another one, possibly through NAT? This analysis is implemented as follows. The query is transformed in a Z3 constraint on the packet variables and the solver is invoked to check its satisfiability with respect to the abstract specification;
- *Implication*: is the policy P_1 implied by a policy P_2? The answer is obtained by asking the solver if there exists no pair of packets that is accepted by P_2 and rejected by P_1;
- *Equivalence*: are two policies equivalent? Just check double implication;
- *Differences*: given two policies, we can synthesize them and show the differences in the extracted multi-cubes, possibly considering some selected packets, only;
- *Related rules*: to identify the rules affecting the processing of the packets selected by user-provided query, we remove, one at a time, a rule from the policy and check whether the new policy is not equivalent to the original one.

1.6 Generating Target Configurations

The last stage of our pipeline compiles the declarative specification extracted from a firewall policy in stage 2. In fact, starting from the specification we easily build a firewall $\mathcal{F}_S$ whose control diagram has a single node and a single ruleset containing the `ACCEPT` and `NAT` rules, each corresponding to a line of the declarative specification (since rows do not

Algorithm 2 Generation of the rulesets R_{dnat}, R_{fil}, R_{snat}, R_{mark} from R_S

1: $R_{dnat} = R_{fil} = R_{snat} = R_{mark} = \epsilon$
2: **for** r in R_S **do**
3: **if** $r = (\phi, \texttt{ACCEPT})$ **then**
4: add r to R_{fil}
5: **else if** $r = (\phi, \texttt{NAT}(d_n, s_n))$ **then**
6: generate fresh tag m
7: add $(\phi \wedge tag(p) = \bullet, \texttt{MARK}(m))$ to R_{mark}
8: add $(tag(p) = m, \texttt{NAT}(d_n, \star))$ to R_{dnat}
9: add $(tag(p) = m, \texttt{NAT}(\star, s_n))$ to R_{snat}
10: add $(tag(p) \neq \bullet, \texttt{ACCEPT})$ and $(true, \texttt{DROP})$ to R_{fil}
11: prepend R_{mark} to R_{dnat}, R_{fil} and R_{snat}

overlap they may appear in the ruleset in any order). Then, we compile $\mathcal{F}_S$ into a firewall $\mathcal{F}_C$ in the target language and we prove the two equivalent.

The resulting firewall automatically accepts all the packets that belong to established connections with the appropriate translations. This is not a limitation, since it is the default behaviour of some real firewall systems (*e.g.*, `pf`) and it is quite odd to drop packets once the initial connection has been established. Moreover, this is consistent with the over-approximation on the firewall state done in Section 1.5.2.

1.6.1 Compiling a Firewall Specification

We first introduce an algorithm that computes the rulesets of the target firewall $\mathcal{F}_C$. Then, we associate these rulesets with the nodes of its control diagram.

Algorithm 2 expects as input the ruleset R_S derived from a synthesized specification and splits it in the basic rulesets R_{fil}, containing filters, and R_{dnat}, R_{snat} (with default `ACCEPT` policy) for DNAT and SNAT rules. This separation reflects what is done in all the real systems we have analysed. Indeed, NAT rules can be placed only in specific nodes of their control diagrams, *e.g.*, in `iptables`, DNAT is allowed only in rulesets R^{nat}_{PRE} and R^{nat}_{OUT}, while SNAT only in R^{nat}_{INP} and R^{nat}_{POST}.

The algorithm leaves the filtering rules unchanged (line 4). Also, it produces rules that assign different tags to packets that must be processed by different NAT rules (lines 6 and 7). Each NAT rule is split in a DNAT (line 8) and a SNAT (line 9), where the predicate ϕ becomes a check on the tag of the packet. Packets subject to NAT are accepted in R_{fil} while the others are dropped (line 10). We prepend R_{mark} to all rulesets making sure that packets are always marked, independently of which ruleset will be processed first (line 11). The empty tag $\bullet$ identifies untagged packets.

Recall that the @ operator combines rulesets in sequence. Note that R_{fil} drops by default and shadows any ruleset appended to it. In practice, the only interesting rulesets are R_ϵ, R_{fil}, R_{dnat}, R_{snat}, $R_{dnat} \,@\, R_{fil}$, $R_{snat} \,@\, R_{fil}$ where R_ϵ is the empty ruleset with default `ACCEPT` policy. We now introduce the notion of *compiled firewall*.

Definition 9 (Compiled firewall)**.** *Let* $\mathcal{R} = \{R_\epsilon, R_{fil}, R_{dnat}, R_{snat}, R_{dnat} @ R_{fil}, R_{snat} @ R_{fil}\}$*. A firewall* $\mathcal{F}_C = (\mathcal{C}, \rho, c)$ *with control diagram* $\mathcal{C} = (Q, A, q_i, q_f)$ *is a* compiled firewall *if*

- $c(q_i) = c(q_f) = R_\epsilon$
- $c(q) \in \mathcal{R}$ *for all* $q \in Q$
- *every path* π *from* q_i *to* q_f *in the control diagram* $\mathcal{C}$ *traverses a node* $q \in Q$ *such that* $c(q) \in \{R_{fil}, R_{dnat} @ R_{fil}, R_{snat} @ R_{fil}\}$

Intuitively, the above definition requires that only rulesets in $\mathcal{R}$ are associated with the nodes in the control diagram and that all paths pass at least once through a node with the filtering ruleset.

Example 4. *Now we map the rulesets to the nodes of the control diagrams of the real systems presented in Section 1.4.1. For* `iptables` *we have:*

$$c(Pre^n) = R_{dnat} \qquad c(Out^n) = R_{dnat} \qquad c(Inp^n) = R_{snat} \qquad c(Post^n) = R_{snat}$$
$$c(Fwd^f) = R_{fil} \qquad c(Inp^f) = R_{fil} \qquad c(Out^f) = R_{fil}$$

while the remaining nodes get the empty ruleset R_ϵ*. For* `pf` *we have:*

$$c(Inp^n) = R_{dnat} \qquad c(Out^n) = R_{snat} \qquad c(Inp^f) = R_{fil} \qquad c(Out^f) = R_{fil}$$

Finally, in `ipfw`*:*

$$c(Inp) = R_{dnat} @ R_{fil} \qquad c(Out) = R_{snat} @ R_{fil}$$

1.6.2 Correctness of the Compiled Firewall

We start by showing that a compiled firewall $\mathcal{F}_C$ accepts the *same* packets as the original abstract firewall $\mathcal{F}_S$, possibly with a different translation. The differences may show up because the source and the target firewall systems have dissimilar expressivity, *e.g.*, when they impose diverse constraints on which kinds of packets can be translated and where.

Lemma 2. *Let* $\mathcal{F}_C$ *be a* compiled firewall *of* $\mathcal{F}_S$ *and let* p *be a packet, then*

$$\exists p'. \mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \exists p''. \mathcal{P}_{\mathcal{F}_C}(p, p'').$$

It is convenient introducing a few auxiliary definitions. Let $\mathcal{T} = \{id, dnat, snat, nat\}$ be the set of translations of a packet while it traverses a firewall. The first, *id*, represents the identity, *dnat* and *snat* are for DNAT and SNAT, while *nat* represents both. Also, let $(\mathcal{T}, <)$ be the partial order such that $id < dnat$, $id < snat$, $dnat < nat$ and $snat < nat$. Finally, given a packet p and a firewall $\mathcal{F}$, let $\pi_{\mathcal{F}}(p)$ be the unique path in the control diagram of $\mathcal{F}$ along which p is processed. Note that there exists a unique path for each packet because the control diagram is deterministic.

The following function computes the *translation capability* of a path π, *i.e.*, which translations can be performed on packets processed along π.

Definition 10 (Translation capability)**.** *Let $\pi = \langle q_1, \ldots, q_n \rangle$ be a path on the control diagram of a compiled firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$. The* translation capability *of π is*

$$tc(\pi) = lub \bigcup_{q_j \in \pi} \gamma(c(q_j))$$

where lub is the least upper bound operator on $(\mathcal{T}, <)$ and γ is defined as

$$\begin{aligned} \gamma(R) &= \{id\} \text{ for } R \in \{R_e, R_{fil}\} \\ \gamma(R_t) &= \{t\} \text{ for } t \in \{dnat, snat\} \\ \gamma(R_1 \,@\, R_2) &= \gamma(R_1) \cup \gamma(R_2) \end{aligned}$$

Let $p \approx p'$ hold iff $p' = p[tag \mapsto m]$ for some tag m; given a packet p and its translation p', let t_β be defined as follows, where $\beta \in \mathcal{T}$:

$$\begin{aligned} t_{id}(p, p') &= p & \qquad t_{dnat}(p, p') &= p[da \mapsto da(p')] \\ t_{nat}(p, p') &= p' & \qquad t_{snat}(p, p') &= p[sa \mapsto sa(p')] \end{aligned}$$

The following theorem describes the relationship between a compiled firewall $\mathcal{F}_C$ and the firewall $\mathcal{F}_S$. Intuitively, $\mathcal{F}_S$ accepts a packet p as p' if and only if $\mathcal{F}_C$ accepts a packet p as p'' where p' and p'' only differ on marking and NAT. More specifically, p'' is derived from p by applying all the translations available on the path $\pi_{\mathcal{F}_C}(p)$ in the control diagram of $\mathcal{F}_C$, along which p is processed.

Theorem 3. *Let p be a packet accepted by both $\mathcal{F}_S$ and $\mathcal{F}_C$; let $\beta = tc(\pi_{\mathcal{F}_C}(p))$; and let $p'' \approx t_\beta(p, p')$ for some p'. We have that*

$$\mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \mathcal{P}_{\mathcal{F}_C}(p, p'')$$

with $p' = p''$ when $\beta = nat$ or $p = p'$.

Example 5. *Consider again Example 4. Any path π in* `iptables` *has $tc(\pi) = nat$, which implies $p' \approx p''$,* i.e.*, $\mathcal{F}_C$ behaves exactly as $\mathcal{F}_S$. Interestingly, paths $\pi_1 = \langle q_i, Inp^n, Inp^f, q_o \rangle$ and $\pi_2 = \langle q_i, Out^n, Out^f, q_o \rangle$ in* `pf` *have $tc(\pi)$ equal to dnat and snat, respectively. In fact,* `pf` *cannot perform snat and dnat on packets directed to and generated from the host, respectively.*

1.7 Experimental Evaluation

We have used our tool on several policies to assess how FWS scales to real-world scenarios. Our tests focus on the first two stages of our pipeline since the compiler is still under development. We have performed our tests on a desktop PC (running Ubuntu 16.04.2) equipped with an Intel i7-3770 CPU and 16 GB of RAM.

TABLE 1.5: Tests performed on our department policy.

Analysis	Multi-cubes	Time (m:s.cs)
$N_1 \rightarrow N_2$	35	0:53.73
$N_1 \rightarrow N_3$	28	0:37.77
$N_1 \rightarrow Out$	25	1:20.65
$N_2 \rightarrow N_1$	45	0:45.32
$N_2 \rightarrow N_3$	39	0:34.27
$N_2 \rightarrow Out$	31	0:57.40
$N_3 \rightarrow N_1$	47	2:19.16
$N_3 \rightarrow N_2$	17	0:05.68
$N_3 \rightarrow Out$	8	0:09.45
$Out \rightarrow N_1$	52	6:02.08
$Out \rightarrow N_2$	10	0:11.41
$Out \rightarrow N_3$	8	0:08.12
Complete policy	138	17:09.31

1.7.1 DAIS Department Policy

The network of the DAIS department at Ca' Foscari is logically partitioned in the main network N_1, the labs network N_2 and a mixed network N_3. The firewall acts as a router between these networks and is connected to the Internet via other routers. The policy is written in `iptables`, consists of 530 rules (including both SNAT and DNAT) and contains 5 user-defined chains. In Table 1.5 we report the execution times and the sizes of the obtained specifications when running our tool on the policy projected on specific source and destination networks, as well as the time required to synthesize the entire firewall policy. The analysis on specific source and destination networks takes less that one minute most of the times and six minutes in the worst case.

1.7.2 Stanford University Backbone Network

It is a medium-sized network that contains 16 operational zone Cisco routers [174]. From the configuration files of these routers we have extracted 252 ACL policies containing 1916 filtering rules in total. Our tool separately synthesized all the policies in 2 minutes and 17.46 seconds; the largest ACL, made of 111 rules, has been analysed in 16.36 seconds and the corresponding specification consists of 12 multi-cubes. The encoding of the ACL policies in our framework has required a simple, mechanized syntactic translation from the Cisco routers configuration syntax into the intermediate language.

1.7.3 Other Real-World Policies

The authors of [54] have collected a set of anonymized `iptables` configurations from several institutions and from the Internet. Table 1.6 reports the time needed to perform a complete synthesis for a selected subset of these policies, together with their size and the number of multi-cubes of the synthesized specification.

TABLE 1.6: Tests performed on real-world policies.

Description	Rules	Multi-cubes	Time (m:s.cs)
Policy from Github	15	11	00:00.765
Ticket from OpenWRT	65	11	00:01.519
Kerberos server	8	14	00:01.635
Policy from a blog	28	25	00:02.572
Eduroam laptop	21	15	00:01.018
Memphis testbed	34	15	00:01.233
Kornwall	52	23	00:02.362
Shorewall	77	48	00:28.154
Home router	76	36	00:05.879
Medium-sized company	90	20	00:25.289
`veroneau.net`	263	7	05:55.690

TABLE 1.7: Tests performed on the *Chair for Network Architectures and Services* firewall policy.

Analysis	<1m	1-3m	3-5m	5-10m	10-20m
Subnet → Subnet	0	405	37	20	0
Subnet → Internet	14	5	1	1	0
Internet → Subnet	5	13	1	0	2

The repository also contains the firewall configuration of the lab the authors of [54] are affiliated to. The firewall has 22 network interfaces and its policy consists of 4841 `iptables` rules. We have slightly modified the policy to remove checks on MAC addresses since they are currently not supported by FWS. In Table 1.7 we provide a summary of the time required to produce a synthesis for each possible pair of input/output interfaces and to communicate with the Internet. Most of the analyses terminate in less than 3 minutes and just a couple of cases involving particularly complex subnets take more than 10 minutes to be completed.

1.7.4 Queries

We have performed some tests to evaluate the expressiveness of the output produced by FWS. For instance, in the *Home router* example, we can check which hosts in the private LAN are reachable via the public IP address of the router by running the query

```
dstIp == 117.195.222.105 && state == NEW
```

FWS succinctly reports that external hosts can access the internal server 192.168.1.130 on ports 22, 80, 443 and 1194 via DNAT. For hosts in the private LAN 192.168.1.0/24, both SNAT and DNAT are applied to connections towards the public IP address to avoid the problem of asymmetric routing using a technique known as NAT reflection. Additional examples are available online [32].

1.8 Related Work

To the best of our knowledge, ours is the first proposal providing at the same time:

1. a language for analysing multiple firewall systems;
2. an effective technique for synthesizing abstract policies;
3. support for NAT;
4. a formal characterization of firewall behaviour;
5. a technique to generate policies in multiple target languages.

In the following we briefly review the existing approaches for the analysis of existing configurations and the techniques proposed for compiling abstract filtering policies to real firewall systems.

1.8.1 Analysis of Firewall Configurations

The literature has many proposals for simplifying and analyzing firewall configurations. Some are based on formal methods, others consist of *ad hoc* configuration and analysis tools. Many works take a top-down approach to facilitate firewall management and detect misconfigurations in existing policies, such as [212, 83, 135, 183, 153, 170, 105, 63, 64]. In our work we follow a dual, bottom-up approach: we synthesize a specification from the actual firewall configuration. Below, we revise papers that take a bottom-up approach and adopt formal methods.

Some researchers focused on analyzing `iptables`: Jeffrey *et al.*introduce in [92] a formal model of firewall policy, based on `iptables`, and investigate the properties of reachability and cyclicity of firewall configurations. The proposal by Diekmann *et al.* [54] has some similarities with ours. In particular, the authors provide a "cleaned" ruleset that an automatic tool can easily analyse, using a formal semantics of `iptables` and a semantics-preserving ruleset simplification (*e.g.*, chain unfolding) with a treatment of unknown match conditions, due to a ternary logic. They give a semantics to a subset of `iptables` that includes access control flow actions, but not packet modification such as NAT. Our approach supports NAT and is based on a generic language that can target languages different from `iptables`. ITVal [118] is a tool that parses `iptables` rules and can be queried to discover host reachability. The tool is specific for `iptables` and does not aim at synthesizing an abstract firewall specification.

Other proposals in the literature are more general and target, in principle, various firewall systems. A model-driven approach is proposed in [150] to derive network access-control policies from real firewall configuration. A proof of concept is given only for `iptables`. Moreover, compared to our proposal this paper does not address NAT. In [50] the authors propose an algorithmic solution to detect and correct specific anomalies on stateful firewalls. However, the proposed approach does not aim at synthesizing an abstract specification, as we do. FIREMAN [210] is a tool that detects inconsistencies and

inefficiencies of firewall policies, but it does not provide support for NAT. In [133] the Margrave policy analyser is applied to the analysis of IOS firewalls. The approach is rather general and extensible to other languages, however the analysis focuses on finding specific problems in policies rather then synthesizing a high-level policy specification. A framework for the static analysis of networks is proposed in [100]. It provides sophisticated insights about network configurations but does not specifically analyse real firewall configurations and, as for the previous papers, there is no synthesis of high-level specifications. Fang [119] is another tool for querying real policies in order to discover anomalies. Authors state that it synthesizes an abstract policy that resembles the one we propose here, but the tool is unavailable and the paper does not describe the tool internals, making any comparison with our approach impossible.

Jayaraman *et al.* [88] propose an approach for validating network connectivity policies, implemented by the tool SECGURU. They extract logical specifications from real Cisco IOS routers and solve them in Z3. In our paper we have extended their approach under two main aspects:

1. we provide support for NAT. This is non trivial and required to model logical predicates on pairs of packets as discussed in Section 1.5.2;

2. we perform our analysis on a generic language that can be used to represent various real configuration languages, by taking into account the platform-depended packet processing flow.

1.8.2 Compilation of Firewall Configurations

Compared to the analysis of existing configurations, less effort has been devoted to the development of approaches aimed at compiling high-level policy specifications into real firewall systems.

In [17] Bartal *et al.* propose Firmato, a firewall management toolkit that allows to specify an abstract model of the desired security policy and compiles it into firewall-specific configuration rules. The authors claim that their approach is general enough to support multiple existing systems, though they have implemented only a back-end for the Lucent Managed Firewall. Similarly, Adão *et al.* [3] present a declarative language to specify firewall policies and a compiler for `iptables`. More generally, NetKat [5] proposes linguistic constructs for programming a network as a whole within the SDN paradigm.

All these approaches propose their own high-level language with a formal semantics, and then compile it to a specific target language (*cf.* our stage 3). Instead, IFCL intermediates between real source and target languages. It thus takes from real languages actions both for filtering/rewriting packets (notably NAT and MARK) and for controlling the inspection flow, widely used in practice.

Chapter 2

Surviving the Web: A Journey into Web Session Security

2.1 Introduction

The Web is the primary access point to on-line data and applications. It is extremely complex and variegate, as it integrates a multitude of dynamic contents by different parties to deliver the greatest possible user experience. This heterogeneity makes it very hard to effectively enforce security, since putting in place novel security mechanisms typically prevents existing websites from working correctly or negatively affects the user experience, which is generally regarded as unacceptable, given the massive user base of the Web. However, this continuous quest for usability and backward compatibility had a subtle effect on web security research: designers of new defensive mechanisms have been extremely cautious and the large majority of their proposals consists of very local patches against very specific attacks. This piecemeal evolution hindered a deep understanding of many subtle vulnerabilities and problems, as testified by the proliferation of different threat models against which different proposals have been evaluated, occasionally with quite diverse underlying assumptions. It is easy to get lost among the multitude of proposed solutions and almost impossible to understand the relative benefits and drawbacks of each single proposal without a full picture of the existing literature.

In this chapter we take the delicate task of performing a systematic overview of a large class of common attacks targeting the current Web and the corresponding security solutions proposed so far. We focus on attacks against *web sessions*, *i.e.*, attacks which target honest web browser users establishing an authenticated session with a trusted web application. This kind of attacks exploits the intrinsic complexity of the Web by tampering, *e.g.*, with dynamic contents, client-side storage or cross-domain links, so as to corrupt the browser activity and/or network communication. Our choice is motivated by the fact that attacks against web sessions cover a very relevant subset of serious web security incidents [148] and many different defenses, operating at different levels, have been proposed to prevent these attacks.

We consider typical attacks against web sessions and we systematise them based on: (i) their attacker model and (ii) the security properties they break. This first classification is useful to understand precisely which intended security properties of a web session can

be violated by a certain attack and how. We then survey existing security solutions and mechanisms that prevent or mitigate the different attacks and we evaluate each proposal with respect to the security guarantees it provides. When security is guaranteed only under certain assumptions, we make these assumptions explicit. For each security solution, we also evaluate its impact on both *compatibility* and *usability*, as well as its *ease of deployment*. These are important criteria to judge the practicality of a certain solution and they are useful to understand to which extent each solution, in its current state, may be amenable for a large-scale adoption on the Web. Since there are several proposals in the literature which aim at providing robust safeguards against multiple attacks, we also provide an overview of them in a separate section. For each of these proposals, we discuss which attacks it prevents with respect to the attacker model considered in its original design and we assess its adequacy according to the criteria described above.

Finally, we synthesize from our survey a list of five guidelines that, to different extents, have been taken into account by the designers of the different solutions. We observe that none of the existing proposals follows all the guidelines and we argue that this is due to the high complexity of the Web and the intrinsic difficulty in securing it. We believe that these guidelines can be helpful for the development of innovative solutions approaching web security in a more systematic and comprehensive way.

2.1.1 Scope of the Work

Web security is complex and web sessions can be attacked at many different layers. To clarify the scope of the present work, it is thus important to discuss some assumptions we make and their import on security:

1. *perfect cryptography*: at the network layer, web sessions can be harmed by network sniffing or man-in-the-middle attacks. Web traffic can be protected using the HTTPS protocol, which wraps the traffic within a SSL/TLS encrypted channel. We do not consider attacks to cryptographic protocols. In particular, we assume that the attacker cannot break cryptography to disclose, modify or inject the contents sent to a trusted web application over an encrypted channel. However, we do not assume that HTTPS is always configured correctly by web developers, since this is quite a delicate task, which deserves to be discussed in the present survey;

2. *the web browser is not compromised by the attacker*: web applications often rely on the available protection mechanisms offered by standard web browsers, like the same-origin policy or the `HttpOnly` cookie attribute. We assume that all these defenses behave as intended and the attacker does not make advantage of browser exploits, otherwise even secure web applications would fail to be protected;

3. *trusted web applications may be affected by content injection vulnerabilities*: this is a conservative assumption, since history teaches us that it is almost impossible to guarantee that a web application does not suffer from this kind of threats. We focus on content injection vulnerabilities which ultimately target the web browser, like

cross-site scripting attacks (XSS). Content injections affecting the backend of the web application, like SQL injections, are not covered.

2.1.2 Structure of the Chapter

Section 2.2 provides some background on the main building blocks of the Web. Section 2.3 presents the attacks. Section 2.4 classifies attack-specific solutions with respect to their security guarantees, their level of usability, compatibility and ease of deployment. Section 2.5 carries out a similar analysis for defenses against multiple attacks. Section 2.6 presents five guidelines for future web security solutions.

2.2 Background

We provide a brief overview of the basic building blocks of the web ecosystem and their corresponding security cornerstones.

2.2.1 Languages for the Web

Documents on the Web are provided as *web pages*, hypertext files connected to other documents via hyperlinks. Web pages embody several languages affecting different aspects of the documents. The *Hyper Text Markup Language* (HTML) [197] or a comparable markup language (*e.g.*, XHTML) defines the structure of the page and the elements it includes, while *Cascading Style Sheets* (CSS) [190] are used to add style information to web pages (*e.g.*, fonts, colors, position of elements).

JavaScript [57] is a programming language which allows the development of rich, interactive web applications. JavaScript programs are included either directly in the web page (*inline* scripts) or as external resources, and can dynamically update the contents in the user browser by altering the *Document Object Model* (DOM) [194, 195, 196], a tree-like representation of the web page. Page updates are typically driven by user interaction or by asynchronous communications with a remote web server based on *AJAX* requests (via the XMLHttpRequest API).

2.2.2 Locating Web Resources

Web pages and the contents included therein are hosted on *web servers* and identified by a *Uniform Resource Locator* (URL). A URL specifies both the location of a resource and a mechanism for retrieving it. A typical URL includes: (1) a *protocol*, defining how the resource should be accessed; (2) a *host*, identifying the web server hosting the resource; and (3) a *path*, localizing the resource at the web server.

Hosts belong to *domains* identifying an administrative realm on the Web, typically controlled by a specific company or organization. Domain names are organised hierarchically: subdomain names can be defined from a domain name by prepending it a string, separated by a period. For example, the host `www.google.com` belongs to the domain `google.com` which is a subdomain of the top-level domain `com`.

Domains that are controlled by domain registrars are known as *public suffixes*. Internet users can only register names that are subdomains of a public suffix. For instance, users cannot register the domain `co.uk` since it is a public suffix, but they can register `example.co.uk`. Browsers vendors maintain a publicly available list to keep track of all existing public suffixes [129]. Two domains are said to be *related* if the longest common suffix they share is not a public suffix. For instance, domains `www.example.com` and `atk.example.com` are related, while `example.co.uk` and `attacker.co.uk` are not.

2.2.3 Hyper Text Transfer Protocol (HTTP)

Web contents are requested and served using the *Hyper Text Transfer Protocol* (HTTP), a text-based request-response protocol based on the client-server paradigm. The client (browser) initiates the communication by sending an HTTP request for a resource hosted on the server; the server, in turn, provides an HTTP response containing the completion status information of the request and its result. HTTP defines *methods* to indicate the action to be performed on the identified resource, the most important ones being GET and POST. GET requests should only retrieve data and have no other import, while side-effects at the server-side should only be triggered by POST requests, though web developers do not always comply with this convention. Both GET and POST requests may include custom parameters which can be processed by the web server.

HTTP is a *stateless* protocol, *i.e.*, it treats each request as independent from all the others. Some applications, however, need to remember information about previous requests, for instance to track whether a user has already authenticated and grant her access to her profile page. HTTP *cookies* are the most widespread mechanism employed on the Web to maintain state information about the requesting clients [18]. Roughly, a cookie is a key-value pair, which is set by the server into the client and automatically attached by it to all subsequent requests to the server. Cookies can be set via the `Set-Cookie` HTTP header or by using JavaScript. Cookies may also have *attributes* which restrict the way the browser handles them (*cf.* Section 2.2.4).

2.2.4 Security Cornerstones and Subtleties

HTTPS

Since all the HTTP traffic flows in the clear, the HTTP protocol does not guarantee several desirable security properties, such as the confidentiality and the integrity of the communication, and the authenticity of the involved parties. To protect the exchanged data, the *HTTP Secure* (HTTPS) protocol [155] wraps plain HTTP traffic within a SSL/TLS encrypted channel. A web server may authenticate itself at the client by using public key certificates; when the client is unable to verify the authenticity of a certificate, a warning message is displayed and the user can decide whether to proceed with the communication or not.

Mixed Content Websites

A *mixed content* page is a web page that is received over HTTPS, but loads some of its contents over HTTP. The browser distinguishes two types of contents depending on their capabilities on the including page: *passive contents* like images, audio tracks or videos cannot modify other portions of the page, while *active contents* like scripts, frames or stylesheets have access to (parts of) the DOM and may be exploited to alter the page. While the inclusion of passive contents delivered over HTTP into HTTPS pages is allowed by the browser, active mixed contents are blocked by default [198].

Same-Origin Policy

The *same-origin policy* (SOP) [128] is a standard security policy implemented by all major web browsers: it enforces a strict separation between contents provided by unrelated sites, which is crucial to ensure their confidentiality and integrity. SOP allows scripts running in a first web page to access data in a second web page only if the two pages have the same *origin*. An origin is defined as the combination of a protocol, a host and a port number [19]. SOP applies to many operations in the browser, most notably DOM manipulations and cookie accesses. However, some operations are not subject to same-origin checks, *e.g.*, cross-site inclusion of scripts and submission of forms are allowed, thus leaving space to potential attacks.

Cookies

Cookies use a separate definition of origin, since cookies set for a given domain are normally shared across all the ports and protocols on that domain. By default, cookies set by a page are only attached by the browser to requests sent to the same domain of the page. However, a page may also set cookies for a parent domain via the `Domain` cookie attribute, as long as the parent domain is not a public suffix: these cookies are shared between the parent domain and all its subdomains, and we refer to them as *domain cookies*.

Cookies come with two security mechanisms: the `Secure` attribute identifies cookies which must only be sent over HTTPS, while the `HttpOnly` attribute marks cookies which cannot be accessed via non-HTTP APIs, *e.g.*, via JavaScript. Perhaps surprisingly, the `Secure` attribute does not provide integrity guarantees, since secure cookies can be overwritten over HTTP [18].

2.3 Attacking Web Sessions

A *web session* is a semi-permanent information exchange between a browser and a web server that involves multiple HTTP(S) requests and responses and is bound to a user identity known to the server. As anticipated, stateful sessions on the Web are typically identified by one or more *cookies* stored in the user browser. When the user authenticates to a website by providing some valid credentials, *e.g.*, a username-password pair, a fresh

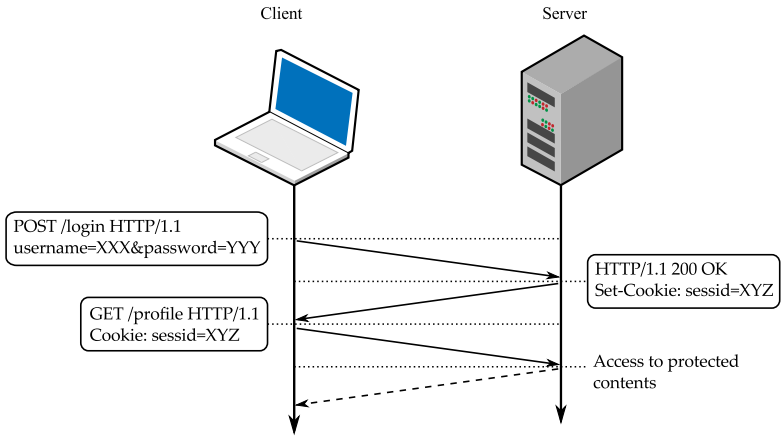


FIGURE 2.1: Cookie-based User Authentication.

cookie is generated by the server and sent back to the browser. Further requests originating from the browser automatically include the cookie as a proof of being part of the session established upon password-based authentication. This common authentication scheme is depicted in Figure 2.1.

Since the cookie essentially plays the role of the password in all the subsequent requests to the web server, it is enough to discover its value to hijack the session and fully impersonate the user, with no need to compromise the low level network connection or the server. We call *authentication cookie* any cookie which identifies a web session.

2.3.1 Security Properties

We consider two standard security properties formulated in the setting of web sessions. They represent typical targets of web session attacks:

- *Confidentiality*: data transmitted inside a session should not be disclosed to unauthorized users;
- *Integrity*: data transmitted inside a session should not be modified or forged by unauthorized users.

Interestingly, the above properties are not independent and a violation of one might lead to the violation of the other. For example, compromising session confidentiality might reveal authentication cookies, which would allow the attacker to perform arbitrary actions on behalf of the user, thus breaking session integrity. Integrity violations, instead, might cause the disclosure of confidential information, *e.g.*, when sensitive data is leaked via a malicious script injected in a web page by an attacker.

2.3.2 Threat Model

We focus on two main families of attackers: *web attackers* and *network attackers*. A web attacker controls at least one web server that responds to any HTTP(S) requests sent to it with arbitrary malicious contents chosen by the attacker. We assume that a web attacker

can obtain trusted HTTPS certificates for all the web servers under his control and is able to exploit content injection vulnerabilities on trusted websites. A slightly more powerful variation of the web attacker, known as the *related-domain attacker* [35], can host malicious web pages on a related domain of the target website. This means in particular that the attacker can set (domain) cookies for the target website [18]. These cookies are indistinguishable from other cookies set by the target website and are automatically sent to the latter by the browser. Hereafter, we explicitly distinguish a related-domain attacker from a standard web attacker only when the specific setting is relevant to carry out an attack.

Network attackers subsume the capabilities of traditional web attackers and are also capable to inspect, forge and corrupt all the HTTP traffic sent on the network, as well as the HTTPS traffic which does not make use of certificates signed by a trusted certification authority. It is common practice in web security to distinguish between *passive* and *active* network attackers, with the first ones lacking the ability of forging or corrupting the unprotected network traffic. From now on, when generically speaking about network attackers, we implicitly refer to active network attackers.

2.3.3 Web Attacks

Content Injection

This wide class of attacks allows a web attacker to inject harmful contents into trusted web applications. Content injections can be mounted in many different ways, but they are always enabled by an improper or missing sanitization of some attacker-controlled input in the web application, either at the client side or at the server-side. These attacks are traditionally assimilated to Cross-Site Scripting (XSS), *i.e.*, injections of malicious JavaScript code; however, the lack of a proper sanitization may also affect HTML contents (markup injection) or even CSS rules [211, 82].

To exemplify how an XSS works, consider a website `vuln.com` hosting a simple search engine. Queries are performed via a GET request including a search parameter which is shown in the result page headline "`Search results for foo:`", where `foo` is the value of the search parameter. An attacker can then attempt to inject contents into `vuln.com` just by providing to the user a link including a script as the search term. If the search page does not properly sanitize such an input, the script will be included in the headline of the results page and it will run on behalf of `vuln.com`, thus allowing the attacker to sidestep SOP: for instance, the injected script will be entitled to read the authentication cookies set by `vuln.com`.

XSS attacks are usually classified as either *reflected* or *stored*, depending on the persistence of the threat. Reflected XSS attacks correspond to cases like the one above, where part of the input supplied by the request is "reflected" into the response without proper sanitization. Stored XSS attacks, instead, are those where the injected script is permanently saved on the target server, *e.g.*, in a message appearing on a discussion board. The script is then automatically executed by any browser visiting the attacked page.

Security properties: since content injections allow an attacker to elude SOP, which is the baseline security policy of standard web browsers, they can have catastrophic consequences on both the confidentiality and the integrity of a web session. Specifically, they can be used to steal sensitive data from trusted websites, such as authentication cookies and user credentials, and to actively corrupt the page contents, so as to undermine the integrity of a web session.

Cross-Site Request Forgery (CSRF)

A CSRF is an instance of the "confused deputy" problem [79] in the context of web browsing. In a CSRF, the attacker forces the user browser into sending HTTP(S) requests to a website where the user has already established an authenticated session: it is enough for the attacker to include HTML elements pointing to the vulnerable website in his own web pages. When rendering or accessing these HTML elements, the browser will send HTTP(S) requests to the target website and these requests will automatically include the authentication cookies of the user. From the target website perspective, these forged requests are indistinguishable from legitimate ones and thus they can be abused to trigger a dangerous side-effect, *e.g.*, to force a bank transfer from the user account to the attacker account. Notably, the attacker can forge these malicious requests without any user intervention, *e.g.*, by including in a page under his control some `<img>` tags or a hidden HTML form submitted via JavaScript.

Security properties: a CSRF attack allows the attacker to inject an authenticated message into a session with a trusted website, hence it constitutes a threat to session integrity. It is less known that CSRFs may also be employed to break confidentiality by sending cross-site requests that return sensitive user data bound to the user session. Normally, SOP (Section 2.2.4) prevents a website from reading responses returned by a different site, but websites may explicitly allow cross-site accesses using the Cross-Origin Request Sharing (CORS) standard [193] or mechanisms like JSON with Padding (JSONP) [85] which can be abused to break session confidentiality. For instance, a CSRF attack leaking the stored files has been reported on the cloud service SpiderOak [13].

Login CSRF

A peculiar instance of CSRF, known as login CSRF, is a subtle attack first described by Barth *et al.*, where the victim is forced to interact with the target website within the attacker session [20]. Specifically, the attacker uses his own credentials to silently log in the user browser at the target website, for instance by forcing it into submitting an invisible login form. The outcome of the attack is that the user browser is forced into an *attacker* session: if the user is not careful, she might be tricked into storing sensitive information, like her credit card number, into the attacker account.

Security properties: though this attack does not compromise existing sessions, it fools the browser into establishing a new attacker-controlled (low integrity) session with a

trusted website. Login CSRFs may enable confidentiality violations in specific application scenarios, like in the credit card example given above.

Cookie Forcing

A web attacker exploiting a code injection vulnerability may directly impose his own authentication cookies in the victim browser, thus forcing it into the attacker session and achieving the same results of a successful login CSRF, though exploiting a different attack vector. Related-domain attackers are in a privileged position for these attacks, since they can set cookies for the target website from a related-domain host.

Security properties: see login CSRF above.

Session Fixation

A session fixation attack allows an attacker to impersonate a user by imposing in the user browser a known session identifier, which is not refreshed upon successful authentication with the vulnerable website. Typically, the attacker first contacts the target site and gets a valid cookie which is then set (*e.g.*, via an XSS attack on the site) into the user browser *before* the initial password-based authentication step is performed. If the website does not generate a fresh cookie upon authentication, the user session will be identified by a cookie known to the attacker. Related-domain attackers have easy access to these attacks, since they can set cookies on behalf of the victim website.

Security properties: by letting the attacker fully impersonate the user at the target website, session fixation harms both the confidentiality and the integrity of the user session, just as if the authentication cookies were disclosed to the attacker.

2.3.4 Network Attacks

Though network attacks are arguably more difficult to carry out on the Web than standard web attacks, they typically have a tremendous impact on both the confidentiality and the integrity of the user session. Since the HTTP traffic is transmitted in clear, a network attacker, either passive or active, can eavesdrop sensitive information and compromise the confidentiality of HTTP sessions. Websites which are served on HTTP or on a mixture of HTTPS and HTTP are prone to expose non-secure cookies or user credentials to a network attacker: in these cases, the attacker will be able to fully impersonate the victim at the target website. An active network attacker can also mount man-in-the-middle attacks via *e.g.*, ARP spoofing, DNS cache poisoning or by setting up a fake wi-fi access point. By interposing himself between the victim and the server, this attacker can arbitrarily modify HTTP requests and responses exchanged by the involved parties, thus breaking the confidentiality and the integrity of the session. Also, active network attackers can compromise the integrity of cookies [18].

A notable example of network attack is *SSL stripping* [117], which is aimed at preventing web applications from switching from HTTP to HTTPS. The attack exploits the

fact that the initial connection to a website is typically initiated over HTTP and the protocol upgrade is done through HTTP redirect messages, links or HTML forms targets. By corrupting the first server response, an active attacker can force the session in clear by replacing all the HTTPS references with their HTTP version and then forward the traffic received by the user to the real web server, possibly over HTTPS. The same operation will then be performed for each request/response in the session, hence the web application will work seamlessly, but the communication will be entirely under the control of the attacker. This attack is particularly subtle, since the user might fail to notice the missing usage of HTTPS, which is only notified by some components in the user interface of the browser (*e.g.*, a padlock icon).

2.4 Protecting Web Sessions

2.4.1 Evaluation Criteria

We evaluate existing defenses along four different axes:

1. *protection*: we assess the effectiveness of the proposed defense against the conventional threat model of the attack, *e.g.*, the web attacker for CSRF. If the proposal does not prevent the attack in the most general case, we discuss under which assumptions it may still be effective;

2. *usability*: we evaluate whether the proposed mechanism affects the end-user experience, for instance by impacting on the perceived performances of the browser or by involving the user into security decisions;

3. *compatibility*: we discuss how well the defense integrates into the web ecosystem with respect to the current standards, the expected functionalities of websites, and the performances provided by modern network infrastructures. For example, solutions that prevent some websites from working correctly are not compatible with the existing Web. On the other hand, a minor extension to a standard protocol which does not break backward compatibility, such as the addition of new HTTP headers that can be ignored by recipients not supporting them, is acceptable;

4. *ease of deployment*: we consider how practical would be a large-scale deployment of the defensive solution by evaluating the overall effort required by web developers and system administrators for its adoption. If they have to pay an unacceptably high cost, the solution will likely never be deployed on a large scale.

We deem a negative impact on server-side performances as a compatibility problem rather than a usability problem when the overall response time can be kept constant by increasing the computational resources of the server, thus keeping the user experience unaffected. To provide a concise yet meaningful evaluation of the different proposals, usability, compatibility and ease of deployment are assigned a score from a three-levels scale: Low, Medium and High. Table 2.1 provides the intuition underlying these scores.

	Usability	*Compatibility*	*Ease of Deployment*
Low	Users must take several security decisions	The correct functioning of some websites is precluded	Applications need to be heavily rewritten, complex security policies must be deployed
Medium	Perceivable slowdown of performances that affects the client	Moderate increase of the server workload	Moderate server-side modifications, small declarative policies have to be written
High	The user experience is not affected in any way	The defense fits the web ecosystem, no impact on server workload	The protection can be enabled just by installing an additional component or by minimal server-side modifications

TABLE 2.1: Evaluation Criteria.

We exclude from our survey several solutions which would require major changes to the current Web, such as new communication protocols or authentication mechanisms replacing cookies and passwords [96, 77, 171, 52].

2.4.2 Content Injection: Mitigation Techniques

Given the critical impact of content injection attacks, there exist many proposals which focus on them. In this section we discuss those solutions which do not necessarily prevent a content injection, but rather mitigate its malicious effects, *e.g.*, by thwarting the leakage of sensitive data.

HttpOnly Cookies

`HttpOnly` cookies have been introduced in 2002 with the release of Internet Explorer 6 SP1 to prevent the theft of authentication cookies via content injection attacks. Available on all major browsers, this simple yet effective mechanism limits the scope of cookies to HTTP(S) requests, making them unavailable to malicious JavaScript injected in a trusted page.

The protection offered by the `HttpOnly` attribute is only limited to the theft of authentication cookies. The presence of the attribute is transparent to users, hence it has no usability import. Also, the attribute perfectly fits the web ecosystem in terms of compatibility with legacy web browsers, since unknown cookie attributes are ignored. Finally, the solution is easy to deploy, assuming there is no need of accessing authentication cookies via JavaScript for generic reasons [216].

SessionShield and Zan

SessionShield [137] is a client-side proxy preventing the leakage of authentication cookies via XSS attacks. It operates by automatically identifying these cookies in incoming response headers, stripping them from the responses, and storing them in a private

database inaccessible to scripts. SessionShield then reattaches the previously stripped cookies to outgoing requests originating from the client to preserve the session. A similar idea is implemented in Zan [178], a browser-based defense which (among other things) automatically applies the `HttpOnly` attribute to the authentication cookies detected through the usage of a heuristic. As previously discussed, `HttpOnly` cookies cannot be accessed by JavaScript and will only be attached to outgoing HTTP(S) requests.

The protection offered by SessionShield and Zan is limited to the improper exfiltration of authentication cookies. These defenses do not prompt the user with security decisions, neither slow down perceivably the processing of web pages, hence they are fine from a usability point of view. However, the underlying heuristic for detecting authentication cookies poses some compatibility concerns, since it may break websites when a cookie is incorrectly identified as an authentication cookie and made unavailable to legitimate scripts that need to access it. Both SessionShield and Zan are very easy to deploy, given their purely client-side nature.

Request Filtering Approaches

Noxes is one of the first developed client-side defenses against XSS attacks [104]. It is implemented as a web proxy installed on the user machine, aimed at preserving the confidentiality of sensitive data in web pages, such as authentication cookies and session IDs. Instead of blocking malicious script execution, Noxes analyses the pages fetched by the user in order to allow or deny outgoing connections on a whitelist basis: only local references and static links embedded into a page are automatically considered safe with respect to XSS attacks. For all the other links, Noxes resorts to user interaction to take security decisions which can be saved either temporarily or permanently. Inspired by Noxes, Vogt *et al.* introduce a modified version of Firefox [188] where they combine dynamic taint tracking and lightweight static analysis techniques to track the flow of a set of sensitive data sources (*e.g.*, cookies, document URLs) within the scripts included in a page. When the value of a tainted variable is about to be sent to a third-party domain, the user is required to authorize or deny the communication.

The protection offered by these approaches is not limited to authentication cookies, but it prevents the exfiltration of arbitrary sensitive data manipulated by web pages. According to the authors, the solutions are not affected by performance problems, however Noxes still suffers from usability issues, as it requires too much user interaction given the high number of dynamic links in modern web pages [137]. The modified Firefox in [188] attempts to lower the number of security questions with respect to Noxes, but still many third-party domains such as `.google-analytics.com` should be manually whitelisted to avoid recurring alert prompts. On the other hand, due to the fine-grained control over the filtering rules, both mechanisms are deemed compatible, assuming that the user takes the correct security decisions. Both solutions are easy to deploy, since no server-side modification is required and users simply need to install an application on their machines.

Critical Evaluation

The exfiltration of sensitive data is a typical goal of content injection attacks. Preventing authentication cookie stealing is simple nowadays, given that the `HttpOnly` attribute is well supported by all modern browsers, and several languages and web frameworks allow the automatic enabling of the attribute for all the authentication cookies [147]. Conversely, solutions aimed at providing wider coverage against general data leakage attacks never gained popularity, mainly due to their impact on the user experience.

2.4.3 Content Injection: Prevention Techniques

While the proposals discussed in the previous section are designed to block leakages of sensitive data, the defenses presented in this section attempt to prevent the execution of malicious contents injected into web pages.

Client-side Filtering

XSS filters like IE XSS Filter [157] and WebKit XSSAuditor [21] are useful to prevent reflected XSS attacks. Before interpreting the JavaScript code in a received page, these client-side filters check whether potentially dangerous payloads included in the HTTP request are also found within the response body: if a match is detected, the payload is typically stripped from the rendered page without asking for user intervention. The NoScript extension for Firefox [116] applies an even stricter policy, since it directly prevents script execution, thus blocking both stored and reflected XSS attacks. This policy can be relaxed on selected domains, where only XSS filtering mechanisms are applied.

XSS filtering proved to be quite effective in practice, despite not being always able to prevent all the attacks. A typical example is a web application which takes a `base64` encoded string via a GET variable and includes the decoded result in the generated page: an attacker may easily bypass the XSS filter by supplying the `base64` encoding of a malicious JavaScript which will, in turn, be decoded by the server and included in the response body. Additionally, XSS filters have also been exploited to introduce new flaws in otherwise secure websites, *e.g.*, by disabling legitimate scripts found in the original pages [132, 94].

The filtering approach against reflected XSS attacks showed no negative impact on the user experience and a good compatibility with modern web applications. Indeed, IE XSS Filter and WebKit XSSAuditor have been included in major browsers. The additional security features offered by NoScript however come at a cost on usability, since the user is involved in the process of dynamically populating the whitelist of the extension whenever a blocked script is required to preserve the functionality of the website. Nevertheless, it is possible to relax the behaviour of NoScript to improve the user experience, by configuring the extension so that it only applies filtering against reflected XSS attacks.

Server-side Filtering

An alternative to the in-browser filtering approach is to perform attack detection on the server-side. Xu *et al.* present a method based on fine-grained taint tracking analysis [206] which improves an earlier solution named CSSE [152]. This approach is designed to prevent a variety of attacks including content injections. The idea is to apply a source-to-source transformation of server-side C programs to track the flow of potentially malicious input data and enforce taint-enhanced security policies. By marking every byte of the user input as tainted, reflected XSS attacks can be prevented by policies that forbid the presence of tainted dangerous HTML tag patterns inside the web application output.

The protection offered by this approach and its ease of deployment crucially depend on the enforced security policy. A simple policy preventing user-provided `<script>` tags from appearing in the web page is trivial to write, but ineffective against more sophisticated attacks. However, writing a more comprehensive set of rules while maintaining the full functionalities of websites is considered a challenging task [110]. The existence of ready-to-use policies would make it easier to apply the security mechanism. Still, server modifications are required to enable support for the protection mechanism on the script language engine, which brings a significant performance overhead on CPU intensive applications, reported to be between 50% and 100%. This partially hinders both compatibility and ease of deployment.

XSS-Guard

The idea of server-side source-to-source program transformation is also employed in XSS-Guard [28], a solution for Java applications aimed at distinguishing malicious scripts reflected into web pages from legitimate ones. For each incoming request, the rewritten application generates two pages: the first includes the original user input, while the second is produced using input strings not including harmful characters (*e.g.*, sequences of `A`'s). The application checks the equivalence of the scripts contained in the two pages by string matching or, in case of failure, by comparing their syntactic structure. Additional or modified scripts found within the real page are considered malicious and stripped from the page returned to the user.

The protection offered by XSS-Guard is good, but limited to reflected XSS attacks. Moreover, since the script detection procedure is borrowed from the Firefox browser, some quirks specific to other browsers may allow to escape the mechanism. However, XSS-Guard is usable, since the browsing experience is not affected by its server-side adoption. The performance overhead caused by the double page generation ranges from 5% to 24%, thus increasing the server workload: this gives rise to some concerns about compatibility. On the other hand, enabling the solution on existing Java programs is simple, since no manual code changes are required and web developers only need to automatically translate their applications.

BEEP

Browser-Enforced Embedded Policies (BEEP) [93] is a hybrid client-server approach that hinges on the assumption that web developers have a precise understanding of which scripts should be trusted for execution. Websites provide a filtering policy to the browser in order to allow the execution of trusted scripts only, thus blocking any malicious scripts injected into the page. The policy is embedded in web pages through a specific JavaScript function which is invoked by a specially-modified browser every time a script is found during the parsing phase. This function accepts as parameters the code and the DOM element of the script and returns a boolean value which determines whether the execution is allowed or not.

The proposed mechanism exhibits some security defects, as shown in [11]. For instance, an attacker may reuse whitelisted scripts in an unanticipated way to alter the behaviour of the application. Regarding usability, the adoption of this solution may cause some slowdowns at the client-side when accessing websites which heavily rely on inline JavaScript contents. Compatibility however is preserved, since browsers not compliant with BEEP will still render pages correctly without the additional protection. The deployment of BEEP is not straightforward, since the effort required to modify existing web applications to implement the security mechanism depends on the complexity of the desired policy.

Blueprint

Blueprint [110] tackles the problem of denying malicious script execution by relieving the browser from parsing untrusted contents: indeed, the authors argue that relying on the HTML parsers of different browsers is inherently unsafe, due to the presence of numerous browser quirks. In this approach, web developers annotate the parts of the web application code which include a block of user-provided content in the page. For each block, the server builds a parse tree of the user input, stripped of all the dynamic contents (*e.g.*, JavaScript, Flash). This sanitized tree is encoded as a `base64` string and included in the page within an invisible `<code>` block. This `base64` data is then processed by a client-side JavaScript which is in charge of reconstructing the DOM of the corresponding portion of the page.

Despite providing strong protection against stored and reflected XSS attacks, Blueprint suffers from performance issues which impact on both usability and compatibility [202]. Specifically, the server workload is increased by a 35%-55% due to the parse tree generation, while the page rendering time is significantly affected by the amount of user contents to be dynamically processed by the browser. Also, Blueprint requires a considerable deployment effort, since the web developer must manually identify and update all the code portions of web applications that write out the user input.

Noncespaces

Along the same line of research, Noncespaces [75] is a hybrid approach that allows web clients to distinguish between trusted and untrusted contents to prevent content injection attacks. This solution provides a policy mechanism which enables web developers to declare granular constraints on elements and attributes according to their trust class. All the (X)HTML tags and attributes are associated to a specific trust class by automatically enriching their names with a random string, generated by the web application, that is unknown to the attacker. In case of XHTML documents, the random string is applied as a namespace prefix (`<r617:h1 r617:id=‘‘Title’’> Title </r617:h1>`), while in the HTML counterpart the prefix is simply concatenated (`<r617h1 r617id=‘‘Title’’> Title </r617h1>`). The server sends the URL of the policy and the mapping between trust classes and random strings via custom HTTP headers. A proxy installed on the user machine validates the page according to the policy and returns an empty page to the browser in case of violations, *i.e.*, if the page contains a tag or attribute with a random string which is invalid or bound to an incorrect trust class.

The solution is an improvement over BEEP in preventing stored and reflected XSS. Since random prefixes are not disclosed to the attacker, Noncespaces is not affected by the exploits introduced in [11]. Additionally, the mechanism allows web developers to permit the inclusion of user-provided HTML code in a controlled way, thus offering protection also against markup injections. Although the impact on server-side performance is negligible, the policy validation phase performed by the proxy on the client-side introduces a noticeable overhead which may range from 32% to 80%, thus potentially affecting usability. Furthermore, though Noncespaces can be safely adopted on XHTML websites, it is affected by compatibility problems on HTML pages, due to the labelling process which disrupts the names of tags and attributes, and thus the page rendering, on unmodified browsers. Web developers are required to write security policies and revise web applications to support Noncespace, hence the ease of deployment depends on the granularity of the enforced policy.

DSI

In parallel with the development of Noncespaces, Nadji *et al.* proposed a similar solution based on the concept of document structure integrity (DSI) [130]. The approach relies on server-side taint-tracking to mark nodes generated by user-inserted data, so that the client is able to recognize and isolate them during the parsing phase to prevent unintended modifications to the document structure. Untrusted data is delimited by special markers, *i.e.*, sequences of randomly chosen Unicode whitespace characters. These markers are shipped to the browser in the `<head>` section of the requested page along with a simple policy which specifies the allowed HTML tags within untrusted blocks. The policy enforcement is performed by a modified browser supporting the security mechanism which is also able to track dynamic updates to the document structure.

This solution shares with Noncespaces a similar degree of protection. Nevertheless, from a performance standpoint, the defense introduces only a limited overhead on the client-side, since the policies are simpler with respect to Noncespaces and the enforcement mechanism is integrated in the browser instead of relying on an external proxy. As a result, the user experience is not affected. Compatibility is preserved, given that the labelling mechanism does not prevent unmodified browsers from rendering correctly DSI-enabled web applications. Finally, even the deployment is simplified, since no changes to the applications are required and the policy language is more coarse grained than the one proposed in Noncespaces.

Content Security Policy

The aforementioned proposals share the idea of defining a client-side security policy [202]. The same principle is embraced by the Content Security Policy (CSP) [191], a web security policy standardized by the W3C and adopted by all major browsers. CSP is deployed via an additional HTTP response header and allows the specification of the trusted origins from which the browser is permitted to fetch the resources included in the page. The control mechanism is fairly granular, allowing one to distinguish between different types of resources, such as JavaScript, CSS and XHR targets. By default, CSP does not allow inline scripts and CSS directives (which can be used for data exfiltration) and the usage of particularly harmful JavaScript functions (*e.g.*, `eval`). However, these constraints can be disabled by using the `'unsafe-inline'` and the `'unsafe-eval'` rules. With the introduction of CSP Level 2 [192], it is possible to selectively white-list inline resources without allowing indiscriminate content execution. Permitted resources can be identified in the policy either by their hashes or by random nonces included in the web page as attributes of their enclosing tags.

When properly configured, CSP provides an effective defense against XSS attacks. Still, general content injection attacks, such as markup code injections, are not prevented. CSP policies are written by web developers and transparent to users, so their design supports usability. Compatibility and deployment cost are better evaluated together for CSP. On the one hand, it is easy to write a very lax policy which allows the execution of inline scripts and preserves the functionality of web applications by putting only mild restrictions on cross-origin communication: this ensures compatibility. On the other hand, an effective policy for legacy applications can be difficult to deploy, since inline scripts and styles should be removed or manually white-listed, and trusted origins for content inclusion should be carefully identified [202]. As of now, the deployment of CSP is not particularly significant or effective [204, 40]. That said, the standardization of CSP by the W3C suggests that the defense mechanism is not too hard to deploy on many websites, at least to get some limited protection.

Critical Evaluation

Content injection is one of the most widespread threats to the security of web sessions [148]. Indeed, modern web applications include contents from a variety of sources, burdening the task of identifying malicious contents. Few proposals attempt to provide a comprehensive defense against content injection and the majority of the most popular solutions are only effective against reflected XSS or have very limited scope. Indeed, among the surveyed solutions, client-side XSS filters are by far the most widespread protection mechanisms, implemented by the majority of the web browsers. Under the current state of the art, achieving protection against stored injections while preserving the application functionality requires the intervention of web developers.

Although several of the discussed approaches were only proposed in research papers and never embraced by the industry, some of them contributed to the development of existing web standards. For instance, the hash-based whitelisting approach of inline scripts supported by CSP has been originally proposed as an example policy in the BEEP paper [93]. More research is needed to provide more general defenses against a complex problem like content injection.

2.4.4 Cross-Site Request Forgery and Login CSRF

We now discuss security solutions which are designed to protect against CSRF and login CSRF. We treat these two attacks together, since security solutions which are designed to protect against one of the attacks are typically also effective against the other. In fact, both CSRF and login CSRF exploit cross-site requests which trigger dangerous side-effects on a trusted web application.

Purely Client-side Solutions

Several browser extensions and client-side proxies have been proposed to counter CSRF attacks, including RequestRodeo [95], CsFire [164, 163] and BEAP [115]. All of these solutions share the same idea of stripping authentication cookies from potentially malicious cross-site requests sent by the browser. The main difference between these proposals concerns the way cross-site requests are deemed malicious: different, more or less accurate heuristics have been put forward for the task.

These solutions are designed to protect against web attackers who host on their web servers pages that include links to a victim website, in the attempt of fooling the browser into sending malicious authenticated requests towards the victim website. Unfortunately, this protection becomes ineffective if a web attacker is able to exploit a content injection vulnerability on the target website, since it may force the browser into sending authenticated requests originating from a *same-site* position.

A very nice advantage of these client-side defenses is their usability and ease of deployment: the user can just install the extension/proxy on her machine and she will be automatically protected from CSRF attacks. On the other hand, compatibility may be at

harm, since any heuristic for determining whenever a cross-site request should be considered malicious is bound to (at least occasionally) produce some false positives. To the best of our knowledge, the most sophisticated heuristic is implemented in the latest release of CsFire [163], but a large-scale evaluation on the real Web has unveiled that even this approach may sometimes break useful functionalities of standard web browsing: for instance, it breaks legitimate accesses to Flickr or Yahoo via the OpenID single sign-on protocol [51].

Allowed Referrer Lists (ARLs)

ARLs have been proposed as a client/server solution against CSRF attacks [51]. Roughly, an ARL is just a whitelist that specifies which origins are entitled to send authenticated requests to a given website. The whitelist is compiled by web developers willing to secure their websites, while the policy enforcement is done by the browser. If no ARL is specified for a website, the browser behaviour is unchanged when accessing it, *i.e.*, any origin is authorized to send authenticated requests to the website.

ARLs are effective against web attackers, provided that no content injection vulnerability affects any of the whitelisted pages. Their design supports usability, since their enforcement is lightweight and transparent to browser users. Moreover, compatibility is ensured by the enforcement of security restrictions only on websites which explicitly opt-in to the protection mechanism. The ease of deployment of ARLs is acceptable in most cases. Users must adopt a security-enhanced web browser, but ARLs do not require major changes to the existing ones: the authors implemented ARLs in Firefox with around 700 lines of C++ code. Web developers, instead, must write down their own whitelists. We believe that for many websites this process requires only limited efforts: for instance, e-commerce websites may include in their ARL only the desired e-payment providers, *e.g.*, Paypal. However, notice that a correct ARL for Paypal may be large and rather dynamic, since it should enlist all the websites relying on Paypal for payment facilities.

Tokenization

Tokenization is a popular server-side countermeasure against CSRF attacks [20]. The idea is that all the requests that might change the state of the web application should include a secret token randomly generated by the server for each session and, possibly, each request: incoming requests that do not include the correct token are rejected. The inclusion of the token is transparently done by the browser during the legitimate use of the website, *e.g.*, every security-sensitive HTML form in the web application is extended to provide the token as a hidden parameter. It is crucial that tokens are bound to a specific session, otherwise an attacker could legitimately acquire a valid token for his own session and transplant it into the user browser to fool the web application into accepting malicious authenticated requests as part of the user session.

Tokenization is robust against web attackers only if we assume they cannot perform content injection attacks. In fact, a content injection vulnerability might give access to all

the secret tokens, given that they are included in the DOM of the web page. The usage of secret tokens is completely transparent to the end-user, so there are no usability concerns. However, tokenization may be hard to deploy for web developers. The manual insertion of secret tokens is tedious and typically hard to get right. Some web development frameworks offer automatic support for tokenization, but this is not always comprehensive and may leave room for attacks. These frameworks are language-dependent and may not be powerful enough for sophisticated web applications developed using many different languages [51].

NoForge

NoForge [98] is a server-side proxy sitting between the web server and the web applications to protect. It implements the tokenization approach against CSRF on all requests, without requiring any change to the web application code. NoForge parses the HTTP(S) responses sent by the web server and automatically extends each hyperlink and form contained in them with a secret token bound to the user session; incoming requests are then delivered to the web server only if they contain a valid token.

The protection and the usability offered by NoForge are equivalent to what can be achieved by implementing tokenization at the server-side. The adoption of a proxy for the tokenization task significantly simplifies the deployment of the defensive solution, but it has a negative impact on compatibility, since HTML links and forms which are dynamically generated at the client side will not be rewritten to include the secret token. As a result, any request sent by clicking on these links or by submitting these forms will be rejected by NoForge, thus breaking the web application. The authors of NoForge are aware of this problem and state that it can be solved by manually writing scripts which extend links and forms generated at the client side with the appropriate token [98]. However, if this need is pervasive, the benefits on deployment offered by NoForge can be easily voided. For this reason we argue that the design of NoForge is not compatible with the modern Web.

Origin Checking

Origin checking is a popular alternative to tokenization [20]. Modern web browsers implement the `Origin` header, identifying the security context (origin) that caused the browser to send an HTTP(S) request. For instance, if a link to `http://b.com` is clicked on a page downloaded from `http://a.com`, the corresponding HTTP request will include `http://a.com` in the `Origin` header. Web developers may inspect this header to detect whether a potentially dangerous cross-site request has been generated by a trusted domain or not.

Origin checking is robust against web attackers without scripting capabilities in any of the domains trusted by the target website. Server-side origin checking is entirely transparent to the end-user and has no impact on the navigation experience, so it may not hinder usability. This solution is simpler to deploy than tokenization, since it can be implemented by using a web application firewall like ModSecurity[1]. Unfortunately, the `Origin` header is not attached to all the cross-origin requests: for instance, the initial proposal of the header was limited to POST requests [20] and current web browser implementations still do not ensure that the header is always populated [19]. Web developers should be fully aware of this limitation and ensure that all the state-changing operations in their applications are triggered by requests bearing the `Origin` header. In practice, this may be hard to ensure for legacy web applications [51].

Critical Evaluation

Effectively preventing CSRFs and login CSRFs is surprisingly hard. Even though the root cause of the security problem is well-understood for these attacks, it is challenging to come up with a solution which is at the same time usable, compatible and easy to deploy. At the time of writing, Allowed Referrer Lists (ARLs) represent the most promising defensive solution against CSRFs and login CSRFs. They are transparent to end-users, respectful towards legacy technology and do not require changes to web application code. Unfortunately, ARLs are not implemented in major web browsers, so in practice tokenization and origin checking are the most widespread solutions nowadays. These approaches however may be hard to deploy on legacy web applications.

2.4.5 Cookie Forcing and Session Fixation

We collect together the defenses proposed against cookie forcing and session fixation. In fact, both the attacks rely on the attacker capability to corrupt the integrity of the authentication cookies set by a trusted website.

Cookies Renewal

The simplest and most effective defense against session fixation is implemented at the server-side, by ensuring that the authentication cookies identifying the user session are refreshed when the level of privilege changes, *i.e.*, when the user provides her password to the web server and performs a login [97]. If this is done, no cookie fixed by an attacker before the first authentication step may be used to identify the user session. Notice that this countermeasure does not prevent cookie forcing, since the attacker can first authenticate at the website using a standard web browser and then directly force his own cookies into the user browser.

Renewing authentication cookies upon password-based authentication is a recommended security practice and it is straightforward to implement for new web applications. However, retrofitting a legacy web application may require some effort, since

[1] https://www.modsecurity.org/

the authentication-related parts of session management must be clearly identified and corrected. It may actually be more viable to keep the application code unchanged and operate at the framework level or via a server-side proxy, to enforce the renewal of the authentication cookies whenever an incoming HTTP(S) request is identified as a login attempt [97]. Clearly, these server-side solutions must ensure that login attempts are accurately detected to preserve compatibility: this is the case, for instance, when the name of the POST parameter bound to the user password is known.

Serene

The Serene browser extension offers automatic protection against session fixation attacks [165]. It inspects each outgoing request sent by the browser and applies a heuristic to identify cookies which are likely used for authentication purposes: if any of these cookies was not set via HTTP(S) headers, it is stripped from the outgoing request, hence cookies which have been fixated or forced by a malicious script cannot be used to authenticate the client. The key observation behind this design is that existing websites set their authentication cookies using HTTP(S) headers in the very large majority of cases.

The solution is designed to be robust against web attackers, since they can only set a cookie for the website by exploiting a markup/script injection vulnerability. Conversely, Serene is not effective against related-domain attackers who might use their sites to legitimately set cookies for the whole domain using HTTP headers. The main advantages of Serene are its usability and ease of deployment: users only need to install Serene in their browser and it will provide automatic protection against session fixation for any website, though the false negatives produced by the heuristic for authentication cookies detection may still leave room for attacks. The compatibility of Serene crucially depends on its heuristic: false positives may negatively affect the functionality of websites, since some cookies which should be accessed by the web server are never sent to it. In practice, it is impossible to be fully accurate in the authentication cookie detection process, even using sophisticated techniques [41].

Origin Cookies

Origin cookies have been proposed to fix some known integrity issues affecting cookies [35]. We have already discussed that standard HTTP cookies do not provide strong integrity guarantees against related-domain attackers and active network attackers. The observation here is that these attackers exploit the relaxation of the same-origin policy applied to cookies (*cf.* Section 2.2.4). Origin cookies, instead, are bound to an exact web origin. For instance, an origin cookie set by `https://example.com` can only be overwritten by an HTTPS response from `example.com` and will only be sent to `example.com` over HTTPS. Origin cookies can be set by websites simply by adding the `Origin` attribute to standard cookies. Origin cookies are sent by the browser inside a new custom header `Origin-Cookie`, thus letting websites distinguish origin cookies from normal ones.

Since origin cookies are isolated between origins, the additional powers of related-domain attackers and active network attackers in setting or overwriting cookies are no longer a problem. The use of origin cookies is transparent to users and their design supports backward compatibility, since origin cookies are treated as standard cookies by legacy browsers (unknown cookie attributes are ignored). Origin cookies are easy to deploy on websites entirely hosted on a single domain and only served over a single protocol: for such a website, it would be enough to add the `Origin` attribute to all its cookies. On the other hand, if a web application needs to share cookies between different protocols or related domains, then the web developer is forced to implement a protocol to link together different sessions built on distinct origin cookies. This may be a non-trivial task to carry out for existing websites.

Cookie Prefixes

Cookie prefixes [205] are a recent proposal aimed at providing strong integrity guarantees for cookies whose names start with particular prefixes. When instructed to set a cookie with a name starting with `__Host-`, a compliant browser verifies that *i)* the cookie is set over HTTPS and the `Secure` attribute is used; *ii)* the `Domain` attribute is omitted; *iii)* the `Path` attribute is equal to `/`, *i.e.*, the cookie is attached to all requests sent to the host. The cookie is rejected by the browser if any of these requirements is not satisfied. Thus, `__Host-` cookies cannot be set by a related domain or by a network attacker, successfully preventing both session fixation and cookie forcing.

Although the specification is currently just a draft, cookie prefixes are already supported by all major browsers except Microsoft Edge and Internet Explorer. Cookie prefixes do not impact in any way the user experience and are designed to be backward compatible since non-compliant web browsers will simply treat them as traditional cookies. Regarding ease of deployment, the same considerations made for origin cookies apply: adoption is straightforward for websites hosted over HTTPS on a single domain, while it may require considerable efforts for web applications shared over multiple related domains.

Critical Evaluation

Session fixation is a dangerous attack, but it is relatively easy to prevent. Renewing authentication cookies upon user authentication is the most popular, effective and widespread solution against these attacks. The only potential issue with this approach is implementing a comprehensive protection for legacy web applications [97]. Cookie forcing, instead, is much trickier to defend against. Although the integrity problems of cookies are well-known to security experts [214], only recently browser started supporting protections to rectify these issues.

2.4.6 Network Attacks

HTTPS with Secure Cookies

Though it is obvious that websites concerned about network attackers should make use of HTTPS, there are some points worth discussing. For instance, while it is well-understood that passwords should only be sent over HTTPS, web developers often underestimate the risk of leaking authentication cookies in clear, thus undermining session confidentiality and integrity. As a matter of fact, many websites are still only partially deployed over HTTPS, either to increase performances or because only a part of their contents needs to be secured. However, cookies set by a website are by default attached to *all* the requests sent to it, irrespectively of the communication protocol. If a web developer wants to deliver a non-sensitive portion of her website over HTTP, it is still possible to protect the confidentiality of the authentication cookies by setting the `Secure` attribute, which instructs the browser to send these cookies only over HTTPS connections. Even if a website is fully deployed over HTTPS, the `Secure` attribute should be set on its authentication cookies, otherwise a network attacker could still force their leakage in clear by injecting non-existing HTTP links to the website in unrelated web pages [86].

Activating HTTPS support on a server requires little technical efforts, but needs a signed public key certificate: while the majority of HTTPS-enabled websites employ certificates signed by recognized certification authorities, a non-negligible percentage uses certificates that are self-signed or signed by CAs whose root certificate is not included in major web browsers [59]. Unless explicitly included in the OS or in the browser keychain, these certificates trigger a warning when the browser attempts to validate them, similarly to what happens when a network attacker acts as a man-in-the-middle and provides a fake certificate: in such a case, a user that proceeds ignoring the warning may be exposed to the active attacker, as if the communication was performed over an insecure channel. The adoption of `Secure` cookies is straightforward whenever the entire website is deployed over HTTPS, since it is enough to add the `Secure` attribute to all the cookies set by the website. For mixed contents websites, `Secure` cookies cannot be used to authenticate the user on the HTTP portion of the site, hence they may be hard to deploy, requiring a change to the cookie scheme.

HProxy

HProxy is a client-side solution which protects against SSL stripping by analyzing the browsing history to produce a profile for each website visited by the user [136]. HProxy inspects all the responses received by the user browser and compares them against the corresponding profiles: divergences from the expected behaviour are evaluated through a strict set of rules to decide whether the response should be accepted or not.

HProxy is effective only on already-visited websites and the offered protection crucially depends on the completeness of the detection ruleset. From a usability perspective, the browsing experience may be affected by the adoption of the proposed defense mechanism, as it introduces an average overhead of 50% on the overall page load time. The

main concern however is about compatibility, since it depends on the ability of HProxy to tell apart legitimate modifications in the web page across consecutive loads from malicious changes performed by the attacker. False positives in this process may break the functionality of benign websites. HProxy is easy to deploy, since the user only needs to install the software on her machine and configure the browser proxy settings to use it.

HTTP Strict Transport Security

HSTS is a security policy implemented in all modern web browsers which allows a web server to force a client to subsequently communicate only over a secure channel [84]. The policy can be delivered solely over HTTPS using a custom header, where it is possible to specify whether the policy should be enforced also for requests sent to subdomains (*e.g.*, to protect cookies shared with them) and its lifetime. When the browser performs a request to a HSTS host, its behaviour is modified so that every HTTP reference is upgraded to the HTTPS protocol before being accessed; TLS errors (*e.g.*, self-signed certificates) terminate the communication session and the embedding of mixed contents (*cf.* Section 2.2.4) is prohibited.

Similarly to the previous solution, HSTS is not able to provide any protection against active network attackers whenever the initial request to a website is carried out over an insecure channel: to address this issue, browsers vendors include a list of known HSTS hosts, but clearly the approach cannot cover the entire Web. Additionally, a recently introduced attack against HSTS [169] exploits a Network Time Protocol weakness found on major operating systems that allows to modify the current time via a man-in-the-middle attack, thus making HSTS policies expire. Usability and compatibility are both high, since users are not involved in security decisions and the HTTP(S) header for HSTS is ignored by browsers not supporting the mechanism. The ease of deployment is high, given that web developers can enable the additional HTTP(S) header with little effort by modifying the web server configuration.

HTTPS Everywhere

This extension for Firefox, Chrome and Opera [58] performs URL rewriting to force access to the HTTPS version of a website whenever available, according to a set of hard-coded rules supplied with the extension. Essentially, HTTPS Everywhere applies the same idea of HSTS, with the difference that no instruction from the website is needed: the hard-coded ruleset is populated by security experts and volunteers.

HTTPS Everywhere is able to protect only sites included in the ruleset: even if the application allows the insertion of custom rules, this requires technical skills that a typical user does not have. In case of partial lack of HTTPS support, the solution may break websites and user intervention is required to switch to the usual browser behaviour; these problems can be rectified by refining the ruleset. The solution is very easy to deploy: the user is only required to install the extension to enforce the usage of HTTPS on supported websites.

Critical Evaluation

HTTPS is pivotal in defending against network attacks: all the assessed solutions try to promote insecure connections to encrypted ones or force web developers to deploy the whole application on HTTPS. Mechanisms exposing compatibility problems are unlikely to be widely adopted, as in the case of HProxy due to its heuristic approach. All the other defenses, instead, are popular standards or enjoy a large user base. Academic solutions proved to be crucial for the development of web standards: HSTS is a revised version of ForceHTTPS [86] in which a custom cookie was used in place of an HTTP header to enable the protection mechanism.

Summary

We summarize in Table 2.2 all the defenses discussed so far. We denote with ★ those solutions whose ease of deployment depends on the policy complexity. When the adoption of a security mechanism is much harder on legacy web applications with respect to newly developed or modern ones, we annotate the score with †.

2.5 Defenses Against Multiple Attacks

All the web security mechanisms described so far have been designed to prevent or mitigate very specific attacks against web sessions. In the literature we also find proposals providing a more comprehensive solution to a range of different threats. These proposals are significantly more complex than those in the previous section, hence it is much harder to provide a schematic overview of their merits and current limitations.

Origin-Bound Certificates

Origin-Bound Certificates (OBC) [55] have been proposed as an extension to the TLS protocol that binds authentication tokens to trusted encrypted channels. The idea is to generate, on the client side, a different certificate for every web origin upon connection. This certificate is sent to the server and used to cryptographically bind authentication cookies to the channel established between the browser and that specific origin. The browser relies on the same certificate when arranging a TLS connection with a previously visited origin. The protection mechanism implemented by OBC is effective at preventing the usage of authentication cookies outside of the intended channel: for instance, a cookie leaked via a content injection vulnerability cannot be reused by an attacker to identify himself as the victim on the vulnerable website, since the victim certificate is not disclosed. Similarly, session fixation attacks are defeated by OBC, given that the cookie value associated to the attacker channel cannot be used within the victim TLS connection.

The presence of OBC is completely transparent to the user and the impact on performances is negligible after certificate generation, so the usability of the solution is high. Compatibility is not at harm, since the browser and the server must explicitly agree on the use of OBC during the TLS handshake. One problem is represented by domain cookies,

Attack	*Defense*	*Type*	*Usability*	*Compatibility*	*Ease of Deployment*
Content injection mitigation	HttpOnly cookies	hybrid	H	H	H
	SessionShield/Zan	client	H	L	H
	Requests filtering	client	L	H	H
Content injection prevention	Client-side XSS filters	client	H	H	H
	Server-side filtering	server	H	M	L/M*
	XSS-Guard	server	H	M	H
	BEEP	hybrid	M	H	L/M*
	Blueprint	hybrid	M	M	L
	Noncespaces	hybrid	M	L	L/M*
	DSI	hybrid	H	H	M
	CSP	hybrid	H	H	L/M*
CSRF	Client-side defenses	client	H	L	H
	Allowed referrer lists	hybrid	H	H	L/M*
Login CSRF	Tokenization	server	H	H	L/H†
	NoForge	server	H	L	H
	Origin checking	server	H	H	L/H†
Session fixation	Cookies renewal	server	H	H	M/H†
Cookie forcing	Serene	client	H	L	H
Session fixation	Origin cookies	hybrid	H	H	M/H†
	Cookie prefixes	hybrid	H	H	M/H†
Network attacks	HTTPS w. secure cookies	hybrid	H	H	M/H†
	HProxy	client	M	L	H
	HSTS	hybrid	H	H	H
	HTTPS Everywhere	client	M	H	H

TABLE 2.2: Analysis of Proposed Defenses.

i.e., cookies accessed by multiple origins: to overcome this issue, the authors suggested a *legacy mode* of OBC in which the client generates certificates bound to the whole domain instead of a single origin. Being an extension to the TLS protocol, OBC requires changes to both parties involved in the encrypted channel initiation. The authors successfully implemented the described mechanism on the open-source browser Chromium and on OpenSSL by altering approximately 1900 and 320 lines of code, respectively. However, web developers are not required to adapt their applications to use OBC, which has a beneficial impact on ease of deployment.

Browser-based Information Flow Control

Browser-based information flow control is a promising approach to uniformly prevent a wide class of attacks against web sessions. FlowFox [71] was the first web browser implementing a full-fledged information flow control framework for confidentiality policies on JavaScript code. Later work on the same research line includes JSFlow [81], COWL [175]

and an extension of Chromium with information flow control [22], which we refer to as ChromiumIFC. These solutions explore different points of the design space:

- FlowFox is based on *secure multi-execution*, a dynamic approach performing multiple runs of a given program (script) under a special policy for input/output operations ensuring non-interference [53]. To exemplify, assume the existence of two security levels Public and Secret, then the program is executed twice (once per level) under the following regime: (1) outputs marked Public/Secret are only done in the execution at level Public/Secret; and (2) inputs at level Public are fed to both the executions, while inputs at level Secret are only fed to the execution at level Secret (a default value for the input is provided to the Public execution). This ensures by construction that Private inputs do not affect Public outputs;

- JSFlow is based on a dynamic *type system* for JavaScript. JavaScript values are extended with a security label representing their confidentiality level and labels are updated to reflect the computational effects of the monitored scripts. Labels are then dynamically checked to ensure that computations preserve non-interference;

- COWL performs a *compartmentalization* of scripts and assigns security labels at the granularity of compartments encapsulating contents from a single origin. It enforces coarse-grained policies on communication across compartments and towards remote origins via label checking;

- ChromiumIFC implements a lightweight dynamic *taint tracking* technique to constrain information flows within the browser and prevent the leakage of secret information. In contrast to previous proposals, this solution is not limited to JavaScript, but it spans all the most relevant browser components.

The different design choices taken by the reviewed solutions have a clear import on our evaluation factors. In terms of protection, enforcing information flow control on scripts is already enough to prevent many web threats. For instance, assuming an appropriate security policy, web attackers cannot leak authentication cookies using XSS [71] or run CSRF attacks based on JavaScript [103]. This is true also in presence of stored XSS attacks, provided that information flow control is performed on the injected scripts. However, there are attack vectors which go beyond scripts, *e.g.*, a web attacker can carry out a CSRF by injecting markup elements. Preventing these attacks requires a more extensive monitoring of the web browser, as the one proposed by ChromiumIFC.

To the best of our knowledge, there has been no thorough usability study for any of the cited solutions. It is thus unclear if and to which extent users need to be involved in security decisions upon normal browsing. However, degradation of performances caused by information flow tracking may hinder the user experience and negatively affect usability. For instance, the performances of FlowFox are estimated to be around 20% worse than those of a standard web browser, even assuming only policies with two security levels [71]. Better performances can be achieved by using simpler enforcement

mechanisms and by lowering the granularity of enforcement, for instance the authors of COWL performed a very promising performance evaluation of their proposal [175].

Compatibility and ease of deployment are better evaluated together, since there is a delicate balance between the two in this area, due to the flexibility of information flow policies. On the one hand, inaccurate information flow policies can break existing websites upon security enforcement, thus affecting compatibility. On the other hand, accurate information flow policies may be large and hard to get right, thus hindering deployment. We think that a set of default information flow policies may already be enough to stop or mitigate a wide class of attacks against web sessions launched by malicious scripts: for instance, cookies could be automatically marked as private for the domain which set them. Indeed, a preliminary experiment with FlowFox on the top 500 sites of Alexa shows that compatibility is preserved for a very simple policy which marks as sensitive any access to the cookie jar [71]. Reaping the biggest benefits out of information flow control, however, necessarily requires some efforts by web developers.

Security Policies for JavaScript

Besides information flow control, in the literature there are several frameworks for enforcing general security policies on untrusted JavaScript code [120, 209, 111, 151, 186]. We just provide a brief overview on them here and we refer the interested reader to a recent survey by Bielova [27] for additional details. The core idea behind all these proposals is to implement a runtime monitor that intercepts the API calls made by JavaScript programs and checks whether the sequence of such calls complies with an underlying security policy. This kind of policies has proved helpful for protecting access to authentication cookies, thus limiting the dangers posed by XSS, and for restricting cross-domain communication attempts by untrusted code, which helps at preventing CSRF attacks. We believe that other useful policies for protecting web sessions can be encoded in these rather general frameworks, though the authors of the original papers do not discuss them in detail. Since all these proposals assume that JavaScript code is untrusted, they are effective even in presence of stored XSS attacks, provided that the injected scripts are subject to policy enforcement.

As expected, security policies for JavaScript share many of the strengths and weaknesses of browser-based information flow control in terms of protection, usability and compatibility. Ease of deployment, instead, deserves a more careful discussion, since it fundamentally depends on the complexity of the underlying policy language. For instance, in [120] security policies are expressed in terms of JavaScript code, while the framework in [209] is based on *edit automata*, a particular kind of state machine with a formal semantics. Choosing the right policy language may significantly improve the ease of deployment, though we believe that meaningful security policies require some efforts by web developers. There is some preliminary evidence that useful policies can be automatically synthesized by static analysis or runtime training: the idea is to monitor normal JavaScript behaviour and to deem as suspicious all the unexpected script

behaviours [120]. However, we believe more research is needed to draw a fair conclusion on how difficult it is to deploy these mechanisms in practice.

AJAX Intrusion Detection System

Guha *et al.* proposed an AJAX intrusion detection system based on the combination of a static analysis for JavaScript and a server-side proxy [74]. The static analysis is employed by web developers to construct the control flow graph of the AJAX application to protect, while the proxy dynamically monitors browser requests to prevent violations to the expected control flow of the web application. The solution also implements defenses against *mimicry attacks*, in which the attacker complies with legitimate access patterns in his malicious attempts. This is done by making each session (and thus each graph) slightly different than the other ones by placing unpredictable, dummy requests in selected points of the control flow. The JavaScript code of the web application is then automatically modified to trigger these requests which cannot be predicted by the attacker.

The approach is deemed useful to mitigate the threats posed by content injection and to prevent CSRF, provided that these attacks are launched via AJAX. Since the syntax of the control flow graph explicitly tracks session identifiers, session fixation attacks can be prevented: indeed, in these attacks there is a mismatch between the cookie set in the first response sent by the web server and the cookie which is included by the browser in the login request, hence a violation to the intended control flow will be detected. The approach is effective even against stored XSS attacks exploiting AJAX requests, whenever they are mounted after the construction of the control flow graph.

The solution offers high usability, since it is transparent to users and the runtime overhead introduced by the proxy is minimal. According to the authors, the adoption of a context-sensitive static analysis for JavaScript makes the construction of the control flow graph very precise, which is crucial to preserve the functionality of the web application and ensure compatibility. The authors claim that the solution is easy to deploy, since the construction of the control flow graph is totally automatic and the adoption of a proxy does not require changes to the web application code.

Escudo

Escudo [89] is an alternative protection model for web browsers, extending the standard same-origin policy to rectify several of its known shortcomings. By noticing a strong similarity between the browser and an operating system, the authors of Escudo argue for the adoption of a protection mechanism based on hierarchical rings, whereby different elements of the DOM are placed in rings with decreasing privileges; the definition of the number of rings and the ring assignment for the DOM elements is done by web developers. Developers can also assign protection rings to their cookies, while the internal browser state containing, *e.g.*, the history, is set by default in ring 0. Access to objects in a given ring is only allowed to subjects being in the same or lower rings.

Escudo is designed to prevent XSS and CSRF attacks. Untrusted web contents should be assigned to the least privileged ring, so that scripts crafted by exploiting a reflected XSS vulnerability would do no harm. Similarly, requests from untrusted web pages should be put in a low privilege ring without access to authentication credentials, thus preventing CSRF attacks. Notice, however, that stored XSS vulnerabilities may be exploited to inject code running with high privileges in trusted web applications and attack them. The authors of Escudo do not discuss network attacks.

Escudo does not require user interventions for security enforcement and it only leads to a slight overhead on page rendering (around 5%). This makes the solution potentially usable. However, deploying ring assignments for Escudo looks challenging. The authors evaluated this aspect by retrofitting two existing opensource applications: both experiments required around one day of work, which looks reasonable. On the other hand, many web developers are not security experts and the fine-grained policies advocated by Escudo may be too much of a burden for them: without tool support for annotating the DOM elements, the deployment of Escudo may be complicated, especially if a comprehensive protection is desired. Escudo is designed to be backward compatible: Escudo-based web browsers are compatible with non-Escudo applications and vice-versa; if an appropriate policy is put in place, no compatibility issue will arise.

CookiExt

CookiExt [37] is a Google Chrome extension protecting the confidentiality of authentication cookies against both web and network attacks. The extension adopts a heuristic to detect authentication cookies in incoming responses: if a response is sent over HTTP, all the identified authentication cookies are marked as `HttpOnly`; if a response is sent over HTTPS, these cookies are also marked as `Secure`. In the latter case, to preserve the session, CookiExt forces an automatic redirection over HTTPS for all the subsequent HTTP requests to the website, since these requests would not include the cookies which have been extended with the `Secure` attribute. In order to preserve compatibility, the extension implements a fallback mechanism which removes the `Secure` attribute automatically assigned to authentication cookies in case the server does not support HTTPS for some of the web pages. The design of CookiExt has been formally validated by proving that a browser with CookiExt satisfies non-interference with respect to the value of the authentication cookies. In particular, it is shown that what an attacker can observe of the CookiExt browser behaviour is unaffected by the value of authentication cookies. CookiExt does not protect against CSRF and session fixation: it just ensures the confidentiality of the authentication cookies.

CookiExt does not require any user interaction and features a lightweight implementation, which guarantees a high level of usability. Preliminary experiments performed by the authors show good compatibility results on existing websites from Alexa, since only minor annoyances due to the security enforcement have been found; however, a large-scale evaluation of the extension is still missing. Being implemented as a browser extension, CookiExt is very easy to deploy.

SessInt

SessInt [38] is an extension for Google Chrome providing a purely client-side countermeasure against the most common attacks targeting web sessions. The extension prevents the abuse of authenticated requests and protects authentication credentials. It enforces web session integrity by combining access control and taint tracking mechanisms in the browser. The security policy applied by SessInt has been verified against a formal threat model including both web and network attackers. As a distinguishing feature with respect to other client-side solutions, SessInt is able to stop CSRF attacks even when they are launched by exploiting reflected XSS vulnerabilities. On the other hand, no protection is given against stored XSS.

The protection provided by SessInt is fully automatic: its security policy is uniformly applied to every website and no interaction with the web server or the end-user is required. Also, the performance overhead introduced by the security checks of SessInt is negligible and no user interaction is needed. However, the protection offered by SessInt comes at a cost on compatibility: the current prototype of the extension breaks several useful web scenarios, including single sign-on protocols and e-payment systems. The implementation as a browser extension makes SessInt very easy to deploy.

Same Origin Mutual Approval

SOMA [142] is a research proposal describing a simple yet powerful policy for content inclusion and remote communication on the Web. SOMA enforces that a web page from a domain d_1 can include contents from an origin o hosted on domain d_2 only if both the following checks succeed: *i)* d_1 has listed o as an allowed source of remote contents; *ii)* d_2 has listed d_1 as an allowed destination for content inclusion. SOMA is designed to offer protection against web attackers: developers can effectively prevent CSRF attacks and mitigate the threats posed by content injection vulnerabilities, including stored XSS, by preventing the injected contents from communicating with attacker-controlled pages.

The protection offered by SOMA does not involve user intervention and the performances of the solution look satisfactory, especially on cached page loads, where only an extra 5% of network latency is introduced. This ensures that SOMA can be a usable solution. Moreover, if a SOMA policy correctly includes all the references to the necessary web resources, no compatibility issues will occur. Writing correct policies looks feasible in practice, since similar specifications are also used by popular web standards like CSP. The deployment of SOMA would not be trivial, but acceptable: browsers must be patched to support the mutual approval policy described above, while web developers should identify appropriate policies for their websites. These policies are declarative in nature and expected be relatively small in practice; most importantly, no change to the web application code is required.

App Isolation

App Isolation [45] is a defense mechanism aimed at offering, within a single browser, the protection granted by the usage of different browsers for navigating websites at separate levels of trust. If one "sensitive" browser is only used to navigate trusted websites, while another "non-sensitive" browser is only used to access potentially malicious web pages, many of the threats posed by the latter are voided by the absence of shared state between the two browsers. For instance, CSRF attacks would fail, since they would be launched from an attacker-controlled web page in the non-sensitive browser, but the authentication cookies for all trusted web applications would only be available in the sensitive browser. Enforcing this kind of guarantees within a single browser requires two ingredients: *i)* a strong *state isolation* among web applications; *ii)* an *entry point restriction*, preventing the access to sensitive web applications from maliciously crafted URLs. Indeed, in the example above, protection would be voided if the link mounting the CSRF attack was opened in the sensitive browser. This design is effective at preventing reflected XSS attacks, session fixation and CSRF. However, stored XSS attacks against trusted websites will bypass the protection offered by App Isolation, since the injected code would be directly delivered from a trusted position.

The usability of App Isolation looks promising, since the protection is applied automatically and the only downside is a slight increase in the loading time of the websites, due to the additional round-trip needed to fetch the list of allowed entry points. The compatibility of the solution is ensured by the fact that supporting browsers only enforce protection when explicitly requested by the web application. Web developers, however, should compile a list of entry points defining the allowed landing pages of their web applications. This is feasible and easy to do only for non-social websites, *e.g.*, online banks, which are typically accessed only from their homepage, but it is prohibitively hard for social networks or content-oriented sites, *e.g.*, newspapers websites, where users may want to jump directly to any URL featuring an article. The ease of deployment thus crucially depends on the nature of the web application to protect.

Summary

We summarize our observations about the described solutions in Table 2.3. Again, we denote with ★ the solutions where the ease of deployment is affected by the policy complexity. Additionally, we use a dash symbol whenever we do not have any definite evidence about a specific aspect of our investigation based on the existing literature. Most notably, we leave empty the Usability and Compatibility entries for browser-based information flow control and JavaScript security policies, since they depend too much on the specific implementation choices and the policies to enforce. More research is needed to understand these important aspects.

		Attacks				*Evaluation*		
Defense	*Type*	*Content injection*	*CSRF Login CSRF*	*Session fixation Cookie forcing*	*Network attacks*	*Usability*	*Compatibility*	*Ease of Deployment*
OBC	hybrid	✓	✗	✓	✓	H	H	M
Browser IFC	hybrid	✓	✓	✓	✗	-	-	L/M*
JS Policies	hybrid	✓	✓	✓	✗	-	-	L/M*
Ajax IDS	server	✓	✓	✓	✗	H	H	H
Escudo	hybrid	✓	✓	-	✗	H	H	L/M*
CookiExt	client	✓	✗	✗	✓	H	M	H
SessInt	client	✓	✓	✓	✓	H	L	H
SOMA	hybrid	✓	✓	✗	✗	H	H	M
App Isolation	hybrid	✓	✓	✓	✗	H	H	L/M*

TABLE 2.3: Defenses Against Multiple Attacks.

2.6 Perspective

Having examined different proposals, we now identify five guidelines for the designers of novel web security mechanisms. This is a synthesis of sound principles and insights which have, to different extents, been taken into account by all the designers of the proposals we surveyed.

2.6.1 Transparency

We call *transparency* the combination of high usability and full compatibility: we think this is the most important ingredient to ensure a large scale deployment of any defensive solution for the Web, given its massive user base and its heterogeneity. It is well-known that security often comes at the cost of usability and that usability defects ultimately weaken security, since users resort to deactivating or otherwise sidestepping the available protection mechanisms [180]. The Web is extremely variegate and surprisingly fragile even to small changes: web developers who do not desire to adopt new defensive technologies should be able to do so, without any observable change to the semantics of their web applications when these are accessed by security-enhanced web browsers; dually, users who are not willing to update their web browsers should be able to seamlessly navigate websites which implement cutting-edge security mechanisms not supported by their browsers.

All the security decisions must be ultimately taken by web developers. On the one hand, users are not willing or do not have the expertise to be involved in security decisions. On the other hand, it is extremely difficult for browser vendors to come up with "one size fits all" solutions which do not break any website. Motivated web developers, instead, can be fully aware of their web application semantics, thoroughly test new proposals and configure them to support compatibility.

Examples: Hybrid client/server solutions like ARLs (Section 2.4.4), CSP (Section 2.4.3) and SOMA (Section 2.5) are prime examples of proposals which ensure transparency, since they do not change the semantics of web applications not adopting them. Conversely, purely client-side defenses like Serene (Section 2.4.5) and SessInt (Section 2.5) typically present some compatibility issues, since they lack enough contextual information to be always precise in their security decisions: this makes them less amenable for a large-scale deployment.

2.6.2 Security by Design

Supporting the current Web and legacy web applications is essential, but developers of new websites should be provided with tools which allow them to realize applications which are secure *by design*. Our feeling is that striving for backward compatibility often hinders the creation of tools which could actually improve the development process of new web applications. Indeed, backward compatibility is often identified with problem-specific patches to known issues, which developers of existing websites can easily plug into their implementation to retrofit it. The result is that developing secure web applications using the current technologies is a painstaking task, which involves actions at too many different levels. Developers should be provided with tools and methodologies which allow them to take security into account from the first phases of the development process. This necessarily means deviating from the low-level solutions advocated by many current technologies, to rather focus on more high-level security aspects of the web application, including the definition of principals and their trust relations, the identification of sensitive information, etc.

Examples: Proposals which are secure by design include the non-interference policies advocated by FlowFox (Section 2.5) and several frameworks for enforcing arbitrary security policies on untrusted JavaScript code (Section 2.5). Popular examples of solutions which are not secure by design include the usage of secret tokens against CSRF attacks (Section 2.4.4): indeed, not every token generation scheme is robust [20] and ensuring the confidentiality of the tokens may be hard, even though this is crucial for the effectiveness of the solution.

2.6.3 Ease of Adoption

Server-side solutions should require a limited effort to be understood and adopted by web developers. For instance, the usage of frameworks which automatically implement recommended security practices, often neglected by web developers, can significantly simplify the development of new secure applications. For client-side solutions, it is important that they work out of the box when they are installed in the user browser: proposals which are not fully automatic are going to be ignored or misused. Any defensive solution which involves both the client and the server is subject to both the previous

observations. Since it is unrealistic that a single protection mechanism is able to accommodate all the security needs, it is crucial to design the defensive solution so that it gracefully interacts with existing proposals which address orthogonal issues and which may already be adopted by web developers.

Examples: Many client-side defenses are easy to adopt, since they are deployed as browser extensions which automatically provide additional protection: this is the case of tools like CsFire (Section 2.4.4) and CookiExt (Section 2.5). Server-side or hybrid client/server solutions are often harder to adopt, for different reasons: some proposals like Escudo (Section 2.5) are too fine-grained and thus require a huge configuration effort, while others like FlowFox (Section 2.5) may be hard for web developers to understand. Good examples of hybrid client/server solutions which promise an easy adoption, since they speak the same language of web developers, include SOMA (Section 2.5) and HSTS (Section 2.4.6). Origin checking is often straightforward to implement as a server-side defense against CSRF attacks (Section 2.4.4).

2.6.4 Declarative Nature

To support a large-scale deployment, new defensive solutions should be *declarative* in nature: web developers should be given access to an appropriate policy specification language, but the enforcement of the policy should not be their concern. Security checks should not be intermingled with the web application logic: ideally, no code change should be implemented in the web application to make it more secure and a thorough understanding of the web application code should not be necessary to come up with reasonable security policies. This is dictated by very practical needs: existing web applications are huge and complex, are often written in different programming languages and web developers may not have full control over them.

Examples: Whitelist-based defenses like ARLs (Section 2.4.4) and SOMA (Section 2.5) are declarative in nature, while the tokenization (Section 2.4.4) is not declarative at all, since it is a low-level solution and it may be hard to adopt on legacy web applications.

2.6.5 Formal Specification and Verification

Formal models and tools have been recently applied to the specification and the verification of new proposals for web session security [34, 4, 61, 38]. While a formal specification may be of no use for web developers, it assists security researchers in understanding the details of the proposed solution. Starting from a formal specification, web security designers can be driven by the enforcement of a clear *semantic* security property, *e.g.*, noninterference [71] or session integrity [38], rather than by the desire of providing ad-hoc solutions to the plethora of low-level attacks which currently target the Web.

This is not merely a theoretical exercise, but it has clear practical benefits. First, it allows a comprehensive identification of *all* the attack vectors which may be used to violate the intended security property, thus making it harder that subtle attacks are left

undetected during the design process. Second, it forces security experts to focus on a rigorous threat model and to precisely state all the assumptions underlying their proposals: this helps making a critical comparison of different solutions and simplifies their possible integration. Third, more speculatively, targeting a property rather than a mechanism allows to get a much better understanding of the security problem, thus fostering the deployment of security mechanisms which are both more complete and easier to use for web developers.

Examples: To the best of our knowledge, only very few of the proposals we surveyed are backed up by a solid formal verification. Some notable examples include CookiExt (Section 2.5), SessInt (Section 2.5), FlowFox (Section 2.5) and CsFire (Section 2.4.4).

2.6.6 Discussion

Retrospectively looking at the solutions we reviewed, we identify a number of carefully crafted proposals which comply with several of the guidelines we presented. Perhaps surprisingly, however, we also observe that *none* of the proposals complies with all the guidelines. We argue that this is not inherent to the nature of the guidelines, but rather the simple consequence of web security being hard: indeed, many different problems at very different levels must be taken into account when targeting the largest distributed system in the world.

The Challenges of the Web Platform

Nowadays, there is a huge number of different web standards and technologies, and most of them are scattered across different RFCs. This makes it hard to get a comprehensive picture of the web platform and, conversely, makes it extremely easy to underestimate the impact of novel defense mechanisms on the web ecosystem. Moreover, the sheer size of the Web makes it difficult to assume typical use case scenarios, since large-scale evaluations often reveal surprises and contest largely accepted assumptions [156, 138, 41].

Particular care is needed when designing web security solutions given the massive user base of the Web whose popularity heavily affects what security researchers and engineers may actually propose to improve its security. Indeed, one may argue that the compatibility and the usability of a web defense mechanism may even be more important than the protection it offers. This may be hard to accept, since it partially limits the design space for well-thought solutions tackling the root cause of a security issue. However, the quest for usability and compatibility is inherently part of the web security problem and it should never be underestimated.

The Architecture of an Effective Solution

Purely client-side solutions are likely to break compatibility, since the security policy they apply should be acceptable for every website, but "one size fits all" solutions do not work in a heterogeneous environment like the Web. The best way to ensure that a client-side

defense preserves compatibility is to adopt a whitelist-based approach, so as to avoid that the defensive mechanism is forced to guess the right security decision. However, the protection offered by a whitelist is inherently limited to a known set of websites.

Similarly, purely server-side approaches have their limitations. Most of the server-side solutions we surveyed are hard to adopt and not declarative at all. When this is not the case, like in NoForge (Section 2.4.4), compatibility is at risk. Indeed, just as client-side solutions are not aware of the web application semantics, server-side approaches have very little knowledge of the client-side code running in the browser.

Based on our survey and analysis, we confirm that hybrid client/server designs hold great promise in being the most effective solution for future proposals [202]. We observe that it is relatively easy to come up with hybrid solutions which are compliant with the first four guidelines: SOMA (Section 2.5), HSTS (Section 2.4.6) and ARLs (Section 2.4.4) are good examples.

A Note on Formal Verification

It may be tempting to think that proposals which comply with the first four guidelines are already good enough, since their formal verification can be performed a posteriori. However, this is not entirely true: solutions which are not designed with formal verification in mind are often over-engineered and very difficult to prove correct, since it is not obvious what they are actually trying to enforce. For many solutions, we just know that they prevent some attacks, but it is unclear whether other attacks are feasible under the same threat model and there is no assurance that a sufficiently strong security property can be actually proved for them.

We thus recommend to take formal verification into account from the first phases of the design process. A very recent survey discusses why and how formal methods can be fruitfully applied to web security and highlights open research directions [36].

Open Problems and New Research Directions

We have observed that, at the moment, there exist no solution complying with the five guidelines above and that solutions complying with the first four guidelines still miss a formal treatment. One interesting line of research would be to try to formally state the security properties provided by those solutions under various threat models. As we discussed, proving formal properties of existing mechanisms is not trivial (and sometimes not even feasible) and requires, in the first place, to come up with a precise statement of the security goals. SOMA (Section 2.5), HSTS (Section 2.4.6) and ARLs (Section 2.4.4) are certainly good candidates for this formal analysis.

However, having a single solution covering the five guidelines would be far from providing a universal solution for web session security. We have seen that most of the proposals target specific problems and attacks. The definition of a general framework for studying, comparing, and composing web security mechanisms might help understanding in which extent different solutions compose and what would be the resulting

security guarantee. Modular reasoning looks particularly important in this respect, since the web platform includes many different components and end-to-end security guarantees require all of them to behave correctly. This would go in the direction of securing web sessions in general, instead of just preventing classes of attacks.

For what concerns new solutions, we believe that they should be supported by a formal specification and a clear statement of the security goals and of the threat model. The development of new, well-founded solutions would certainly benefit from the investigation and formal analysis of existing, practical solutions. However, new solutions should try to tackle web session security at a higher level of abstraction, independently of the specific attacks. They should be designed with all of the above guidelines in mind which, in turn, suggests a hybrid approach. The formal model would clarify what are the critical components to control and what (declarative) server-side information is necessary to implement a transparent, secure by design and easy to adopt solution.

Chapter 3

WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring

3.1 Introduction

Web protocols are security protocols deployed on top of HTTP and HTTPS, most notably to implement authentication and authorization at remote servers. Popular examples of web protocols include OAuth 2.0, OpenID Connect, SAML 2.0 and Shibboleth, which are routinely used by millions of users to access security-sensitive functionalities on their personal accounts.

Unfortunately, designing and implementing web protocols is a particular error-prone task even for security experts, as witnessed by the large number of vulnerabilities reported in the literature [177, 12, 13, 215, 106, 107, 207, 199]. The main reason is that web protocols involve communication with a web browser which does not strictly follow the protocol specification but reacts asynchronously to any input it receives, producing messages which may have an import on protocol security. Reactiveness is dangerous because the browser is agnostic to the web protocol semantics: it does not know when the protocol starts, nor when it ends, and is unaware of the order in which messages should be processed, as well as of the confidentiality and integrity guarantees desired for a protocol run. For example, in the context of OAuth 2.0, Bansal *et al.* [12] discussed *token redirection attacks* enabled by the presence of open redirectors, while Fett *et al.* [60] presented *state leak attacks* enabled by the communication of the `Referer` header; these attacks are not apparent from the protocol specification, but arise from subtleties of the browser behaviour.

Major service providers try to aid software developers to correctly integrate web protocols in their websites by means of APIs; however, web developers are not forced to use them, can still use them incorrectly [200], and the APIs themselves do not necessarily implement the best security practices [177]. This unfortunate situation led to the proliferation of attacks against web protocols even at popular services.

In this chapter, we propose a fundamental paradigm shift to strengthen the security guarantees of web protocols. The key idea we put forward is to extend browsers with a security monitor which is able to enforce the compliance of browser behaviours with respect to the web protocol specification. This approach brings two main benefits:

1. web applications are automatically protected against a large class of bugs and vulnerabilities on the browser-side, since the browser is aware of the intended protocol flow and any deviation from it is detected at runtime;

2. protocol specifications can be written and verified once, possibly as a community effort, and then uniformly enforced at a number of different websites.

Remarkably, though changing the behaviour of web browsers is always delicate for backward compatibility, the security monitor we propose is carefully designed to interact gracefully with existing websites so that their functionalities are preserved unless they critically deviate from the intended protocol specification. Moreover, the monitor can be implemented as a browser extension, thereby offering immediate protection to Internet users and promising a significant practical impact.

3.1.1 Contributions

We make the following contributions:

1. we identify three fundamental browser-side security properties for web protocols, that is, the *confidentiality* and *integrity* of message components, as well as the compliance with the intended *protocol flow*. We discuss concrete examples of their import in the popular authorization protocol OAuth 2.0;

2. we semantically characterize these properties and formally prove that their enforcement suffices to protect web applications from a wide range of protocol implementation bugs and attacks on the application code running in the browser;

3. we propose the Web Protocol Security Enforcer, or WPSE for short, a browser-side security monitor designed to enforce the aforementioned security properties, which we implement as a publicly available Google Chrome extension;

4. we experimentally assess the effectiveness of WPSE by testing it against 90 popular websites making use of OAuth 2.0 to implement single sign-on at major identity providers. We identified security flaws in 55 websites (61.1%), including new critical vulnerabilities caused by tracking libraries such as Facebook Pixel, all of which fixable by WPSE. We show that WPSE works flawlessly on 83 websites (92.2%), with the 7 compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability;

5. to show the generality of our approach we also considered SAML 2.0, a popular web authorization protocol: while formalizing its specification, we found a new attack on the Google implementation of SAML 2.0 that has been awarded a bug bounty according to the Google Vulnerability Reward Program.[1]

[1] `https://www.google.com/about/appsecurity/reward-program/`

3.1.2 Structure of the Chapter

Section 3.2 presents the critical challenges for the enforcement of security in web protocols exemplified in the context of OAuth 2.0. The design and the implementation of WPSE is discussed in Section 3.3. In Section 3.4 we present the attacks from the literature that can be successfully defeated by WPSE. Additionally, we introduce a novel attack against Google implementation of SAML 2.0 that is prevented by our solution. Section 3.5 reports on the experimental evaluation of WPSE on real websites to assess our solution in terms of security and compatibility with existing websites. In Section 3.6 we formally characterize the security guarantees provided by our monitoring technique. Related work is discussed in Section 3.7.

3.2 Security Challenges in Web Protocols

The design of web protocols comes with various security challenges which can often be attributed to the presence of the web browser that acts as a non-standard protocol participant. In the following, we discuss three crucial challenges using the OAuth 2.0 authorization protocol as illustrative example.

3.2.1 Background on OAuth 2.0

OAuth 2.0 [78] is a web protocol that enables resource owners to grant controlled access to resources hosted at remote servers. Typically, OAuth 2.0 is also used for authenticating the resource owner to third parties by giving them access to the resource owner's identity stored at an identity provider. This functionality is known as Single Sign-On (SSO). Using standard terminology, we refer to the third-party application as *relying party* (*RP*) and to the website storing the resources, including the identity, as *identity provider* (*IdP*).[2]

The OAuth 2.0 specification defines four different protocol flows, also known as *grant types* or *modes*. We focus on the *authorization code* mode and the *implicit* mode since they are the most commonly used by websites.

The authorization code mode is intended for a *RP* whose main functionality is carried out at the server-side. The high-level protocol flow is depicted in Figure 3.1. For the sake of readability, we introduce a simplified version of the protocol abstracting from some implementation details that are presented in Section 3.4.1. The protocol works as follows:

① the user *U* sends a request to *RP* for accessing a remote resource. The request specifies the *IdP* that holds the resource. In the case of SSO, this step determines which *IdP* should be used;

② *RP* redirects *U* to the login endpoint of *IdP*. This request contains the *RP*'s identity at *IdP*, the URI that *IdP* should redirect *U* to after successful login and an optional state parameter for CSRF protection that should be bound to *U*'s state;

[2] The OAuth 2.0 specification distinguishes between *resource servers* and *authorization servers* instead of considering one identity provider that stores the user's identity as well as its resources [78], but it is common practice to unify resource and authorization servers as one party [60, 177, 107].

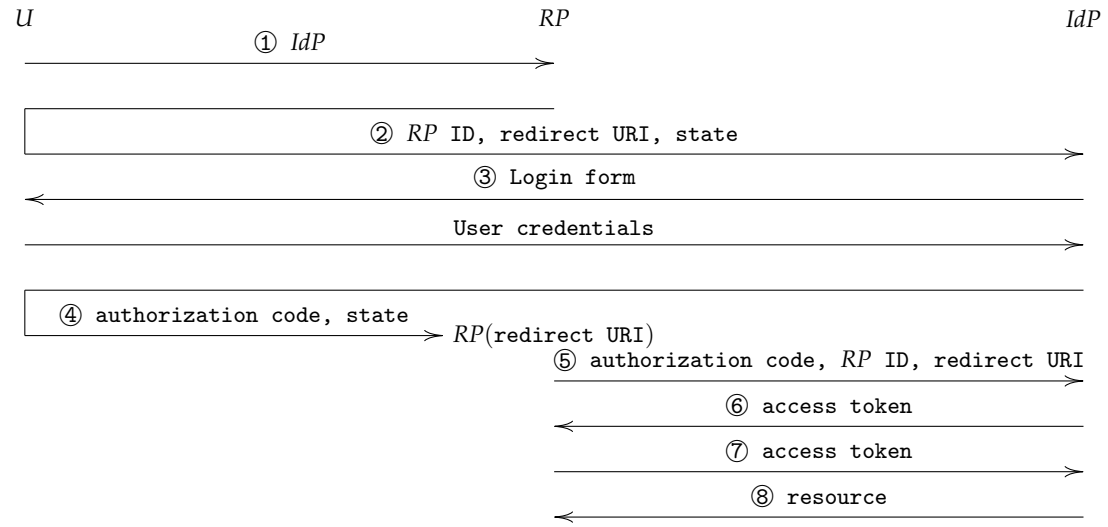


FIGURE 3.1: OAuth 2.0 (authorization code mode).

③ *IdP* answers to the authorization request with a login form and the user provides her credentials;

④ *IdP* redirects *U* to the URI of *RP* specified at step ②, including the previously received state parameter and an authorization code;

⑤ *RP* makes a request to *IdP* with the authorization code, including its identity, the redirect URI and optionally a shared secret with the *IdP*;

⑥ *IdP* answers with an access token to *RP*;

⑦ *RP* makes a request for the user's resource to *IdP* using the access token;

⑧ *IdP* answers *RP* with the user's resource at *IdP*.

The implicit mode differs from the authorization code mode in steps ④-⑥. Instead of granting an authorization code to *RP*, the *IdP* provides an access token in the fragment identifier of the redirect URI. A piece of JavaScript code embedded in the page located at the redirect URI extracts the access token and communicates it to the *RP*.

3.2.2 Challenge #1: Protocol Flow

Protocols are specified in terms of a number of sequential message exchanges which honest participants are expected to follow, but the browser is not forced to comply with the intended protocol flow.

Example in OAuth 2.0. The use of the state parameter is recommended to prevent attacks leveraging this idiosyncrasy. When OAuth is used to implement SSO and *RP* does not provide the state parameter in its authorization request to *IdP* at step ②, it is possible to force the honest user's browser to authenticate as the attacker. This attack is known as *session swapping* [177].

We give a short overview on this attack against the authorization code mode. A web attacker *A* initiates SSO at *RP* with an identity provider *IdP*, performs steps ①-③ of the protocol and learns a valid authorization code for her session. Next, *A* creates a page on her website that, when visited, automatically triggers a request to the redirect URI of *RP* and includes the authorization code. When a honest user visits this page, the login procedure is completed at *RP* and an attacker session is established in the user's browser.

3.2.3 Challenge #2: Secrecy of Messages

The security of protocols typically relies on the confidentiality of cryptographic keys and credentials, but the browser is not aware of which data must be kept secret.

Example in OAuth 2.0. The secrecy of the authorization credentials (namely authorization codes and access tokens) is crucial for meeting the protocol security requirements, since their knowledge allows an attacker to access the user's resources. The secrecy of the state parameter is also important to ensure session integrity.

An example of an unintended secrets leakage is the *state leak* attack described in [60]. If the page loaded at the redirect URI in step ④ loads a resource from a malicious server, the state parameter and the authorization code (that are part of the URL) are leaked in the `Referer` header of the outgoing request. The learned authorization code can potentially be used to obtain a valid access token for *U* at *IdP*, while the leaked state parameter enables the session swapping attack discussed previously.

3.2.4 Challenge #3: Integrity of Messages

Protocol participants are typically expected to perform a number of runtime checks to prove the integrity of the messages they receive and ensure the integrity of the messages they send, but the browser cannot perform these checks unless they are explicitly carried out in a JavaScript implementation of the web protocol.

Example in OAuth 2.0. The *naïve RP session integrity* attack presented in [60] exploits this weakness. Suppose that *RP* supports SSO with various identity providers and uses different redirect URIs to distinguish between them. In this case, an attacker controlling a malicious identity provider *AIdP* can confuse the *RP* about which provider is being used and force the user's browser to login as the attacker.

To this end, the attacker starts a SSO login at *RP* with an honest identity provider *HIdP* to obtain a valid authorization code for her account. If an honest user starts a login procedure at *RP* with *AIdP*, in step ④ *AIdP* is expected to redirect the user to *AIdP*'s redirect URI at *RP*. If *AIdP* redirects to the redirect URI of *HIdP* with the authorization code from the attacker session, then *RP* mistakenly assumes that the user intended to login with *HIdP*. Therefore, *RP* completes the login with *HIdP* using the attacker's account.

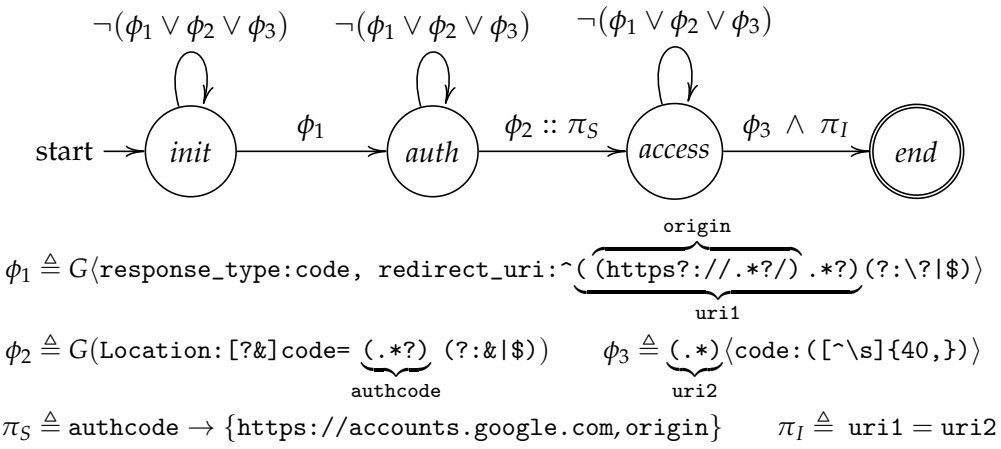


FIGURE 3.2: Automaton for OAuth 2.0 (authorization code mode) where G is the OAuth endpoint at Google.

3.3 WPSE: Design and Implementation

The *Web Protocol Security Enforcer* (WPSE) is the first browser-side security monitor addressing the peculiar challenges of web protocols. The current prototype is implemented as an extension for Google Chrome which we make available online [44].

3.3.1 Key Ideas of WPSE

We illustrate WPSE on the authorization code mode of OAuth 2.0 where Google is used as identity provider and the state parameter is not used (since it is not mandatory at Google). For simplicity, here we show only the most common scenario where the user has an ongoing session with the identity provider and the authorization to access the user's resources on the provider has been previously granted to the relying party.

Protocol Flow

WPSE describes web protocols in terms of the HTTP(S) exchanges observed by the web browser, following the so-called *browser relayed messages* methodology first introduced by Wang *et al.* [199]. The specification of the protocol flow defines the syntactic structure and the expected (sequential) order of the HTTP(S) messages, supporting the choice of different execution branches when a particular protocol message is sent or received by the browser. The protocol specification is given in XML but, for the sake of readability, here we use an equivalent representation in terms of finite state automata, like the one depicted in Figure 3.2. Intuitively, each state of the automaton represents one stage of the protocol execution in the browser. By sending an HTTP(S) request or receiving an HTTP(S) response as dictated by the protocol, the automaton steps to the next state until it reaches a final state denoting the end of the protocol run. Afterwards, the automaton moves back to the initial state and a new protocol run can start.

The edges of the automaton are labeled with *message patterns*, describing the expected shape of the protocol messages at each state. We represent HTTP(S) requests as $e\langle a \rangle$,

where e is the remote endpoint to which the message is sent and a is a list of parameters, while HTTP(S) responses are noted $e(h)$, where e is the remote endpoint from which the message is received and h is a list of headers.[3] The syntactic structure of e, a, h can be described using regular expressions. The message patterns should be considered as *guards* of the transitions, which are only enabled for messages matching the pattern. For instance, the pattern ϕ_2 in Figure 3.2 matches a response from the endpoint G with a `Location` header that contains a URL with a parameter named `code`. If an HTTP(S) request or response does not satisfy any of the patterns of the outgoing transitions of the current state, it is blocked and the automaton is reset to the initial state, *i.e.*, the protocol run is aborted. In case of branches with more than one transition enabled at a given state, we solve the non-determinism by picking the first matching transition according to the order defined in the XML specification. Patterns can be composed using standard logical connectives.

Each state of the automaton also allows for pausing the protocol execution in presence of requests and responses that are unrelated to the protocol. Unrelated messages are not of the shape of any valid message in the protocol specification. In the automaton, this is expressed by having a self-loop for each state, labeled with the negated disjunction of all patterns describing valid protocol messages. This is important for website functionality, because the input/output behaviour of browsers on realistic websites is complex and hard to fully determine when writing a protocol specification. Also, the same protocol may be run on different websites, which need to fetch different resources as part of their protocol-unrelated functionalities, and we would like to ensure that the same protocol specification can be enforced uniformly on all these websites.

Security Policies

To incorporate secrecy and integrity policies in the automaton, we allow for binding parts of message patterns to *identifiers*. For instance, in Figure 3.2 we bind the identifier `origin` to the content of the `redirect_uri` parameter, more precisely to the part matching the regular expression group `(https?://.*?/)`. The scope of an identifier includes the state where it is first introduced and all its successor states, where the notion of successor is induced by the tree structure of the automaton. For instance, the scope of the identifier `origin` introduced in ϕ_1 includes the states *auth*, *access*, *end*.

The *secrecy policy* defines which parts of the HTTP(S) responses included in the protocol specification must be confidential among a set of web origins. We express secrecy policies π_S with the notation $\mathtt{x} \rightarrow S$ to denote that the value bound to the identifier `x` can be disclosed only to the origins specified in the set S. We call S the *secrecy set* of identifier `x` and represent such a policy on the message pattern where the identifier `x` is first introduced, using a double colon symbol :: as a separator. For instance, in Figure 3.2 we require that the value of the authorization code, which is bound to the identifier `authcode` introduced in ϕ_2, can be disclosed only to Google (at `https://accounts.google.com`) and

[3] Our implementation supports HTTP headers also in requests. Here we omit them since they are not used in the analysed protocols.

the relying party (bound to the identifier `origin`). Confidential message components are stripped from HTTP(S) responses and substituted by random placeholders, so that they are isolated from browser accesses, *e.g.*, computations performed by JavaScript. When the automaton detects an HTTP(S) request including one of the generated placeholders, it replaces the latter with the corresponding original value, but only if the HTTP(S) request is directed to one of the origins which is entitled to learn it. A similar idea was explored by Stock and Johns to strengthen the security of password managers [176]. Since the substitution of confidential message components with placeholders changes the content of the messages, potentially introducing deviations with respect to the transition labels, the automaton processes HTTP(S) responses before stripping confidential values and HTTP(S) requests after replacing the placeholders with the original values. This way, the input/output behaviour of the automaton matches the protocol specification.

The *integrity policy* defines runtime checks over the contents of HTTP(S) messages. These checks allow for the comparison of incoming messages with those received in previous steps of the protocol execution. If any of the integrity checks fails, the corresponding message is not processed and the protocol run is aborted. To express integrity policies π_I in the automaton, we enrich the message patterns to include comparisons ranging over the identifiers introduced by preceding messages. In the case of OAuth 2.0, we would like to ensure that the browser is redirected by the *IdP* to the redirect URI specified in the first step of the protocol. Therefore, in Figure 3.2 the desired integrity policy is modeled by the condition $\texttt{uri1} = \texttt{uri2}$.

Enforcing Multiple Protocols

There are a couple of delicate points to address when multiple protocol specifications $P_1, \ldots, P_n$ must be enforced by WPSE:

1. if two different protocols P_i and P_j share messages with the same structure, there might be situations where WPSE does not know which of the two protocols is being run, yet a message may be allowed by P_i and disallowed by P_j or vice-versa;

2. if WPSE is enforcing a protocol P_i, it must block any message which may be part of another protocol P_j, otherwise it would be trivial to sidestep the security policy of P_i by first making the browser process the first message of P_j.

Both problems are solved by replacing the protocol specifications $P_1, \ldots, P_n$ with a single specification P with n branches, one for each P_i. Using this construction, any ambiguity on which protocol specification should be enforced is solved by the determinism of the resulting finite state automaton. Moreover, the self loops of the automaton will only match the messages which are not part of any of the n protocol specifications, thereby preventing unintended protocol interleavings. Notice that the semantics of WPSE depends on the order of $P_1, \ldots, P_n$, due to the way we enforce determinism on the compiled automaton: if P_i starts with a request to u including two parameters a and b, while P_j starts with a request to u including just the parameter a, then P_i should occur before P_j to ensure it is actually taken into account.

3.3.2 Discussion

A number of points of the design and the implementation of WPSE are worth discussing more in detail.

Protocol Flow

WPSE provides a significant improvement in security over standard web browsers, as we show in the remainder of the paper, but the protection it offers is not for free because it requires the specification of a protocol flow and a security policy. We think that it is possible to develop automated techniques to reconstruct the intended protocol flow from observable browser behaviours, while synthesizing the security policy looks more difficult. Manually finding the best security policy for a protocol may require significant expertise, but even simple policies can be useful to prevent a number of dangerous attacks, as we demonstrate in Section 3.4.

The specification style of the protocol flow supported by WPSE is simple, because it only allows sequential composition of messages and branching. As a result, our finite state automata are significantly simpler than the request graphs proposed by Guha *et al.* [74] to represent legitimate browser behaviours (from the server perspective). For instance, our finite state automata do not include loops and interleaving of messages, because it seems that these features are not extensively used in web protocols. Like standard security protocols, web protocols are typically specified in terms of a fixed number of sequential messages, which are appropriately supported by our specification language.

Secrecy Enforcement

The implementation of the secrecy policies of WPSE is robust, but restrictive. Since WPSE substitutes confidential values with random placeholders, only the latter are exposed to browser-side scripts. Shielding secret values from script accesses is crucial to prevent confidentiality breaches via untrusted scripts or XSS, but it might also break the website functionality if a trusted script needs to compute over a secret value exchanged in the protocol. The current design of WPSE only supports a limited use of secrets by browser-side scripts, *i.e.*, scripts can only forward secrets unchanged to the web origins entitled to learn them. We empirically show that this is enough to support existing protocols like OAuth 2.0 and SAML, but other protocols may require more flexibility.

Dynamic information flow control deals with the problem of letting programs compute over secret values while avoiding confidentiality breaches and it has been applied in the context of web browsers [71, 80, 26, 154, 22]. We believe that dynamic information flow control can be fruitfully combined with WPSE to support more flexible secrecy policies. This integration can also be useful to provide confidentiality guarantees for values which are generated at the browser-side and sent in HTTP(S) requests, rather than received in HTTP(S) responses. We leave the study of the integration of dynamic information flow control into WPSE to future work.

Detected Violation	Attack
Protocol flow deviation	Session swapping [177] Social login CSRF on stateless clients [12] IdP mix-up attack (web attacker) [60]
Secrecy violation	Unauthorized login by authentication code redirection [12] Resource theft by access token redirection [12] 307 redirect attack [60] State leak attack [60]
Integrity violation	Cross social-network request forgery [12] Naïve RP session integrity attack [60]

TABLE 3.1: Overview of the attacks against OAuth 2.0.

Extension APIs

The current prototype of WPSE suffers from some limitations due to the Google Chrome extension APIs. In particular, the body of HTTP messages cannot be modified by extensions, hence the secrecy policy cannot be implemented when secret values are embedded in the page contents or the corresponding placeholders are sent as POST parameters. Currently, we protect secret values contained in the HTTP headers of a response (*e.g.*, cookies or parameters in the URL of a `Location` header) and we only substitute the corresponding placeholders when they are communicated via HTTP headers or as URL parameters. Clearly this is not a limitation of our general approach but rather one of the extension APIs, which can be solved by implementing the security monitor directly in the browser or as a separate proxy application. Despite these limitations, we were able to test the current prototype of WPSE on a number of real-world websites with very promising results, as reported in Section 3.5.

3.4 Fortifying Web Protocols with WPSE

To better appreciate the security guarantees offered by WPSE, we consider two popular web protocols: OAuth 2.0 and SAML 2.0. The security of both protocols has already been studied in depth, so they are an excellent benchmark to assess the effectiveness of WPSE: we refer to [12, 60, 177] for security analyses of OAuth 2.0 and to [9, 10] for research studies on SAML. Remarkably, by writing down a precise security policy for SAML, we were able to expose a new critical attack against the Google implementation of the protocol.

3.4.1 Attacks Against OAuth 2.0

We review in this section several attacks against OAuth 2.0 from the literature, analysing whether they are prevented by our extension. We focus in particular on those presented

in [12, 60, 177], since they apply to the OAuth 2.0 flows presented in this work. In Table 3.1 we provide an overview of the attacks that WPSE is able to prevent, grouped according to the type of violation of the security properties that they expose.

Protocol Flow Deviations

This category covers attacks that force the user's browser to skip messages or to accept them in a wrong order. For instance, some attacks, *e.g.*, some variants of CSRF and session swapping, rely on completing a login in the user's browser that was not initiated before. This is a clear deviation from the intended protocol flow and, as a consequence, WPSE blocks these attacks.

We exemplify on the session swapping attack discussed in Section 3.2.2. Here the attacker tricks the user into sending a request containing the attacker's authorization credential (*e.g.*, the authorization code) to *RP* (step ④ of the protocol flow). Since the state parameter is not used, the *RP* cannot verify whether this request was preceded by a login request by the user. Our security monitor blocks the (out-of-order) request since it matches the pattern ϕ_3, which is allowed by the automaton in Figure 3.2 only in state *access*. Thus, the attack is successfully prevented.

Secrecy Violations

This category covers attacks where sensitive information is unintentionally leaked, *e.g.*, via the Referer header or because of the presence of open redirectors at *RP*. Sensitive data can either be leaked to untrusted third parties that should not be involved in the protocol flow (as in the state leak attack) or protocol parties that are not trusted for a specific secret (as in the 307 redirect attack). WPSE can prevent this class of attacks since the secrecy policy allows one to specify the origins that are entitled to receive a secret.

We illustrate how the monitor prevents these attacks in case of the state leak attack discussed in Section 3.2.3, focusing on the authorization code. In the attack, the authorization code is leaked via the `Referer` header of the request fetching a resource from the attacker website which is embedded in the page located at the redirect URI of *RP* (step ④ of the protocol). When the authorization code (`authcode`) is received (step ②), the monitor extracts it from the `Location` header and replaces it with a random placeholder before the request is processed by the browser. After step ④, the request to the attacker's website is sent, but the monitor does not replace the placeholder with the actual value of the authorization code since the secrecy set associated to `authcode` in π_S does not include the domain of the attacker.

Integrity Violations

This category contains attacks that maintain the general protocol flow, but the contents of the exchanged messages do not satisfy some integrity constraints required by the protocol. WPSE can prevent these attacks by enforcing browser-side integrity checks.

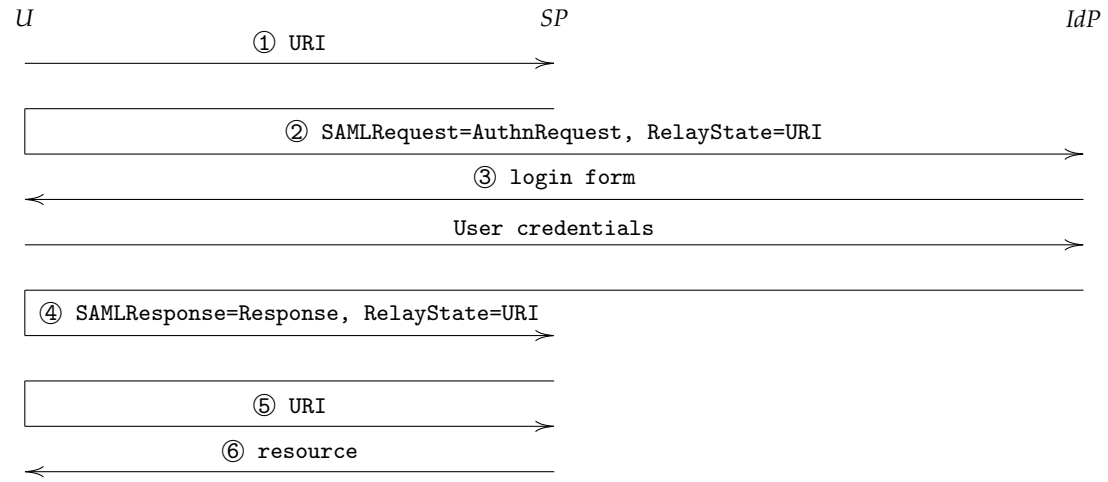


FIGURE 3.3: SAML 2.0 SP-Initiated SSO with Redirect/POST Bindings.

Consider the naïve RP session integrity attack presented in Section 3.2.4. In this attack, the malicious identity provider *AIdP* redirects the user's browser to the redirect URI of the honest identity provider *HIdP* at *RP* during step ④ of the protocol. At step ②, the redirect URI is provided to *AIdP* as parameter. This request corresponds to the pattern ϕ_1 of the automation and the redirect URI associated to *AIdP* is bound to the identifier `uri1`. At step ④, *AIdP* redirects the browser to a different redirect URI, which is bound to the identifier `uri2`. Although the shape of the request satisfies pattern ϕ_3, the monitor cannot move from state *access* to state *end* since the constraint `uri1 = uri2` in the integrity policy π_I is violated. Thus, no transition is enabled for the state *access* and the request is blocked by WPSE, therefore preventing the attack.

3.4.2 Attacks Against SAML 2.0

The *Security Assertion Markup Language* (SAML) 2.0 [141] is an open standard for sharing authentication and authorization across a multitude of domains. SAML is based on XML messages called *assertions* and defines different *profiles* to account for a variety of use cases and deployment scenarios. SSO functionality is enabled by the SAML 2.0 web browser SSO profile, whose typical use case is the *SP*-Initiated SSO with Redirect/POST Bindings [140, 10]. Similarly to OAuth 2.0, there are three entities involved: a user controlling a web browser (*U*), an identity provider (*IdP*) and a service provider (*SP*). The protocol prescribes how *U* can access a resource provided by *SP* after authenticating with *IdP*.

The relevant steps of the protocol are depicted in Figure 3.3. In step ①, *U* requests from *SP* the resource located at `URI`; in ② the *SP* redirects the browser to the *IdP* sending an `AuthnRequest` XML message and a `RelayState` parameter; *U* provides her credentials to the *IdP* in step ③ where they are verified; in step ④ the *IdP* causes the browser to issue a POST request to the Assertion Consumer Service at the *SP* containing the `SamlResponse` and the `RelayState` parameters; in ⑤ the *SP* processes the response, creates a security

context at the service provider and redirects U to the target resource at `URI`; given that a security context is in place, the *SP* provider returns the resource to U.

`RelayState` is a mechanism for preserving some state information at the *SP* such as the resource URI requested by the user [69]. If the `RelayState` parameter is used within a request message, then subsequent responses must maintain the exact value received with the request [139]. A violation of this constraint enables attacks such as [9], in which U requests a resource $\texttt{URI}_A$ hosted by a malicious service provider *AP*. *AP* pretends to be U at the honest *SP* and requests a different resource at *SP* located at $\texttt{URI}_S$ which is returned to *AP*. At step ② the malicious service provider replies to U by providing $\texttt{URI}_S$ in place of $\texttt{URI}_A$ as value of the *RelayState* parameter. The result is that U forcibly accesses a resource at *SP*, while she originally asked for a resource from *AP*. Interestingly, by using WPSE it is possible to instruct the browser with knowledge of the protocol in such a way that the client can verify whether the requests at steps ②,④ are related to the initial request ①. We distilled a simple policy for the SAML 2.0 web browser SSO profile that enforces an integrity constraint on the value of the `RelayState` parameter, thus blocking requests to undesired resources due to a violation of the policy.

Furthermore, SAML 2.0 does not specify any way to maintain a contextual binding between the request at step ② and the request at step ④. It follows that only the `SAMLResponse` and `RelayState` parameters are enough to allow U to access the resource at `URI`. We discovered that this shortcoming in the protocol has a critical impact on real *SP*s using the SAML-based SSO profile described in this section. Indeed, we managed to mount an attack against Google that allows a web attacker to authenticate any user on Google's suite applications under the attacker's account, with effects similar to a Login CSRF attack. Since Google can act as a Service Provider (*SP*) with a third party *IdP*, an attacker registered to a given *IdP* can simulate a login attempt with his legitimate credentials to obtain a valid POST request to the Google assertion consumer service (step ④). Using the learned parameters the attacker can construct a web page that cause the victim's browser to automatically issue a request to the Google assertion consumer service, thus forcing the victim inside the attacker session. From the browser standpoint, this attack is clearly caused by a violation of the protocol flow given that steps ①-③ are carried out by the attacker and step ④ and subsequent ones involve the victim. WPSE identifies the outgoing request to the *IdP* as a protocol flow deviation, thereby preventing the attack.

The vulnerability can be exploited by any web attacker with a valid account on a third party *IdP* that uses Google as *SP*. In particular, our university uses SAML 2.0 with Google as a Service provider to offer email and storage facilities to students and employees. We have implemented the attack by constructing a malicious webpage that silently performs a login on Google's suite applications using one of our personal accounts. The vulnerability allows the attacker to access private information of the victim that has been saved in the account, such as activity history, notes and documents. We have responsibly reported this vulnerability to Google who rewarded us according to their bug bounty program. Interestingly, the vulnerability cannot be fixed exclusively at the server-side, but a patch to

the browser is also required. In particular, Google has modified its browser Chrome to monitor the execution of SAML 2.0 and prompt a popup window to the user just before completing the protocol to confirm that she really wanted to authenticate using the specified account [68].

3.4.3 Out-of-Scope Attacks

We have shown that WPSE is able to block a wide range of attacks on existing web protocols. However, some classes of attacks cannot be prevented by browser-side security monitoring. Specifically, WPSE cannot prevent:

1. attacks which do not deviate from the expected protocol flow. An example of such an attack against OAuth 2.0 is the *automatic login CSRF* attack presented in [12], which exploits the lack of CSRF protection on the login form of the relying party to force an authentication to the identity provider. This class of attacks can be prevented by implementing appropriate defenses against known web attacks;

2. attacks which cause deviations from the expected protocol flow that are not observable by the browser. In particular, this class of attacks includes *network attacks*, where the attacker corrupts the traffic exchanged between the protocol participants. For instance, a network attacker can run the *IdP mix-up* attack from [60] when the first step of OAuth 2.0 is performed over HTTP. This class of attacks can be prevented by making use of HTTPS, preferably backed up by HSTS;

3. attacks which do not involve the user's browser at all. An example is the *impersonation* attack on OAuth 2.0 discussed in [177], where public information is used for authentication. Another example is the *DuoSec* vulnerability found on several SAML implementations [112] that exploits a bug in the XML libraries used by SPs to parse SAML messages. This class of attacks must be necessarily solved at the server-side.

3.5 Experimental Evaluation

Having discussed how WPSE can prevent several real-world attacks presented in the literature, we finally move to on-field experiments. The goal of the present section is assessing the practical security benefits offered by WPSE on existing websites in the wild, as well as to test the compatibility of its browser-side security monitoring with current web technologies and programming practices. To this end, we experimentally assessed the effectiveness of WPSE by testing it against websites using OAuth 2.0 to implement SSO at high-profile *IdP*s.

3.5.1 Experimental Setup

We developed a crawler to automatically identify existing OAuth 2.0 implementations in the wild. Our analysis is not meant to provide a comprehensive coverage of the deployment of OAuth 2.0 on the web, but just to identify a few popular identity providers and their relying parties to carry out a first experimental evaluation of WPSE.

We started from a comprehensive list of OAuth 2.0 identity providers[4] and we collected for each of them the list of the HTTP(S) endpoints used in their implementation of the protocol. Inspired by [185], our crawler looks for login pages on websites to find syntactic occurrences of these endpoints: after accessing a homepage, the crawler extracts a list of (at most) 10 links which may likely point to a login page, using a simple heuristic. It also retrieves, using the Bing search engine, the 5 most popular pages of the website. For all these pages, the crawler checks for the presence of the OAuth 2.0 endpoints in the HTML code and in the 5 topmost scripts included by them. By running our crawler on the Alexa 100k top websites, we found that Facebook (1,666 websites), Google (1,071 websites) and VK (403 websites) are the most popular identity providers in the wild.

We then developed a faithful XML representation of the OAuth 2.0 implementations available at the selected identity providers. There is obviously a large overlap between these specifications, though slight differences are present in practice, *e.g.*, the use of the `response_type` parameter is mandatory at Google, but can be omitted at Facebook and VK to default to the authorization code mode. For the sake of simplicity, we decided to model the most common use case of OAuth 2.0, *i.e.*, we assume that the user has an ongoing session with the identity provider and that authorization to access the user's resources on the provider has been previously granted to the relying party. For each identity provider we devised a specification that supports the OAuth 2.0 authorization code and implicit modes, with and without the optional state parameter, leading to 4 possible execution paths. Finally, we created a dataset of 90 websites by sampling 30 relying parties for each identity provider, covering both the authorization code mode and the implicit mode of OAuth 2.0. We have manually visited these websites with a browser running WPSE both to verify if the protocol run was completed successfully and to assess whether all the functionalities of the sites were working properly. In the following we report on the results of testing our extension against these websites from both a security and a compatibility point of view.

3.5.2 Security Analysis

We devised an automated technique to check whether WPSE can stop dangerous real-world attacks. Since we did not want to attack the websites, we focused on two classes of vulnerabilities which are easy to detect just by navigating the websites when using WPSE. The first class of vulnerabilities enables confidentiality violations: it is found when one of the placeholders generated by WPSE to enforce its secrecy policies is sent to an unintended web origin. The second class of vulnerabilities, instead, is related to the

[4] `https://en.wikipedia.org/wiki/List_of_OAuth_providers`

use of the state parameter: if the state parameter is unused or set to a predictable static value, then session swapping becomes possible (see Section 3.2.2). We can detect these cases by checking which protocol specification is enforced by WPSE and by making the state parameter secret, so that all the values bound to it are collected by WPSE when they are substituted by the placeholders used to enforce the secrecy policy.

We observed that our extension prevented the leakage of sensitive data on 4 different relying parties. Interestingly, we found that the security violation exposed by the tool are in all cases due to the presence of tracking or advertisements libraries such as Facebook Pixel,[5] Google AdSense,[6] Heap[7] and others. For example, this has been observed on `ticktick.com`, a website offering collaborative task management tools. The leakage is enabled by two conditions:

1. the website allows its users to perform a login via Google using the implicit mode;
2. the Facebook tracking library is embedded in the page which serves as redirect URI.

Under these settings, right after step ④ of the protocol, the tracking library sends a request to `https://www.facebook.com/tr/` with the full URL of the current page, which includes the access token issued by Google. We argue that this is a critical vulnerability, given that leaking the access token to an unauthorized party allows unintended access to sensitive data owned by the users of the affected website. We promptly reported the issue to the major tracking library vendors and the vulnerable websites. Library vendors informed us that they are not providing any fix since it is a responsibility of web developers to include the tracking library only in pages without sensitive contents.[8]

For what concerns the second class of vulnerabilities, 55 out of 90 websites have been found affected by the lack or misuse of the state parameter. More in detail, we identified 41 websites that do not support it, while the remaining 14 websites miss the security benefit of the state parameter by using a predictable or constant string as a value. We claim that such disheartening situation is mainly caused by the identity providers not setting this important parameter as mandatory. In fact, the state parameter is listed as recommended by Google and optional by VK. On the other hand, Facebook marks the state parameter as mandatory in its documentation, but our experiments showed that it fails to fulfill the requirement in practice. Additionally, it would be advisable to clearly point out in the OAuth 2.0 documentation of each provider the security implications of the parameter. For instance, according to the Google documentation,[9] the state parameter can be used "for several purposes, such as directing the user to the correct resource in your application, sending nonces, and mitigating cross-site request forgery": we believe that this description is too vague and opens the door to misunderstandings.

[5] `https://www.facebook.com/business/a/facebook-pixel`

[6] `https://www.google.com/adsense`

[7] `https://heapanalytics.com/`

[8] See, for instance, Google AdSense program policy available at `https://support.google.com/adsense/topic/6162392`

[9] `https://developers.google.com/identity/protocols/OAuth2WebServer`

3.5.3 Compatibility Analysis

To detect whether WPSE negatively affects the web browser functionality, we performed a basic navigation session on the websites in our dataset. This interaction includes an access to their homepage, the identification of the SSO page, the execution of the OAuth 2.0 protocol, and a brief navigation of the private area of the website. In our experiments, the usage of WPSE did not impact in a perceivable way the browser performance or the time required to load webpages. We were able to navigate 81 websites flawlessly, but we found 9 websites where we did not manage to successfully complete the protocol run.

In all the cases, the reason for the compatibility issues was the same, *i.e.*, the presence of an HTTP(S) request with a parameter called `code` after the execution of the protocol run. This message has the same syntactic structure as the last request sent as part of the authorization code mode of OAuth 2.0 and is detected as an attack when our security monitor moves back to its initial state at the end of the protocol run, because the message is indistinguishable from a session swapping attempt (*cf.* Section 3.2.2). We manually investigated these cases: 2 of them were related to the use of the Gigya social login provider, which offers a unified access interface to many identity providers including Facebook and Google; the other 7, instead, were due to a second exchange of the authorization code at the end of the protocol run. We were able to solve the first issue by writing a specification for Gigya (limited to Facebook and Google), while the other cases openly deviate from the OAuth 2.0 specification, where the authorization code is only supposed to be sent to the redirect URI and delivered to the relying party from there. These custom practices are hard to explain and to support and, unsurprisingly, may introduce security flaws. In fact, one of the websites deviating from the OAuth 2.0 specification suffers from a serious security issue since the authorization code is first communicated to the website over HTTP before being sent over HTTPS, thus becoming exposed to network attackers. We responsibly disclosed this security issue to the website owners.

In the end, all the compatibility issues we found boil down to the fact that a web protocol message has a relatively weak syntactic structure, which may end up matching a custom message used by websites as part of their functionality. We think that most of these issues can be robustly solved by using more explicit message formats for standardized web protocols like OAuth 2.0: explicitness is indeed widely recognized as a prudent engineering practice for traditional security protocols [1]. Having structured message formats could be extremely helpful for a precise browser-side fortification of web protocols which minimizes compatibility issues.

3.6 Formal Guarantees

Now we formally characterize the security guarantees offered by our monitoring technique. Here we provide an intuitive description of the formal result, referring the interested reader to [44] for a complete account.

The formal result states that given a web protocol that is proven secure for a set of network participants and an uncorrupted client, by our monitoring approach we can

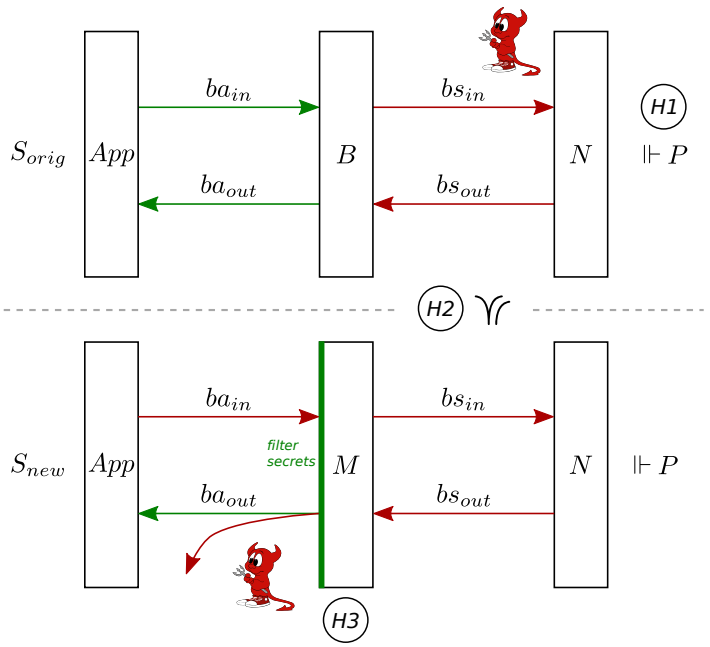


FIGURE 3.4: Visual description of Theorem 4.

achieve the same security guarantees given a corrupted client (*e.g.*, due to XSS attacks). More precisely this means that all attacks not occurring in presence of an ideally behaving client can be fixed by our monitor. Of course, these security guarantees only span the run of the protocol that is proven secure and its protocol-specific secrets. So the monitor can, *e.g.*, ensure that the OAuth 2.0 protocol is securely executed in the presence of compromised scripts which might result in successful authentication and the setting of a session cookie. However, the monitor cannot prevent that this session cookie is leaked by a malicious script after the protocol run is over. So other security techniques like the `HttpOnly` attribute for cookies have to be in place or the protocol specification must be extended to include the subsequent application steps (*e.g.*, we can protect session cookies like we do for access tokens).

Our theory is elaborated within the applied pi calculus [162], a popular process calculus for the formal analysis of cryptographic protocols which is supported by various automated verifiers such as ProVerif [29]. Bansal *et al.* [12] have recently presented a technique to leverage ProVerif for the analysis of web protocol specifications, including OAuth.

We give an overview on the theorem in Figure 3.4. We assume that the protocol specification has already been proven secure in a setting where the browser-side application is well-behaved and, in particular, follows the protocol specification (S_{orig}). Intuitively, our theorem says that security carries over to a setting (S_{new}) where the browser-side application is totally under the control of the attacker (*e.g.*, because of XSS attacks or a simple bug in the code) but the communication between the browser and the other protocol parties is mediated by our monitor. Specifically, S_{orig} includes a browser B and an uncompromised application *App* that exchange messages via private communication channels ba_{in}, ba_{out}.

The communication between the browser B and the network N is performed via the public channels bs_{in}, bs_{out} that can be observed and infiltrated by the network attacker. S_{new} shows the setting in which the application is compromised: channel ba_{in} for requests from the application to the browser is made public, modeling that arbitrary requests can be performed on it by the attacker. In addition, we assume the channel ba_{out} leaks all messages, thus modeling a compromised application that might disclose these secrets. Indeed, the compromised application can communicate with the network attacker, who can in turn use the learned information to attack the protocol. The following theorem states the correctness of our monitoring approach.

Theorem 4 (Monitor Correctness)**.** *Let processes App, N, B and M as defined in S_{orig} and P be a property on execution traces against a network attacker. Assume that the following conditions hold:*

(H1) $S_{orig} \vDash P$ *("S_{orig} satisfies P")*

(H2) $M \downarrow bs_{in}, bs_{out} \preccurlyeq S_{orig} \downarrow bs_{in}, bs_{out}$ *("the set of requests/responses on bs_{in},bs_{out} allowed by M are a subset of those produced by S_{orig}")*

(H3) M does not leak any secrets (i.e., *messages initially unknown to the attacker) on ba_{out}*

Then it also holds that:

(C) $S_{new} \vDash P$ *("S_{new} satisfies P")*

Assumption (H1) states that the process as shown in S_{orig} satisfies a certain trace property. In the applied pi calculus, this is modeled by requiring that each partial execution trace of S_{orig} in parallel with an arbitrary network attacker satisfies the trace predicate P. Assumption (H2) states that the requests/responses allowed by the monitor M on the channels bs_{in}, bs_{out}, which model the communication between the browser and the network, are a subset of those possibly performed by the process S_{orig}. Intuitively, this means that the monitor allows for the intended protocol flow, filtering out messages deviating from it. Formally this is captured by projecting the execution traces of the corresponding processes to those components that model the input and output behaviour on bs_{in} and bs_{out} and by requiring that for every such execution trace of M there is a corresponding one for S_{orig}. Finally, assumption (H3) states that the monitor M should not leak any secrets on channel ba_{out}. In applied pi calculus this is captured by requiring that the outputs of M on channel ba_{out} do not increase the attacker's knowledge.

Together these assumptions ensure that the monitored browser behaves as the ideal protocol participant in S_{orig} towards the network and additionally assure that an attacker cannot gain any additional knowledge via a compromised application that could enable her to perform attacks against the protocol over the network. Formally, this is captured in conclusion (C) that requires the partial execution traces of S_{new} to satisfy the trace predicate P.

3.6.1 Discussion

Our formal result is interesting for various reasons. First, it allows us to establish formal security guarantees in a stronger attacker model by checking certain semantic conditions on the monitor, without having to prove from scratch the security of the protocol with the monitor in place on the browser-side. Second, the theorem demonstrates that enforcing the three security properties identified in Section 3.2 does indeed suffice to protect web protocols from a large class of bugs and vulnerabilities on the browser-side: (H2) captures the compliance with the intended protocol flow as well as data integrity, while (H3) characterizes the secrecy of messages.

Finally, the three hypotheses of the theorem are usually extremely easy to check. For instance, let us consider the OAuth protocol. As previously mentioned, this has been formally analysed in [12], so (H1) holds true. In particular, the intended protocol flow is directly derivable from the applied pi calculus specification. The automaton in Figure 3.2 only allows for the intended protocol flow, which is clearly contained in the execution traces analysed in [12]. Hence (H2) holds true as well. Finally, the only secrets in the protocol specification are those subject to the confidentiality policy in the automaton in Figure 3.2: as previously mentioned, these are replaced by placeholders, which are then passed to the web application. Hence no secret can ever leak, which validates (H3).

3.7 Related Work

3.7.1 Analysis of Web Protocols

The first paper to highlight the differences between web protocols and traditional cryptographic protocols is due to Gross *et al.* [72]. The paper presented a model of web browsers, based on a formalism reminiscent of input/output automata, and applied it to the analysis of password-based authentication. The model was later used to formally assess the security of the WSFPI protocol [73].

Traditional protocol verification tools have been successfully applied to find attacks in protocol specifications. For instance, Armando *et al.* analysed both the SAML protocol and a variant of the protocol implemented by Google using the SATMC model-checker [10]. Their analysis exposed an attack against the authentication goals of the Google implementation. Follow-up work by the same group used a more accurate model to find an authentication flaw also in the original SAML specification [9]. Akhawe *et al.* used the Alloy framework to develop a core model of the web infrastructure, geared towards attack finding [4]. The paper studied the security of the WebAuth authentication protocol among other case studies, finding a login CSRF attack against it. The WebSpi library for ProVerif by Bansal *et al.* has been successfully applied to find attacks against existing web protocols, including OAuth 2.0 [12] and cloud storage protocols [13]. Fett *et al.* developed the most comprehensive model of the web infrastructure available to date and fruitfully applied it to the analysis of a number of web protocols, including BrowserID [61], SPRESSO [62] and OAuth 2.0 [60].

Protocol analysis techniques are useful to verify the security of protocols, but they assume websites are correctly implemented and do not depart from the specification, hence many security researchers performed empirical security assessments of existing web protocol implementations, finding dangerous attacks in the wild. Protocols which deserved attention by the research community include SAML [172], OAuth 2.0 [177, 107] and OpenID Connect [106]. Automated tools for finding vulnerabilities in web protocol implementations have also been proposed by security researchers [199, 215, 207, 114]. None of these works, however, presented a technique to protect users accessing vulnerable websites in their browsers.

3.7.2 Security Automata

The use of finite state automata for security enforcement is certainly not new. The pioneering work in the area is due to Schneider [167], which first introduced a formalization of security automata and studied their expressive power in terms of a class of enforceable policies. Security automata can only stop a program execution when a policy violation is detected; later work by Ligatti *et al.* extended the class of security automata to also include edit automata, which can suppress and insert individual program actions [109]. Edit automata have been applied to the web security setting by Yu *et al.*, who used them to express security policies for JavaScript code [209]. The focus of their paper, however, is not on web protocols and is only limited to JavaScript, because input/output operations which are not JavaScript-initiated are not exposed to their security monitor.

Guha *et al.* also used finite state automata to encode web security policies [74]. Their approach is based on three steps: first, they apply a static analysis for JavaScript to construct the control flow graph of an AJAX application to protect and then they use it to synthesize a request graph, which summarizes the expected input/output behaviour of the application. Finally, they use the request graph to instruct a server-side proxy, which performs a dynamic monitoring of browser requests to prevent observable violations to the expected control flow. The security enforcement can thus be seen as the computation of a finite state automaton built from the request graph. Their technique, however, is only limited to AJAX applications and operates at the server-side, rather than at the browser-side.

3.7.3 Browser-Side Defenses

Our work positions itself in the popular research line of extending web browsers with stronger security policies. To the best of our knowledge, this is the first work which explicitly focuses on web protocols, but a number of other proposals on browser-side security are worth mentioning. Enforcing information flow policies in web browsers is a hot topic nowadays and a few fairly sophisticated proposals have been published as of now [71, 80, 26, 154, 22]. Information flow control can be used to provide confidentiality and integrity guarantees for browser-controlled data, but it cannot be directly used to detect deviations from expected web protocol executions, which instead are naturally

captured by security automata. Combining our approach with browser-based information flow control can improve its practicality, because a more precise information flow tracking would certainly help a more permissive security enforcement.

A number of browser changes and extensions have been proposed to improve web session security, both from the industry and the academia. Widely deployed industrial proposals include Content Security Policy (CSP) and HTTP Strict Transport Security (HSTS). Notable proposals from the academia include Allowed Referrer Lists [51], SessionShield [137], Zan [178], CSFire [163], Serene [165], CookiExt [37] and SessInt [38]. We refer to Chapter 2 for a comprehensive overview of the existing techniques. None of these works, however, tackles web protocols.

Chapter 4

Mind Your Keys? A Security Evaluation of Java Keystores

4.1 Introduction

Cryptography is a fundamental technology for IT security. Even if there are well established standards for cryptographic operations, cryptography is complex and variegated, typically requiring a non-trivial combination of different algorithms and mechanisms. Moreover, cryptography is intrinsically related to the secure management of cryptographic keys which need to be protected and securely stored by applications. Leaking cryptographic keys, in fact, voids any advantage of cryptography, allowing attackers to break message confidentiality and integrity, to authenticate as legitimate users or impersonate legitimate services. Quoting [168], "key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system".

In the recent years we have faced a multitude of flaws related to cryptography (*e.g.*, [24, 14, 122, 121]). Some of these are due to the intrinsic complexity of cryptography, that makes it hard to design applications that adopt secure combinations of mechanisms and algorithms. For example, in padding oracle attacks, the usage of some (standard) padding for the plaintext combined with a specific algorithm or mechanism makes it possible for an attacker to break a ciphertext in a matter of minutes or hours [187, 30, 14]. Most of the time this is not a developer fault as, unfortunately, there are well-known flawed mechanisms that are still enabled in cryptographic libraries. In other cases, the attacks are due to flaws in protocols or applications. The infamous Heartbleed bug allowed an attacker to get access to server private keys through a simple over-read vulnerability. Once the private key was leaked, the attacker could decrypt network traffic or directly impersonate the attacked server [122].

Thus, breaking cryptography is not merely a matter of breaking a cryptographic algorithm: the attack surface is quite large and the complexity of low-level details requires abstractions. Crypto APIs offer a form of abstraction to developers that allows to make use of cryptography in a modular and implementation-independent way. The Java platform, for example, provides a very elegant abstraction of cryptographic operations that makes it possible, in many cases, to replace a cryptographic mechanism or its implementation with a different one without modifying the application code.

Crypto APIs, however, do not usually provide security independently of the low-level implementation: default mechanisms are often the weakest ones, thus developers have to face the delicate task of choosing the best mechanism available for their needs. For example, in the Java Cryptography Architecture (JCA), ECB is the default mode of operation for block ciphers [87] and PKCS#1 v1.5 is the default padding scheme for RSA [90], which is well know to be subject to padding oracle attacks [30]. Additionally, crypto APIs that promise to provide security for cryptographic keys have often failed to do so: in PKCS#11, the standard API used with cryptographic tokens, it is possible to wrap a sensitive key under another key and then just ask the device to decrypt it, obtaining the value of the sensitive key in the clear [47], and violating the requirement that "sensitive keys cannot be revealed in plaintext off the token" [158].

In this chapter we analyse in detail the security of key management in the Java ecosystem and, in particular, of Java keystores. Password-protected keystores are the standard way to securely manage and store cryptographic keys in Java: once the user (or the application) provides the correct password, the keys in the keystore become available and can be used to perform cryptographic operations, such as encryption and digital signature. The `KeyStore` Java class abstracts away from the actual keystore implementation, which can be either in the form of an encrypted file or based on secure hardware. As discussed above, this abstraction is very important for writing code that is independent of the implementation but developers are still required to select among the various keystore *types* offered by Java. Unfortunately, the information in the keystore documentation is not enough to make a reasoned and informed choice among the many alternatives. More specifically, given that the Java Keystore API does not provide control over the cryptographic mechanisms and parameters employed by each keystore, it is crucial to assess the security provided by the different implementations, which motivated us to perform the detailed analysis reported in this Chapter. In fact, our work is the first one studying the security of keystores for general purpose Java applications.

We have estimated the adoption rate and analysed the implementation details of seven different Java keystores offered by the Oracle JDK and by Bouncy Castle, a famous cryptographic library. Keystores are used by hundreds of commercial applications and open-source projects, as assessed by scraping GitHub, including leading web applications servers and frameworks, *e.g.*, Tomcat [7], Spring [173], Oracle Weblogic [201]. Additionally, keystores have been found to be widespread among security-critical custom Java software for large finance, government and healthcare companies.

The security of keystores is achieved by performing a cryptographic operation C under a key derived from a password through a function F called Key Derivation Function (KDF). The aim of the cryptographic operation C is to guarantee confidentiality and/or integrity of the stored keys. For example, a Password-Based Encryption (PBE) scheme is used to protect key confidentiality: in this case C is typically a symmetric cipher, so that keys are encrypted using the provided password before being stored in the keystore. In order to retrieve and use that key, the keystore implementation will perform the following steps: *i)* obtain the password from the user; *ii)* derive the encryption key from

the password using *F*; *iii)* decrypt the particular keystore entry through *C* and retrieve the actual key material. Notice that different passwords can be used to protect different keys and/or to achieve integrity. To prevent attacks, it is recommended that *C* and *F* are implemented using standard, state-of-the-art cryptographic techniques [126, 160].

Interestingly, we have found that the analysed keystores use very diverse implementations for *C* and *F* and in several cases they do not adhere to standards or use obsolete and ad-hoc mechanisms. We show that, most of the time, keystores using weak or custom implementations for the key derivation function *F* open the way to password brute-forcing. We have empirically measured the speed-up that the attacker achieves when these flawed keystores are used and we show that, in some cases, brute-forcing is three orders of magnitude faster with respect to the keystores based on standard mechanisms. We even found keystores using the deprecated cipher RC2 that enables an attacker to brute-force the 40-bit long key in a matter of hours using a standard desktop computer.

Our analysis has also pointed out problems related to availability and malicious code execution, which are caused by *type-flaws* in the keystore, *i.e.*, bugs in which an object of a certain type is interpreted as one of a different type. In particular, by directly tampering with the keystore file, an attacker could trigger denial of service (DoS) attacks or even arbitrary code execution. Interestingly, we also found that the use of standard key derivation functions can sometimes enable DoS attacks. These functions are parametrized by the number of internal iterations, used to slow down brute-forcing, which is stored in the keystore file. If the number of iterations is set to a very big integer, the key derivation function will hang, blocking the whole application.

Unless stated otherwise, our findings refer to Oracle JDK 8u144 and Bouncy Castle 1.57, the two latest releases available in August 2017.

4.1.1 Contributions

Our contributions can be summarized as follows:

1. we define a general threat model for password-protected keystores and we distill a set of significant security properties and consequent rules that any secure keystore should adhere to;

2. we perform a thoughtful analysis of seven keystores, we report undocumented details about their cryptographic implementations and we classify keystores based on our properties and rules;

3. we report on unpublished attacks and weaknesses found in the analysed keystores. For each attack we point out the corresponding violations of our proposed properties and rules and we provide a precise attacker model;

4. we empirically estimate the speed-up due to bad cryptographic implementations and we show that, in some cases, this allows to decrease the guessing time of three orders of magnitude with respect to the most resistant keystore, and four orders of magnitude with respect to NIST recommendations; interestingly, the attack on the

Oracle JKS keystore that we present in this Chapter and previously mentioned in a blog post [49], has been integrated into the Hashcat password recovery tool;

5. we discuss the advancements on the security of Oracle and Bouncy Castle keystore implementations following our responsible disclosure. The Oracle Security Team acknowledged the reported issues by assigning three CVE IDs [123, 124, 125] and released fixes in October 2017 and April 2018 [144, 145]. Bouncy Castle developers patched the reported vulnerabilities in version 1.58 and later releases.

4.1.2 Structure of the Chapter

We discuss related work in Section 4.2; in Section 4.3 we define the security properties of interest, the rules for the design of secure keystores and the threat model; in Section 4.4 we report on our analysis of seven Java keystores; in Section 4.5 we describe unpublished attacks on the analysed keystores; in Section 4.6 we make an empirical comparison of the password cracking speed among the keystores; in Section 4.7 we discuss the improvements implemented by Oracle and Bouncy Castle following our responsible disclosure.

4.2 Related Work

Cooijmans *et al.* [48] have studied various key storage solutions in Android, either provided as an operating system service or through the Bouncy Castle cryptographic library. The threat model is very much tailored to the Android operating system and radically different from the one we consider in our work. Offline brute-forcing, for example, is only discussed marginally in the paper. Interestingly, authors show that under a root attacker (*i.e.*, an attacker with root access to the device), the Bouncy Castle software implementation is, in some respect, more secure than the Android OS service using TrustZone capabilities, because of the possibility to protect the keystore with a user-supplied password. Differently from our work, the focus of the paper is not on the keystore design and the adopted cryptographic mechanisms.

Sabt *et al.* [166] have recently found a forgery attack in the Android KeyStore service, an Android process that offers a keystore service to applications and is out of the scope of our work. However, similarly to our results, the adopted encryption scheme is shown to be weak and not compliant to the recommended standards, enabling a forgery attack that make apps use insecure cryptographic keys, voiding any benefit of cryptography.

Li *et al.* [108] have analysed the security of web password managers. Even if the setting is different, there are some interesting similarities with keystores. In both settings a password is used to protect sensitive credentials, passwords in one case and keys in the other. So the underlying cryptographic techniques are similar. However the kind of vulnerabilities found in the paper are not related to cryptographic issues. Gasti *et al.* [67] have studied the format of password manager databases. There is some similarity with our work for what concerns the threat model, *e.g.*, by considering an attacker that can

tamper with the password database. However, the setting is different and the paper does not account for cryptographic weaknesses and brute-forcing attacks.

Many papers have studied password resistance to guessing, *e.g.*, [101, 39, 203, 213]. While this is certainly a very important subject, our work takes a complementary perspective: we analyse whether Java keystores provide a sufficient resistance to brute-forcing, compared to existing standards and recommendations. Of course, using a tremendously weak password would make it possible for the attacker to guess it, independently of the keystore implementation. Similarly, if the password is very long and with high entropy, the guess would be infeasible anyway. However, when a password is reasonably strong, the actual implementation makes a difference: brute-force is prevented only when key derivation is done accordingly to recommendations.

Kelsey *et al.* introduced the notion of *key stretching*, a mechanism to increase the time of brute-forcing for low entropy keys [102]. The basic idea is that key derivation should iterate the core derivation function l times so to multiply the computational cost of brute-forcing by l and make it equivalent to the cost of brute-forcing a password with additional $log_2 l$ bits. Intuitively, through this strategy, brute-forcing each password requires the same time as brute-forcing l passwords. Combined with standard random salting (to prevent precomputation of keys), key stretching effectively slows down brute-forcing, and prevents guessing the password even when its complexity is not very high. This idea is at the base of modern, state-of-the-art key derivation functions. In [2, 208, 23], this mechanism has been formalized and analysed, providing formal evidence of its correctness. Standard key derivation functions are all based on key stretching and salting to slow down brute-forcing [126, 160]. In our work we advocate the use of these standard mechanisms for keystores security.

4.3 Security Properties and Threat Model

In this section, we identify a set of fundamental security properties that should be guaranteed by any keystore (Section 4.3.1). We then distill rules that should be followed when designing a keystore in order to achieve the desired security properties (Section 4.3.2). Finally, we introduce the threat model covering a set of diverse attacker capabilities that enable realistic attack scenarios (Section 4.3.3).

4.3.1 Security Properties

We consider standard security properties such as confidentiality and integrity of keys and keystore entries. Breaking confidentiality of sensitive keys allows an attacker to intercept all the encrypted traffic or to impersonate the user. Breaking integrity has similar severe consequences as it might allow an attacker to import fake CA certificates and old expired keys. Additionally, since the access to a keystore is mediated by a software library or an application, we also consider the effect that a keystore has on the execution environment. Thus, we target the following properties:

P1 Confidentiality of encrypted entries

P2 Integrity of keystore entries

P3 System integrity

Property **P1** states that the value of an encrypted entry should be revealed only to authorized users, who know the correct decryption password. According to **P2**, keystore entries should be modified, created or removed only by authorized users, who know the correct integrity password, usually called *store password*. Property **P3** demands that the usage of a keystore should always be tolerated by the environment, *i.e.*, interacting with a keystore, even provided by an untrusted party, should not pose a threat to the system, cause misbehaviours or hang the application due to an unsustainable performance hit.

A keystore file should be secured similarly to a password file: the sensitive content should not be disclosed even when the file is leaked to an attacker. In fact, it is often the case that keystores are shared in order to provide the necessary key material to various corporate services and applications. Thus, in our threat model we will always assume that the attacker has read access to the keystore file (*cf.* Section 4.3.3). For this reason we require that the above properties hold even in the presence of offline attacks. The attacker might, in fact, brute-force the passwords that are used to enforce confidentiality and integrity and, consequently, break the respective properties.

4.3.2 Design Rules

We now identify a set of core rules that should be embraced by the keystore design in order to provide the security guarantees of Section 4.3.1:

R1 Use standard, state-of-the-art cryptography

R2 Choose strong, future-proof cryptographic parameters, while maintaining acceptable performance

R3 Enforce a typed keystore format

Rule **R1** dictates the use of modern and verified algorithms to achieve the desired keystore properties. It is well-known that the design of custom cryptography is a complex task even for experts, whereas standard algorithms have been carefully analysed and withstood years of cracking attempts by the cryptographic community [15]. In this context, the National Institute of Standards and Technology (NIST) plays a prominent role in the standardization of cryptographic algorithms and their intended usage [16], engaging the cryptographic community to update standards according to cryptographic advances. For instance, NIST declared SHA1 unacceptable to use for digital signatures beginning in 2014, and more recently, urged all users of Triple-DES to migrate to AES for encryption as soon as possible [184] after the findings published in [25]. The KDF function recommended by NIST [182] is PBKDF2, as defined in the PKCS#5 standard, which supersedes

the legacy PBKDF1. Another standard KDF function is defined in PKCS#12, although it has been deprecated for confidentiality purposes in favour of PBKDF2.

Key derivation functions combine the password with a randomly generated salt and iteratively apply a pseudorandom function (*e.g.*, a hash function) to produce a cryptographic key. The salt allows the generation of a large set of keys corresponding to each password [208], while the high number of iterations is introduced to hinder brute-force attacks by significantly increasing computational times. Rule **R2** reflects the need of choosing parameters to keep pace with the state-of-the-art in cryptographic research and the advances in computational capabilities. The latest NIST draft on Digital Identity Guidelines [70] sets the minimum KDF iteration count to 10,000 and the salt size to 32 bits. However, such lower bounds on the KDF should be significantly raised for critical keys according to [182] which suggests to set the number of iterations as high as can be tolerated by the environment, while maintaining acceptable performance. For instance, Apple iOS derives the decryption key for the device from the user password using a KDF with an iteration count calculated by taking into account the computational capabilities of the hardware and the impact on the user experience [8].

Finally, rule **R3** states that the keystore format must provide strong typing for keystore content, such that cryptographic objects are stored and read unambiguously. Despite some criticism over the years [76], the PKCS#12 standard embraces this principle providing precise types for storing many cryptography objects. Additionally, given that keystore files are accessed and modified by different parties, applications parsing the keystore format must be designed to be robust against malicious crafted content.

Interestingly, not following even one of the aforementioned rules may lead to a violation of confidentiality and integrity of the keystore entries. For instance, initializing a secure KDF with a constant or empty salt, which violates **R2**, would allow an attacker to precompute the set of possible derived keys and take advantage of *rainbow tables* [143] to speed up the brute-force of the password. On the other hand, a KDF with strong parameters is useless once paired with a weak cipher, since it is easier to retrieve the encryption key rather than brute-forcing the password. In this case only **R1** is violated.

Additionally, disrespecting Rule **R3** may have serious consequences on system integrity (breaking property **P3**), which range from applications crashing due to parsing errors while loading a malicious keystore to more severe scenarios where the host is compromised. An attacker exploiting type-flaw bugs could indirectly gain access to the protected entries of a keystore violating the confidentiality and integrity guarantees. System integrity can additionally be infringed by violating Rule **R2** with an inadequate parameter choice, *e.g.*, an unreasonably high iteration count value might hang the application, slow down the system or prevent the access to cryptographic objects stored in a keystore file due to an excessive computational load. In Section 4.5 we show how noncompliance to these rules translates into concrete attacks.

4.3.3 Threat Model

In our standard attacker model we always assume that the attacker has read access to the keystore file, either authorized or by means of a data leakage. We also assume that the attacker is able to perform offline brute-force attacks using a powerful system of her choice. We now present a list of interesting attacker settings, that are relevant with respect to the security properties defined in Section 4.3.1:

S1 Write access to the keystore

S2 Integrity password is known

S3 Confidentiality password of an entry is known

S4 Access to previous legitimate versions of the keystore file

Setting **S1** may occur when the file is shared over a network filesystem, *e.g.*, in banks and large organizations. Since keystores include mechanisms for password-based integrity checks, it might be the case that they are shared with both read and write permissions, to enable application that possess the appropriate credentials (*i.e.*, the integrity password) to modify them. We also consider the case **S2** in which the attacker possesses the integrity password. The password might have been leaked or discovered through a successful brute-force attack. The attacker might also know the password as an insider, *i.e.*, when she belongs to the organization who owns the keystore. Setting **S3** refers to a scenario in which the attacker knows the password used to encrypt a sensitive object. Similarly to the previous case, the password might have been accessed either in a malicious or in an honest way. For example, the password of the key used to sign the `apk` of an Android application [6] could be shared among the developers of the team.

In our experience, there exists a strong correlation between **S2** and **S3**. Indeed, several products and frameworks use the same password both for confidentiality and for integrity, *e.g.*, Apache Tomcat for TLS keys and IBM WebSphere for LTPA authentication. Additionally, the standard utility for Java keystores management (*keytool*) supports this practice when creating a key: the tool invites the user to just press the `RETURN` key to reuse the store password for encrypting the entry.

To summarize, our standard attacker model combined with **S1**-**S3** covers both reading and writing capabilities of the attacker on the keystore files together with the possibility of passwords leakage. On top of these settings, we consider the peculiar case **S4** that may occur when the attacker has access to backup copies of the keystore or when the file is shared over platforms supporting version control such as *Dropbox*, *ownCloud* or *Seafile*.

4.4 Analysis of Java Keystores

The Java platform exposes a comprehensive API for cryptography through a *provider*-based framework called Java Cryptography Architecture (JCA). A provider consists of a set of classes that implement cryptographic services and algorithms, including keystores.

In this section, we analyse the most common Java software keystores implemented in the Oracle JDK and in a widespread cryptographic library called Bouncy Castle that ships with a provider compatible with the JCA. In particular, since the documentation was not sufficient to assess the design and cryptographic strength of the keystores, we performed a comprehensive review of the source code exposing, for the first time, implementation details such as on-disk file structure and encoding, standard and proprietary cryptographic mechanisms, default and hard-coded parameters.

For reader convenience, we provide a brief summary of the cryptographic mechanisms and acronyms used in this section: Password-Based Encryption (PBE) is an encryption scheme in which the cryptographic key is derived from a password through a Key Derivation Function (KDF); a Message Authentication Code (MAC) authenticates data through a secret key and HMAC is a standard construction for MAC which is based on cryptographic hash functions; Cipher Block Chaining (CBC) and Counter with CBC-MAC (CCM) are two standard modes of operation for block ciphers, the latter is designed to provide both authenticity and confidentiality.

4.4.1 Oracle Keystores

The Oracle JDK offers three keystore implementations, namely JKS, JCEKS and PKCS12, which are respectively made available through the providers SUN, SunJCE and SunJSSE [146]. While JKS and JCEKS rely on proprietary algorithms to enforce both the confidentiality and the integrity of the saved entries, PKCS12 relies on open standard format and algorithms as defined in [159].

JKS

Java KeyStore (JKS) is the first official implementation of a keystore that appeared in Java since the release of JDK 1.2. It is the default keystore up to Java 8 when no explicit choice is made by the developer. It supports encrypted private key entries and public key certificates stored in the clear. The file format consists of a header containing the magic file number, the keystore version and the number of entries, which is followed by the list of entries. The last part of the file is a digest used to check the integrity of the keystore. Each entry contains the type of the object (key or certificate) and the label, followed by the cryptographic data.

Private keys are encrypted using a custom stream cipher designed by Sun, as reported in the OpenJDK source code. In order to encrypt data, a keystream W is generated in 20-bytes blocks with W_0 being a random salt and $W_i = SHA1(password||W_{i-1})$. The encrypted key E is computed as the XOR of the private key K with the keystream W, hence K and E share the same length. The ciphertext is then prepended with the salt and appended with the checksum $CK = SHA1(password||K)$. The block diagram for decryption is shown in Figure 4.1.

The integrity of the keystore is achieved through a custom hash-based mechanism: JKS computes the SHA1 hash of the integrity password, concatenated with the constant

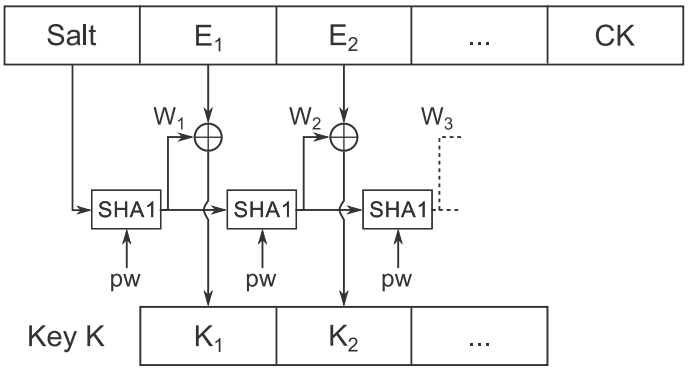


FIGURE 4.1: Decryption in the custom stream cipher used by JKS.

string "`Mighty Aphrodite`" and the keystore content. The result is then checked against the 20 bytes digest at the end of the keystore file.

JCEKS

Java Cryptography Extension KeyStore (JCEKS) has been introduced after the release of JDK 1.2 in the external Java Cryptography Extension (JCE) package and merged into the standard JDK distribution from version 1.4. According to the Java documentation, it is an alternate proprietary keystore format to JKS "that uses much stronger encryption in the form of Password-Based Encryption with Triple-DES" [87]. Besides the improved PBE mechanism, it allows for storing also symmetric keys.

The file format is almost the same of JKS with a different magic number in the file header and support for the symmetric key type. The integrity mechanism is also borrowed from JKS.

JCEKS stores certificates as plaintext, while the PBE used to encrypt private keys, inspired by PBES1 [126], is based on 20 MD5 iterations and a 64 bits salt. Given that Triple-DES is used to perform the encryption step, the key derivation process must be adapted to produce cipher parameters of the adequate size. In particular, JCEKS splits the salt in two halves and applies the key derivation process for each of them. The first 192 bits of the combined 256 bits result are used as the Triple-DES key, while the remaining 64 bits are the initialization vector.

PKCS12

The PKCS12 keystore supports both private keys and certificates, with support for secret keys added in Java 8. Starting from Java 9, Oracle replaced JKS with PKCS12 as the default keystore type [91].

The keystore file is encoded as an ASN.1 structure according to the specification given in [159]. It contains the version number of the keystore, the list of keys and the certificates. The last part of the keystore contains an HMAC (together with the parameters for its computation) used to check the integrity of the entire keystore by means of a password.

The key derivation process, used for both confidentiality and integrity, is implemented as described in the PKCS#12 standard [159] using SHA1 as hashing function, 1024 iterations and a 160 bit salt. Private keys and secret keys (when supported) are encrypted using Triple-DES in CBC mode. Certificates are encrypted in a single encrypted blob, using the RC2 cipher in CBC mode with a 40-bit key. While each key can be encrypted with a different password, all the certificates are encrypted reusing the store password.

4.4.2 Bouncy Castle Keystores

Bouncy Castle is a widely used open-source crypto API. As of 2014, it provides the base implementation for the crypto library used in the Android operating system [48]. It supports four different keystore types via the BC provider: BKS, UBER, BCPKCS12 and the new FIPS-compliant BCFKS. Similarly to Oracle keystores, all BC keystores rely on passwords to enforce confidentiality over the entries and to verify keystore integrity.

BKS

The Bouncy Castle Keystore (BKS) allows to store public/private keys, symmetric keys and certificates. BKS relies on a custom file structure to store the entries. The file contains the version number of the keystore, the list of stored cryptographic entries and an HMAC, along with its parameters, computed over the entries as integrity check.

Only symmetric and private keys can be encrypted in BKS, with Triple-DES in CBC mode. The key derivation schema is taken from PKCS#12 v1.0, using SHA1 as hashing function, a random number of iterations between 1024 and 2047 which is stored for each entry and a 160 bit salt.

The integrity of the keystore is provided by an HMAC using the same key derivation scheme used for encryption and applied to the integrity password. For backward compatibility, the current version of BKS still allows to load objects encrypted under a buggy PBE mechanism used in previous versions of the keystore[1]. If the key is recovered using an old mechanisms, it is immediately re-encrypted with the newer PBE scheme.

UBER

UBER shares most of its codebase with BKS, thus it supports the same types of entries and PBE. Additionally, it provides an extra layer of encryption for the entire keystore file, which means that all metadata around the keys and certificates are encrypted as well. The PBE mechanism used for encrypting the file is Twofish in CBC mode with a key size of 256 bits. The KDF is PKCS#12 v1.0 with SHA1 using a 160 bits salt and a random number of iterations in the range 1024–2047.

The integrity of the keystore is checked after successful decryption using the store password. The plaintext consists of the keystore entries followed by their SHA1 checksum. UBER recomputes the hash of the keystore and compares it with the stored digest.

[1] `https://github.com/bcgit/bc-java/blob/master/prov/src/main/java/org/bouncycastle/jce/provider/BrokenPBE.java`

BCFKS

BCFKS is a new FIPS-compliant [131] keystore introduced in the version 1.56 of Bouncy Castle[2] offering similar features to UBER. This keystore provides support for secret keys in addition to asymmetric keys and certificates.

The entire keystore contents is encrypted using AES in CCM mode with a 256 bits key, so to provide protection against introspection. After the encrypted blob, the file contains a block with a HMAC-SHA512 computed over the encrypted contents to ensure the keystore integrity. The store password is used to derive the two keys for encryption and integrity.

All key derivation operations use PBKDF2 with HMAC-SHA512 as pseudorandom function, 512 bits of salt and 1024 iterations. Each key entry is separately encrypted with a different password using the same algorithm for the keystore confidentiality, while this possibility is not offered for certificates.

BCPKCS12

The BCPKCS12 keystore aims to provide a PKCS#12-compatible implementation. It uses the same algorithms and default parameters for key derivation, cryptographic schemes and file structure of the Oracle JDK version detailed in Section 4.4.1. Compared to Oracle, the Bouncy Castle implementation lacks support for symmetric keys and the possibility to protect keys with different passwords, since all the entries and certificates are encrypted under the store password. The BC provider also offers a variant of the PKCS#12 keystore that allows to encrypt certificates using the same PBE of private keys, that is Triple-DES in CBC mode.

4.4.3 Keystores Adoption

We have analysed 300 Java projects supporting keystores that are hosted on Github to estimate the usage of the implementations examined in this work. Applications range from amateur software to well-established libraries by Google, Apache and Eclipse.

We searched for occurrences of known patterns used to instantiate keystores in the code of each project. We have found that JKS is the most widespread keystore with over 70% of the applications supporting it. PKCS12 is used in 32% of the analysed repositories, while JCEKS adoption is close to 10%. The Bouncy Castle keystores UBER and BCPKCS12 are used only in 3% of the projects, while BKS can be found in about 6% of the examined software. Finally, since BCFKS is a recent addition to the Bouncy Castle library, none of the repositories is supporting it.

4.4.4 Summary

In Tables 4.1a and 4.1b we summarize the features and the algorithms (rows) offered by the keystore implementations (columns) analysed in this section. Table 4.1a does not

[2] `https://github.com/bcgit/bc-java/commit/80fd6825`

contain the row "Store Encryption" since none of the JDK keystores provides protection against introspection.

4.5 Attacks

In the previous section, we have shown that the analysed keystores use very diverse key derivation functions and cryptographic mechanisms and, in several cases, they do not adhere to standards or use obsolete and ad-hoc mechanisms. We now discuss how this weakens the overall security of the keystore and enables or facilitates attacks. In particular, we show that keystores using weak or ad-hoc implementations for password-based encryption or integrity checks open the way to password brute-forcing. During the in-depth analysis of keystores, we have also found security flaws that can be exploited in practice to mount denial of service and code execution attacks.

Attacks in this section are organized according to the security properties violated, as defined in Section 4.3.1. For each attack we provide a detailed description discussing the attacker settings and the rules that are not followed by the keystore implementation (*cf.* Section 4.3.2). We conclude with some general security considerations that are not specific to any particular attack.

Table 4.2 provides a high-level overview of the properties which are guaranteed by the analysed keystores with respect to the attacks presented in this section. We consider versions of Oracle JDK and Bouncy Castle before and after disclosing our findings to the developers. Specifically, we refer to JDK 8u144 and 8u181 for Oracle, while version 1.57 of Bouncy Castle is compared against release 1.59. We use the symbol → to point out improvements in newer versions. Details of the changes are listed in Section 4.7. The ✓✓ symbol denotes that a property is satisfied by the keystore under any attacker setting and the implementation adhere to the relevant design rules listed in Section 4.3.2. We use ✓ when no clear attack can be mounted but design rules are not completely satisfied, *e.g.* a legacy cipher like Triple-DES is used. The ✗ symbol indicates that the property is broken under the standard attacker model. When a property is broken only under a specific setting **Sx**, we report it in the table as $\text{✗}_{\mathbf{Sx}}$. If a more powerful attack is enabled by additional settings, we clarify in the footnotes.

As an example, consider the system integrity property (**P3**) in the JCEKS keystore: up to JDK 8u144 included, write capabilities (**S1**) allow to DoS the application loading the keystore; when integrity and key confidentiality passwords are known (**S2** and **S3**), the attacker can also achieve arbitrary code execution on the system (*cf.* note 3 in the table). The rightmost side of the arrow indicates that the code execution attack has been patched in JDK 8u181.

TABLE 4.1: Summary of the keystores.

(A) Oracle JDK 8u144 and below.

		JKS	JCEKS	PKCS12
Provider		Sun	SunJCE	SunJSSE
Support for secret keys		✗	✓	✓*
Keys PBE	KDF	Custom (SHA1)	Custom (MD5)	PKCS12 (SHA1)
	Salt	160b	64b	160b
	Iterations	-	20	1024
	Cipher	Stream cipher	3DES (CBC)	3DES (CBC)
	Key size	-	192b	192b
Certificates PBE	KDF	✗	✗	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Cipher			RC2 (CBC)
	Key size			40b
Store Integrity	KDF	SHA1 with password	SHA1 with password	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Mechanism			HMAC (SHA1)

* since Java 8

(B) Bouncy Castle 1.57 and below.

		BKS	UBER	BCFKS	BCPKCS12
Provider		Bouncy Castle	Bouncy Castle	Bouncy Castle	Bouncy Castle
Support for secret keys		✓	✓	✓	✗
Keys PBE	KDF	PKCS12 (SHA1)	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b	160b	512b	160b
	Iterations	1024–2047	1024–2047	1024	1024
	Cipher	3DES (CBC)	3DES (CBC)	AES (CCM)	3DES (CBC)
	Key size	192b	192b	256b	192b
Certificates PBE	KDF	✗	✗	✗	PKCS12 (SHA1)
	Salt				160b
	Iterations				1024
	Cipher				RC2 / 3DES (CBC)
	Key size				40b / 192b
Store Encryption	KDF	✗	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	✗
	Salt		160b	512b	
	Iterations		1024–2047	1024	
	Cipher		Twofish (CBC)	AES (CCM)	
	Key size		256b	256b	
Store Integrity	KDF	PKCS12 (SHA1)	SHA1 after decrypt	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b		512b	160b
	Iterations	1024–2047		1024	1024
	Mechanism	HMAC (SHA1)		HMAC (SHA512)	HMAC (SHA1)

TABLE 4.2: Properties guaranteed by keystores with respect to attacks, before and after updates listed in Section 4.7.

	JKS	JCEKS	PKCS12	BKS	UBER	BCFKS	BCPKCS12
(**P1**) Entries confidentiality	✗	✗→✓	✓[1]	✓	✓	✓→✓✓	✓[1]
(**P2**) Keystore integrity	✗[2]	✗[2]	✓→✓✓	✓	✓	✓→✓✓	✓→✓✓
(**P3**) System integrity	✓✓	✗_{S1}^{3}→✓✓	✗_{S1}→✓✓	✗_{S1}	✓✓	✓✓	✗_{S1}→✓✓

1 only confidentiality of certificates can be violated
2 under settings **S1** or **S4** it might be possible to use rainbow tables
3 under settings **S2** and **S3** it is possible to achieve code execution on JDK $\leq$ 8u162
✓✓ property is always satisfied
✓ no clear attacks but rules not completely satified
✗ property is broken in the standard attacker model
✗_{Sx} property is broken under a attacker setting **Sx**

4.5.1 Attacks on Entries Confidentiality (P1)

JKS Password Cracking

The custom PBE mechanism described in Section 4.4.1 for the encryption of private keys is extremely weak. The scheme requires only one SHA1 hash and a single XOR operation to decrypt each block of the encrypted entry resulting in a clear violation of rule **R1**. Since there is no mechanism to increase the amount of computation needed to derive the key from the password, also rule **R2** is neglected.

Despite the poor cryptographic scheme, each attempt of a brute-force password recovery attack would require to apply SHA1 several times to derive the whole keystream used to decrypt the private key. As outlined in Figure 4.1, a successful decryption is verified by matching the last block (*CK*) of the protected entry with the hash of the password concatenated with the decrypted key. For instance, a single password attempt to decrypt a 2048 bit RSA private key entry requires over 60 SHA1 operations.

We found that such password recovery attack can be greatly improved by exploiting the partial knowledge over the plaintext of the key. Indeed, the ASN.1 structure of a key entry enables to efficiently test each password with a single SHA1 operation. In JKS, private keys are serialized as DER-encoded ASN.1 objects, along the PKCS#1 standard [127]. For instance, an encoded RSA key is stored as a sequence of bytes starting with byte `0x30` which represent the ASN.1 type `SEQUENCE` and a number of bytes representing the length of the encoded key. Since the size of the encrypted key is the same as the size of the plaintext, these bytes are known to the attacker. On average, given n bytes of the plaintext it is necessary to continue decryption beyond the first block only for one password every 256^n attempts.

The pseudocode of the attack is provided in Algorithm 3, using the same notation introduced in Section 4.4.1. We assume that the algorithm is initialized with the salt, all the blocks of the encrypted key and the checksum. The XOR operation between the known plaintext and the first encrypted block (line 3) is performed only once for all the

Algorithm 3 JKS 1-block Crack

```
1:  procedure JKS_1BlockCrack(Salt, E_{1..n}, CK)
2:      known_plaintext ← 0x30 || length(E)
3:      test_bytes ← known_plaintext ⊕ E_1
4:      for password in passwords do
5:          W_1 ← SHA1(password || Salt)
6:          if W_1 = test_bytes then
7:              K ← Decrypt(Salt, E, password)
8:              checksum ← SHA1(password || K)
9:              if CK = checksum then
10:                 return password
```

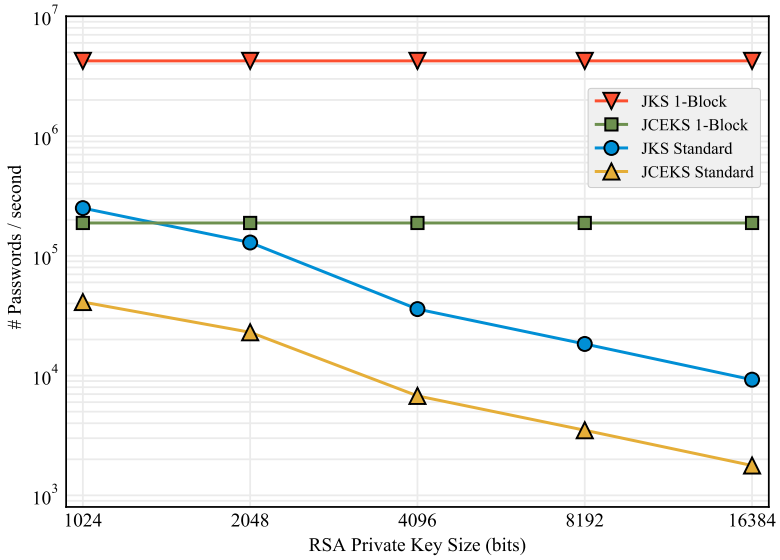


FIGURE 4.2: Performance comparison of password cracking for private RSA keys on JKS and JCEKS using both the standard and the improved 1-block method on a Intel Core i7 6700 CPU.

possible passwords. As a halt condition, the result is then compared against the digest of the salt concatenated to the tested password (lines 5-6). To further verify the correctness of the password, a standard decrypt is performed.

A comparison between the standard cracking attack and our improved version is depicted in Figure 4.2. From the chart it is possible to see that the cost of the single block attack (referred to as 1-block) is independent from the size of the encrypted entry, while the number of operations required to carry out the standard attack is bound to the size of the DER-encoded key. As an example, for a 4096 bit private RSA key, the 1-block approach is two orders of magnitude faster than the standard one.

Based on our findings, this attack has been recently integrated into Hashcat 3.6.0[3] achieving a speed of 8 billion password tries/sec with a single NVIDIA GTX 1080 GPU.

[3]https://hashcat.net/forum/thread-6630.html

JCEKS Password Cracking

The PBE mechanism discussed in Section 4.4.1 uses a custom KDF that performs 20 MD5 iterations to derive the encryption key used in the Triple-DES cipher. This value is three orders of magnitude lower than the iteration count suggested in [70], thus violating both rules **R1** and **R2**. Given that keys are DER-encoded as well, it is possible to speed up a brute-force attack using a technique similar to the one discussed for JKS. Figure 4.2 relates the standard cracking speed to the single block version. Notice that the cost of a password-recovery attack is one order of magnitude higher than JKS in both variants due to the MD5 iterations required by the custom KDF of JCEKS.

PKCS#12 Certificate Key Cracking

Oracle PKCS12 and BCPKCS12 keystores allow for the encryption of certificates. The PBE is based on the KDF defined in the PKCS#12 standard paired with the legacy RC2 cipher in CBC mode with a 40 bit key, resulting in a clear violation of rule **R1**. Due to the reduced key space, the protection offered by the KDF against offline attacks can be voided by directly brute-forcing the cryptographic key. Our serialized tests, performed using only one core of an Intel Core i7 6700 CPU, show that the brute-force performance is 8,300 $passwords/s$ for password testing (consisting of a KDF and decryption run), while the key cracking speed is 1,400,000 $keys/s$. The worst-case scenario that requires the whole 40-bits key space to be exhausted, requires about 9 days of computation on our system. This time can be reduced to about 1 day by using all eight cores of our processor. We estimate that a modern high-end GPU should be able to perform this task in less than one hour. Notice, however, that although finding the key so easily makes the encryption of certificates pointless, an attacker cannot use the key value to reduce the complexity of cracking the integrity password since the random salt used by the KDF makes it infeasible to precompute the mapping from passwords to keys.

4.5.2 Attacks on Keystore Integrity (P2)

JKS/JCEKS Integrity Password Cracking

The store integrity mechanism used by both JKS and JCEKS (*cf.* Section 4.4.1) only relies on the SHA1 hash digest of the integrity password, concatenated with the constant string "`Mighty Aphrodite`" and with the keystore data. In contrast with rule **R1**, this technique based on a single application of SHA1 enables to efficiently perform brute-force attacks against the integrity password. Section 4.6 reports on the computational effort required to attack the integrity mechanism for different sizes of the keystore file.

Additionally, since SHA1 is based on the Merkle-Damgard construction, this custom approach is potentially vulnerable to extension attacks [56]. For instance, it may be possible for an attacker with write access to the keystore (**S1**) to remove the original digest at the end of the file, extend the keystore content with a forged entry and recompute a valid hash without knowing the keystore password. Fortunately, this specific attack

is prevented in JKS and JCEKS since the file format stores the number of entries in the keystore header.

JKS/JCEKS Integrity Digest Precomputation

The aforementioned construction to ensure the integrity of the keystore suffers from an additional problem. Assume the attacker has access to an empty keystore, for example when an old copy of the keystore file is available under a file versioning storage (**S4**). Alternatively, as special case of **S1**, the attacker may be able to read the file, but the interaction with the keystore is mediated by an application that allows to remove entries without disclosing the store password. This file consists only of a fixed header followed by the SHA1 digest computed using the password, the string "`Mighty Aphrodite`" and the header itself. Given that there is no random salting in the digest computation, it would be possible to mount a very efficient attack to recover the integrity password by exploiting precomputed hash chains, as done in rainbow tables [143].

4.5.3 Attacks on System Integrity (P3)

JCEKS Code Execution

A secret key entry is stored in a JCEKS keystore as a Java object having type `SecretKey`. First, the key object is serialized and wrapped into a `SealedObject` instance in an encrypted form; next, this object is serialized again and saved into the keystore.

When the keystore is loaded, all the serialized Java objects stored as secret key entries are evaluated. An attacker with write capabilities (**S1**) may construct a malicious entry containing a Java object that, when deserialized, allows her to execute arbitrary code in the application context. Interestingly, the attack is not prevented by the integrity check since keystore integrity is verified only after parsing all the entries.

The vulnerable code is in the `engineLoad` method of the class `JceKeyStore` implemented by the SunJCE provider.[4] In particular, the deserialization is performed as follows:

```
// read the sealed key
try {
    ois = new ObjectInputStream(dis);
    entry.sealedKey = (SealedObject) ois.readObject();
    ...
```

Notice that the cast does not prevent the attack since it is performed after the object evaluation.

To stress the impact of this vulnerability, we provide three different attack scenarios: *i)* the keystore is accessed by multiple users over a shared storage. An attacker can replace or add a single entry of the keystore embedding the malicious payload, possibly

[4] `http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/5534221c23fc/src/share/classes/com/sun/crypto/provider/JceKeyStore.java`

gaining control of multiple hosts; *ii)* a remote application could allow its users to upload keystores for cryptographic purposes, such as importing certificates or configuring SSL/TLS. A crafted keystore loaded by the attacker may compromise the remote system; *iii)* an attacker may even forge a malicious keystore and massively spread it like a malware using email attachments or instant messaging platforms. Users with a default application associated to the keystore file extension (*e.g.*, keystore inspection utilities such as KSE[5]) have a high probability of being infected just by double clicking on the received keystore. Interestingly, all the malicious keystores generated during our tests did not raise any alert on antivirus tools completing a successful scan by `virustotal.com`.

We checked the presence of the vulnerability from Java 6 onwards. We were able to achieve arbitrary command execution on the host with JDK $\leq$ 7u21 and JDK $\leq$ 8u20 by forging a payload with the tool *ysoserial*.[6] Newer versions up to 8u162 are still affected by the vulnerability, but the JDK classes exploited to achieve code execution have been patched. Since the deserialization occurs within a Java core class, the classpath is restricted to bootstrap and standard library classes. However, by embedding a recursive object graph in a JCEKS entry, an attacker can still hang the deserialization routine consuming CPU indefinitely and thus causing a DoS in the target machine. We were able to mount this attack on any version of the Oracle JDK $\leq$ 8u144.

The implementation choice for storing secret keys in JCEKS is a clear violation of Rule **R3**, since these entities are essentially stored as Java code. The correct approach is to adopt standard formats and encodings, such as the PKCS#8 format used in the PKCS12 keystore.

JCEKS Code Execution After Decryption

When the attacker knows the integrity password and the confidentiality password of a secret key entry (**S2**, **S3**) in addition to **S1**, the previous attack can be further improved to achieve arbitrary command execution up to JDK 8u162 (released on January 2018). This variant of the attack assumes that the application loading the JCEKS keystore makes use of one of the widespread third-party libraries supported by *ysoserial*, such as *Apache Commons Collections* or the *Spring* framework: such libraries have been found [189] to contain vulnerable gadget chains that can be exploited by the malicious payload.

When a `SealedObject` wrapping a secret key is successfully loaded and decrypted, an additional deserialization call is performed over the decrypted content. The `SealedObject` class extends the classpath to allow the deserialization of any class available in the application scope, including third-party libraries. By exploiting this second deserialization step, an attacker may construct more powerful payloads to achieve command execution.

The exploitation scenarios are similar to the ones already discussed in the previous variant of the attack. Additionally, we point out that even an antivirus trained to detect

[5] `http://keystore-explorer.org`
[6] `https://github.com/frohoff/ysoserial`

deserialization signatures would not be able to identify the malicious content since the payload is stored in encrypted form in the keystore.

DoS by Integrity Parameters Abuse

Many keystores rely on a keyed MAC function to ensure the integrity of their contents. The parameters of the KDF used to derive the key from the store password are saved inside the file. Thus, an attacker with write capabilities (**S1**) may tamper with the KDF parameters to affect the key derivation phase that is performed before assessing the integrity of the file. In particular, the attacker may set the iteration count to an unreasonably high value in order to perform a DoS attack on applications loading the keystore.

We found that Oracle PKCS12, BKS and BCPKCS12 implementations are affected by this problem. Starting from valid keystore files, we managed to set the iteration count value to $2^{31} - 1$. Loading such keystores required around 15 minutes at full CPU usage on a modern computer. According to [182] the iteration count should not impact too heavily on the user-perceived performance, thus we argue that this is a violation of Rule **R2**.

4.5.4 Bad Design Practices

During our analysis we found that some of the keystores suffered from bad design decisions and implementation issues that, despite not leading to proper attacks, could cause serious security consequences.

Our review of the Oracle PKCS12 keystore code showed that the KDF parameters are not treated uniformly among MAC, keys and certificates. During a store operation, the Oracle implementation does not preserve the original iteration count and salt size for MAC and certificates that has been found at load time in the input keystore file. Indeed, iteration count and salt size are silently set to the hard-coded values of 1024 and 20 byte, respectively. Since this keystore format is meant to be interoperable, this practice could have security consequences when dealing with keystores generated by third-party tools. For instance, PKCS12-compatible keystores generated by OpenSSL default to 2048 iterations: writing out such keystore with the Oracle JDK results in halving the cost of a password recovery attack.

The Bouncy Castle BCPKCS12 implementation suffers a similar problem: in addition to MAC and certificate parameters, also the iteration count and the salt size used for private keys are reverted to default values when the keystore is saved to disk. Following our report to the Bouncy Castle developers, this behaviour has been addressed by preserving the original parameters whenever possible.

Lastly, the construction of the integrity mechanism for the UBER keystore could cause an information leakage under specific circumstances. After a successful decryption using the store password, UBER recomputes the hash of the keystore and compares it with the stored digest. This MAC-then-encrypt approach is generally considered a bad idea, since it can lead to attacks if, for example, there is a perceptible difference in behaviour (an error message, or execution time) between a decryption that fails because the padding is

invalid, or a decryption that fails because the hash is invalid (a so-called padding oracle attack [187]).

4.5.5 Security Considerations

We now provide general considerations on the security of Java keystores. The first one is about using the same password for different purposes. If the integrity password is also used to ensure the confidentiality of encrypted entries, then the complexity of breaking either the integrity or the confidentiality of stored entries turns out to be the one of attacking the weakest mechanism. For instance, we consider a keystore where cracking the integrity password is more efficient than recovering the password used to protect sensitive entries: as shown in Section 4.6, this is the case of PKCS12 and BCPKCS12 keystores. Under this setting, sensitive keys can be leaked more easily by brute-forcing the integrity password. Although this is considered a bad practice in general [108], all the keystores analysed permit the use of the same password to protect sensitive entries and to verify the integrity of the keystore. This practice is indeed widespread [67] and, as already stated in Section 4.3.3, prompted by *keytool* itself. Furthermore, our analysis found that the BCPKCS12 keystore forcibly encrypts keys and certificates with the store password. For these reasons, we argue that using the same password for integrity and confidentiality is not a direct threat to the security of stored keys when both mechanisms are resistant to offline attacks and a strong password is used. Still the security implications of this practice should be seriously considered.

The second consideration regards how the integrity of a keystore is assessed. Indeed, a poorly designed application may bypass the integrity check on keystores by providing a null or empty password to the Java `load()` function. All the Oracle keystores analysed in the previous section and BouncyCastle BKS are affected by this problem. On the other hand, keystores providing protection to entries inspection, such as UBER and BCFKS, cannot be loaded with an empty password since the decryption step would fail. Lastly, BCPKCS12 throws an exception if an attempt of loading a file with an empty password is made. Clearly, if the integrity check is omitted, an attacker can trivially violate Property **P2** by altering, adding or removing any entry saved in the clear. Conversely, the integrity of encrypted sensitive keys is still provided by the decryption mechanism that checks for the correct padding sequence at the end of the plaintext. Since the entries are typically encoded (*e.g.*, in ASN.1), a failure in the parse routine could also indicate a tampered ciphertext.

We also emphasize that the 1-block cracking optimization introduced in Section 4.5.1 is not limited to JKS and JCEKS. Indeed, by leveraging the structure of saved entries, all the analysed keystores enable to reduce the cost of the decrypt operation to check the correctness of a password. However, excluding JKS and JCEKS, this technique only provides a negligible speed-up given that the KDF is orders of magnitude slower than the decrypt operation.

Finally, we point out that the current design of password-based keystores cannot provide a proper key-revocation mechanism without a trusted third-party component. For

instance, it may be the case that a key has been leaked in the clear and subsequently substituted with a fresh one in newer versions of a keystore file. Under settings **S1** and **S4**, an attacker may replace the current version of a keystore with a previously intercepted valid version, thus restoring the exposed key. The integrity mechanism is indeed not sufficient to distinguish among different versions of a keystore protected with the same store password. For this reason, the store password must be updated to a fresh one every time a rollback of the keystore file is not acceptable by the user, which is typically the case of a keystore containing a revoked key.

4.6 Estimating Brute-Force Speed-Up

We have discussed how weak PBEs and integrity checks in keystores can expose passwords to brute-forcing. In this section we make an empirical comparison of the cracking speed to bruteforce both the confidentiality and integrity mechanisms in the analysed keystores. We also compute the speed-up with respect to BCFKS, as it is the only keystore using a standard and modern KDF, *i.e.*, PBKDF2, which provides the best brute-forcing resistance. Notice, however, that the latest NIST draft on Digital Identity Guidelines [70] sets the minimum KDF iteration count to 10,000 which is one order of magnitude more than what is used in BCFKS (*cf.* Table 4.1b). Thus all the speed-up values should be roughly multiplied by 10 if compared with a baseline implementation using PBKDF2 with 10,000 iterations.

It is out of the scope of this work to investigate brute-forcing strategies. Our tests only aim at comparing, among the different keystores, the actual time to perform the key derivation step and the subsequent cryptographic operations, including the check to assess key correctness. Our study is independent of the actual password guessing strategy adopted by the attacker.

4.6.1 Test Methodology

We developed in C a compatible implementation of the key decryption and the integrity check for each keystore type. Each implementation is limited to the minimum steps required to check the correctness of a test password. This procedure is then executed within a loop to evaluate the cracking speed. Algorithms 4 and 5 show the pseudocode of our implementations. Note that, in both algorithms, we set the password length to 10 bytes because it is an intermediate value between trivial and infeasible. Similarly, since the iteration count in BKS and UBER is chosen randomly in the range 1024–2047, we set it to the intermediate value 1536.

Confidentiality

The code for brute-forcing the confidentiality password (Algorithm 4) is logically divided into three steps: key derivation, decryption and a password correctness check. The last step is included in the loop only to account for its computational cost in the results. Both

Algorithm 4 Confidentiality password cracking benchmark

```
1: procedure BENCHCONFIDENTIALITY(test_duration)
2:     encrypted_entry ← (B_1, ..., B_2000)
3:     passwords ← (pw_1, ..., pw_n)                    ▷ all 10-bytes passwords
4:     salt ← constant
5:     counter ← 0
6:     while ELAPSEDTIME < test_duration do
7:         password ← next(passwords)
8:         key ← KDF_key(password, salt)
9:         iv ← KDF_iv(password, salt)                  ▷ not in JKS, BCFKS
10:        plaintext ← DECRYPTBLOCK(encrypted_entry, key, iv)
11:        VERIFYKEY(plaintext)
12:        counter ← counter + 1
13:    return counter
```

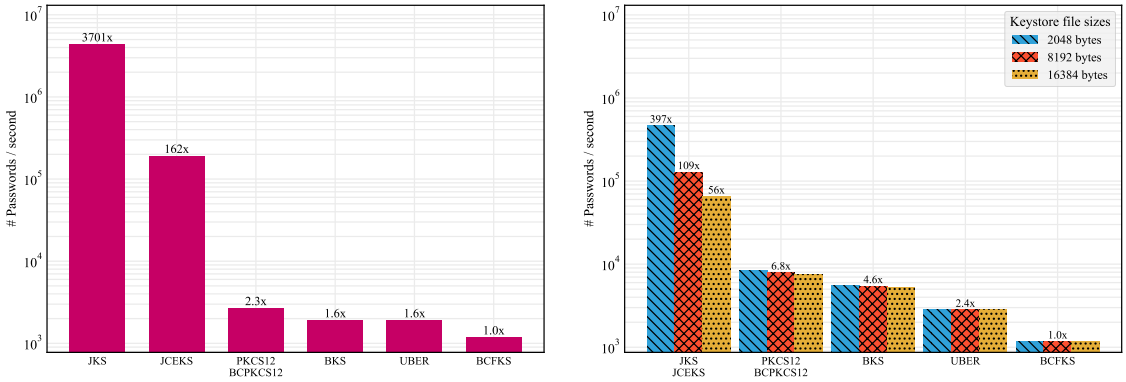
Algorithm 5 Integrity password cracking benchmark

```
1: procedure BENCHINTEGRITY(test_duration)
2:     keystore_content_small ← (B_1, ..., B_2048)
3:     keystore_content_medium ← (B_1, ..., B_8192)
4:     keystore_content_large ← (B_1, ..., B_16384)
5:     passwords ← (pw_1, ..., pw_n)                    ▷ all 10-bytes passwords
6:     salt ← constant
7:     counter_(small,medium,large) ← 0
8:     for all keystore_content, counter do
9:         while ELAPSEDTIME < test_duration do
10:            password ← next(passwords)
11:            key ← KDF_mac(password, salt)            ▷ not in JKS, JCEKS
12:            mac ← MAC(keystore_content, key)
13:            VERIFYMAC(mac)
14:            counter ← counter + 1
15:    return counter_(small,medium,large)
```

PBES1 (PKCS#5) and PKCS#12 password-based encryption schemes, used in all keystores but BCFKS, require to run the KDF twice to derive the decryption key and the IV. On the other hand, in BCFKS the initialization vector is not derived from the password but simply stored with the ciphertext. During our tests we set *encrypted_entry* to a fixed size to resemble an on-disk entry containing a 2048 bits RSA key. However, in Section 4.5.1 we have shown how the partial knowledge of the plaintext structure of a JKS key entry can be leveraged to speed-up brute-forcing. This shortcut can be applied to all the analysed keystores in order to decrypt only the first block of *encrypted_entry*. For this reason, the key size becomes irrelevant while testing for a decryption password.

Integrity

Similarly, the code for cracking the integrity password (Algorithm 5) is organized in three steps: key derivation, a hash/MAC computation and the password correctness check. The key derivation step is run once to derive the MAC key in all keystores, with the exception of JKS and JCEKS where the password is fed directly to the hash function (*cf.*



(A) Speed comparison of password recovery attack for key encryption (confidentiality).

(B) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

FIGURE 4.3: Comparison of keystores password cracking speed. Bar labels indicate the speed-up compared to the strongest BCFKS baseline.

Section 4.4.1). As described later in this section, the speed of KDF plus MAC calculation can be highly influenced by the keystore size, thus we performed our tests using a *keystore_content* of three different sizes: 2048, 8192 and 16384 bytes.

Test configuration

We relied on standard implementations of the cryptographic algorithms to produce comparable results: the OpenSSL library (version 1.0.2g) provides all the needed hash functions, ciphers and KDFs, with the exception of Twofish where we used an implementation from the author of the cipher.[7] All the tests were performed on a desktop computer running Ubuntu 16.04 and equipped with an Intel Core i7 6700 CPU; the source code of our implementations has been compiled with GCC 5.4 using `-O3 -march=native` optimizations. We run each benchmark on a single CPU core because the numeric results can be easily scaled to a highly parallel systems. To collect solid and repeatable results each benchmark has been run for 60 seconds.

4.6.2 Results

The charts in Figure 4.3 show our benchmarks on the cracking speed for confidentiality (Figure 4.3a) and integrity (Figure 4.3b). On the x-axis there are the 7 keystore types: we group together different keystores when the specific mechanism is shared among the implementations, *i.e.*, PKCS12/BCPKCS12 for both confidentiality and integrity and JKS/JCEKS for integrity. On the y-axis we report the number of tested passwords per second doing a serial computation on a single CPU core: note that the scale of this axis is logarithmic. We stress that our results are meant to provide a relative, inter-keystore comparison rather than an absolute performance index. To this end, a label on top of each bar indicates the speed-up relative to the strongest BCFKS baseline. Absolute performance

[7] `https://www.schneier.com/academic/twofish/download.html`

can be greatly improved using both optimized parallel code and more powerful hardware which ranges from dozens of CPU cores or GPUs to programmable devices such as FPGA or custom-designed ASICs [99, 46, 113].

Confidentiality

From the attack described in Section 4.5.1, it follows that cracking the password of an encrypted key contained in JKS, the default keystore up to Java 8, is at least three orders of magnitude faster than in BCFKS. Even without a specific attack, recovering the same password from JCEKS is over one hundred times faster due to its low (20) iteration count. By contrast, the higher value (1024 or 1024–2047) used in PKCS12, BKS and UBER translates into a far better offline resistance as outlined in the chart.

Integrity

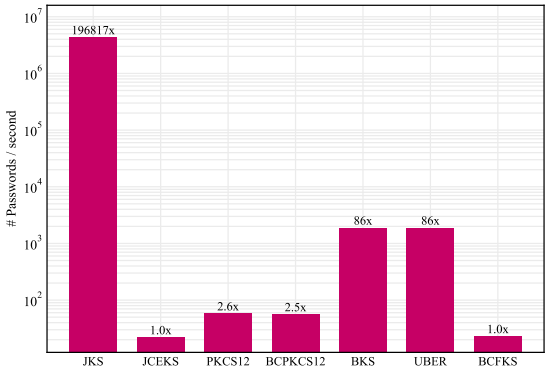
Similar considerations can be done for the resistance of the integrity password. Finding this password in all keystores but JKS is equivalent, or even faster than breaking the confidentiality password. Moreover, the performance of these keystores is influenced by the size of the file due to the particular construction of the MAC function (*cf.* Section 4.4.1). The speed gain (with respect to confidentiality) visible in PKCS12, BKS and UBER is caused by the missing IV derivation step which, basically, halves the number or KDF iterations. Interestingly, in BCFKS there is no difference between the two scores: since the whole keystore file is encrypted, we can reduce the integrity check to a successful decryption, avoiding the computation overhead of the HMAC on the entire file.

4.7 Disclosure and Security Updates

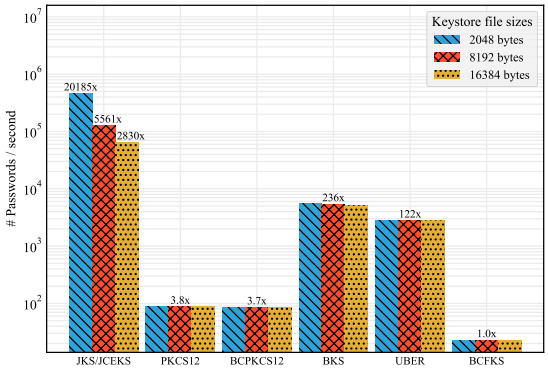
We have timely disclosed our findings to Oracle and Bouncy Castle developers in May 2017. The Oracle Security Team has acknowledged the reported issues with CVE IDs [123, 124, 125] and has fixed all the vulnerabilities between October 2017 and April 2018 [144, 145]. In the following list, we summarize the changes already published by Oracle:

- *keytool* suggests to switch to PKCS12 when JKS or JCEKS keystores are used;
- improved KDF strength of the PBE in JCEKS by raising the iteration count to 200,000. Added a ceiling value of 5 millions to prevent parameter abuse;
- in PKCS12 the iteration count has been increased to 50,000 for confidentiality and 100,000 for integrity. The same upper bound as in JCEKS is introduced;
- fixed the JCEKS deserialization vulnerabilities described in Section 4.5.3 by checking that the objects being deserialized are of the correct type and by imposing a recursion limit to prevent infinite loops.

In version 1.58 of the library, Bouncy Castle developers fixed the parameter abuse vulnerability of BCPKCS12 by adding an optional Java system property that imposes an upper



(A) Speed comparison of password recovery attack for key encryption (confidentiality).



(B) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

FIGURE 4.4: Revised password cracking benchmarks after library updates.

bound for the KDF iteration count. In version 1.59 they have implemented the following changes:

- in BCFKS, the iteration count is raised to 51,200 for both confidentiality and integrity;
- in BCPKCS12, the iteration count is increased to 51,200 and 102,400 for confidentiality and integrity, respectively.

Table 4.2 outlines the improved security guarantees offered by keystore implementations following the fixes released by Oracle and Bouncy Castle. Additionally, in Figure 4.4 we show the updated results of the brute-force resistance benchmarks to reflect the improved KDF parameters. JCEKS and BCFKS now offer the best resistance to offline brute-force attacks of the confidentiality password. However, JCEKS still provides the weakest integrity mechanism. Thus, if the same password is used both for key encryption and for keystore integrity, then the increased protection level can easily be voided by attacking the latter mechanism. On the other hand, both the confidentiality and the integrity mechanisms have been updated in PKCS12. This keystore, which is now the default in Java 9, offers a much higher security level with respect to the previous release.

Conclusion

In this thesis we have provided significant contributions on three relevant topics related to the analysis of security-critical systems, namely techniques for the analysis of firewall configurations, web security and secure storage of cryptographic keys.

In particular, we proposed the first transcompilation pipeline to aid network administrators during the analysis, maintenance and porting of firewall configurations. Our approach is particularly interesting because of it is independent from any specific firewall system. Independence is achieved by defining `IFCL`, an intermediate language for configuring firewalls with a formal semantics. We implemented the first two stages of our pipeline in FWS, a tool that analyses real firewall configurations to produce an abstract specification that emphasizes the meaning of the policy and discards all low-level details. We showed how an administrator can use our tool to simplify maintenance routines by exploiting the features of FWS and we performed various tests on several firewall configurations to assess how our solution scales to real-world scenarios.

Regarding web security, we took a retrospective look at different attacks against web sessions and we surveyed the most popular solutions against them. For each solution, we discussed its security guarantees against different attacker models, its impact on usability and compatibility, and its ease of deployment. We then synthesized five guidelines which can help web security experts in the development of novel solutions that are both more effective and amenable for a large-scale adoption. Additionally, we presented WPSE, the first browser-side security monitor designed to address the security challenges of web protocols, and we showed that the security policies enforceable by WPSE can prevent a large number of real-world attacks. Our work encompasses a thorough review of well-known attacks reported in the literature and an extensive experimental analysis performed in the wild, which exposed several undocumented security vulnerabilities fixable by WPSE in existing OAuth 2.0 implementations. We also discovered a new attack on the Google implementation of SAML 2.0 by formalizing its specification in WPSE. In terms of compatibility, we showed that WPSE works flawlessly on the majority of the existing websites, with the few compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability.

The last contribution provided in this thesis consists in an exhaustive analysis of the Java keystore implementations from the Oracle JDK and the Bouncy Castle library. We have pointed out that several implementations adopt non-standard mechanisms and we have shown how this affects keystores security in terms of speed required to brute-force

the keystore passwords. Additionally, we reported novel attacks that range from breaking the confidentiality of stored keys to arbitrary code execution on remote systems and denial of service. Finally we discuss the improvements on keystore security after the fixes implemented by Oracle and Bouncy Castle developers following our responsible disclosure.

Appendix A

Proofs of Chapter 1

A.1 Correctness of Unfolding

The following property is used in the proof of Lemma 4 below. Intuitively, it says that if given a ruleset R a packet p matches no control flow rule, then the packet matches a rule in the unfolded ruleset $[\![R]\!]_I^{true}$ with the same target.

Property 3. *Given a ruleset R, if $p \models_R (t, i)$ with $t \in \{\texttt{ACCEPT}, \texttt{DROP}, \texttt{CHECK-STATE}(X), \texttt{NAT}(n_1, n_2), \texttt{MARK}\}$ then for all sets I, $p \models_{[\![R]\!]_I^{true}} (t, j)$ with $i \leq j$.*

Proof. We know that $R = [(\phi_1, t_1), \ldots, (\phi_n, t_n)]$ and that $\phi_k(p)$ does not hold for $k < i$. Furthermore, we know that our unfolding algorithm replaces each rule $r_k = (\phi_k, t_k)$ where $t \in \{\texttt{CALL(R')}, \texttt{GOTO(R')}\}$ with a list of rules whose predicates have ϕ_k as a conjunct if R' is not in I; otherwise r_k is replaced with $(\phi_k \wedge \mathit{true}, \texttt{DROP}) = (\phi_k, \texttt{DROP})$. Since $\phi_k(p)$ is false for $k < i$, the new rules do not apply, whereas the rule r_i still applies. Moreover, consider the case when a rule $r_k = (\phi_k, \texttt{RETURN})$ occurs in the ruleset for $k < i$. In the resulting ruleset this rule is canceled and the rules that follow it are rewritten by adding the conjunct $\neg\phi_k$. Since r_k does not apply, we know that $\neg\phi_k(p)$ is *true*, and this does not affect the evaluation of the other rules, so that the rule r_i still matches. □

The following lemma shows the correctness of our unfolding procedure. Intuitively, it tells us that packet p is evaluated in the same way by a ruleset R and by $[\![R]\!]$.

Lemma 4. *Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and given a ruleset R such that $c(q) = R$ for some node q of $\mathcal{C}$, we have that for $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$*

$$\forall p, s. p, s \models_R^{\epsilon} (t, p') \iff p, s \models_{[\![R]\!]}^{\epsilon} (t, p')$$

Proof. We prove the following stronger statement from which the lemma follows as a particular case:

$$\forall p, s, S.\ p, s \models_R^S (t, p') \iff p, s \models_{[\![R]\!]_I^{true}}^S (t, p')$$

with $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$ and $I = \mathit{flatten}(S) \cup \{R\}$, where the function *flatten* is recursively defined as follows, where $X \in \{R, \overline{R}, \overline{R_i}, R_{i+1}\}$:

$$\mathit{flatten}(\epsilon) = \emptyset \qquad \mathit{flatten}(X \cdot S) = \mathit{flatten}(S) \cup \{X\}$$

($\Rightarrow$) We proceed by induction on the derivations of $p, s \models^S_R$ and then by cases on the last rule applied.

- **Rule (1)**. By the premise of the rule, it holds that $p \models_R t$ where $t \in \{\texttt{ACCEPT}, \texttt{DROP}\}$. The thesis follows by applying Property 3. Note that Property 3 holds for all sets I and in particular for $I = \mathit{flatten}(S) \cup \{R\}$.

- **Rules (2) and (4)**. Similar to rule (1).

- **Rule (3)**. By the premise of the rule, we have that $p \models_R (\texttt{CHECK-STATE}(X), i)$, $p \not\models_s$ and $p, s \models^S_{R_{i+1}} (t, p')$. By Property 3, we have that $p \models_{[\![R]\!]^{true}_I} (\texttt{CHECK-STATE}(X), j)$ holds and, by induction hypothesis, that $p, s \models^S_{[\![R_{i+1}]\!]^{true}_I} (t, p')$ for $I = \mathit{flatten}(S) \cup \{R\}$. When recursively unfolding R_{i+1} starting from R, we have a generic predicate ϕ in place of *true*, *i.e.*, $[\![R_{i+1}]\!]^{\phi}_I$. According to the unfolding procedure, ϕ may include the negation of the predicates of rules r_j with $j < i$. Since r_i is the first rule matching p in R we know that $\phi(p)$ evaluates to *true*, hence $p, s \models^S_{[\![R_{i+1}]\!]^{\phi}_I}$ iff $p, s \models^S_{[\![R_{i+1}]\!]^{true}_I}$. Thus, the thesis holds by applying the rule (3).

- **Rule (5)**. By the premise of the rule, it holds $p \models_R (\texttt{GOTO(R')}, i)$, *i.e.*, the rule $r_i = (\phi_i, \texttt{GOTO(R')})$ matches, $R' \notin S$ and $p, s \models^{\overline{R} \cdot S}_{R'} (t, p')$. By induction hypothesis we have that $p, s \models^{\overline{R} \cdot S}_{[\![R']\!]^{true}_{I'}} (t, p')$ with $I' = \mathit{flatten}(\overline{R} \cdot S) \cup \{R'\}$. From the definition of the unrolling procedure we know that for all I not including R' the ruleset $[\![R]\!]^{true}_I$ includes all the rules of $[\![R']\!]^{true}_{I \cup \{R'\}}$ prefixed by the predicate ϕ_i as a conjunct. Since ϕ_i is true for p, it is not change the validity of the rule matching p in $[\![R']\!]^{true}_{I'}$ that continues to match p also in $[\![R]\!]^{true}_I$. Thus, we have that $p \models_{[\![R]\!]^{true}_I} (\texttt{GOTO(R')}, j)$ for some j and for all I not including R'. So we conclude the thesis $p, s \models^S_{[\![R]\!]^{true}_I} (t, p')$ by taking as $I = \mathit{flatten}(S) \cup \{R\}$.

- **Rule (7)**. Similar to rule (5).

- **Rule (6)**. By the premise of the rule, we know that $p \models_R (\texttt{GOTO(R')}, i)$, *i.e.*, there exists $\phi_i(p)$ that holds, and $R' \in S$. Hence, taking $I = \mathit{flatten}(S) \cup \{R\}$ we have that $R' \in I$, and in $[\![R]\!]^{true}_I$ the rule matching r_i is replaced with $(\mathit{true} \wedge \phi_i, \texttt{DROP})$, so $p \models_{[\![R]\!]^{true}_I} (\texttt{DROP}, j)$ for some j and by applying rule (1) we obtain the thesis.

- **Rule (8)**. Similar to rule (6).

- **Rule (9)**. By the premise of the rule we know that $p \models_R (\texttt{RETURN}, i)$, $pop^*(S) = (R', S')$ and $p, s \models^{S'}_{R'} (t, p')$. In the ruleset $[\![R]\!]$ the rule r_i is skipped and all the rules after the r_i are rewritten by adding the negation of the predicate ϕ_i of r_i as a conjunct; in these way none of these new rules match p. Thus, $p \not\models_{[\![R]\!]^{true}_I}$ for any I. The thesis follows by applying the induction hypothesis and then rule (11).

- **Rule (10)**. It is essentially as the proof of rule (9) except that at the end the thesis follows by applying rule (12).

- **Rule (11)**. By the premise of the rule we know $p \not\models_R$, $pop^*(S) = (R', S')$ and $p, s \models^{S'}_{R'} (t, p')$. Trivially it holds that if $p \not\models_R$ then $p \not\models_{[\![R]\!]^{true}_I}$ for all I. Thus, the thesis follows by applying the induction hypothesis and rule (11).
- **Rule (12)**. By the premise of the rule it holds that $p \not\models_R$ and $S = \epsilon \vee pop^*(S) = \epsilon$. Since if $p \not\models_R$ then $p \not\models_{[\![R]\!]^{true}_I}$ for all I, the thesis holds by applying rule (12).
- **Rule (13)**. Immediate by induction hypothesis.

($\Leftarrow$) We proceed by contradiction by assuming that $p, s \models^{S}_{[\![R]\!]^{true}_I} (t, p')$ for $I = \mathit{flatten}(S) \cup \{R\}$, but that $p, s \models^{S}_{R} (t', p'')$ with $(t, p') \neq (t', p'')$. By applying the just proved case "if" of the Lemma we know that $p, s \models^{\epsilon}_{[\![R]\!]^{true}_{I'}} (t', p'')$ for $I' = \mathit{flatten}(S) \cup \{R\}$. Contradiction. □

The following lemma guarantees that the evaluations in the slave transition system for a firewall and its unfolded version are the same.

Lemma 5. *Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $[\![\mathcal{F}]\!]$ its unfolding. Let $(q, s, p) \rightarrow_X (q', s, p')$ be a step of the slave transition system performed by the firewall $X \in \{\mathcal{F}, [\![\mathcal{F}]\!]\}$. Given a node q of $\mathcal{C}$ we have that*

$$(q, s, p) \rightarrow_{\mathcal{F}} (q', s, p') \iff (q, s, p) \rightarrow_{[\![\mathcal{F}]\!]} (q', s, p').$$

Proof. Assume $(q, s, p) \rightarrow_{\mathcal{F}} (q', s, p')$. By definition of the slave transition system we know that $c(q) = R$, $p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p')$ and $\delta(q, p') = q'$. By Definition 7 and by Lemma 4, we have $c'(q) = [\![R]\!]$ and $p, s \models^{\epsilon}_{[\![R]\!]} (\texttt{ACCEPT}, p')$, so we can prove $(q, s, p) \rightarrow_{[\![\mathcal{F}]\!]} (q', s, p')$. The case "only if" follows the same schema. □

Now we can easily prove Theorem 1.

Theorem 1 (Correctness of unfolding)**.** *Let $\mathcal{F} = (\mathcal{C}, \rho, c)$ be a firewall and $[\![\mathcal{F}]\!]$ its unfolding. Let $s \xrightarrow{p,p'}_X s'$ be a step of the master transition system performed by the firewall $X \in \{\mathcal{F}, [\![\mathcal{F}]\!]\}$. Then, it holds*

$$s \xrightarrow{p,p'}_{\mathcal{F}} s' \iff s \xrightarrow{p,p'}_{[\![\mathcal{F}]\!]} s'.$$

Proof. Assume $s \xrightarrow{p,p'}_{\mathcal{F}} s'$. By the premise of the rule, we know that there exists a sequence of steps of the slave transition system such that $(q_i, s, p) \rightarrow^{+}_{\mathcal{F}} (q_f, s, p')$. By repeatedly using Lemma 5 we have that $(q_i, s, p) \rightarrow^{+}_{[\![\mathcal{F}]\!]} (q_f, s, p')$. Thus, the thesis follows by applying the rule of the master transition system. The "only if" case follows the same schema. □

A.2 Correctness of the Logical Characterization

The following property is an immediate consequence of Definition 3.

Property 6. *Given a ruleset R and a rule $r' = (\phi', t')$, if $p \models_R (t, i)$ and $\phi'(p)$ does not hold then $p \models_{r';R} (t, i)$.*

The following lemma guarantees the correctness of the predicate definition in Section 1.5.2.

Lemma 1. *Given a ruleset R we have that*

1. $\forall p, s.\ p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p') \implies P_R(p, p')$; *and*
2. $\forall p, p'.\ P_R(p, p') \implies \exists s.\ p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p')$

Proof. **(1)**. By induction on the depth of the derivation of $p, s \models^{\epsilon}_{R} (\texttt{ACCEPT}, p')$ and by cases on the last rule applied. Since we consider unfolded firewalls, the only relevant rules are (1), (2), (3), (4), (12) and (13).

- **Rule (1)**. By the premise of the rule, we know that $p \models_R (\texttt{ACCEPT}, i)$, *i.e.*, there exists $r_i = (\phi_i, \texttt{ACCEPT})$ and $\phi_i(p)$ holds and $p = p'$. Additionally, we know that the formula P_R has a disjunct $\phi_i(p) \wedge (p = p')$ that holds, proving the thesis.
- **Rule (2)**. By the premise of the rule $p \models_R (\texttt{CHECK-STATE}(X), i)$, $p \vdash_s \alpha$ and $p' = \mathit{establ}(\alpha, X, p)$. The thesis follows because P_R contains a disjunct $\phi(p) \wedge p' \in \mathit{tr}(p, *{:}*, *{:}*, X)$ that holds.
- **Rules (3) and (13)**. Trivial using the induction hypothesis.
- **Rule (4)**. By the rule, we have $p \models_R (\texttt{NAT}(n_1, n_2), i)$ and $p' = \mathit{nat}(p, s, d_s, s_n)$. We know that P_R contains a disjunct $\phi(p) \wedge p' \in \mathit{tr}(p, d_s, s_n, \leftrightarrow)$ that holds, thus proving the thesis.
- **Rule (12)**. This case applies if the default policy of the ruleset is ACCEPT and $p' = p$. We know that the disjunct $dp(R) \wedge p' = p$ holds, proving the thesis.

(2). By induction on the length of the ruleset R and then by cases on its first rule. When R is empty, the formula P_R is $dp(R) \wedge p = \tilde{p}$ and $p \not\models_R$. Thus, the thesis follows for each state s by using the rule (12) with an empty stack. If R is not empty we consider all the possible different cases. Note that apart from the case for DROP our translation procedure creates mutually exclusive disjunctions.

- **Case** $(\phi, \texttt{ACCEPT})$. If the first disjunct holds, the thesis follows by applying the rule (1) in any state s. If the second one holds, the thesis follows by induction hypothesis and by Property 6.
- **Case** $(\phi, \texttt{DROP})$. Just as the second case above.
- **Case** $(\phi, \texttt{NAT}(d_s, s_n))$. If the first disjunct holds we know that $p \models_R (\texttt{NAT}(n_1, n_2), i)$ and that p' is one of the possible translation of p. To prove the thesis it suffices to apply the rule (4), taking a state s such that $p' = \mathit{nat}(p, s, d_n, s_n)$. If the second disjunct holds, the thesis follows by induction hypothesis and by applying the Property 6.
- **Case** $(\phi, \texttt{CHECK-STATE}(X))$. If the first disjunct holds, we have that $\phi(p)$ is true and p' is obtained by rewriting p. Thus, the thesis follows by applying the rule (2) in a state s where $p \vdash_s \alpha$ and $p' = \mathit{establ}(\alpha, X, p)$. If the second disjunct holds, the thesis follows by induction hypothesis and by Property 6.

- **Case** $(\phi, \texttt{MARK}(m))$. By induction hypothesis, similarly to the `ACCEPT` case.

□

The following auxiliary lemma establishes the correspondence between the executions in the slave transition system from a node q and the formula built for the same node q.

Lemma 7. *Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and a node q of $\mathcal{C}$, we have that*

1. $\forall s, p.\ (q, s, p) \rightarrow^* (q_f, s, \tilde{p}) \implies \mathcal{P}_q(p, \tilde{p})$
2. $\forall q, p, \tilde{p}.\ \mathcal{P}_q(p, \tilde{p}) \implies \exists s.(q, s, p) \rightarrow^* (q_f, s, \tilde{p})$

Proof. **(1)**. By induction on the length of the derivation of $(q, s, p) \rightarrow^n (q_f, s, \tilde{p})$. In the case of $n = 0$ the thesis trivially holds. We assume that the statement holds for derivation of length n and we prove the case $n + 1$. Thus, there is a derivation $(q, s, p) \rightarrow (q', s, p') \rightarrow^n (q_f, s, \tilde{p})$. By the premise of the slave transition system, we know that $p, s \models^{\epsilon}_{c(q)} (\texttt{ACCEPT}, p')$ and $\delta(q, p') = q'$. By Lemma 6 (1), we know that $P_{c(q)}(p, p')$ is true and by Definition 5 that $\psi(p')$ holds. By applying the induction hypothesis we have $\mathcal{P}_{q'}(p, \tilde{p})$ and the thesis follows.
(2). Note that since $\mathcal{P}_q(p, \tilde{p})$ holds there exist n packets p_i and a path of length n in the control diagram of the firewall $q_1 \xrightarrow{\psi_1(p_1)} q_2 \xrightarrow{\psi(p_2)} \dots \xrightarrow{\psi(p_n)} q_f$ such that $q_1 = q$, $p_1 = p$, $p_n = \tilde{p}$ and $\wedge_{i=1}^{n} \psi_i(p_i)$ holds. We proceed by induction on the length n of this path. For $n = 0$ the thesis trivially holds. We assume the statement valid for n and consider given a path of length $n + 1$: $q \xrightarrow{\psi(p)} q_1 \xrightarrow{\psi_1(p_1)} q_2 \xrightarrow{\psi(p_2)} \dots \xrightarrow{\psi(p_n)} q_f$ with $p_n = \tilde{p}$. Since $\mathcal{P}_q(p, \tilde{p})$ holds, we know that $P_{c(q)}(p, p')$ holds for some p'. By applying Lemma 1 (2), we know that there exists a state s such that $p, s \models^{\epsilon}_{c(q)} (\texttt{ACCEPT}, p')$. Since in the path we have the arc $q \xrightarrow{\psi(p)} q_1$ we also know that $\delta(q, p') = q_1$. The thesis follows by taking $p' = p_1$ and by induction hypothesis. □

Now we can prove the following theorem, which states the correspondence between the logical formulation and the operational semantics.

Theorem 2 (Correctness of the logical characterization)**.** *Given a firewall $\mathcal{F} = (\mathcal{C}, \rho, c)$ and its corresponding predicate $\mathcal{P}_{\mathcal{F}}$ we have:*

1. $s \xrightarrow{p,p'} s \uplus (p, p') \implies \mathcal{P}_{\mathcal{F}}(p, p')$
2. $\forall p, p'.\ \mathcal{P}_{\mathcal{F}}(p, p') \implies \exists s.s \xrightarrow{p,p'} s \uplus (p, p')$

Proof. **(1)**. The thesis follows by taking the premise of the master transition system and by applying Lemma 7 (1). **(2)**. The thesis follows by applying first the Lemma 7 (2), and then the rule of the master transition system, using the same state s given by Lemma 7 (2). □

A.3 Correctness of Policy Generation

Lemma 2. *Let $\mathcal{F}_C$ be a* compiled firewall *of $\mathcal{F}_S$ and let p be a packet, then*

$$\exists p'.\, \mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \exists p''.\, \mathcal{P}_{\mathcal{F}_C}(p, p'').$$

Proof. ($\Rightarrow$) We show that a packet p accepted by $\mathcal{F}_S$ never matches a `DROP` rule in the rulesets of $\mathcal{F}_C$. Since rulesets have a default `ACCEPT` policy, we consider only R_{fil}, $R_{dnat} \,@\, R_{fil}$ and $R_{snat} \,@\, R_{fil}$ which are the only rulesets containing the rule $(true, \texttt{DROP})$ at the end. We distinguish two cases:

- p is accepted by rule $r = (\phi, \texttt{NAT}(d_n, s_n)) \in R_S$ in $\mathcal{F}_S$. By construction of the rulesets, in $\mathcal{F}_C$ the packet will be tagged with some mark m in the first non-empty ruleset in the path $\pi_{\mathcal{F}_C}(p)$; then:
  - if processed by R_{fil}, it is accepted by the rule $(tag(p) \neq \bullet, \texttt{ACCEPT})$;
  - if processed by $R_{dnat} \,@\, R_{fil}$, it is accepted (and translated) according to the NAT rule $(tag(p) = m, \texttt{NAT}(d_n, \star))$ in R_{dnat};
  - similarly, if processed by $R_{snat} \,@\, R_{fil}$, it is accepted by rule $(tag(p) = m, \texttt{NAT}(\star, s_n))$ in R_{snat}.
- p is accepted by rule $r = (\phi, \texttt{ACCEPT}) \in R_S$ in $\mathcal{F}_S$, hence it is not subject to NAT. Algorithm 2 places the rule r in R_{fil}, therefore the packet is accepted when rulesets R_{fil}, $R_{dnat} \,@\, R_{fil}$ are $R_{snat} \,@\, R_{fil}$ are traversed.

($\Leftarrow$) Since $\mathcal{P}_{\mathcal{F}_C}(p, p'')$, there exist a path $\pi_{\mathcal{F}_C}(p) = \langle q_1, \ldots, q_n \rangle$ and $n+1$ packets $p_1, \ldots, p_{n+1}$ such that $P_{c(q_j)}(p_j, p_{j+1})$ for $j \in [1..n]$ where $p_1 = p$ and $p_{n+1} = p''$. By definition of compiled firewall, there exists $j \in [1..n]$ such that $c(q_j) \in \{R_{fil}, R_{dnat} \,@\, R_{fil}, R_{snat} \,@\, R_{fil}\}$. Assume $c(q_j) = R_{fil}$, we distinguish two cases:

- $tag(p_{j+1}) \neq \bullet$: by Algorithm 2, there exists a `MARK` rule in the first non-empty ruleset of $\pi_{\mathcal{F}_C}(p)$ such that $\phi(p)$ holds. Hence, there exists some rule $r = (\phi, \texttt{NAT}(d_n, s_n)) \in R_S$ that accepts p translated as p'.
- $tag(p_{j+1}) = \bullet$: packet p_{j+1} is accepted by some rule $r = (\phi, \texttt{ACCEPT}) \in R_{fil}$; we also have $p_{j+1} = p$ since the packet is not tagged, thus it is not transformed by any NAT in $\mathcal{F}_C$. Since rule $r \in R_S$ by construction, $\mathcal{F}_S$ accepts p without translation.

The cases $c(q_j) \in \{R_{dnat} \,@\, R_{fil}, R_{snat} \,@\, R_{fil}\}$ follow a similar scheme. □

We define an auxiliary function tt that, given two packets p, p', returns the type of translation required to transform p into p'. Additionally, we define and prove the following property about packet marking that is useful in the proof of Theorem 3.

Property 8. *Let p be a packet accepted by a compiled firewall $\mathcal{F}_C$. We have that:*

1. *if p is accepted without NATs it is never tagged by the firewall;*

2. *if p accepted with NATs, it is tagged exactly once in the first non-empty ruleset of* $\pi_{\mathcal{F}_C}(p)$.

Proof. **(1)** If a packet is not tagged in the first non-empty ruleset of the path $\pi_{\mathcal{F}_C}(p)$, all the conditions ϕ in the `MARK` rules do not apply. Thus, none of the (eventual) `NAT` rules applies and the packet is left unchanged. Therefore, subsequent evaluations of the marking rules still do not apply. **(2)** Straightforward, `MARK` rules include the check $tag(p) = \bullet$ in their conditions. Marking occurs in the first node of $\pi_{\mathcal{F}_C}(p)$ that contains a non-empty ruleset, *i.e.*, a ruleset different from $R_\epsilon(p)$. □

Now we can prove Theorem 3.

Theorem 3. *Let p be a packet accepted by both* $\mathcal{F}_S$ *and* $\mathcal{F}_C$*; let* $\beta = tc(\pi_{\mathcal{F}_C}(p))$*; and let* $p'' \approx t_\beta(p, p')$ *for some* p'*. We have that*

$$\mathcal{P}_{\mathcal{F}_S}(p, p') \Leftrightarrow \mathcal{P}_{\mathcal{F}_C}(p, p'')$$

with $p' = p''$ *when* $\beta = nat$ *or* $p = p'$.

Proof. ($\Rightarrow$) Assume $\mathcal{P}_{\mathcal{F}_S}(p, p')$. Let $\pi_{\mathcal{F}_C}(p) = \langle q_1, \ldots, q_n \rangle$ with $q_1 = q_i$ and $q_n = q_f$ and we consider $n + 1$ intermediate packets $p_1, \ldots, p_{n+1}$ with $p_1 = p$ and $p_{n+1} = p''$. We proceed by cases on the translation $tt(p, p')$.

- If $tt(p, p') = id$, by Property 8 we know that p is never tagged by $\mathcal{F}_C$. Since NAT rules are applied only to tagged packets, all rulesets in $\pi_{\mathcal{F}_C}(p)$ accept the packet without translations, *i.e.*, $\mathcal{P}_{\mathcal{F}_C}(p, p)$ holds. We have $t_\beta(p, p') = p' = p$ for all possible values of β, therefore the thesis hold.

- If $tt(p, p') = nat$, by Property 8, we know that p is tagged in the first non-empty ruleset $c(q_j)$ by rule $(\phi \wedge tag(p) = \bullet, \texttt{MARK}(m))$, *i.e.*, $tag(p_{j+1}) = m$.

  - If $\beta = id$, we have $p_{j+1} = \cdots = p_{n+1} = p''$ since the packet traverses and is accepted by rulesets that do not contain NAT rules. Therefore $p'' \approx t_{id}(p, p')$ and the thesis hold.
  - If $\beta = nat$, let $k, l \in [j + 1..n]$ the smallest indexes such that $dnat \in \gamma(c(q_k))$ and $snat \in \gamma(c(q_l))$. Without loss of generality, we assume $k < l$ (the other case is analogous). We have that $p_{j+1} = \cdots = p_k$. Packet p_k is processed by ruleset $c(q_k)$ that applies the DNAT translation associated with tag m, *i.e.*, $p_{k+1} \approx p[da \mapsto da(p')]$. Then we have $p_{k+1} = \cdots = p_l$. Packet p_l is processed by ruleset $c(q_l)$ that applies the DNAT translation associated with tag m, thus we have $p_{l+1} \approx p'$. Finally we have $p_{l+1} = \cdots = p_{n+1} \approx p'$. Since $t_{nat}(p, p') = p'$, the thesis hold.
  - For $\beta \in \{dnat, snat\}$, the proof is a simplified version of $\beta = nat$.

- Proofs for cases $tt(p, p') = snat$ and $tt(p, p') = dnat$ are similar to the case $tt(p, p') = nat$.

($\Leftarrow$) Assume $\mathcal{P}_{\mathcal{F}_C}(p, p'')$. We distinguish two cases, depending on the fact that p'' is tagged or not.

- If $tag(p'') = \bullet$, we know that p'' has been accepted without NATs, *i.e.*, $p = p''$. By definition of $\mathcal{F}_C$, the path $\pi_{\mathcal{F}_C}(p)$ has a node associated with a ruleset R in $\{R_{fil}, R_{dnat} @ R_{fil}, R_{snat} @ R_{fil}\}$. Since p is accepted by R, it means that p is accepted by one of the filtering rules taken from R_S. Therefore we have $\mathcal{P}_{\mathcal{F}_S}(p, p)$, $p'' \approx t_\beta(p, p)$ for any β and the thesis hold.

- Let $tag(p'') = m$. By Property 8 we know that the packet is tagged only once during its processing and tagging occurs inside the first non-empty ruleset of $\pi_{\mathcal{F}_C}(p)$. By Algorithm 2, we know that there exists a rule $r = (\phi, \texttt{NAT}(d_n, s_n)) \in R_S$ for some d_n, s_n that accepts p as p', *i.e.*, $\mathcal{P}_{\mathcal{F}_S}(p, p')$. Moreover, the same ranges are used in the NAT rules that have translated p into p'' during the traversal of the path $\pi_{\mathcal{F}_C}(p)$. Hence we have $p'' \approx t_\beta(p, p')$ for any β and the thesis follows.

□

Bibliography

[1] Martín Abadi and Roger M. Needham. "Prudent Engineering Practice for Cryptographic Protocols". In: *IEEE Transactions on Software Engineering* 22.1 (1996), pp. 6–15.

[2] Martín Abadi and Bogdan Warinschi. "Password-Based Encryption Analyzed". In: *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming, ICALP 2005*. 2005, pp. 664–676.

[3] Pedro Adão et al. "Mignis: A Semantic Based Tool for Firewall Configuration". In: *Proceedings of the 27th IEEE Computer Security Foundations Symposium CSF*. 2014, pp. 351–365.

[4] Devdatta Akhawe et al. "Towards a Formal Foundation of Web Security". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*. 2010, pp. 290–304.

[5] Carolyn Jane Anderson et al. "NetkAT: Semantic Foundations for Networks". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. 2014, pp. 113–126.

[6] *Android Studio User Guide: Sign Your App*. URL: `https://developer.android.com/studio/publish/app-signing.html`.

[7] *Apache Tomcat 7 Documentation: SSL/TLS Configuration*. 2017. URL: `https://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html`.

[8] Apple Inc. *iOS Security Guide*. Tech. rep. Mar. 2017. URL: `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`.

[9] Alessandro Armando et al. "An Authentication Flaw in Browser-Based Single Sign-On protocols: Impact and Remediations". In: *Computers & Security* 33 (2013), pp. 41–58.

[10] Alessandro Armando et al. "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 2008)*. 2008, pp. 1–10.

[11] Elias Athanasopoulos, Vasilis Pappas, and Evangelos P. Markatos. "Code-Injection Attacks in Browsers Supporting Policies". In: *Proceedings of the 2009 IEEE Web 2.0 Security and Privacy Workshop*. 2009.

[12] Chetan Bansal et al. "Discovering Concrete Attacks on Website Authorization by Formal Analysis". In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[13] Chetan Bansal et al. "Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage". In: *Proceedings of the 2nd International Conference on Principles of Security and Trust, POST 2013*. 2013, pp. 126–146.

[14] Romain Bardou et al. "Efficient Padding Oracle Attacks on Cryptographic Hardware". In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*. 2012, pp. 608–625.

[15] Elaine B. Barker. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. 2016. URL: `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B.pdf`.

[16] Elaine B. Barker and Allen L. Roginsky. *Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths (Rev. 1)*. 2015. URL: `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf`.

[17] Yair Bartal et al. "Firmato: A Novel Firewall Management Toolkit". In: *ACM Transactions on Computer Systems* 22.4 (2004), pp. 381–420.

[18] Adam Barth. *HTTP State Management Mechanism*. 2011. URL: `http://tools.ietf.org/html/rfc6265`.

[19] Adam Barth. *The Web Origin Concept*. 2011. URL: `http://tools.ietf.org/html/rfc6454`.

[20] Adam Barth, Collin Jackson, and John C. Mitchell. "Robust Defenses for Cross-Site Request Forgery". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*. 2008, pp. 75–88.

[21] Daniel Bates, Adam Barth, and Collin Jackson. "Regular Expressions Considered Harmful in Client-side XSS Filters". In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010*. 2010, pp. 91–100.

[22] Lujo Bauer et al. "Run-time Monitoring and Formal Analysis of Information Flows in Chromium". In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015*. 2015.

[23] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. "Multi-instance Security and Its Application to Password-Based Cryptography". In: *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*. 2012, pp. 312–329.

[24] Benjamin Beurdouche et al. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P 2015*. 2015, pp. 535–552.

[25] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*. 2016, pp. 456–467.

[26] Abhishek Bichhawat et al. "Information Flow Control in WebKit's JavaScript Bytecode". In: *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST 2014)*. 2014, pp. 159–178.

[27] Nataliia Bielova. "Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser". In: *Journal of Logic and Algebraic Programming* 82.8 (2013), pp. 243–262.

[28] Prithvi Bisht and V. N. Venkatakrishnan. "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks". In: *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2008*. 2008, pp. 23–43.

[29] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*. 2001, pp. 82–96.

[30] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '98*. 1998, pp. 1–12.

[31] Chiara Bodei et al. "Language-Independent Synthesis of Firewall Policies". In: *Proceedings of the 3rd IEEE European Symposium on Security and Privacy (EuroS&P 2018)*.

[32] Chiara Bodei et al. *FireWall Synthesizer (FWS): Tool and Examples*. URL: `https://github.com/secgroup/fws`.

[33] Chiara Bodei et al. "Transcompiling Firewalls". In: *Proceedings of the 7th International Conference on Principles of Security and Trust (POST 2018)*. 2018, pp. 303–324.

[34] Aaron Bohannon and Benjamin C. Pierce. "Featherweight Firefox: Formalizing the Core of a Web Browser". In: *USENIX Conference on Web Application Development, WebApps 2010*. 2010.

[35] Andrew Bortz, Adam Barth, and Alexei Czeskis. "Origin Cookies: Session Integrity for Web Applications". In: *Web 2.0 Security & Privacy Workshop (W2SP 2011)*. 2011.

[36] Michele Bugliesi, Stefano Calzavara, and Riccardo Focardi. "Formal Methods for Web Security". In: *Journal of Logical and Algebraic Methods in Programming* 87 (2017), pp. 110–126.

[37] Michele Bugliesi et al. "CookiExt: Patching the Browser against Session Hijacking Attacks". In: *Journal of Computer Security* 23.4 (2015), pp. 509–537.

[38] Michele Bugliesi et al. "Provably Sound Browser-Based Enforcement of Web Session Integrity". In: *Proceedings of the IEEE 27th Computer Security Foundations Symposium, CSF 2014*. 2014, pp. 366–380.

[39] William Burr et al. *Electronic Authentication Guideline - Special Publication 800-63-2*. 2012.

[40] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. "Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild". In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security, CCS 2016*. 2016, pp. 1365–1375.

[41] Stefano Calzavara et al. "Quite a Mess in My Cookie Jar!: Leveraging Machine Learning to Protect Web Authentication". In: *Proceedings of the 23rd International World Wide Web Conference, WWW 2014*. 2014, pp. 189–200.

[42] Stefano Calzavara et al. "Surviving the Web: A Journey into Web Session Security". In: *ACM Computing Surveys* 50.1 (2017), 13:1–13:34.

[43] Stefano Calzavara et al. "WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring". In: *Proceedings of the 27th USENIX Security Symposium*. 2018, pp. 1493–1510.

[44] Stefano Calzavara et al. *WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring - Technical report*. URL: `https://sites.google.com/site/wpseproject/`.

[45] Eric Yawei Chen et al. "App isolation: Get the Security of Multiple Browsers with Just One". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. 2011, pp. 227–238.

[46] Richard Clayton and Mike Bond. "Experience Using a Low-Cost FPGA Design to Crack DES Keys". In: *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002*. 2002, pp. 579–592.

[47] Jolyon Clulow. "On the Security of PKCS#11". In: *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*. 2003, pp. 411–425.

[48] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. "Analysis of Secure Key Storage Solutions on Android". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM 2014*. 2014, pp. 11–20.

[49] Cryptosense S.A. *Mighty Aphrodite – Dark Secrets of the Java Keystore*. 2016. URL: `https://cryptosense.com/mighty-aphrodite-dark-secrets-of-the-java-keystore/`.

[50] Frédéric Cuppens et al. "Handling Stateful Firewall Anomalies". In: *Proceedings of the 27th IFIP Conference on Information Security and Privacy Conference (SEC 2012)*. 2012, pp. 174–186.

[51] Alexei Czeskis et al. "Lightweight Server Support for Browser-Based CSRF Protection". In: *Proceedings of the 22nd International World Wide Web Conference, WWW 2013*. 2013, pp. 273–284.

[52] Italo Dacosta et al. "One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens". In: *ACM Transactions on Internet Technology* 12.1 (2012), pp. 1–24.

[53] Dominique Devriese and Frank Piessens. "Noninterference through Secure Multi-execution". In: *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*. 2010, pp. 109–124.

[54] Cornelius Diekmann et al. "Verified iptables Firewall Analysis". In: *Proceedings of the 15th IFIP Networking Conference*. 2016, pp. 252–260.

[55] Michael Dietz et al. "Origin-Bound Certificates: a Fresh Approach to Strong Client Authentication for the Web". In: *Proceedings of the 21th USENIX Security Symposium, USENIX 2012*. 2012, pp. 317–331.

[56] Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. "Salvaging Merkle-Damgard for Practical Applications". In: *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2009*. 2009, pp. 371–388.

[57] ECMA. *ECMAScript Language Specification Standard, 5.1 Edition*. 2011. URL: `http://www.ecma-international.org/ecma-262/5.1/`.

[58] EFF. *HTTPS Everywhere*. Electronic Frontier Foundation, 2011. URL: `https://www.eff.org/https-everywhere`.

[59] Sascha Fahl et al. "Why Eve and Mallory (Also) Love Webmasters: A Study on the Root Causes of SSL Misconfigurations". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2014*. 2014, pp. 507–512.

[60] Daniel Fett, Ralf Küsters, and Guido Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*. 2016, pp. 1204–1215.

[61] Daniel Fett, Ralf Küsters, and Guido Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy, S&P 2014*. 2014, pp. 673–688.

[62] Daniel Fett, Ralf Küsters, and Guido Schmitz. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*. 2015, pp. 1358–1369.

[63] *Firestarter*. URL: `http://www.fs-security.com/`.

[64] *Firewall Builder*. URL: `http://www.fwbuilder.org/`.

[65] Riccardo Focardi et al. "Mind Your Keys? A Security Evaluation of Java Keystores". In: *Proceedings of the 25th Network and Distributed System Security Symposium, NDSS 2018*. 2018.

[66] Open Networking Foundation. *Software-Defined Networking (SDN) Definition*. URL: https://www.opennetworking.org/sdn-resources/sdn-definition.

[67] Paolo Gasti and Kasper Bonne Rasmussen. "On the Security of Password Manager Database Formats". In: *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*. 2012, pp. 770–787.

[68] Google. *A More Secure Sign-in Flow on Chrome*. 2018. URL: https://gsuiteupdates.googleblog.com/2018/04/more-secure-sign-in-chrome.html.

[69] Google. *GSuite Administrator Help, Set up SSO via a third party Identity provider*. 2018. URL: https://support.google.com/a/answer/6262987.

[70] Paul A. Grassi et al. *Digital Identity Guidelines: Authentication and Lifecycle Management*. 2017. URL: https://pages.nist.gov/800-63-3/sp800-63b.html\#sec5.

[71] Willem De Groef et al. "FlowFox: a Web Browser with Flexible and Precise Information Flow Control". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 2012, pp. 748–759.

[72] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. "Browser Model for Security Analysis of Browser-Based Protocols". In: *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*. 2005, pp. 489–508.

[73] Thomas Groß, Birgit Pfitzmann, and Ahmad-Reza Sadeghi. "Proving a WS-Federation Passive Requestor Profile with a Browser Model". In: *Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005*. 2005, pp. 54–64.

[74] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. "Using Static Analysis for Ajax Intrusion Detection". In: *Proceedings of the 18th International Conference on World Wide Web, WWW 2009*. 2009, pp. 561–570.

[75] Matthew Van Gundy and Hao Chen. "Noncespaces: Using Randomization to Defeat Cross-site Scripting Attacks". In: *Computers & Security* 31.4 (2012), pp. 612–628.

[76] Peter Gutmann. "Lessons Learned in Implementing and Deploying Crypto Software". In: *Proceedings of the 11th USENIX Security Symposium*. 2002, pp. 315–325.

[77] Per A. Hallgren, Daniel T. Mauritzson, and Andrei Sabelfeld. "GlassTube: A Lightweight Approach to Web Application Integrity". In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013*. 2013, pp. 71–82.

[78] Dick Hardt. *The OAuth 2.0 Authorization Framework*. 2012. URL: http://tools.ietf.org/html/rfc6749.

[79] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)". In: *Operating Systems Review* 22.4 (1988), pp. 36–38.

[80] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. "Information-flow Security for JavaScript and its APIs". In: *Journal of Computer Security* 24.2 (2016), pp. 181–234.

[81] Daniel Hedin et al. "JSFlow: Tracking Information Flow in JavaScript and its APIs". In: *Proceedings of the 29th Symposium on Applied Computing, SAC 2014*. 2014, pp. 1663–1671.

[82] Mario Heiderich et al. "Scriptless Attacks: Stealing the Pie Without Touching the Sill". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 2012, pp. 760–771.

[83] *High Level Firewall Language*. URL: `http://www.hlfl.org`.

[84] Jeff Hodges, Collin Jackson, and Adam Barth. *HTTP Strict Transport Security (HSTS)*. 2012. URL: `http://tools.ietf.org/html/rfc6797`.

[85] Bob Ippolito. *JSONP*. 2015. URL: `http://json-p.org/`.

[86] Collin Jackson and Adam Barth. "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks". In: *Proceedings of the 17th International Conference on World Wide Web, WWW 2008*. 2008, pp. 525–534.

[87] *Java Cryptography Architecture (JCA) Reference Guide*. 2016. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[88] Karthick Jayaraman et al. *Automated Analysis and Debugging of Network Connectivity Policies*. Tech. rep. Microsoft, 2014.

[89] Karthick Jayaraman et al. "ESCUDO: A Fine-Grained Protection Model for Web Browsers". In: *Proceedings of the 2010 International Conference on Distributed Computing Systems, ICDCS 2010*. 2010, pp. 231–240.

[90] *JDK 7 Security Enhancements*. 2016. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-7.html`.

[91] *JDK 9 Early Access Release Notes*. 2017. URL: `http://jdk.java.net/9/release-notes`.

[92] Alan Jeffrey and Taghrid Samak. "Model Checking Firewall Policy Configurations". In: *Proceedings of the 10th IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2009)*. 2009, pp. 60–67.

[93] Trevor Jim, Nikhil Swamy, and Michael Hicks. "Defeating Script Injection Attacks with Browser-enforced Embedded Policies". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*. 2007, pp. 601–610.

[94] Martin Johns, Ben Stock, and Sebastian Lekies. "A Tale of the Weaknesses of Current Client-Side XSS Filtering". In: *Blackhat USA 2014*. 2014.

[95] Martin Johns and Justus Winter. "RequestRodeo: Client Side Protection against Session Riding". In: *Proceedings of the OWASP Europe 2006 Conference* (2006), pp. 5–17.

[96] Martin Johns et al. "BetterAuth: Web Authentication Revisited". In: *Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012*. 2012, pp. 169–178.

[97] Martin Johns et al. “Reliable Protection Against Session Fixation Attacks”. In: *Proceedings of the 26th ACM Symposium on Applied Computing, SAC 2011*. 2011, pp. 1531–1537.

[98] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. “Preventing Cross Site Request Forgery Attacks”. In: *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks, SecureComm 2006*. 2006, pp. 1–10.

[99] Jens P. Kaps and Christof Paar. “Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine”. In: *Proceedings of the 5th Annual International Workshop in Selected Areas in Cryptography, SAC'98*. 1999, pp. 234–247.

[100] Peyman Kazemian, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking for Networks”. In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*. 2012, pp. 113–126.

[101] Patrick Gage Kelley et al. “Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms”. In: *Proceedings of the 33rd IEEE Symposium on Security and Privacy, S&P 2012*. 2012, pp. 523–537.

[102] John Kelsey et al. “Secure Applications of Low-Entropy Keys”. In: *Proceedings of the 1st International Workshop on Information Security, ISW '97*. 1997, pp. 121–134.

[103] Wilayat Khan et al. “Client Side Web Session Integrity as a Non-interference Property”. In: *Proceedings of the 10th International Conference on Information Systems Security, ICISS 2014*. 2014, pp. 89–108.

[104] Engin Kirda et al. “Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006*. 2006, pp. 330–337.

[105] *KMyFirewall*. URL: `https://sourceforge.net/projects/kmyfirewall/`.

[106] Wanpeng Li and Chris J. Mitchell. “Analysing the Security of Google's Implementation of OpenID Connect”. In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2016)*. 2016, pp. 357–376.

[107] Wanpeng Li and Chris J. Mitchell. “Security Issues in OAuth 2.0 SSO Implementations”. In: *Proceedings of the 17th International Conference in Information Security (ISC 2014)*. 2014, pp. 529–541.

[108] Zhiwei Li et al. “The Emperor's New Password Manager: Security Analysis of Web-based Password Managers”. In: *Proceedings of the 23rd USENIX Security Symposium*. 2014, pp. 465–479.

[109] Jay Ligatti, Lujo Bauer, and David Walker. “Edit Automata: Enforcement Mechanisms for Run-Time Security Policies”. In: *International Journal of Information Security* 4.1-2 (2005), pp. 2–16.

[110] Mike Ter Louw and V. N. Venkatakrishnan. "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers". In: *Proceedings of the 30th IEEE Symposium on Security and Privacy, S&P 2009*. 2009, pp. 331–346.

[111] Mike Ter Louw et al. "SafeScript: JavaScript Transformation for Policy Enforcement". In: *Proceedings of the 18th Nordic Conference on Secure IT Systems, NordSec 2013*. 2013, pp. 67–83.

[112] Kelby Ludwig. *Duo Finds SAML Vulnerabilities Affecting Multiple Implementations*. 2018. URL: `https://duo.com/blog/duo-finds-saml-vulnerabilities-affecting-multiple-implementations`.

[113] Ikuo Magaki et al. "ASIC Clouds: Specializing the Datacenter". In: *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA 2016*. 2016, pp. 178–190.

[114] Christian Mainka et al. "SoK: Single Sign-On Security–An Evaluation of OpenID Connect". In: *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017)*. 2017, pp. 251–266.

[115] Ziqing Mao, Ninghui Li, and Ian Molloy. "Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection". In: *Proceedings of the 13th International Conference on Financial Cryptography and Data Security, FC 2009*. 2009, pp. 238–255.

[116] Giorgio Maone. *The NoScript Firefox Extension*. 2004. URL: `http://noscript.net/`.

[117] Moxie Marlinspike. "New Tricks for Defeating SSL in Practice". In: *BlackHat DC 2009*. 2009.

[118] Robert M. Marmorstein. "Formal Analysis of Firewall Policies". PhD thesis. College of William and Mary, 2008.

[119] Alain J. Mayer, Avishai Wool, and Elisha Ziskind. "Fang: A Firewall Analysis Engine". In: *proc. of the 21st IEEE S&P 2000*. 2000, pp. 177–187.

[120] Leo A. Meyerovich and V. Benjamin Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser". In: *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010*. 2010, pp. 481–496.

[121] MITRE. *CVE-2012-4929: CRIME attack*. 2012. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929`.

[122] MITRE. *CVE-2014-0160: Heartbleed bug*. 2013. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`.

[123] MITRE. *CVE-2017-10345*. 2017. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10345`.

[124] MITRE. *CVE-2017-10356*. 2017. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10356`.

[125] MITRE. *CVE-2018-2794*. 2018. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-2794`.

[126] Kathleen M. Moriarty, Burt Kaliski, and Andreas Rusch. *PKCS#5: Password-Based Cryptography Specification (Version 2.1)*. 2017. URL: https://www.ietf.org/rfc/rfc8018.txt.

[127] Kathleen M. Moriarty et al. *PKCS#1: RSA Cryptography Specifications (Version 2.2)*. 2016. URL: https://www.ietf.org/rfc/rfc8017.txt.

[128] Mozilla. *Same-Origin Policy*. 2015. URL: http://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[129] Mozilla Corporation. *Public Suffix List*. URL: https://publicsuffix.org.

[130] Yacin Nadji, Prateek Saxena, and Dawn Song. "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*. 2009.

[131] National Institute of Standards and Technology. *Annex A: Approved Security Functions for FIPS PUB 140-2, Security Requirements for Cryptographic Modules*. 2018. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf.

[132] Eduardo Vela Nava and David Lindsay. "Our Favorite XSS Filters and How to Attack Them". In: *Blackhat USA 2009*. 2009.

[133] Timothy Nelson et al. "The Margrave Tool for Firewall Analysis". In: *Proceedings of the 24th Large Installation System Administration Conference, LISA 2010*. 2010.

[134] *Netfilter*. URL: https://www.netfilter.org/.

[135] *NeTSPoC: A Network Security Policy Compiler*. URL: http://netspoc.berlios.de.

[136] Nick Nikiforakis, Yves Younan, and Wouter Joosen. "HProxy: Client-Side Detection of SSL Stripping Attacks". In: *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2010*. 2010, pp. 200–218.

[137] Nick Nikiforakis et al. "SessionShield: Lightweight Protection against Session Hijacking". In: *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems, ESSoS 2011*. 2011, pp. 87–100.

[138] Nick Nikiforakis et al. "You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS 2012*. 2012, pp. 736–747.

[139] OASIS. *Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0*. 2015. URL: http://www.oasis-open.org/committees/download.php/56779/sstc-saml-bindings-errata-2.0-wd-06.pdf.

[140] OASIS. *Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0*. 2005. URL: http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf.

[141] OASIS. *Security Assertion Markup Language (SAML) v2.0*. 2005. URL: https://www.oasis-open.org/standards#samlv2.0.

[142] Terri Oda et al. "SOMA: Mutual Approval for Included Content in Web Pages". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008*. 2008, pp. 89–98.

[143] Philippe Oechslin. "Making a Faster Cryptanalytic Time-Memory Trade-Off". In: *Proceedings of the 23rd Annual International Cryptology Conference on Advances in Cryptology, CRYPTO 2003*. 2003, pp. 617–630.

[144] Oracle Corporation. *Critical Patch Updates, Security Alerts and Third Party Bulletin*. 2017. URL: http://www.oracle.com/technetwork/security-advisory/cpuoct2017-3236626.html.

[145] Oracle Corporation. *Critical Patch Updates, Security Alerts and Third Party Bulletin*. 2018. URL: http://www.oracle.com/technetwork/security-advisory/cpuapr2018-3678067.html.

[146] Oracle Corporation. *Java Cryptography Architecture, Standard Algorithm Name Documentation for JDK 8*. 2014. URL: http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html\#KeyStore.

[147] OWASP. *HttpOnly*. 2014. URL: https://www.owasp.org/index.php/HttpOnly.

[148] OWASP. *Top 10 Security Threats*. 2013. URL: https://www.owasp.org/index.php/Top_10_2013-Top_10.

[149] *Packet Filter (PF)*. URL: https://www.openbsd.org/faq/pf/.

[150] Salvador Martínez Perez et al. "A Model-Driven Approach for the Extraction of Network Access-Control Policies". In: *Proceedings of the Workshop on Model-Driven Security (MDsec 2012)*. 2012, pp. 1–6.

[151] Phu H. Phung, David Sands, and Andrey Chudnov. "Lightweight Self-protecting JavaScript". In: *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2009*. 2009, pp. 47–60.

[152] Tadeusz Pietraszek and Chris Vanden Berghe. "Defending Against Injection Attacks Through Context-Sensitive String Evaluation". In: *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, RAID 2005*. 2005, pp. 124–145.

[153] *Pyroman*. 2011. URL: http://pyroman.alioth.debian.org/.

[154] Vineet Rajani et al. "Information Flow Control for Event Handling and the DOM in Web Browsers". In: *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF 2015)*. 2015, pp. 366–379.

[155] Eric Rescorla. *HTTP Over TLS*. 2000. URL: https://tools.ietf.org/html/rfc2818.

[156] Gregor Richards et al. "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications". In: *Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP 2011*. 2011, pp. 52–78.

[157] David Ross. *IE 8 XSS Filter Architecture / Implementation*. 2008. URL: http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[158] RSA Laboratories. *PKCS#11 v2.30: Cryptographic Token Interface Standard*. 2009.

[159] RSA Laboratories. *PKCS#12: Personal Information Exchange Syntax Standard (Version 1.0)*. 1999.

[160] RSA Laboratories. *PKCS#12: Personal Information Exchange Syntax Standard (Version 1.1)*. 2012.

[161] Rusty Russell. *Linux 2.4 Packet Filtering HOWTO*. 2002. URL: http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html.

[162] Mark D. Ryan and Ben Smyth. "Applied Pi Calculus". In: *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press, 2011. Chap. 6.

[163] Philippe De Ryck et al. "Automatic and Precise Client-Side Protection against CSRF Attacks". In: *Proceedings of the 16th European Symposium on Research in Computer Security, ESORICS 2011*. 2011, pp. 100–116.

[164] Philippe De Ryck et al. "CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests". In: *Proceedings of Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010*. 2010, pp. 18–34.

[165] Philippe De Ryck et al. "Serene: Self-Reliant Client-Side Protection against Session Fixation". In: *Proceedings of the 2012 Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012*. 2012, pp. 59–72.

[166] Mohamed Sabt and Jacques Traoré. "Breaking into the KeyStore: A Practical Forgery Attack Against Android KeyStore". In: *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*. 2016, pp. 531–548.

[167] Fred B. Schneider. "Enforceable Security Policies". In: *ACM Transactions on Information and System Security* 3.1 (2000), pp. 30–50.

[168] Bruce Schneier. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.

[169] Jose Selvi. "Bypassing HTTP Strict Transport Security". In: *BlackHat DC 2014*. 2014.

[170] *Shorewall*. URL: http://www.shorewall.net/.

[171] Kapil Singh et al. "Practical End-to-End Web Content Integrity". In: *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012*. 2012, pp. 659–668.

[172] Juraj Somorovsky et al. "On Breaking SAML: Be Whoever You Want to Be". In: *Proceedings of the 21th USENIX Security Symposium*. 2012, pp. 397–412.

[173] *Spring Crypto Utils Documentation: Keystore*. 2017. URL: http://springcryptoutils.com/keystore.html.

[174] *Stanford University Backbone Network Configuration Ruleset*. URL: https://bitbucket.org/peymank/hassel-public/.

[175] Deian Stefan et al. "Protecting Users by Confining JavaScript with COWL". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014*. 2014, pp. 131–146.

[176] Ben Stock and Martin Johns. "Protecting users against XSS-based password manager abuse". In: *Proceedings of the 9th ACM Asia Conference on Information, Computer and Communications Security (AsiaCCS 2014)*. 2014, pp. 183–194.

[177] San-Tsai Sun and Konstantin Beznosov. "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems". In: *Proceedings of the 19th ACM Conference on Computer and Communications Security, (CCS'12)*. 2012, pp. 378–390.

[178] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. "Fortifying web-based applications automatically". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*. 2011, pp. 615–626.

[179] *The IPFW Firewall*. URL: `https://www.freebsd.org/doc/handbook/firewalls-ipfw.html`.

[180] Mary Frances Theofanos and Shari Lawrence Pfleeger. "Guest Editors' Introduction: Shouldn't All Security Be Usable?" In: *IEEE Security & Privacy* 9.2 (2011), pp. 12–17.

[181] *Traversing of Tables and Chains*. URL: `http://www.iptables.info/en/structure-of-iptables.html`.

[182] Meltem Sonmez Turan et al. *Recommendation for Password-Based Key Derivation. Part 1: Storage Applications*. 2010. URL: `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`.

[183] *Uncomplicated Firewall*. URL: `https://help.ubuntu.com/community/UFW`.

[184] *Update to Current Use and Deprecation of TDEA*. 2017. URL: `https://beta.csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA`.

[185] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. "Measuring Login Webpage Security". In: *Proceedings of 32nd ACM Symposium on Applied Computing (SAC 2017)*. 2017, pp. 1753–1760.

[186] Steven Van Acker et al. "WebJail: Least-privilege Integration of Third-party Components in Web Mashups". In: *Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011*. 2011, pp. 307–316.

[187] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Proceedings of the 21st International Conference on the Theory and Applications of Cryptographic Techniques Advances in Cryptology, EUROCRYPT 2002*. 2002, pp. 534–546.

[188] Philipp Vogt et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *Proceedings of the 14th Network and Distributed System Security Symposium, NDSS 2007*. 2007.

[189] *Vulnerability Note VU#576313*. 2015. URL: https://www.kb.cert.org/vuls/id/576313.

[190] W3C. *Cascading Style Sheets*. 2014. URL: http://www.w3.org/Style/CSS/.

[191] W3C. *Content Security Policy*. 2012. URL: http://www.w3.org/TR/CSP/.

[192] W3C. *Content Security Policy Level 2*. 2015. URL: http://www.w3.org/TR/CSP2/.

[193] W3C. *Cross-Origin Resource Sharing*. 2014. URL: http://www.w3.org/TR/cors.

[194] W3C. *Document Object Model (DOM) Level 1 Specification*. 1998. URL: http://www.w3.org/TR/REC-DOM-Level-1.

[195] W3C. *Document Object Model (DOM) Level 2 Core Specification*. 2000. URL: http://www.w3.org/TR/DOM-Level-2-Core.

[196] W3C. *Document Object Model (DOM) Level 3 Core Specification*. 2004. URL: http://www.w3.org/TR/DOM-Level-3-Core.

[197] W3C. *HTML5: A Vocabulary and Associated APIs for HTML and XHTML*. 2014. URL: http://www.w3.org/TR/html5/.

[198] W3C. *Mixed Content*. 2015. URL: http://www.w3.org/TR/2015/CR-mixed-content-20151008/.

[199] 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 33rd IEEE Symposium on Security and Privacy (S&P 2012)*. 2012, pp. 365–379.

[200] Rui Wang et al. "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization". In: *Proceedings of the 22th USENIX Security Symposium*. 2013, pp. 399–314.

[201] *WebLogic Integration 7.0: Configuring the Keystore*. URL: http://docs.oracle.com/cd/E13214_01/wli/docs70/b2bsecur/keystore.htm.

[202] Joel Weinberger, Adam Barth, and Dawn Song. "Towards Client-side HTML Security Policies". In: *6th USENIX Workshop on Hot Topics in Security, HotSec 2011*. 2011.

[203] Matt Weir et al. "Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*. 2010, pp. 162–175.

[204] Michael Weissbacher, Tobias Lauinger, and William K. Robertson. "Why Is CSP Failing? Trends and Challenges in CSP Adoption". In: *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2014*. 2014, pp. 212–233.

[205] Mike West. *Cookie Prefixes*. URL: {https://tools.ietf.org/html/draft-west-cookie-prefixes-05}.

[206] Wei Xu, Sandeep Bhatkar, and R. Sekar. "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks". In: *Proceedings of the 15th USENIX Security Symposium, USENIX 2006*. 2006, pp. 121–136.

[207] Ronghai Yang et al. "Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations". In: *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (AsiaCCS 2016)*. 2016, pp. 651–662.

[208] Frances F. Yao and Yiqun Lisa Yin. "Design and Analysis of Password-Based Key Derivation Functions". In: *IEEE Transactions on Information Theory* 51.9 (2005), pp. 3292–3297.

[209] Dachuan Yu et al. "JavaScript Instrumentation for Browser Security". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*. 2007, pp. 237–249.

[210] Lihua Yuan et al. "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis". In: *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P 2006)*. 2006, pp. 199–213.

[211] Michal Zalewski. *Postcards From the Post-XSS World*. `http://lcamtuf.coredump.cx/postxss/`. 2011.

[212] Bin Zhang et al. "Specifications of a High-Level Conflict-Free Firewall Policy Language for Multi-Domain Networks". In: *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*. 2007, pp. 185–194.

[213] Yinqian Zhang, Fabian Monrose, and Michael K. Reiter. "The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*. 2010.

[214] Xiaofeng Zheng et al. "Cookies Lack Integrity: Real-World Implications". In: *Proceedings of the 24th USENIX Security Symposium, USENIX 2015*. 2015, pp. 707–721.

[215] Yuchen Zhou and David Evans. "SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities". In: *Proceedings of the 23rd USENIX Security Symposium*. 2014, pp. 495–510.

[216] Yuchen Zhou and David Evans. "Why Aren't HTTP-only Cookies More Widely Deployed?" In: *Web 2.0 Security and Privacy Workshop, W2SP 2010*. 2010.