Proceedings of the 5th Workshop on Hot Issues in Security Principles and Trust

Hotspot 2017

April 23rd 2017, Uppsala, Sweden

Preface

This volume contains the papers presented at the 5th Workshop on Hot Issues in Security Principles and Trust (HotSpot 2017) held on April 23rd, 2017 in Uppsala, Sweden.

The program includes one invited talk and eight accepted regular papers. Each submission was reviewed by three program committee members.

The workshop is intended to be a less formal counterpart to the Principles of Security and Trust (POST) conference at ETAPS with an emphasis on ”hot topics”, both of security and of its theoretical foundations and analysis.

I would like to thank Andrei Sabelfeld for accepting to give an invited talk at this work, the authors for their contributions and for making this workshop possible, and the program committee members for their excellent work in reviewing the submitted papers and for their help in putting together an exciting program. I would also like to thank the members of my group, in particular Daniel Rausch for maintaining the workshop's web site.

March 10, 2017 Ralf Küsters
Stuttgart

Program Committee

Aslan Askarov	Aarhus University
David Basin	ETH Zurich
Veronique Cortier	CNRS, Loria
Cas Cremers	University of Oxford
Riccardo Focardi	Università Ca' Foscari, Venezia
Joshua Guttman	Worcester Polytechnic Institute
Boris Köpf	IMDEA Software Institute
Ralf Küsters	University of Stuttgart (PC chair)
Ninghui Li	Purdue University
Frank Piessens	Katholieke Universiteit Leuven
Tamara Rezk	INRIA
Mark Ryan	University of Birmingham
P. Y. A. Ryan	University of Luxembourg
Geoffrey Smith	Florida International University
Nikhil Swamy	Microsoft Research

Table of Contents

Taint Tracking without Tracking Taints

Andrei Sabelfeld

Chalmers University of Technology, Gothenburg, Sweden

Taint tracking has been successfully deployed in a range of security applications to track data dependencies in hardware and machine-, binary-, and high-level code. Precision of taint tracking is key for its success in practice: being a vulnerability analysis, false positives must be low for the analysis to be practical. This talk presents an approach to taint tracking, which, remarkably, does not involve tracking taints throughout computation. Instead, we include shadow memories in the execution context, so that a single run of a program has the effect of computing on both tainted and untainted data. On the theoretical side, we present a general framework and establish its soundness with respect to explicit secrecy, a policy for preventing insecure data leaks, and its precision showing that runs of secure programs are never modified. We show that the technique can be used for attack detection with no false positives. On the practical side, we present DroidFace, leveraging the approach by a source-to-source transform, and benchmark its precision and performance with respect to state-of-the-art static and dynamic taint trackers for Android apps. The results indicate that the performance penalty is tolerable, while achieving no false positives/negatives on the standard benchmarks.

The talk draws on the work reported in the following two publications: *Explicit Secrecy: A Policy for Taint Tracking*, published with Daniel Schoepe, Musard Balliu, and Benjamin C. Pierce in the *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, and *Let's Face It: Faceted Values for Taint Tracking*, published with Daniel Schoepe, Musard Balliu, and Frank Piessens in the *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2016.

Combining Graph-Based and Deduction-Based Information-Flow Analysis

Bernhard Beckert, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
`{beckert,simon.bischof,herda,kirsten}@kit.edu`,
`marko@kleinebuening.de`

Abstract. Information flow control (IFC) is a category of techniques for ensuring system security by enforcing information flow properties such as non-interference. Established IFC techniques range from fully automatic approaches with much over-approximation to approaches with high precision but potentially laborious user interaction. A noteworthy approach mitigating the weaknesses of both automatic and interactive IFC techniques is the hybrid approach, developed by Küsters et al., which – however – is based on program modifications and still requires a significant amount of user interaction.
In this paper, we present a combined approach that works without any program modifications. It minimizes potential user interactions by applying a dependency-graph-based information-flow analysis first. Based on over-approximations, this step potentially generates false positives. Precise non-interference proofs are achieved by applying a deductive theorem prover with a specialized information-flow calculus for checking that no path from a secret input to a public output exists. Both tools are fully integrated into a combined approach, which is evaluated on a case study, demonstrating the feasibility of automatic and precise non-interference proofs for complex programs.

1 Introduction

When sensitive information leaks to unauthorized parties, this is often a result from bugs and errors introduced in the system during software development. In order to prevent such a leakage, we analyze systems for the absence of illegal information flows, also defined as confidentiality [20]. An established approach for proving confidentiality for a system is to prove the *non-interference* property. Non-interference holds if there is no illegal information flow from a secret (high) input to a public (low) output of the system.

There are a variety of different techniques for checking non-interference. Within this work, we distinguish these techniques by their required user interaction and the precision of their checks. These are (a) fully automatic approaches based on type checking [23] or dependency graphs [7], which require no user interaction. Due to decidability problems, this automation comes at the cost of over-approximation and therefore can generate *false positives*. A false positive is a situation where the approach indicates an illegal information flow, even though there is none. Tools implementing such techniques are, e.g., JIF [18] and JOANA [14]. There are also (b) interactive techniques based on theorem provers that do not run automatically, but achieve more precise non-interference checks [3]. Interactive approaches do not generate false positives, but require a high degree of time- and user-interaction.

Küsters et al. developed a hybrid approach that combines an automatic dependency-graph analysis with a theorem prover to minimize the user-effort of non-interference proofs [16]. The hybrid approach attempts to show non-interference with the dependency-graph analysis tool and –if not successful– the user must extend the program such that the affected low output is overwritten independently of the high inputs. Hence, the proof task is divided into two parts, one equivalence proof for the theorem prover, and one non-interference check with the dependency-graph analysis. In this paper we provide an alternative way to combine both tools, thereby not requiring the user to modify the original program in any way, and thus automating large parts of the combination that requires user interaction.

Our contribution is a new combined approach that improves state-of-the-art approaches regarding the automation while maintaining the same level of precision. The combined approach attempts to show the non-interference property of a program using the dependency-graph tool. If the attempt fails, the reported leaks are disproved with the theorem prover. The communication between the two approaches is fully automated: the reported leaks are analyzed one by one and the information flow proof obligations necessary to disprove the leaks are generated and given to the theorem prover. The user can simply provide the program (with high sources and low sinks annotated) to our tool-chain and attempt to prove non-interference. The user does not need to analyze the output of the dependency-graph analysis tool and manually extend the program. The proof obligations for the theorem prover consist of code with generated specification that need to be proven in order to show that the reported leaks are false positives. However, the user might need to provide additional auxiliary specification (e.g., loop invariants) in order for the proof attempt to succeed, this is generally an undecidable problem.

We implemented this new combined approach for the Java programming language focusing on sequential and terminating programs. For the implementation, we chose JOANA [14] as the dependency-graph analysis tool and KeY [1] as the theorem prover, both based on state-of-the-art approaches. The approach is evaluated based on examples and a small case study for which JOANA by itself returns false positives. For these examples, a direct non-interference proof with KeY would require a high degree of user interaction in the form of specifications and proof-interactions. Based on these examples, we show that our approach can prove non-interference automatically but also indicate limits in automation.

We give definitions and background information on logic- and graph-based information-flow analysis underlying the IFC tools JOANA and KeY in Section 2.

In Section 3, we present our combined approach and its guaranteed properties. The implementation including the specification generation and heuristics for an efficient selection of proof obligations is described in Section 4, and evaluated on a number of case studies in Section 5. In Section 6, we discuss related work, and finally conclude in Section 7.

2 Non-Interference

In order to prevent sensitive information from leaking to unauthorized parties, we analyze programs for the absence of illegal information flow. If such an analysis succeeds, we have shown that this program maintains confidentiality of the specified sensitive information with respect to the specified unauthorized parties. In general, the situation can be more complex, and not only two, but a multitude of different sensitivity levels may exist as, e.g., already expressed in the Bell-LaPadula security model [4] establishing access control mechanisms, and extended by the lattice-model thereby establishing a formal notion of information flow [8]. This emerged to more general techniques, denoted by information flow control (IFC), which check that no secret input channel of a given program may influence what is passed to a public output channel [10].

In its simplest form, i.e., there is no information flow, this is known as *non-interference*. With this generalized notion, it suffices to regard only two security or confidentiality levels, in the following referred to as *high* confidentiality and *low* confidentiality, as we abstract from the leakage itself, and instead analyze the program's potential for information leakage from a specified (information) *source* to a specified (information) *sink*. A sink specifies the (program) location, where an unauthorized party may be able to *observe* the potentially secret information, i.e., we call this a publicly observable location. We hence want to check that any two different executions of a program P with different secret inputs (i.e., coming from sources specified as high) i_h, i'_h and the same public input (i.e., coming from sources specified as low) i_ℓ must be indistinguishable in their publicly-observable output (i.e., sinks specified as low sinks). Note that this is stronger than dynamically searching for illegal flows during run-time, as when proven to be non-interferent, no illegal information flow is possible for *any* execution of the program. A given security lattice can hence also be specified as pairs of sources and sinks.

Within this work, we examine and make use of two different language-based types of techniques for IFC [21], namely dependency-graph-based techniques transforming the program into a graph and hence performing specialized graph traversal algorithms [12], as well as logic-based techniques based on a deductive theorem-prover approach symbolically executing the program twice and hence performing a logic-based calculus on the composition of both executions [6]. One main difference is that the dependency-graph analysis is done on the whole system, and the self-composition is done modularly for each involved method, allowing for reasoning about the whole system based on specialized method contracts. Note that we focus on techniques operating directly on either the program's source or byte code without the need for any manual program modifications. Language-based approaches, in our sense, refer to IFC techniques considering potential attackers being able to evaluate expressions, but not able to observe changes in the memory directly. Furthermore, we only consider deterministic sequential programs and do not regard concurrent flows.

2.1 SDG-based Approaches

JOANA is a tool for checking the non-interference property for a given program. It builds a system dependency graph (SDG) from the program code. A formal definition of an SDG is given in [9]. The nodes represent statements and the edges represent dependencies between those statements. JOANA is able to detect direct dependencies, which are also called data dependencies [11], and indirect or control dependencies [11]. Furthermore, there are special nodes, e.g., for method calls, field accesses and exceptions.

For a method we have special formal-in nodes and formal-out nodes. Formal-in nodes represent all direct inputs that influence the method execution. These are the input parameters, used fields, other classes that are called during execution and the class in which the method is executed. The formal-out nodes represent the influence of the method. In most cases the formal-out node represents the method's return value. Other possibilities are that the method influences global variables, fields in other classes or terminates with an exception.

```
int f(int x, int y) { return x; }

void caller() { ...
  f(a,b); ...
}
```

Listing 1. Method call

For function `f` in Listing 1, we would have two formal-in nodes for `x` and `y` and one formal-out node for the return value of `f`. At each method call site, we have actual-in nodes representing the arguments and actual-out nodes representing the returned values. For a given method site, each actual-in node corresponds to a formal-in node of the callee and vice versa. The same holds for actual-out and formal-out nodes. For the call in Listing 1, there are actual-in nodes for `a` and `b`, corresponding to the formal-in nodes of `f` for `x` and `y`, respectively. We also have one actual-out node representing the return value of `f`, which corresponds to the single formal-out node of `f`. For every method call we also have so called *summary edges* [9] in the SDG from any actual-in node to any actual-out node of the method whenever the tool finds a flow between the corresponding formal-in to the formal-out node of the called method. In Listing 1, we have a flow in `f` from `x` to the result, so a summary edge is inserted at the call site, namely from the actual-in node representing `a` to the single actual-out node. For a complex method there can be a huge number of actual-in and actual-out nodes and therefore an even greater number of summary edges. For our combined approach, we focus on summary edges that belong to a *chop* between high and low and it thus is sufficient to regard only a smaller subset of these edges. A chop from a node s to a node t consists of all nodes on paths from s to t in the SDG. It is commonly computed by intersecting the backward slice for t with the forward slice for s. An example of an SDG generated from a program is given in Giffhorn's thesis on page 18 [9].

Through graph analysis, namely through *slicing* and *chopping* on a syntactic level, [11] JOANA is able to detect an information flow. As with KeY, there are some specifications required. But in comparison to KeY, these are rather lightweight. The user must annotate which variables contain secure (high) or public (low) information. After these annotations have been made, JOANA can run the information flow analysis automatically. If the analysis returns that there is no illegal information flow, JOANA guarantees that the program is secure.

Before we give specific property definitions, we introduce the relation *low-equivalent* ($\sim_L$) for the term *state*. We base our definitions on Wasserab's thesis [24]. A state s is a program state, consisting of variable values and storage locations. We assume that the input of a program is included in the initial state and the output of a program is included in the final state. Two states s, s' are low-equivalent if all low variables have the same value.

We only regard sequential programs here. Thus, we want to prove a property called *sequential non-interference* or *classical non-interference* as shown in Definition 1 [24]. If for a sequential program, JOANA returns that there is no illegal information flow, sequential non-interference holds for that program [24]. Note that this definition is equivalent to Definition 4 and hence also guaranteed by non-interference proofs done with KeY.

Definition 1 (Sequential non-interference (SNI)). *Let P be a program. Let s, s' be initial program states, let $[\![P]\!](s), [\![P]\!](s')$ be the final states after executing P in state s resp. s'. Non-interference holds iff*

$$s \sim_L s' \Rightarrow [\![P]\!](s) \sim_L [\![P]\!](s')\,.$$

JOANA guarantees that for a program P it finds secure, if two initial states are low-equivalent then the final states, after executing P from each of the two initial states independently, are also low-equivalent. In case SNI is violated, JOANA generates at least one violation, and can calculate the respective violation chops as well.

2.2 Logic-based Approaches

When attempting to prove non-interference with respect to existing software programs, precision can only be attained by taking functional properties into account. For example, a program such as "`low = high * 0;`" can only be proven to be secure with knowledge about the functionality of `*`. Similarly, for proving non-interference of the program "`if (high) low = f1(); else low = f2();`", we need to verify that `f1` and `f2` compute the same value.

We start with the standard dynamic logic definition from [6], which defines non-interference as a problem of value independence (Definition 2). Dynamic program logics allows to reason about the program `P` as well as program variables `h` of high confidentiality, and `l` of low confidentiality. The predicate $\doteq$ is to be evaluated in the post-state of `P`.

Definition 2 (Non-Interference as value independence). *When starting* `P` *with arbitrary values* `l`*, then the value* r *of* `l` *– after executing* `P` *– is independent of the choice of* `h`*.*

$$\forall \mathtt{l}\ \exists r\ \forall \mathtt{h}\ \langle P \rangle\ r \doteq \mathtt{l}$$

Non-interference verification using self-composition. Amtoft et al. introduced an approach based on a Hoare-style logic, which formalizes non-interference as an "indistinguishability" relation on program states [2]. As such, the foremost *functional* verification task now becomes *relational* by comparing two runs of the same program, performed by a technique called *self-composition* as proposed, e.g., in [3,6]. Furthermore, we can abstract from a concrete location and instead talk about location sets. Based on the notion of *low-equivalence* as in Definition 3, we obtain the notion given in Definition 4, where low-equivalence refers to identity on all low variables [6,22]. Note that by self-composing the program to two instances, we got rid of the existential quantifier, thereby enabling automatic verification techniques as we avoid the difficult quantifier instantiation.

Definition 3 (Low-equivalence). *Two states* s, s' *are low-equivalent iff they assign the same values to low variables (with* L *denoting the set of all low variables in state* s*).*

$$s \simeq_L s' \quad \Leftrightarrow \quad \forall\, v \in L\ (v^s = v^{s'})$$

Definition 4 (Non-Interference as self-composition). *Let* P *be a program and* L_1, L_2 *two sets of low variables. Then starting two instances of* P *in two arbitrary low-equivalent states (on arbitrary high values however) results in two final states that are also low-equivalent.*

$$s_1 \simeq_{L_1} s_1' \Rightarrow [P]s_2 \simeq_{L_2} [P]s_2'$$

These findings were extended by a fully compositional information-flow calculus for Java based on a deductive theorem prover for functional program verification [1,22]. It deals with object-oriented software by allowing for two different semantics, distinguishing on whether object creation is low-observable or not. For the new semantics, we assume that references are opaque, in particular object comparison can only be done via the operator `==`. Furthermore, we assume isomorphisms π_i on objects such that π_1 and π_2 are compatible, i.e., for an object o, $\pi_1(o) = \pi_2(o)$ holds if o is observable in both states s_1 and s_2. Then, low-equivalence can be generalized by Definition 5.

Definition 5 (Low-equivalence with isomorphism). *Two states s, s' are low-equivalent iff they assign the same values to low variables (with L denoting the set of all low variables in state s).*

$$s \simeq_L^\pi s' \quad \Leftrightarrow \quad \forall\, v \in L\; (\pi(v^s) = v^{s'})$$

The calculus and means for specification are implemented in the KeY system [1]. KeY is a deductive theorem prover for Java programs based on JavaDL, a first-order dynamic logic for Java, which allows to reason directly about Java programs on a language-level with an explicit heap variable and changes to the program state translated into so-called *updates* operating on the heap. Thereby, the program can be symbolically executed directly in the logic.

In JavaDL, we can express non-interference based on Definition 4 using heap variables within update operations as given in Definition 6. The updates as a means to change program states are denoted by curly braces.

Definition 6 (Non-Interference as self-composition in JavaDL).

$$\begin{aligned}\forall in_l\; \forall in_h^1\; \forall in_h^2\; \forall out_l^1\; \forall out_l^2\; \{low := in_l\}(&\\ \{high := in_h^1\}[P]\, out_l^1 = low&\\ \wedge\, \{high := in_h^2\}[P]\, out_l^2 = low&\\ \rightarrow\; out_l^1 = out_l^2&\\ )&\end{aligned}$$

The postcondition can be weakened by only proving the variables to be equal up to isomorphism.

The KeY system proves non-interference or other program properties modularly on the program code on Java method level, specified using method contracts as well as auxiliary specifications such as loop invariants inside the method by the modelling language JML*. The formulation of these specifications always depends on the outcome to be proven and describes, e.g., the non-interference property of the program. After the specification is complete, KeY transforms it into equivalent formulas in Dynamic Logic and performs a proof using the sequent calculus. In general, the problem is undecidable and verification sometimes requires some user-interaction. KeY is capable of verifying non-interference for Java programs and covers a wide range of Java features. Proofs are constructed in a precise manner based on a deductive rule base with the possibility of inspecting the proof tree later-on.

Non-interference specification. Information-flow properties are specified in KeY using an extension of the Java Modeling Language (JML) [17], thereby introducing special *determines* clauses for expressing a fine-grained information flow control [22]. These constructs can be used for modular specifications on the method level as well as for enhancing loop invariants for the self-composition of loop statements and block contracts for the self-composition of arbitrarily chosen blocks of statements enclosed by curly braces.

The central specification elements for IFC purposes consist of the two keywords `determines` and `\by` both followed by a comma-separated list of JML expressions. The `determines` clause states that the JML expressions found after the `determines` keyword depend *only* on the JML expressions found after the `\by` keyword. This can furthermore be followed by the keyword `\new_objects` for specifying fresh objects to be included in the isomorphism. With this toolkit, powerful specification elements are given for proving non-interference, also allowing for declassification.

3 The Combined Approach

In this section, we present our approach that combines the advantages of precise logic-based approaches based on theorem provers, such as KeY, and automatic SDG-based approaches using graph-traversal algorithms, such as JOANA. We argue that our approach gurantees the SNI property for a given program and specified sources and sinks.

In the following, we describe our combined approach on the example of proving non-interference for a given program P. In Section 2.1, we established that SDG-based IFC techniques can detect any illegal information flow. Hence, if the SDG-analysis indicates that there is no illegal information flow for the program P, we need no further action as it is guaranteed that non-interference holds. The combined approach is used in case the automatic SDG-based approach detects an illegal information flow and we want to check whether this information flow is a false positive or a genuine leak.

For the information flow check, we first create a system-dependency graph (SDG) as defined in [9]. The created SDG is over-approximated and thus may contain edges which do not represent an actual flow in the program, hence potentially leading to false positives. Our approach assumes that the SDG nodes corresponding to high inputs and low outputs are annotated as high and low respectively. Furthermore, N_h denotes the set of all nodes annotated as high, and N_ℓ the set of all nodes annotated as low. There is an illegal information flow if information may flow from a node that is annotated as high to a node that is annotated as low. If any set of N_h or N_ℓ is empty, there is no illegal information flow.

After the SDG has been annotated by the user, the automated tool runs an information flow check. This check returns a set of *violations*. A violation is a pair (n_h, n_ℓ) of a high node $n_h \in N_h$ (secret source) and a low node $n_\ell \in N_\ell$ (public sink) such that there is a path from n_h to n_ℓ. We then call the set of all nodes lying on a path from n_h to n_ℓ the *violation chop* $c(n_\ell, n_h)$. To keep the notation simple, we will also use $c(n_h, n_\ell)$ for the subgraph induced by those nodes. The set of all violation chops is denoted by C_V. If this set is empty, the SDG-based approach guarantees non-interference, independently from our approach. If – however – there is a false positive, C_V contains at least one chop. The idea of the combined approach is then to validate each violation chop $c(n_h, n_\ell) \in C_V$ and try to prove it does not exist on the semantic level in program P. We show this by verifying each chop to be interrupted (see Definition 7) with the help of a theorem prover.

Definition 7 (Unnecessary summary edge, Interrupted violation chop). *A summary edge $e = (a_i, a_o)$ is called unnecessary if we can prove with a theorem prover that, in the context of the SDG, there is no flow from the formal-in node f_i to the formal-out node f_o corresponding to a_i and a_o, respectively.*

A violation chop is interrupted, if we find a non-empty set S of unnecessary summary edges on this chop, such that after deleting the edges in S from the SDG, no path exists between the source and the sink of the violation chop.

In order to show that a summary edge $e = (a_i, a_o)$ is unnecessary, a proof obligation is generated for the theorem prover. This proof obligation states that there is no information flow from f_i to f_o. The proof is done for the method corresponding to the summary edge e and is generally done for all possible contexts. Additionally, results from software analyses done by the SDG-based approach (e.g., points-to analysis) are used to generate a precondition for the analyzed method thus increasing the likelihood of showing non-interference for that method and interrupting the violation chop.

Our approach attempts to interrupt each violation chop in C_V. For each violation chop a summary edge is taken, the appropriate information flow proof obligation is generated for the method corresponding to the summary edge, and a proof attempt is made using the theorem prover. Our non-interference transformation directly converts the summary edge information to a specification for KeY. If the proof is successful, the summary edge can then be deleted from the SDG based on Definition 7.

Note that this is possible as KeY's (object-sensitive) non-interference property is at least as strict as SNI (Definition 1). This however only holds without the opaqueness assumption, i.e., only for KeY's standard non-interference semantics based on Definition 3 and not Definition 5. If this obligation is chosen, low-equivalence of states from Definition 1 matches low-equivalence of heap locations from Definition 6. In conclusion, we can state that KeY's non-interference property is equivalent to SNI (Definition 1). This implies Theorem 1.

Theorem 1 (Non-Interference Combined Approach). *The combined approach guarantees sequential non-interference.*

We then check whether this violation chop is interrupted, in which case we can proceed to analyse the remaining violation chops until all of them are interrupted. If the violation chop is still not interrupted, or in case the proof attempt is not successful, another summary edge from the violation chop is chosen. If we are able to interrupt every violation chop by deleting unnecessary edges, our approach guarantees non-interference.

Note that each violation chop is guaranteed to contain at least one summary edge, namely the one corresponding to the `main` method. Generating a proof obligation for the main method – however – is equivalent to verifying the entire program with the theorem prover.

Proofs with the theorem prover are often performed fully automatically, but may sometimes need auxiliary specification and user interaction. Therefore, we want to minimize the theorem prover usage as much as possible. We hence developed a number of heuristics for choosing the order in which the edges are checked by the theorem prover.

4 Implementation

We implemented the combined approach using JOANA as the dependency-graph analysis tool and KeY as the theorem prover. In this section, we show how we generate the proof obligations for KeY in the form of specified Java code and also describe the heuristics choosing the summary edges that are to be analyzed by KeY.

4.1 Method Contracts

For the method corresponding to the summary edge selected by the heuristics we generate an information flow method contract such that a successful proof would show that there is in fact no dependency between the formal in and formal out node of the summary edge.

Thus, to show that a summary edge $se(a_i, a_o)$ is unnecessary we prove that there is no information flow between the corresponding formal-in node f_i and formal-out node f_o. In order to achieve this, we generate a JML specification for the appropriate method stating that f_o is determined by all formal in nodes other than f_i, as explained in Definition 8.

Definition 8 (Generation of the determines clause). *Let $se(a_i, a_o)$ be the summary edge to be checked, and let f_i and f_o be the formal nodes corresponding to the actual nodes a_i and a_o. Let L_i be a list of all formal-in nodes f'_i other than f_i of the method belonging to the call site of a_i and a_o. The following determines clause is added to the method contract:* `determines` f_o `\by` L_i.

Should the proof of this property succeed then it would show that f_o does not depend on f_i and therefore a_o does not depend on actual-in parameter a_i. Since there is no dependency between a_i and a_o the summary edge can be safely deleted from the violation chop.

In order to avoid some false positives, JOANA uses a points-to analysis which keeps track of the objects a reference o may point to (the points-to set of o). This information is useful, since it may show that two references cannot be aliased. We use the results of the points-to analysis to generate preconditions for the method contracts, as shown in Definition 9, thus transferring information about the context from JOANA to KeY and increasing the likelihood of a successful proof.

Definition 9 (Generation of preconditions). *Let o be a reference and P_o its points-to set. We generate the following precondition:* $\bigvee_{o' \in P_o} o = o'$

4.2 Loop Invariants

In Section 3, we stated that the proof with the theorem prover can cost a lot of time- and user-effort. The theorem prover needs auxiliary specification like loop invariants or frame conditions. The method contracts generated as described in the previous section are necessary for proving a summary edge is unnecessary, however in the general case they are not sufficient for a successful proof. If the method contains loops of any kind, the theorem prover needs loop-invariants. The automatic generation of loop-invariants is an active research field, see for example [15, 19]. These approaches focus on functional loop-invariants and do not consider information flow loop-invariants.

The determines clause, described in the previous section, can be used to specify the allowed information flows of a loop. The determines clause generated for a loop invariant is similar to the one for method contracts. Because the variables from the formal-in and formal-out nodes may not directly occur in the loop some adjustments are necessary. Definition 10 shows what determines clauses are generated for loops invariants:

Definition 10 (Generation of the determines clause for loop invariants). *Let $se(a_i, a_o)$ be the summary edge to be checked, and let f_i and f_o be the formal nodes corresponding to the actual nodes a_i and a_o. Let L_i be a list of all formal-in nodes f_i' other than f_i of the method belonging to the call site of a_i and a_o. Let V_i be the set of all variables in the loop and let I_i be a list of variables in the method that influence f_o. The following determines clause is added to the loop invariant:* `determines` f_o, V_i `\by` L_i, I_i.

Note that the sets V_i and I_i can be constructed by analysing the SDG.

4.3 Heuristics

The order in which the summary edges of in the violation chops are checked determine the performance of the combined approach. Ideally we would want to avoid proof attempts of methods that do have an information flow or of very large methods that would overwhelm the theorem prover (for example the main method). In order to achieve these goals we developed several heuristics.

A first category of heuristics searches the code for three patterns that are likely to cause false positives by the SDG-based tool . The first pattern focuses on the problem of array-handling. The tool considers the array to be one syntactical construct and ignores the indexes. Thus, for Listing 2, tools like JOANA would detect an information flow from `high` to the *return value*. Thus we consider methods containing array accesses to be more likely to cause a false positives and assign a higher priority to them.

The second pattern in Listing 3 considers infeasible path conditions. Through purely syntactical slicing, it is not possible to detect that there cannot be an illegal information flow in the example below. The current implementation finds simple excluding statements, like “`x < = 0`” and “`x > 0`”. While the heuristic itself does not check wheter a method contains infeasible paths it does assign a higher priority to methods containing complex path conditions.

The second category of heuristics attempts to identify the methods that are likely to run through the theorem prover automatically. Earlier, we mentioned that it is difficult to create precise loop-invariants and thus methods without loops are assigned a higher priority. Furthermore, the method should have as few as possible references to other classes and methods.

A third category of heuristics tries to identify the methods that, if proven non-interferent, would bring the greatest benefit to the goal of proving the entire program non-interferent. We assign a high priority to summary edges which are *bridges* in the SDG, i.e. an edge whose removal from the SDG would result in two unconnected graphs [5].

In the case that there is no bridge, we prefer the method with the highest number of connections i.e. the most often called method.

5 Evaluation

The evaluation is two-parted. First the combined approach was evaluated based on examples that generate false positives for SDG-based tools like JOANA. Second, we applied the combined on a case study based on a simple e-voting system. The simple e-voting system was taken from the information flow examples of the KeY system. Both evaluations were tested on a standard PC (Core i7 2.6GHz, 8GB RAM) and outline the advantages and limitations of the combined approach compared to the state-of-the-art.

5.1 List of Examples

We considered eleven examples, which cover different program structures and reasons for false positives. Each of these examples is not solvable by automated graph based approaches like JOANA.

In Table 1 we have listed the eleven examples. The evaluation is split into automatic mode and interactive mode. In the automatic mode, an attempt is made to prove the generated proof obligations automatically. In the interactive mode, the theorem prover is called for all proof obligations in interactive mode. In this mode, the user can perform automatic or interactive steps and can add auxiliary specification.

```
int[] array = new int[2];
array[0] = high;
array[1] = 3;
return array[1];
```

Listing 2. Array-handling

```
if (x > 0){ y = high; }
if (x <= 0){ low = y; }
return low;
```

Listing 3. Excluding statements

The eleven examples are again divided into two groups. First, there are individual methods that cause false positives. In the method *Identity* the high value is added and subtracted to the low variable such that the low value remains the same. On a syntactical level there is dependency from high to low but in reality there is none. In the method *Precondition* there is an if-condition that can never be true and the method *Excluding Statements* contains if-statements that can not both be true at the same program execution. The example *Loop Override* contains a loop which overrides the low value in the last loop execution. For this example the non-interference loop-invariant was not enough for an automated proof and further functional information had to be given by the user. The last simple method *Array Access* describes the problem described in Section 4.3, it represents the handling of data structures. The second group consists of programs that include these problems in different program structures. Based on the possible SDG, we regard simple flows, branching, nested summary edges and a combination of it all.

Table 1. List of examples

	Automatic Mode			*Interactive Mode*	
Program	**Provable**	**KeY Calls**	**Time**	**Provable**	**KeY Calls**
Individual Methods					
Identity	Yes	1	5 sec.	Yes	1
Precondition	Yes	1	5 sec.	Yes	1
Excluding Statements	Yes	1	5 sec.	Yes	1
Loop Override	No	1	7 sec.	Yes	1
Array Access	Yes	1	6 sec.	Yes	1
Whole Programs					
KeY example	Yes	1	7 sec.	Yes	1
Single Flow	Yes	1	6 sec.	Yes	1
Branching	Yes	2	10 sec.	Yes	2
Nested Methods	Yes	2	10 sec.	Yes	2
Mixture	Yes	4	19 sec.	Yes	3
Mixture with Loops	No	7	20 sec.	Yes	5

The example programs are in the scope of 5 to 30 lines of code. They show that the combined approach can prove programs automatically for which JOANA would generate false positives and KeY would require a significant amount of user interaction.

5.2 Case Study - E-Voting

In addition to the outlined examples a small case study has been conducted. The code used in this case study is attached in Appendix A (Listing 5) and represents a simple implementation of a voting system. The vote of every voter is read and sent over a simulated network. If the read vote is not valid, then 0 is sent instead to indicate abstention. The votes itself and whether a vote is valid is secret. All variables starting with `low` (e. g. `low_sendSuccessful`) are annotated as low and the `high_inputstream` is annotated as high.

In the first step of our combined approach, we use JOANA to analyze the program code based on the mentioned annotations. The SDG-based approach finds 14 violations. All these violations are false positives that occur due to the over-approximation of the SDG. Specifically, they occur because the condition `isValid(high_vote)`, which is high, controls which assignment to `low_sendSuccessful` is executed, so JOANA assumes that this variable depends on a secret input. In reality, the values assigned to `low_sendSuccessful` do not depend on which branch is taken, since they only depend on `low_outputStreamAvailable`, which remains constant during a fixed execution. All violations are different chops from the high input stream `high_inputstream` to the different low output streams at different locations, including one exception that can be thrown when assigning `low_numOfVotes` to a value.

The combined approach tries to validate these chops bottom up and interrupt them if possible. First the heuristic looks for smaller methods like `sendVote(int x)`, `inputVote()` and `isValid(int high_vote)` but all three of them are not secure in regards to our specification and thus cannot be proven secure with the KeY system. The approach then looks at the top-level method `secure_voting()`.

For the method given in Listing 4 our approach is able to generate most of the specifications automatically. The JML method contract can be generated automatically. For the loop-invariant, the current approach is not able to specify functional properties such as the frame condition or the term that is decreasing every loop-run. In Listing 4, the boxed commands must be added manually for a sufficient loop-specification. For the KeY to automatically prove the method contract two *block-contracts* are necessary. These are auxiliary specifications for a group of statements that provide supplementary information to the prover. For the method `secure_voting()` the information flow specification has to be copied manually for both if-blocks as shown in Appendix A.

This specification can then be proven with KeY for the first violation. All other 13 violations are running through the same top-level method and thus are satisfied by the same proof. Thus, we showed that our combined approach can automatically find the causes of false positives by the SDG-based tool and generate the necessary proof obligations in order to disprove the reported leaks.

```
/*@ normal_behavior
  @     determines low_outputStream, low_outputStreamAvailable,
  @                low_NUM_OF_VOTERS, low_numOfVotes,
  @                low_sendSuccessful \by \itself;
  @*/
void secure_voting() {
    /*@ loop_invariant  0 <= i && i <= low_NUM_OF_VOTERS;
      @ loop_invariant \invariant_for(this);
      @ determines low_outputStream, low_outputStreamAvailable,
      @            low_NUM_OF_VOTERS, low_numOfVotes,
      @            low_sendSuccessful, i \by \itself;
      @ decreases low_NUM_OF_VOTERS - i;
      @*/
    for (int i = 0; i < low_NUM_OF_VOTERS; i++) {
        ...
    }
    publishVoterParticipation();
}
```

Listing 4. Method `secure_voting()` with loop invariant

6 Related Work

There exist many different approaches for proving non-interference. In Section 2.2 and Section 2.1 we have introduced logic-based and SDG-based approaches. In addition, we will discuss two other possibilities for proving non-interference.

6.1 The Hybrid Approach

The hybrid approach by Küsters et al. [16] is the work most related to our approach. It combines the same type of tools, i.e. an automatic dependency-graph analysis with a theorem prover, in an attempt to prove non-interference for a given program with minimized user-effort. The hybrid approach attempts to show non-interference with the dependency-graph analysis tool first. If the attempt does not succeed, the user must identify the possible cause of the false positive and extend the program such that the affected low output is overwritten with a value that does not depend on the high inputs. The extension is not allowed to change the state of the original program, it is allowed to use an extended state and is only allowed to read and overwrite variables from the original program. The extended program must be shown to be non-interferent with the dependency-graph analysis tool. In the next step, the theorem prover is used to show that the extended program is equivalent to the original program (modulo the extended state).

Similarly to our approach, the SDG-based tool is called first and if it does not prove non-interference, further action is taken. Unlike our case, the user has to analyze the program and the output of the SDG-based tool carefully in order to find out whether the reported flow is a false positive or not. The user then has to extend the program such that the low output is overwritten with a value such that the SDG-based tool successfully shows non-interference and then use the theorem prover to prove that the extended program is equivalent to the original one. In our approach, the interaction between the SDG-based tool and the theorem prover is automatic, the user needs to provide functional auxiliary specification when necessary.

6.2 Path Conditions

Path conditions [13] are another example of how SDG-based tools can be combined with more precise approaches, in this case constraints solvers, to increase precision. For a violation found by such an analysis, a path condition is a necessary condition that an information flow exists from the source to the sink of the violation. Path conditions can be computed automatically. In the program "`int y = high; if (x < 0) low = y;`", the path condition for an information flow from `high` to `low` would be `x < 0`. A constraint solver can then be used to generate a satisfying assignment for the path condition, which is a potential witness for an illegal flow. If the path condition is not satisfiable, then one can conclude that the violation was a false alarm.

Thus SDG-based non-interference analysis can be improved by path conditions. But it is important to state that the generation of path conditions is non-trivial [13]. The generation and checking of path conditions is not feasible for huge programs and is therefore not fully included in SDG-based tools like JOANA.

6.3 Type Systems

A well established technique is the information flow analysis based on security type systems. Type systems usually use syntactic rules to assign security types, typically low and high, to expressions and statements of a given program. If the program is typeable, the non-interference property holds. Examples of such a security type system are given in [21] or in [23].

The advantages of security type systems is that there is a clear separation between the rules and the concrete program execution. Furthermore, soundness proofs and the verification of a program with type systems are very fast. The disadvantages on the other hand are possible false positives and limitations of type systems. Most type systems are neither flow-, context- nor object sensitive, which degrades precision. Also, there exist languages like *separation logic* for which there is no known type system available.

7 Conclusion and Future Work

In this paper we introduced a new combined approach to prove non-interference with less user interaction while keeping the same precision. Our approach combines an automated SDG-based technique with a deductive theorem prover. We demonstrated that the non-interference properties guaranteed by the two tools are compatible and, thus, that our approach is sound. The combined approach has been developed tool-independently, but implemented and evaluated on a selection of examples as well as a small case study. Although the programs covered in our evaluation do not exceed 100 lines of code and could – as such – also be proven without the help of SDG-based IFC, they could – however – also be embedded in much bigger programs, which – as such – may be clearly too big for the analysis with a theorem prover. Thereby, our evaluation demonstrates promising results for complex programs and we are confident that much bigger programs are in reach.

An extended case study, covering programs too big to be checked by a theorem prover alone, is planned. For future work, the heuristics can be improved by integrating an SMT solver in order to enhance the recognition of excluding statements or further excluding program structures. The user-effort of the approach can be further minimized by automating the generation of functional loop invariants. Furthermore, the approach itself can be extended to also cover non-sequential programs and declassification.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book: From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
2. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3148, pp. 100–115. Springer (2004)
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE. pp. 100–114. IEEE (2004)
4. Bell, D.E., LaPadula, L.J.: Secure computer systems: Mathematical foundations. Tech. rep., DTIC Document (1973)
5. Bollobás, B.: Modern graph theory, vol. 184. Springer Science & Business Media (2013)
6. Darvas, Á., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow, pp. 193–209. Springer (2005)
7. van Delft, B.: Abstraction, objects and information flow analysis. Ph.D. thesis, Chalmers University of Technology, Goeteborg, Sweden (2011)
8. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
9. Giffhorn, D.: Slicing of Concurrent Programs and its Application to Information Flow Control. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)

10. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984. pp. 75–87. IEEE Computer Society (1984)
11. Graf, J., Hecker, M., Mohr, M.: Using joana for information flow control in Java programs-a practical guide. In: Software Engineering (Workshops). pp. 123–138 (2013)
12. Hammer, C.: Experiences with pdg-based IFC. In: Massacci, F., Wallach, D.S., Zannone, N. (eds.) Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5965, pp. 44–60. Springer (2010)
13. Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: IEEE International Symposium on Secure Software Engineering. pp. 87–96 (2006)
14. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8(6), 399–422 (2009)
15. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2006)
16. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of Java programs. Proceedings of the Computer Security Foundations Workshop 2015-Septe, 305–319 (2015)
17. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
18. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. 9(4), 410–442 (2000)
19. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42(4), 443–476 (2007)
20. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE pp. 214–227 (1999)
21. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications 21(1), 5–19 (2003)
22. Scheben, C., Schmitt, P.H.: Verification of information flow properties of Java programs without approximations. In: International Conference on Formal Verification of Object-Oriented Software. pp. 232–249. Springer (2011)
23. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of computer security 4(2-3), 167–187 (1996)
24. Wasserrab, D.: From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (Oct 2010), http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020678

A Source Code for E-Voting Case Study

```
/**
 * Information flow example.
 * The example is a toy implementation of a voting process. The vote of
 * every voter is read and sent over a not further modelled network. If
 * the read vote is not valid, then 0 is sent instead to indicate
 * abstention. The votes itself and whether a vote is valid is secret.
 * At the end the participation is output.
 * Without the optimizations described in the FM-Paper the verification
 * of the method secure_voting() with the help of self-composition is
 * very expensive or even infeasible.
 *
 * @author Christoph Scheben
 */
public class Voter {
    public static int low_outputStream;
    public static boolean low_outputStreamAvailable;
    private static int high_inputStream;

    public static final int low_NUM_OF_VOTERS = 763;
    public static int low_numOfVotes;
    public boolean low_sendSuccessful;

    private boolean high_voteValid;

    public static void main(String[] args) {
        Voter v = new Voter();
        v.secure_voting();
    }

    /*@ normal_behavior
      @     determines low_outputStream, low_outputStreamAvailable,
      @                low_NUM_OF_VOTERS, low_numOfVotes,
      @                low_sendSuccessful \by \itself;
      @*/
    void secure_voting() {
        /*@ loop_invariant 0 <= i && i <= low_NUM_OF_VOTERS
          @             && \invariant_for(this);
          @ determines low_outputStream, low_outputStreamAvailable,
          @            low_NUM_OF_VOTERS, low_numOfVotes,
          @            low_sendSuccessful, i \by \itself;
          @ decreases low_NUM_OF_VOTERS - i;
          @*/
        for (int i = 0; i < low_NUM_OF_VOTERS; i++) {
            int high_vote = inputVote();
            /*@ normal_behavior
              @     determines low_outputStream,
              @                low_outputStreamAvailable,
              @                low_NUM_OF_VOTERS,
              @                low_numOfVotes,
              @                low_sendSuccessful \by \itself;
              @*/
            {
                if (isValid(high_vote)) {
                    high_voteValid = true;
                    low_sendSuccessful = sendVote(high_vote);
                } else {
                    high_voteValid = false;
                    low_sendSuccessful = sendVote(0);
                }
            }
            /*@ normal_behavior
              @     determines low_outputStream,
              @                low_outputStreamAvailable,
              @                low_NUM_OF_VOTERS, low_numOfVotes,
              @                low_sendSuccessful \by \itself;
              @*/
            {
                low_numOfVotes =
                    (low_sendSuccessful ?
                        low_numOfVotes + 1 : low_numOfVotes);
             }
        }
        publishVoterParticipation();
    }

    int inputVote() {
        return high_inputStream;
    }

    boolean sendVote(int x) {
        if (low_outputStreamAvailable) {
            // encrypt and send over some channel
            // (not further modeled here)
            return true;
        } else {
            return false;
        }
    }

    boolean isValid(int high_vote) {
        // vote has to be in range 1..255
        return 0 < high_vote && high_vote <= 255;
    }

    void publishVoterParticipation() {
        low_outputStream =
            low_numOfVotes * 100 / low_NUM_OF_VOTERS;
    }
}
```

Listing 5. Source Code

A Linguistic Framework for Firewall Decompilation and Analysis*

Extended Abstract

Chiara Bodei[1], Pierpaolo Degano[1], Riccardo Focardi[2],
Letterio Galletta[1], Mauro Tempesta[2], and Lorenzo Veronese[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{chiara,degano,galletta}@di.unipi.it
[2] DAIS, Università Ca' Foscari Venezia, Italy
{focardi,tempesta}@unive.it, 852058@stud.unive.it

Abstract

Configuring and maintaining firewall configurations is notoriously complex. Policies are written in low-level, platform-specific languages where firewall rules are inspected and enforced along nontrivial control ow paths. Moreover, firewalls are tightly related to Network Address Translation since filters must be implemented with addresses translations in mind, further complicating the task of administrators. Here we propose a way of decompiling a real firewall configuration into an abstract declarative specification. We define a linguistic framework parametric with respect to the rule inspection control flow and that provides the typical features of real languages. Widely deployed firewalls used in Linux/Unix can be represented in our framework. Given a firewall configuration expressed in this way, we define a logical predicate that characterizes the accepted packets and their possible translations. We build a tool based on this logical characterization that uses the Z3 solver to produce a declarative specification that succinctly represents the firewall behavior. Tests on real configurations show our approach effective: the tool processes our university department policy in about 26 minutes, while subsets of the policy for simple subnets or specific hosts are synthesized in a matter of seconds.

1 Introduction

Firewalls are one of the standard mechanisms for protecting computers and network data but, as any other security mechanism, they become useless when improperly configured. Setting up a firewall can be a challenging task also for skilled network administrators, since configurations typically contain a large number of rules and it is often hard to figure out their impact in terms of firewall behavior. In addition, configurations must be maintained to reflect the updates of the desired security policies. Since rules may also interact with each other, incautious modifications may unexpectedly impact on the overall behavior of the firewall, with possible severe consequences on the connectivity or the security of the network.

Configuration languages are variegated and rather complex, accounting for low level details and supporting nontrivial control flow constructs, such as jumps between rulesets. The way firewall configurations are enforced typically depends on how packets are processed by the network stack of the operating system and needs to take into account Network Address Translation (NAT), the indispensable mechanism for translating addresses and performing port redirection.

Over the past few years, there has been a growing interest in high level languages that allow *programming* the network as a whole, e.g., the Software Defined Network (SDN) paradigm [21].

*Work partially supported by FilieraSicura, a project funded by Cisco Systems Inc. and Leonardo SpA.

Although SDN is spreading fast, it will take time before the "old" technology is dismissed, thus we can expect to still have to face a variety of configuration languages in the next years.

The literature provides many approaches for simplifying and analyzing firewall configurations. Many of them adopt a top-down approach, proposing ways to specify abstract policies that are compiled into real systems [1, 6, 10, 2, 9, 4, 23] or providing tools for aiding firewall management and spotting misconfigurations [1, 3, 31, 11, 20, 29, 24, 12, 15, 7, 8, 16]. Other proposals (like ours) follow a bottom-up approach, by extracting the model of the access control policy from the firewall configuration files [5, 30, 19, 14, 17, 13].

We take here the bottom-up alternative, and we study the problem of "decompiling" a real firewall configuration into an abstract declarative specification with the aim of extracting its *meaning*. To the best of our knowledge, this is the first proposal to synthesize a declarative specification starting from actual policies. A declarative version of the configuration makes it easier for administrators to check whether the firewall is implementing the intended security policy. Moreover, by comparing two specifications one can detect the differences between configurations and verify that updates have the desired effect on the firewall behavior. Decompilation also paves the way to cross-platform re-compilation into a different firewall system. This is particularly useful when migrating to a different infrastructure or to a new network configuration paradigm such as SDN.

2 Contribution

Starting from the most used firewall tools in Linux and Unix [27, 28, 26, 22], we have identified the fundamental ingredients of a typical firewall configuration. We exploit these ingredients to define a generic core language for expressing configurations with an abstract notion of packets, rules and state. Our language supports NAT, invocations to rulesets and stateful filtering, i.e., packet filtering that depends on the history of the previously received packets.

We endow the language with an operational semantics that specifies how packets are dealt with by the firewall in a given state, which is taken as a parameter. In addition, we introduce the notion of *control diagram*, which abstractly represents the rule inspection control flow related to the steps performed on the packets passing through the host.

We can encode a real firewall policy language in our framework by simply instantiating the state and the control diagram, up to minor syntactic transformations. The encoded language inherits our operational semantics and all of its properties. Interestingly, representing different languages in the same linguistic framework allows us to compare them and highlights subtle differences of their rule inspection control flow. We model the firewall tools in Linux/Unix mentioned above in our framework, showing it is sufficiently general and expressive. To the best of our knowledge, we provide for the first time a uniform, formal semantics to these tools.

Given a policy expressed in our framework, we transform it by unfolding all its control flow actions. We have formally proved that the transformation preserves the behavior of the policy. The unfolded policy is used as a starting point to synthesize a logical predicate that determines which are the packets accepted by the firewall policy and how they are possibly transformed due to NAT rules.

We have developed a tool that, given a configuration and a rule inspection control flow, computes the corresponding predicate and uses the Z3 solver [18] to synthesize a declarative firewall specification expressed with Mignis rules [1].

Our tool supports queries that can be used to verify properties of interest in a given policy, as well as to compare policies specified in different languages, once encoded in our framework. As a matter of fact, we used our tool to check whether or not a certain address is reachable from another one (recall that that addresses can be translated); whether a bidirectional communication is enabled; whether a policy implies another or whether they are equivalent, possibly leading to simplifications. We also computed the differences between non equivalent policies, so helping during maintenance operations. It is also worth mentioning that all the results of the above queries are displayed in a simple, human-readable form.

Tests on real configurations show the effectiveness of our approach: our tool can process the 175 ACL policies of the Stanford University backbone network [25] in less than 80 seconds. Also, our tool synthesizes the entire `iptables` policy of the DAIS department of the University of Venezia, which includes NAT rules and user-defined chains, in 26 minutes; fragments of the policy relative to simple subnets or specific hosts can be extracted in a few seconds.

References

[1] Pedro Adão, Claudio Bozzato, G. Dei Rossi, Riccardo Focardi, and Flaminia L. Luccio. MIGNIS: A semantic based tool for firewall configuration. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 351–365, 2014.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. ACM, 2014.

[3] Y. Bartal, A. Mayer, Nissim, and A. Wool: Firmato. A Novel Firewall Management Toolkit. *ACM Transactions on Computer Systems*, 22(4):1237–1251, 2002.

[4] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. *Firmato*: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[5] F. Cuppens, N. Cuppens-Boulahia, J. García-Alfaro, T. Moataz, and X. Rimasson. Handling stateful firewall anomalies. In *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 174–186. Springer, 2012.

[6] F. Cuppens, N. Cuppens-Boulahia, T. Sans, and A. Miège. A formal approach to specify and deploy a network security policy. In *Formal Aspects in Security and Trust (FAST'04)*, pages 203–218, 2004.

[7] Firestarter. `http://www.fs-security.com/`, 2007.

[8] Firewall builder. `http://www.fwbuilder.org/`, 2012.

[9] Simon N. Foley and Ultan Neville. A firewall algebra for openstack. In *2015 IEEE Conference on Communications and Network Security, CNS 2015*, pages 541–549, 2015.

[10] M.G. Gouda and A.X. Liu. Structured firewall design. *Computer Networks*, 51(4):1106–1120, 2007.

[11] High Level Firewall Language. `http://www.hlfl.org`, 2003.

[12] IPtables made easy, Shorewall. `http://www.shorewall.net/`, 2014.

[13] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. Technical report, Microsoft, 2014.

[14] 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, San Jose, CA, USA, April 25-27, 2012*, pages 113–126, 2012.

[15] KMyFirewall. `http://www.kmyfirewall.org/`, 2008.

[16] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15*, pages 499–512. USENIX Association, 2015.

[17] Alain J. Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *2000 IEEE Symposium on Security and Privacy, USA*, pages 177–187, 2000.

[18] Microsoft Research. Z3 theorem prover. https://github.com/Z3Prover/z3.

[19] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA 2010*, 2010.

[20] NeTSPoC: A Network Security Policy Compiler. http://netspoc.berlios.de, 2011.

[21] Open Networking Foundation. Software-Defined Networking (SDN) Definition. https://www.opennetworking.org/sdn-resources/sdn-definition.

[22] Packet Filter (PF). https://www.openbsd.org/faq/pf/.

[23] S. Pozo, R. Ceballos, and R. M. Gasca. Afpl, an abstract language model for firewall acls. In *Proc. of the international conference on Computational Science and Its Applications, Part II*, ICCSA '08, pages 468–483. Springer-Verlag, 2008.

[24] Pyroman. http://pyroman.alioth.debian.org/, 2011.

[25] Stanford University Backbone Network configuration ruleset. https://bitbucket.org/peymank/hassel-public/.

[26] The IPFW Firewall. https://www.freebsd.org/doc/handbook/firewalls-ipfw.html.

[27] The Netfilter Project. https://www.netfilter.org/.

[28] The Nftables Project. https://netfilter.org/projects/nftables/.

[29] Uncomplicated Firewall. https://help.ubuntu.com/community/UFW.

[30] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), USA*, pages 199–213, 2006.

[31] B. Zhang, E. Al-Shaer, R. Jagadeesan, J. Riely, and C. Pitcher. Specifications of a high-level conflict-free firewall policy language for multi-domain networks. In *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*. ACM, 2007.

A Runtime Monitoring System to Secure Browser Extensions

Raúl Pardo[1], Pablo Picazo-Sanchez[1], Gerardo Schneider[1], and Juan Tapiador[2]

[1] Dept. of Computer Science and Engineering,
Chalmers — University of Gothenburg, Sweden.
pardo@chalmers.se, pablop@chalmers.se, gersch@chalmers.se
[2] Dept. of Computer Science, Carlos III University of Madrid
28911 Leganes, Madrid, Spain.
jestevez@inf.uc3m.es

1 Problem

Web browsers are applications originally created to surf over the Internet in a friendly way. Nowadays these browsers have turned into a richer software where, apart from surfing the web, users are provided with a vast variety of small applications, called browsers extensions, that are not maintained by the web browsers. Those browser extensions are usually developed by external developers and directly interact either with the web content or with the users.

Browser extensions considerably increase the functionality of the browser. For instance, using a well known translate browser extension, users can have any web page translated to their preferred language; telephone numbers can be remarked in the web page so that users can click on them and automatically open a desktop application to make that call or browser extensions that block the advertisement that some sites insert in the HTML.

However, the inclusion of these third parties applications in the web browsers poses new security and privacy challenges in terms of malware [4, 5, 3], advertisement [8, 9, 2, 1] or disclosing personal information about the user [6], *e.g.*, getting access to browser history or reading another site's password.

Nevertheless, despite working properly, an extension can have undesirable consequences. Imagine that Alice, who is not a Swedish speaker, visits her bank account website to check her current balance. She gets a HTML content with the latest transactions written in Swedish. Alice decides to use an extension to translate from Swedish to English and confirm that all transactions are correct. To perform this task the extension sends the whole HTML to an external server so that it is automatically translated. Finally, the server sends back the website completely in English. One might argue that this is an undesirable behaviour, since sensitive information— which was not relevant for the task that Alice wanted to perform—was disclosed. In this abstract, we describe preliminary ideas on how to effectively prevent leaks of this kind.

According to the official browser extension developer guidelines[3], a browser extension is an application composed of a manifest file, one or more HTML file(s) and zero

[3] https://developer.chrome.com/extensions

or more JavaScript. A manifest file is where the information about the extension, such as the capabilities that the extension might use (*Content Security Policy (CSP)*), is. The HTML files define the *User Interface (UI)* and it is the link between the extension and the user. Finally, JavaScript files contain the logic and how the extension behaves.

Conforming to the interaction of the extensions with the browser, they can be classified in two different groups: *persistent* or *event*. Both types are based on HTML and Javascript files whose content can be accessible indistinctly. However, persistent extensions are always running, whereas event extensions are opened and closes as they are needed.

Content script is an additional functionality that browser extensions can have. A content script is a Javascript file that can interact with the web content and alter it Interactive extensions. It can be used together with both persistent and event extensions. Communication between the content script and the extension can only be done by using specific calls to the API.

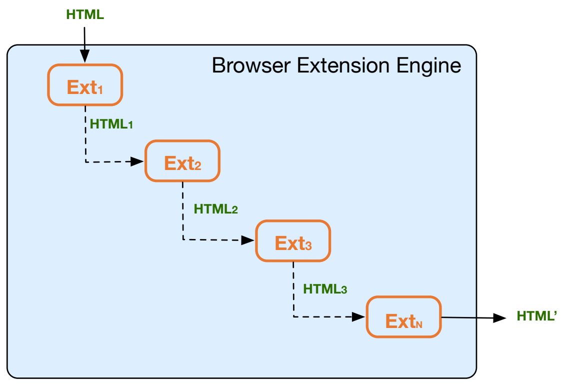

Fig. 1: Browser Extension Engine

Browser extensions are typically executed in a sequential manner. Figure 1 shows the execution order together with their inputs and outputs. Note that, in the figure, Ext_1 takes the original HTML file, performs some actions, and passes the resulting HTML_1 to Ext_2. The actions that extensions are allowed to perform can be controlled by means of CSPs. Roughly speaking, CSPs act as a filter to allow extensions to load and execute external resources.

2 Our Approach

Consider again Alice's example in the introduction. In order to avoid having her current balance sent to the translation server she can activate a policy that says: "Extensions cannot send web pages to the internet". This policy would make the Translate extension unusable, since it relies on external servers to perform translations. Moreover, Alice does not want to forbid the extension to send any kind of information. She only wants to prevent the extension to send the balance in her bank account. In particular, a better policy would be: “When I visit my bank website do not send any numerical digits to the internet”. Note that the previous policy includes a condition that depends on Alice’s visiting her bank website.

In [7] Pardo *et al.* introduced *policy automata*, a formalism to describe policies which depend on the state of the system and events, also known as *evolving policies*. It was applied in the context of social networks. Nevertheless, in this abstract we propose a similar approach to tackle the problems in browser extensions we have described.

Policy automata consists of a set of states, which indicate the policy ($\mathcal{P}$) that must be activated in the system, and transitions between states. Each transition is labelled with an event, a boolean condition and an action, which we denote $e/c/a$. An event e can be any event that is detectable in the browser. For instance, a user’s visiting a website or an extension sending information to an external server. The condition c is any condition regarding the state of the browser. For example, checking whether the tab is in incognito mode or certain extension is enabled. Finally, the action a can be an arbitrary program, which would be executed as a consequence of triggering the transition.

Example 1. Consider the policy we mentioned earlier: “When I visit my bank website do not send any numerical digits to the Internet”. It can be modelled using policy automata as follows:

$$\begin{tikzcd}
\text{start}// \arrow[r] & \boxed{\mathcal{P}_{HTML}} \arrow[r, bend left, "\textit{start Translate}/\textit{https://mybank.com/*} \in \textit{current\_tabs}/"] & \boxed{\mathcal{P}_{HTML-NumericalDigits}} \arrow[l, bend left, "\textit{start Translate}/\textit{https://mybank.com/*} \notin \textit{current\_tabs}/"] \\
& s_0 & s_1
\end{tikzcd}$$

When a user opens the browser, the automaton is in the initial state s_0. The elements inside the states represent policies that must be enforced. In s_0 the policy $\mathcal{P}_{HTML}$ represents that extensions can access all the HTML content that the browser has rendered. The transition from s_0 to s_1 models that, when the translation extension is activated, if *mybank.com* is opened in a tab then the automaton changes to state s_1. In state s_1 the policy $\mathcal{P}_{HTML-NumericalDigits}$ means that extensions can get the whole HTML except for numeric values. Finally, if the automaton is in state s_1 and the translation starts, but the tab *mybank.com* is not present then the automaton enables again the policy $\mathcal{P}_{HTML}$. When an event occurs but the condition is not satisfied the automaton remains in the same state. □

Note that, in the previous example, it is possible for another malicious extension to modify the numerical digits of the HTML so that the information is sent to the translation server. This is known as collusion attack and protecting against this type attacks would require a more sophisticated policy. Policy automata can also be used to prevent instances of this type of attacks. The sequence of events which occurs during the collusion can be specified in the automaton, and the required policies can be activated.

Policy automata define the behaviour of monitors that will run in parallel together with the web browser. The monitors will be in charge of activating and deactivating the static policies according to the specification in the automaton. Policy automata can be automatically compiled to Java monitors using LARVA [7]. However, we are still evaluating other approaches to directly implementing the monitors that are more targeted for our setting.

3 Discussion

We have identified browser extensions as a potential source of personal information leakage. We are currently looking into policies that can be enforceable using our technique. In particular, we are focusing in the implementation of an extension or a plug-in for the web browser Chromium. The events and information that can accessed as well as the (static) policies that can be enforced in Chromium will determine the type of policies that we can effectively implement.

Related Work. In [4] authors proposed an application that classifies extensions according to some parameters (developer reputation, code base or behaviour) as malware and they were automatically removed from the Chrome Store. A similar work named Hulk was proposed in [5] were extensions were classified by identifying suspicious behaviours. In our proposal, we go one step further and not only do we detect whether a browser extension is or is not disclosing sensitive information about the user but we also create a mechanism to avoid that sensitive information — without skipping the execution of the browser extension which is leaking our data — will be sent.

Acknowledgements This research has been supported by the Swedish funding agency SSF under the grant *Data Driven Secure Business Intelligence* and the Swedish Research Council (*Vetenskapsrådet*) under grant Nr. 2015-04154 (*PolUser: Rich User-Controlled Privacy Policies*).

Bibliography

[1] Sajjad Arshad, Amin Kharraz, and William Robertson. *Identifying Extension-Based Ad Injection via Fine-Grained Web Content Provenance*, pages 415–436. Springer International Publishing, Cham, 2016.

[2] Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 481–496, Austin, TX, 2016. USENIX Association.

[3] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[4] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and lessons from three years fighting malicious extensions. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 579–593, 2015.

[5] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 641–654, 2014.

[6] Lei Liu, Xinwen Zhang, Vuclip Inc, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *In 19th Network and Distributed System Security Symposium (NDSS '12*, 2012.

[7] Raúl Pardo, Christian Colombo, Gordon J. Pace, and Gerardo Schneider. An automata-based approach to evolving privacy policies for social networks. In *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings*, pages 285–301, 2016.

[8] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *S&P 2015*, pages 151–167, 2015.

[9] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding malvertising through ad-injecting browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1286–1295, 2015.

Making Decryption Accountable

Mark D. Ryan

University of Birmingham

Abstract. Decryption is accountable if the users that create ciphertexts can gain information about the circumstances of the decryptions that are later obtained. We describe a protocol that forces decryptors to create such information. The information can't be discarded or suppressed without detection. The protocol relies on a trusted hardware device. We describe some applications.

1 Introduction

When I was a teenager, I wanted to be able to go out in the evening and not have to tell my parents where I was going. My parents were understanding about this wish for privacy, but explained that if for some reason I didn't come back at the expected time, they needed to have some clues to give to the police about where I had been. So we came to the following compromise: I would leave a sealed envelope explaining my activities. This would enable them to search for me if they needed to, but if I came back on time I could retrieve the envelope and see that it had not been opened.

To have such a protocol in the digital world, we would need some way of knowing whether someone who has all the needed material to perform a decryption has actually performed it. More generally, we need a way to make decryption key holders accountable in some way for their use of the key. This accountability might take many forms. For example, some applications might need fine-grained accounts of exactly what was decrypted, and when, while in other cases we may be interested only in volumes, frequencies, or patterns of decryption.

In this paper, we informally describe the requirements for making decryptions accountable (section 2), and devise a protocol based on trusted hardware that achieves them (section 3). We describe a few applications at a very high level (section 4).

2 The requirements

We formulate the requirements as follows:

- Users $U_1, \ldots$ create ciphertexts using a public encryption key ek.
- Decrypting agent Y is capable of decrypting the ciphertexts without any help from the users.

– When Y decrypts ciphertexts, it unavoidably creates evidence e that is accessible to the users. The evidence cannot be suppressed or discarded without detection.
– By examining e, the users gain some information about the nature of the decryptions being performed.

Here, the granularity of e is left open. We will see some examples in section 3.2.

3 Protocol design

Intuitively, if Y has a ciphertext and a decryption key, it is impossible to detect whether she applies the key to to ciphertext or not. This implies that the key has to be guarded by some kind of hardware device D that controls its use. In this section, we propose a simple generic design that achieves some of the desired functionality. The hardware device D embodies the secret decryption key dk corresponding to ek. The secret decryption key dk never leaves the device.

In order to make the evidence e persistent, we assume a log L. The log is organised as an append-only Merkle tree as used in, for example, certificate transparency [1]. The log maintainer publishes the root tree hash H of L, and is capable of generating two kinds of proof about the log's behaviour:

– A proof of presence of some data in the log. More precisely, given some data d and a root tree hash H of the log, the log maintainer can produce a compact proof that d is indeed in the log represented by H.
– A proof of extension, that is, a proof that the log is maintained append-only. More precisely, given a previous root tree hash H' and the current one H, the log maintainer can produce a proof that the log represented by H is an append-only extension of the log represented by H'.

(Details of these proofs can be found in e.g. [8].) This means that the maintainer of L is not required to be trusted to maintain the log correctly. It can give proofs about its behaviour.

3.1 Performing decryptions

The decrypting agent Y uses the device D to perform decryptions. The device will perform decryptions only if it has a proof that the decryption request has been entered into the provably-append-only log.

The device maintains a variable containing its record of the most recent root tree hash H that it has seen of the log L. On receiving a set R of decryption requests, the decrypting agent performs the following actions:

– Obtain from the device its last-seen root tree hash H.
– Enter the set R of decryption requests into the log.
– Obtain the current root tree hash H' of the log.
– Obtain from the log a proof π of presence of R in the log with RTH H'.

– Obtain from the log a proof ρ that the log with RTH H' is an append-only extension of the log with RTH H.

The decrypting agent presents (R, H', π, ρ) to the device. The device verifies the proofs, and if they are valid, it performs the requested decryptions R. It updates its record H of the last-seen root tree hash with H'.

3.2 Evidence

Evidence about decryptions is obtained by inspecting the log, which contains the decryption requests. There are many ways that this could be organised. We look at two examples:

Example 1: the log contains a hash of the decrypted ciphertext. This allows a user U to detect if ciphertexts she produced have been decrypted.

Example 2: the log contains a unique value representing the decrypted ciphertext, but the value cannot be tied to a particular ciphertext (for example, the value could be the hash of a re-encryption [7]). This allows users to see the number of ciphertexts decrypted, but not which particular ones.

3.3 Currency

As described so far, the protocol is insecure because the device D could be tracking a version of the log which is different from the version that the users track. Although both the device and users verify proofs that the log is maintained append-only, there is no guarantee that it is the same version log. The log maintainer can bifurcate the log, maintaining each branch independently but append-only.

Gossip protocols of the kind proposed for solving this problem for certificate transparency [5] are insufficient here, because the device D is not capable of reliably participating in them.

To ensure that users track the same version of the log that D tracks, we introduce an additional protocol of D. In this second protocol, D accepts as input a *verifiably current* value v. The value v cannot be predicted in advance, but is verifiable by anyone. D outputs its signature $\mathrm{Sign}_{sk}(v, H)$ on the value v and its current stored root tree hash H of the log. Thus, we require that D has an additional secret key sk for signing. The corresponding verification key vk is published. Like dk, the key sk never leaves the device.

There are several ways in which the verifiably current value v can be constructed. For example, v can be the hash of a data structure containing nonces $v_1, \ldots$, each one produced by one of the users $U_1, \ldots$. Alternatively, v could be the concatenation of the date and the day's closing value of an international stock exchange.

Periodically, the current value of H tracked by the device is published. By means of the proofs of extension, users can verify that it is consistent with their view of the log.

3.4 The trusted hardware device

The protocol described relies on having a trusted hardware device D that performs a specific set of operations that are recapped here. The aim is to keep the functionality of D as small and as simple as possible, while still allowing it to support the variety of applications mentioned below (section 4). In summary, D stores persistent keys dk (decryption) and sk (signing), and the current root tree hash H of a log. It offers two services:

Decryption. It accepts a tuple (R, H', π, ρ) as described in section 3.1. It verifies the proof π that R is present in the log with root tree hash (rth) H', and the proof ρ that H' is the rth of a log that is an extension of the log of which its current rth is the H stored in D. (These verifications consist of some hash calculations and comparisons.) If the verifications succeed, it performs the decryptions R, and replaces its stored H with H'.

Attestation. It accepts a value v, and returns $\mathrm{Sign}_{sk}(v, H)$ on the value v and its current stored rth H.

4 Applications

Most electronic voting protocols begin with voting clients that encrypt votes with a public key, and end with the result being decrypted by a trustworthy party (or, possibly, a set of trustworthy parties each of which holds a share of the decryption key). The decrypting agents are trusted only to decrypt the result, and not the votes of individual voters. A protocol to make decryption accountable could help make this verifiable.

Finance is an area in which privacy and accountability are often required to be balanced. For this reason, the designers of Zerocash have introduced mechanisms which allow selective user tracing and coin tracing in a cryptocurrency [6]. Making decryptions accountable is another technique which could help obtain the desired combination of privacy and accountability.

The UK government has recently passed legislation allowing government agencies to access information about the communications of private citizens [2], in order to solve crimes. In an effort to provide some kind of accountability, there are stipulations in the law to ensure that the provisions of the act are used in ways that are necessary and proportionate to the crimes being addressed. A protocol that makes decryption accountable could make verifiable the quantity and perhaps the nature of decryptions [7].

Making decryptions accountable potentially addresses the problem of having to trust escrow holders, for example in identity-based encryption [4] and elsewhere [3].

5 Conclusion

There seems to be a variety of circumstances in which making decryption accountable is attractive. This paper proposes the design of trusted hardware which would assist in this process.

The idea of the design is that the decrypting agent has no way to decrypt data without leaving evidence in the log, unless it can break the hardware device D. This raises the question of who manufactures the device, and how the relying parties (both users $U_1 \dots$ and decrypting agents Y) can be assured that it will behave as specified. It depends on the sensitivity of the information being processed. One idea is that it is jointly manufactured by an international coalition of companies with a reputation they wish to maintain.

References

1. Certificate transparency. Available: `www.certificate-transparency.org`, 2007.
2. Investigatory Powers Act. Available: `www.legislation.gov.uk/ukpga/2016/25/contents/enacted`, 2016.
3. Harold Abelson, Ross Anderson, Steven M Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Matthew Green, Susan Landau, Peter G Neumann, et al. Keys under doormats: mandating insecurity by requiring government access to all data and communications. *Journal of Cybersecurity*, 2015.
4. Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual International Cryptology Conference*, pages 213–229, 2001.
5. Laurent Chuat, Pawel Szalachowski, Adrian Perrig, Ben Laurie, and Eran Messeri. Efficient gossip protocols for verifying the consistency of certificate logs. In *IEEE Conference on Communications and Network Security (CNS)*, pages 415–423, 2015.
6. Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. *IACR Cryptology ePrint Archive*, 2016:61, 2016.
7. Jia Liu, Mark D. Ryan and Liqun Chen. Balancing Societal Security and Individual Privacy: Accountable Escrow System. In *CSF*, 2014.
8. Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In Network and Distributed System Security (NDSS), 2014.

Securing the End-points of the Signal Protocol using Intel SGX based Containers

Kristoffer Severinsen Christian Johansen* Sergiu Bursuc
Dept. of Informatics, University of Oslo
{kristmse,cristi}@ifi.uio.no
University of Bristol, UK
sbursuc@gmail.com

This paper presents ongoing work on securing the end-points in the Signal messaging protocol using the new hardware security technology provided by Intel Software Guard Extensions (SGX). Signal is a recent secure messaging protocol, descendant from the classical off-the-record protocol, which has lately become popular partly due to the Snowden revelations. Signal, or variants of it, are now implemented, or under way of being implemented, in various major chat products, like from Facebook, WhatsApp, Google. Formal verification of Signal is interesting as well, e.g., two papers being presented at the 2017 European Symposium on Security and Privacy.

However, when studying communication protocols, usually one focuses on the protocol itself, and assumes the end-points to be free from malware or hardware attacks. In contrast, we are here focusing on securing the end-points of such an end-to-end secure communication protocol, i.e., the centralized server application, as well as the client side applications (which now typically run on smart phones). We make use of hardware enabled security features provided by the recent technology of Intel's SGX, part of the newer Skylake architectures. However, working with SGX can be tedious, therefore, we are looking at simpler ways of programming, using the recent SCONE secure containers. These are the secure counterparts of the popular Docker containers, implemented using SGX. Our work of implementing Signal using Intel's SGX can also be seen as an exploration and testing of the new features and performance of this new security technology from Intel.

1 Motivation

Secure messaging protocols have been around for more than a decade, with off-the-record (OTR) protocol[1] [5, 10] being a prominent example. OTR also has been implemented in standard instant messaging clients for quite some time, e.g., in Adium[2] (for MacOS), Jitsi[3] (cross-platform), or through plug-ins in the popular Pidgin[4] (for Linux). However, these have not seen wide adoption, partly due to usability difficulties [21, 25, 24], but also partly due to lack of motivation from the users. The Snowden revelations, however, triggered more concern, and recently we have seen an explosion in secure messaging implementations, with prominent example being the Signal protocol (formerly known as TextSecure). A few recent studies appeared about secure messaging in general [20, 24], as well as formal analysis of Signal/TextSecure [11, 12, 8] (the last two are going to be presented at the 2017 EuroS&P).

*The second author was partially supported by the projects IoTSec – Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the OffPAD project with number E!8324 part of the Eurostars program funded by the EUREKA and European Community.

[1] `https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html`

[2] `https://www.adium.im`

[3] `https://jitsi.org/Main/About`

[4] `https://developer.pidgin.im/wiki/ThirdPartyPlugins`

Submitted to:
HotSpot 2017 – 5th Workshop on Hot Issues in Security Principles and Trust

The Signal messenger, like many other secure messenger applications[5] relies on a centralized infrastructure to achieve asynchronous[6] communications between clients. The content of messages going through the server are end-to-end encrypted, but information about the sender and receiver is known to the server in order to route the messages [20]. This, and other, metadata can be used to obtain sensitive information about the clients [14].

Keeping the metadata secure requires trust in a large software and hardware stack. Even more so when the Signal server is set up in a cloud environment, since the trusted computing base (TCB) will include privileged software like firmware, hypervisor, the providers cloud management software, and in many cases the operating system as well, since cloud-providers supplies pre-built images. Not only do you have to trust the providers hardware and software stack, one also has to trust the cloud-provider's personnel, like the system administrators, and other personnel with physical access to the hardware. New research suggests you even have to trust other customers of the cloud-provider due to attacks like *memory massaging attacks* [19] which can fully compromise co-hosted cloud VMs [22]. To secure the server and metadata in this threat-model we have to remove the privileged software from the TCB, and protect the application's memory from lateral attacks.

In the recent survey [24] of secure messenger protocols and applications, the threat-model assumes that the end-points are secure from malware and hardware attacks. However, considering the large TCB, we would like to remove this assumption by using Intel Software Guard Extension (SGX) [15, 1] to keep the metadata encrypted in memory. Moreover, we think that the same technology of Intel's SGX can be used to similarly secure the desktop Signal clients. However, for mobile clients (e.g., running inside Android environments) one needs to investigate alternatives (e.g., ARM's TrustZone).

Our motivation is similar in spirit to the motivation of the authors of the recent article [7] where they want to secure the data handled by the Apache ZooKeeper (used for coordination of distributed systems) against privileged software, like hypervisors. They compare two approaches, either implementing the whole application inside an SGX enclave, or implementing only specific security functionalities inside enclaves, e.g., for encrypting data before storing or passing it around.

In this talk we plan to introduce the Signal protocol, emphasising the central server end-point, as well as introduce the essential functionalities of Intel's SGX that we plan to use. We will continue to discuss the recent secure containers of [2] and how they could be used for implementing the server-side of Signal. We will end with discussions of drawbacks and alternatives, including the recent SecureKeeper [7] or unikernels [13, 6] (where the recent Graphene [23] is the first, as far as we know, that claims support for Intel SGX).

2 Some technical details for our approach

We first present the technologies that we plan to use, and in the end we give a short presentation of the Signal instant messaging protocol.

[5]`https://en.wikipedia.org/wiki/Comparison_of_instant_messaging_clients#Secure_messengers`

[6]Note that OTR was intended for synchronous communications as with chats, and is thus not usable for securing SMS like asynchronous communications (a few modifications are needed, see [11]).

2.1 Intel Software Guard Extension

The Intel SGX [16] is an ISA extension to the Intel architecture that provides a trusted execution environment (TEE) for user applications. SGX allows applications to create a protected memory area inside its address space called an *enclave*. This protected environment provides confidentiality and integrity even from privileged software such as hypervisors, BIOS, or operating systems. Compared to TPMs [3, 9] the Intel SGX reduces the trust requirements from the CPU & TPM providers to just the CPU provider.

Enclaves can be created by an `ECREATE` instruction, which will create an *SGX enclave control structure* (SECS) in protected memory. Using the `EADD` instruction, memory pages can be added to the enclave. These pages are mapped to an encrypted and integrity protected part of physical memory, called the *enclave page cache* (EPC). On EPC page accesses the CPU will check that the CPU is running in *enclave mode* and proceed to decrypt the page at the granularity of cache lines. SGX only supports a limited amount of physical memory (in the range of 128 MB) for the EPC, but supports a paging mechanism for swapping the encrypted and integrity protected EPC pages in normal memory.

The code running inside an enclave has full access to the address space of the application, but to access the enclave the application must explicitly enter the enclave by calling the `EENTER` instruction. This will put the CPU in *enclave mode* and transfer the execution to a predefined point inside the enclave. The CPU will continue to run in enclave mode until the code explicitly exits by calling `EEXIT`, or until an exception (asynchronous exit) returns the control back to the operating system. After the operating system has handled the exception, the enclave can be resumed by calling `ERESUME` to return the execution inside the enclave.

In order to bootstrap a secure enclave, the enclave code cannot be encrypted, and it is thus open for inspection and modification by the host. To ensure the integrity of the enclave code, a cryptographic measurement is created of all the pages loaded into the enclave by executing the `EINIT` instruction.

When using SGX on a remote host, a trusted third party (Intel) can perform remote attestation to verify that the host is using a genuine implementation of SGX, and that the code sent to the host, is the same code that is loaded into the enclave. After the integrity of the enclave has been established the application can be started, and by establishing a secure communication channel from the client to the enclave using standard TLS, secrets like keys and sensitive data can be sent to the enclave.

If properly implemented, the enclave memory is confidentiality and integrity protected against firmware attacks, privileged software attacks, operator/administrator access and replay attacks. Entering the enclave is only possible at the predefined entry points, and protecting these entry points is the responsibility of the developer.

2.2 Linux Containers

The concept of *software containers* tries to solve the problem of managing software dependencies [17]. Conflicting or missing dependencies can be a big problem when deploying software to different services. If the developer does not have the same versions of the software as running in the production environment, a dependency conflict might break the functionality of the application. To solve this containers uses an isolated runtime environment, and pack the dependencies together with the application inside the container. A popular implementation is the *Docker containers*,[7] which also provide a repository[8] of

[7] `https://www.docker.com`

[8] `https://store.docker.com`

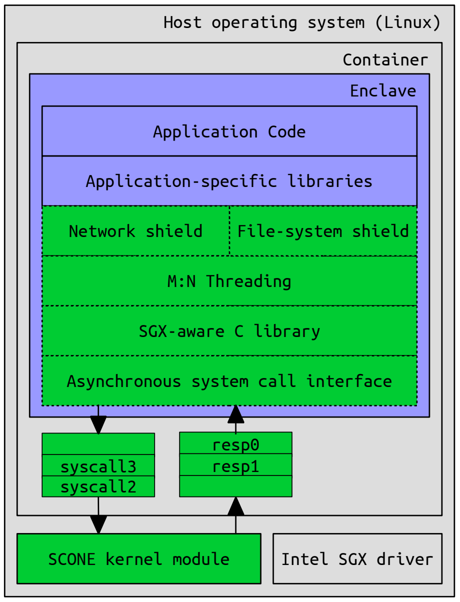

Figure 1: SCONE architecture [2]

Docker images curated by both trusted developers and the Docker community, making it very easy to pull and deploy applications.

Unlike virtual machines, that virtualizes the hardware of a machine, linux container software like Docker uses OS-level virtualization to isolate processes and their runtime environment. This makes containers much more lightweight than VMs since they do not need to boot up a full operating system to start the application. A container has all the files and binaries needed to run the application, but uses the operating system for services like I/O and resource management. Docker builds on the technology of Linux Containers (LXC)[9] to provide containerization. Using *kernel namespaces* the containers get their own view of system resources, and using *control groups* these resources can be limited by the host.

2.3 Approach: SCONE secure containers

An alternative approach to the SecureKeeper re-implementation using small enclaves is to run the entire unmodified application inside one enclave. Previous work on this approach include *Haven* [4], where a library operating system [13] and a shield module that handles scheduling threads, memory management and a file system were included inside the enclave to be able to run unmodified windows applications inside the enclave. A drawback of this solution is the large subset of Windows that the library OS includes, which adds considerable extra code to the TCB.

Recent work [2] in running unmodified applications inside a single SGX enclave make use of Docker containers instead of a library operating system. Thus, the objective of *SCONE* [2] is to make a *secure container* mechanism by placing the application and application-specific libraries of Docker containers inside an enclave.

Running unmodified applications inside enclaves requires a standard C library (libc) interface and an external interface to execute system calls, since enclaves do not support system calls. SCONE includes

[9]https://linuxcontainers.org/lxc/

the *musl*[10] libc library and the *Linux Kernel Library* [18] (LKL) to create a small Linux library OS.

Exiting and entering enclaves is an expensive operations, since it needs to do a context switch from the protected stack, and then sanitize the CPU registers so as to not leak information. To minimize enclave exits and entries, SCONE uses the hybrid (M:N) threading model, and supports asynchronous system calls by writing system calls on a queue outside the enclave. As seen in figure 1, the kernel module on the host will execute the system calls from the call queue, and put responses in the response queue.

To protect the enclave code from a malicious operating system, the system call interface does various checks on the system call parameters and responses, like checking if pointers and buffers resides inside or outside the enclave. The authors of [2] describe three different *shield modules* to protect I/O operations: the *file-system shield*, *network shield* and *console shield*. The file-system shield protects the confidentiality and integrity of files by transparently encrypting files used by the containers overlay file-system, which resides outside the secure enclave. The network shield encrypts the container's network interface transparently using TLS. The console shield encrypts the unidirectional console stream from the application using symmetric-key encryption; enabeling the operator to decrypt the console stream from a trusted environment. The shield modules are extendible, in case the containerized application has additional interfaces that require protection.

To create the secure containers for SCONE, the applications must be built as a SCONE executable by statically compiling them with the application-specific libraries, and the SCONE libraries. There is also need for some additional configurations in order to enable and configure the different shield modules. When complete, the secure Docker image can be published using the standard Docker Store. The secret information needed by the enclave to encrypt the file-system and console stream is provided by a special configuration file called the *startup configuration file* (SCF). The SCF is not included in the image, but is sent to the enclave over the TLS secured channel after SGX has verified the integrity and identity of the enclave.

2.4 Other approaches: SecureKeeper and Graphene libOS

Implementing native SGX support for the Signal server will require some additional code to invoke the special system calls used to create and enter the enclave. One approach is to identify the critical sections in the server that handles the metadata and routing of messages, and implement them using SGX. *SecureKeeper* used this approach to implement *Apache ZooKeeper* using SGX [7]. In SecureKeeper, an SGX enclave is used as a secure entry point for TLS encrypted requests to the server. The requests are decrypted inside the enclave, then the payload and path field of the requests are encrypted again before passing the request out of the enclave to the normal ZooKeeper code. The re-encryption of the payload and path is transparent to the ZooKeeper cluster, and is basically working like a disk-encryption scheme for the cluster. For SecureKeeper this approach worked well, and with little overhead, as measured by the authors. However, this may require more detailed programming knowledge of the system to be secured, as well as a re-implementation.

Scone provides a very small TCB by only including a common C library inside their secure container, but it should also be possible to run the Signal server inside a full library OS (also called unikernels). The Graphene library OS [23] supports multi-process applications, and have recently added support for running unmodified binaries inside SGX Enclaves[11]. Graphene claims to be able to run Java applications on top of OpenJDK inside a enclave with minimal development efforts.

[10]`https://www.musl-libc.org`

[11]`https://github.com/oscarlab/graphene/wiki/Introduction-to-Intel-SGX-Support`

2.5 Signal protocol

Signal is an instant messaging protocol devised with similar goals as the off-the-record protocol, i.e.,

end-to-end security or confidentiality, where only the intended conversation partners are able to read a message; in particular, the message should not be available to a third party like an intermediary server (offering some infrastructure support);

deniability which, given a sequence of messages and the relevant keys, ensures that there is no way for a judge to prove that a certain message was authored by a certain user;

forward secrecy which ensures that previously encrypted messages cannot be decrypted upon obtaining current and/or future keys;

future secrecy which ensures that a message cannot be decrypted even if keys from previous sessions are being compromised.

The Signal protocol goes over several phases, and a third trusted server is also involved, besides the two honest parties participating in the conversation. By honest parties it is only assumed that their long-term cryptographic material is not compromised.

> The *registration phase* involves the Trusted Server, as well as Google Cloud Messaging system (GCM). The trusted server needs the phone number of the participant to which a verification token is sent to check the ownership of the phone; the exchange of messages is done through HTTP and uses basic authentication. Various cryptographic material will be stored on the server for this user, some used to encrypt messages sent to GCM, others, the pre-keys, are used in encrypting messages.
>
> A *key comparing phase* can be done by the human parties, similar to what is done in OTR. In this stage the Signal App can compute a QR code, to make the comparison automatic.
>
> *Sending message phase* involves the trusted server to provide the stored pre-keys to be used in a complex key derivation algorithm called Axolot-ratcheting. This this conversation with the server, more information than just the message is being sent. In particular, identities of the participants.
>
> Sending subsequent messages, or sending reply messages within the same session, does not involve the trusted server.

2.6 Signal server

In [2] the authors built and benchmarked secure Docker images of *Redis*, *NGINX*, and *Memcached*. These applications are implemented in C, and can run natively inside the secure container. The Signal server however, is a Java application, and needs to run on top of a *Java Virtual Machine* (JVM). A lightweight JVM like JamVM[12] could be statically compiled to use the SCONE libraries and system call interface, and included inside the enclave.

The Signal server has at least three security critical parts. When sending a message to some user using Signal, the message is end-to-end encrypted using the recipients public-key, but the message header also contains the phone number of the recipient [20]. The network shield of SCONE will protect the this data, since the traffic is only decrypted inside the enclave. Both the SecureKeeper and the SCONE approach would be able to secure this information, since both will protect the communication end-points. The server already tries to protects the user-data on the server by only storing a hash of the phone numbers.

[12] `http://jamvm.sourceforge.net`

To lookup the contact information would require hashing the phone number and searching the database. Both the computations and the database should be hidden to the host, and the approach used by SCONE should accomplish this by running all of the application inside the enclave, and using the file-system shield of SCONE to protect the database.

To facilitate asynchronous first-time communications between users, the Signal server also stores a number of precomputed Diffie-Hellman public-keys from the users. These *pre-keys* are used to generate a shared secret between the users. Since pre-keys are tied to a user, this information should also be protected, and the same mechanism as above could be used to protect this information.

SCONE have not yet implemented support for SGX remote attestation, but using this feature is quite important if deploying the server to the cloud; remote attestation will confirm that the server is proteced by a genuine SGX implementation, and that the application code has not been modifed.

References

[1] Ittai Anati, Shay Gueron, Simon Johnson & Vincent Scarlata (2013): *Innovative technology for CPU based attestation and sealing*. In: *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, ACM. Available at `https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing`.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch & Christof Fetzer (2016): *SCONE: Secure Linux Containers with Intel SGX*. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, GA, pp. 689–703. Available at `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`.

[3] W. Arthur, D. Challener & K. Goldman (2015): *A Practical Guide to TPM 2.0*. APress, doi:10.1007/978-1-4302-6584-9.

[4] Andrew Baumann, Marcus Peinado & Galen Hunt (2015): *Shielding Applications from an Untrusted Cloud with Haven*. *ACM Trans. Comput. Syst.* 33(3), pp. 8:1–8:26, doi:10.1145/2799647.

[5] Nikita Borisov, Ian Goldberg & Eric Brewer (2004): *Off-the-record Communication, or, Why Not to Use PGP*. In: *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, ACM, New York, NY, USA, pp. 77–84, doi:10.1145/1029179.1029200.

[6] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad & K. Begnum (2015): *IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services*. In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 250–257, doi:10.1109/CloudCom.2015.89.

[7] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch & Rüdiger Kapitza (2016): *SecureKeeper: Confidential ZooKeeper Using Intel SGX*. In: *Proceedings of the 17th International Middleware Conference*, Middleware '16, ACM, pp. 14:1–14:13, doi:10.1145/2988336.2988350.

[8] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt & Douglas Stebila (2017): *A formal security analysis of the Signal messaging protocol*. In: *2nd IEEE European Symposium on Security and Privacy*, IEEE. Available at `https://eprint.iacr.org/2016/1013`.

[9] S. Delaune, S. Kremer, M. D. Ryan & G. Steel (2011): *Formal analysis of protocols based on TPM state registers*. In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, IEEE Computer Society Press, Cernay-la-Ville, France, pp. 66–82, doi:10.1109/CSF.2011.12.

[10] Mario Di Raimondo, Rosario Gennaro & Hugo Krawczyk (2005): *Secure Off-the-record Messaging*. In: *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, ACM, New York, NY, USA, pp. 81–89, doi:10.1145/1102199.1102216.

[11] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk & T. Holz (2016): *How Secure is TextSecure?* In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 457–472, doi:10.1109/EuroSP.2016.41.

[12] N. Kobeissi, K. Bhargavan & B. Blanchet (2017): *Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach*. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. Available at `http://prosecco.gforge.inria.fr/personal/bblanche/publications/KobeissiBhargavanBlanchetEuroSP17.pdf`. To appear.

[13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand & Jon Crowcroft (2013): *Unikernels: Library Operating Systems for the Cloud*. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, ACM, New York, NY, USA, pp. 461–472, doi:10.1145/2451116.2451167.

[14] Jonathan Mayer, Patrick Mutchler & John C. Mitchell (2016): *Evaluating the privacy properties of telephone metadata*. *Proceedings of the National Academy of Sciences* 113(20), pp. 5536–5541, doi:10.1073/pnas.1508081113.

[15] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue & Uday R. Savagaonkar (2013): *Innovative Instructions and Software Model for Isolated Execution*. In: *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, ACM, New York, NY, USA, pp. 10:1–10:1, doi:10.1145/2487726.2488368.

[16] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue & Uday R. Savagaonkar (2013): *Innovative instructions and software model for isolated execution*. In: *HASP@ ISCA*, p. 10. Available at `http://css.csail.mit.edu/6.858/2015/readings/intel-sgx.pdf`.

[17] Dirk Merkel (2014): *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. *Linux J.* 2014(239). Available at `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[18] O. Purdila, L. A. Grijincu & N. Tapus (2010): *LKL: The Linux kernel library*. In: *9th RoEduNet IEEE International Conference*, pp. 328–333. Available at `http://ieeexplore.ieee.org/document/5541547/`.

[19] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida & Herbert Bos (2016): *Flip Feng Shui: Hammering a Needle in the Software Stack*. In: *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, Austin, TX, pp. 1–18. Available at `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi`.

[20] Christoph Rottermanner, Peter Kieseberg, Markus Huber, Martin Schmiedecker & Sebastian Schrittwieser (2015): *Privacy and data protection in smartphone messengers*. In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, ACM, p. 83, doi:10.1145/2837185.2837202.

[21] Ryan Stedman, Kayo Yoshida & Ian Goldberg (2008): *A User Study of Off-the-record Messaging*. In: *Proceedings of the 4th Symposium on Usable Privacy and Security*, SOUPS '08, ACM, New York, NY, USA, pp. 95–104, doi:10.1145/1408664.1408678.

[22] Jakub Szefer, Eric Keller, Ruby B Lee & Jennifer Rexford (2011): *Eliminating the hypervisor attack surface for a more secure cloud*. In: *Proceedings of the 18th ACM conference on Computer and Communications Security*, ACM, pp. 401–412, doi:10.1145/2046707.2046754.

[23] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira & Donald E. Porter (2014): *Cooperation and Security Isolation of Library OSes for Multi-process Applications*. In: *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, ACM, New York, NY, USA, pp. 9:1–9:14, doi:10.1145/2592798.2592812.

[24] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg & M. Smith (2015): *SoK: Secure Messaging*. In: *2015 IEEE Symposium on Security and Privacy*, pp. 232–249, doi:10.1109/SP.2015.22.

[25] Alma Whitten & J. D. Tygar (1999): *Why Johnny Can'T Encrypt: A Usability Evaluation of PGP 5.0*. In: *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, USENIX Asso-

ciation, Berkeley, CA, USA, pp. 14–14. Available at `http://dl.acm.org/citation.cfm?id=1251421.1251435`.

On Composability of Game-based Password Authenticated Key Exchange

Jean Lancrenon[1] and Marjan Škrobot[2]

[1] itrust consulting,
lancrenon.jean@gmail.com
[2] SnT, University of Luxembourg,
marjan.skrobot@uni.lu

Abstract. It is standard practice that the secret key derived from an execution of *Password Authenticated Key Exchange* (PAKE) protocol is used to authenticate and encrypt some data payload using symmetric key protocols. Unfortunately, most PAKEs of practical interest are studied using so-called *game-based* models, which – unlike simulation models – do not guarantee secure composition *per se.* However, Brzuska et al. (CCS 2011) have shown that a middle ground is possible in the case of authenticated key exchange that relies on *Public-Key Infrastructure* (PKI): the game-based models do provide secure composition guarantees when the class of higher-level applications is restricted to symmetric-key protocols. The question that we pose in this paper is whether or not a similar result can be exhibited for PAKE. Our work answers this question positively. More specifically, we show that PAKE protocols secure according to the game-based Real-or-Random (RoR) definition of Abdalla et al. (PKC 2005) allow for safe composition with arbitrary, higher-level symmetric key protocols. Since there is evidence that most PAKEs secure in the Find-then-Guess (FtG) model are in fact secure according to RoR definition, we can conclude that nearly all provably secure PAKEs enjoy a certain degree of composition, one that at least covers the case of implementing secure channels.

Keywords: Cryptographic protocols, Password authenticated key exchange, Composability, Composition Theorem.

1 Introduction

1.1 The problem

The objective of *Password-Authenticated Key Exchange* (PAKE) is to allow secure authenticated session key establishment over insecure networks between two or more parties who only share a low-entropy password. Even though there may be other applications of PAKE, it is common practice that the secret key derived from a PAKE execution is used to authenticate and encrypt some data payload using symmetric key primitives and protocols. For example, two certificate-less

TLS proposals that integrate PAKE as a key exchange mechanism have recently appeared on the IETF [15,16][3]. When looking at these two drafts through the lens of composition, one sees that both of them suggest the PAKE be followed by Authenticated Encryption (AE) algorithms (namely AES-CCM and AES-GCM). Another project that makes use of PAKE is Magic Wormhole [1], the file transfer protocol in which PAKE is composed with NaCl's *crypto_secretbox* containing the stream cipher XSalsa20 and MAC algorithm Poly1305. Consequently, being able to guarantee the overall security of a *composed* protocol, consisting of first running a PAKE and then a symmetric key application, is imperative.

Unfortunately, the provably secure composition is difficult to automatically obtain without using complex, usually simulation-based models. Furthermore, most PAKEs in the literature are studied using so-called *game-based* models, which – while being workable to obtain acceptable proofs – do not guarantee secure composition. The most common such model used is the Find-then-Guess (FtG) model of Bellare et al. [5].

In [11], Brzuska et al. show that a middle ground is possible in the case of *Public-Key Infrastructure*-based key exchange (PKI-KE): Among other things, they define a framework for PKI-KE that (1) is game-based and (2) allows to prove that, under a certain technical condition, secure composition holds when the class of higher-level applications is restricted to symmetric-key protocols. The question is whether or not a similar result can be exhibited for PAKE.

1.2 Our contribution

In this paper, we answer this question positively by essentially adapting the framework in [11] to the password-based case. More specifically, we show that PAKE protocols secure in the sense of the game-based Real-or-Random (RoR) definition of Abdalla et al. [3] allow for automatic, secure composition with arbitrary, higher-level symmetric key protocols according to a security definition very similar to that in [11]. Since in [3] the authors provide evidence that most PAKEs secure in the FtG model of [5] are in fact secure according to RoR, we can conclude that nearly all provably secure PAKEs enjoy a certain degree of composition, one that at least covers the case of implementing secure channels. It should be noted that for our result to hold, we also need the technical condition mentioned earlier to be fulfilled. However, we emphasize that to the best of our knowledge, for nearly all published PAKEs this is always the case. Prominent examples include EKE [7], PAK [19], SPAKE2 [4], Dragonfly [17], and J-PAKE [2,18]. The next paragraph explains our work in more detail.

[3] The reason behind this integration - and not using PAKE with some symmetric cipher over TCP - is to circumvent the need to establish a network protocol for data transfer (i.e. TCP or UDP) and to negotiate symmetric key algorithms (or protocols) on their own.

1.3 Password-induced subtleties

It is well-known that already when dealing with "basic" PAKE definitions, the usual low-entropy nature of the long-term authentication material causes definitional headaches. It is, therefore, no surprise that similar issues should be encountered here. We begin with a simple recap of how PAKE security is defined in the foundational paper [5]. Then, we briefly explain the theorem of Brzuska et al. [11] and show where passwords cause trouble. Finally, we demonstrate how to circumvent this problem, and in particular why RoR is more suitable than FtG.

The Find-then-Guess model for PAKE. As in all reasonable key exchange security models, in [5] the adversary $\mathcal{A}$ is modeled as a network adversary: It can bring to life protocol participants with access to the secret long-term keying material and deliver to these instances messages of its choice. In the event that an instance accepts and computes a session key, $\mathcal{A}$ may ask that this key is revealed, modeling higher-level protocol leakage. In some models, it may even corrupt protocol participants in an effort to account for e.g. forward secrecy.

Crucially, to capture the fundamental notion of *session key semantic security*, $\mathcal{A}$ is allowed to make *a single* Test query, from which it receives either the real session key computed by the target instance or a random key. $\mathcal{A}$'s goal is to determine which it is. Its advantage $\mathbf{Adv}_{\mathrm{P}}^{FtG}(\mathcal{A})$ against protocol P is essentially defined as the distance of its success probability from 1/2.

In PKI-KE, i.e. when users' long-term keys are public key/secret key pairs, it is natural to ask that $\mathbf{Adv}_{\mathrm{P}}^{FtG}(\mathcal{A})$ be a negligible function in the security parameter. When the long-term keys are passwords however – say, uniformly selected from a dictionary Pass of size N – the best we can expect is:

$$\mathbf{Adv}_{\mathrm{P}}^{FtG}(\mathcal{A}) \leq \frac{B \cdot n_{se}}{N} + \varepsilon, \tag{1}$$

where B is some constant, ε is negligible, and n_{se} measures the number of instances $\mathcal{A}$ has tried online attacks on using *guessed* passwords[4]. Note that the first right-hand term is not negligible in general.

The composition result for PKI-KE in [11]. Let S be some arbitrary, two-party, symmetric key protocol and P; S denote its "natural" composition with P. The main theorem established in [11] for the PKI-KE case states that for every efficient adversary $\mathcal{A}$ playing a suitably defined security game against P; S there exist efficient adversaries $\mathcal{B}$ against P and $\mathcal{C}$ against S such that following formula holds:

$$\mathbf{Adv}_{\mathrm{P;S}}(\mathcal{A}) \leq q \cdot \mathbf{Adv}_{\mathrm{P}}^{FtG}(\mathcal{B}) + \mathbf{Adv}_{\mathrm{S}}(\mathcal{C}), \tag{2}$$

where q is the maximum number of instances in play in the key exchange game. Of course, in [11]'s framework, security of the composition holds when the left-hand term is negligible. Therefore, the upper bound implies this under the condition that P and S are secure. Indeed, observe that q is at most polynomial in

[4] B is usually interpreted as the number of passwords that can be tested simultaneously during one log-on attempt.

the security parameter and that $\mathbf{Adv}_{\mathrm{P}}^{FtG}(\mathcal{B})$ is supposed to be *negligible* when using PKI-KE. (And, of course, S is secure if $\mathbf{Adv}_{\mathrm{S}}(\mathcal{C})$ is negligible for all $\mathcal{C}$.) This effectively shows that the security of the composition $\mathrm{P};\mathrm{S}$ is guaranteed by the stand-alone security of P and S.

Two immediate password problems. There are two main obstacles to overcome when trying to get a password analog of Eq. 2 to work, and both stem from the non-negligible term in Eq. 1.

First, it is clear that the term $q \cdot \mathbf{Adv}_{\mathrm{P}}^{FtG}(\mathcal{B})$ cannot be negligible anymore. Thus, it makes no sense to try and deduce from Eq. 2 that the left-hand side is ultimately negligible. The only way out of this is to "boost" the left-hand side. Fortunately, there is a natural way to do this. Indeed, intuitively it should be clear that the composed protocol will also suffer from a breach in the event $\mathcal{A}$ guesses a password and mounts an online attack. Thus, it is the definition of security for the composed protocol that has to change, in that it needs to incorporate the same non-negligible bound as in Eq. 1. In other words, at best we can only require by definition that:

$$\mathbf{Adv}_{\mathrm{P};\mathrm{S}}(\mathcal{A}) \leq \frac{B \cdot n_{se}}{N} + \varepsilon, \tag{3}$$

where B is some constant, ε is negligible, and n_{se} counts $\mathcal{A}$'s online attacks. In short, our first problem is handled at the definition level. But, Eq. 3 leads to our second problem.

If we simply plug our optimal FtG PAKE bound into the right-hand side of 2, we obtain

$$\mathbf{Adv}_{\mathrm{P};\mathrm{S}}(\mathcal{A}) \leq \frac{B \cdot q \cdot n_{se}}{N} + \mathbf{Adv}_{\mathrm{S}}(\mathcal{C}). \tag{4}$$

This is not what we want: The q factor is still making the desired upper bound too large for our purpose! This is where using the RoR model comes in handy.

In the proof of the main theorem in [11], the authors need to make use of a hybrid argument indexed by the instances in play: The idea is to have the simulator plant the only available Test query at the randomly chosen index. This is what makes the q come out. Our observation is that by using the RoR model – in which *multiple* Test queries are allowed – we can avoid having this parasite factor appear.

In short, our main theorem says that for every efficient adversary $\mathcal{A}$ playing against $\mathrm{P};\mathrm{S}$ there exist efficient adversaries $\mathcal{B}$ and $\mathcal{C}$ such that:

$$\mathbf{Adv}_{\mathrm{P};\mathrm{S}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathrm{P}}^{RoR}(\mathcal{B}) + \mathbf{Adv}_{\mathrm{S}}(\mathcal{C}), \tag{5}$$

and from this theorem we get that if P and S are actually secure, the optimal bound stated in Eq. 3 holds[5].

[5] Note that the presence of passwords has no effect on the security of S as a stand-alone primitive. This is why $\mathbf{Adv}_{\mathrm{S}}(\mathcal{C})$ should remain negligible.

The technical condition. Let us briefly return to the "technical condition" mentioned in paragraphs 1.1 and 1.2. Roughly, it states that when observing many PAKE interactions over a network, it is publicly possible to determine pairs of communicants holding the same session key. This property is called *partnering* and is formally described further down. Often in PAKE research, partnering is defined using session identifiers that are locally computed. In practice, most published PAKEs define these identifiers simply by concatenating the PAKE message flows with their identities. Clearly, this is a publicly checkable criterion. Hence, the condition causes no real limitation to our result.

The big picture. We end this section by putting our result in a wider perspective. The RoR model was initially introduced by Abdalla et al. in [3] so that they could prove the security of a three-party PAKE generically constructed from two runs of a two-party PAKE. Also, they demonstrate that RoR security is better than FtG security for all PAKEs. Our work further validates the last claim. Indeed, RoR allows us to prove some composability in a way that seems difficult to adapt to FtG.

1.4 Related work

Here we shall briefly go over the papers that have contributed to secure composition of key exchange with other protocols.

Composition of key exchange. All key exchange models devised in the last two decades (e.g. [6,5,3,13]) support concurrent self-composition of key exchange protocols. The first to successfully provide a framework in the game-based setting that grants stronger composition guarantees were Canetti and Krawczyk [13]. Indeed, they identified a security notion (SK-security) that is sufficient to yield a secure channel when appropriately composed with a secure symmetric encryption algorithm and MAC. As far as we know, this result was never adapted to the password-based case. The simulation-based models of Shoup [20] (for ordinary key exchange) and Boyko et al. [8] (for PAKE) claim to have a "built in" composition guarantee, but this only been informally argued. Later, applying the methodology of Universal Composability (UC) for key exchange [14], a second, stronger simulation-based notion – Universally Composable PAKE – was proposed by [12]. These models' very strong composition guarantees are appealing, but unfortunately the models themselves are harder to work with than the simpler, game-based models. Another shortcoming of this approach is its restrictive nature which yields not overly efficient protocols. For a nice discussion on the limitations of UC and simulation-based AKE in general, we point reader to [10] and [13].

Although key exchange protocols proven in the game-based setting of [6] remained mostly used in practice, it took almost a decade before someone started addressing the problem of studying the composability properties of this setting. Namely, this was done by Brzuska et al. in [11,21,9]. They presented a more general framework which allowed showing that BR-style secure key exchange protocols are composable with a wide class of symmetric key protocols under the condition that a public session matching algorithm for the key exchange protocol exists. In subsequent work [10], the authors have shown that even a weaker notion for key exchange protocols than BR-security would still be enough for composition, and apply this to the TLS handshake. As far as we aware, no similar study has been conducted in the password-based setting. With this work, we aim to beginning filling this gap, by first adapting the results of [11].

2 Composition Result

As already explained in the introduction, for our composition result to hold we need to work with a PAKE model that is stronger than the FtG one from [5], namely the ROR model of [3]. This model guarantees that the adversary who does not know the correct password cannot distinguish *any* honestly generated session key from a random key drawn from the key space. In contrast, in the FtG model, only the session key that is targeted by the single available Test query is indistinguishable from random.

The rest of this section is devoted to present the following theorem:

Theorem 1. *Let $(PWGen, \mathrm{P})$ be a password authenticated key exchange protocol outputting keys according to a distribution $\mathcal{K}$, that is secure according to the RoR game G^{RoR}, and for which an efficient partner matching algorithm exists. Let $(KGen,\mathrm{S})$ be a symmetric key protocol secure according to the game G^{sym}. If the keys used in the symmetric key protocol algorithm S are distributed according to $\mathcal{K}$, then the composed protocol $(CGen, \mathrm{C})$ is secure according to G^{com} and the advantage of any efficient adversary $\mathcal{A}$ against composed protocol satisfies*

$$\boldsymbol{Adv}_{\mathrm{C}}^{com}(\mathcal{A}) \;\leq\; \boldsymbol{Adv}_{\mathrm{P}}^{RoR}(\mathcal{B}) \;+\; \boldsymbol{Adv}_{\mathrm{S}}^{skp}(\mathcal{C}) \tag{6}$$

for some efficient adversaries $\mathcal{B}$ and $\mathcal{C}$.

Intuition behind the proof. To prove Theorem 1, we first argue that all the session keys computed by the PAKE can be randomized, since the protocol is assumed to be RoR-secure (with weak forward secrecy). With this step, we will practically decouple the PAKE and SKP phases of the composed protocol, because the session keys that used in the SKP phase of the composition will, from that point on, be completely independent of those computed in the PAKE phase. Then, in the next step, we will show that the advantage of an adversary against the resulting composed game (with random keys) is upper bounded by the advantage of an adversary against the security game of the underlying symmetric key protocol.

An immediate consequence of our theorem is that preceding a secure symmetric key algorithm with an optimally *RoR-secure* PAKE yields an optimally secure composed protocol according to our definition of composition security.

3 Conclusion

Considered well-studied cryptographic objects in academia, PAKE protocols are just starting to appear more widely as building blocks in commercial real-world applications. They are typically used to generate keys that will grant two (or more) parties means to subsequently establish some type of secure channel between them. However, one cannot directly claim that the security of such a composed protocol holds, since PAKEs that are typically deployed – due to their efficiency and easier setup – are proven secure in game-based models that do not necessarily provide composition guarantees. Therefore, to substantiate the expected security claims, a new security proof would need to be exhibited for the entire protocol from scratch. As a result of Theorem 1, a modular design of more complex protocols is possible: One can obtain the secure protocol that would consist of RoR-secure PAKE protocol followed by a secure symmetric key protocol, without any additional analysis.

As future work, it would be interesting to adapt to the password-based case the study in [10] that among other things aim to take into account the incorporation of certain specific session-key-dependent messages (such as the "Finished" message from TLS). This could be useful for the protocols recently specified on the IETF [15,16]. Another observation worth making is that formally, our result could certainly adapted to the case of any authenticated key exchange.

References

1. Magic Wormhole (2016), `https://github.com/warner/magic-wormhole`
2. Abdalla, M., Benhamouda, F., MacKenzie, P.: Security of the J-PAKE Password-Authenticated Key Exchange Protocol. In: 2015 IEEE Symposium on Security and Privacy, SP 2015. pp. 571–587. IEEE Computer Society (2015)
3. Abdalla, M., Fouque, P., Pointcheval, D.: Password-Based Authenticated Key Exchange in the Three-Party Setting. In: Vaudenay, S. (ed.) Public-Key Cryptography – PKC 2005. LNCS, vol. 3386, pp. 65–84. Springer (2005)
4. Abdalla, M., Pointcheval, D.: Simple Password-Based Encrypted Key Exchange Protocols. In: Menezes, A. (ed.) Topics in Cryptology - CT-RSA 2005. LNCS, vol. 3376, pp. 191–208. Springer (2005)
5. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated Key Exchange Secure Against Dictionary Attacks. In: Advances in Cryptology – EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer (2000)
6. Bellare, M., Rogaway, P.: Entity Authentication and Key Distribution. In: Advances in Cryptology - CRYPTO 1993. pp. 232–249 (1993)
7. Bellovin, S.M., Merritt, M.: Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In: 1992 IEEE Symposium on Research in Security and Privacy, SP 1992. pp. 72–84 (1992)
8. Boyko, V., MacKenzie, P.D., Patel, S.: Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000. LNCS, vol. 1807, pp. 156–171. Springer (2000)
9. Brzuska, C.: On the Foundations of Key Exchange. Ph.D. thesis, Darmstadt University of Technology (2013)

10. Brzuska, C., Fischlin, M., Smart, N.P., Warinschi, B., Williams, S.C.: Less is more: Relaxed yet Composable Security Notions for Key Exchange. International Journal of Information Security 12(4), 267–297 (2013)
11. Brzuska, C., Fischlin, M., Warinschi, B., Williams, S.C.: Composability of Bellare-Rogaway Key Exchange Protocols. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011. pp. 51–62. ACM (2011)
12. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.D.: Universally Composable Password-Based Key Exchange. In: Cramer, R. (ed.) Advances in Cryptology – EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer (2005)
13. Canetti, R., Krawczyk, H.: Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In: Pfitzmann, B. (ed.) Advances in Cryptology - EUROCRYPT 2001. LNCS, vol. 2045, pp. 453–474. Springer (2001)
14. Canetti, R., Krawczyk, H.: Universally Composable Notions of Key Exchange and Secure Channels. In: Knudsen, L.R. (ed.) Advances in Cryptology - EUROCRYPT 2002. LNCS, vol. 2332, pp. 337–351. Springer (2002)
15. Cragie, R., Hao, F.: Elliptic Curve J-PAKE Cipher Suites for Transport Layer Security (TLS) (2016), `https://datatracker.ietf.org/doc/draft-cragie-tls-ecjpake/`
16. Harkins, D.: Secure Password Ciphersuites for Transport Layer Security (TLS) (2016), `https://datatracker.ietf.org/doc/draft-harkins-tls-dragonfly/`
17. Lancrenon, J., Skrobot, M.: On the Provable Security of the Dragonfly Protocol. In: Lopez, J., Mitchell, C.J. (eds.) Information Security – ISC 2015. LNCS, vol. 9290, pp. 244–261. Springer (2015)
18. Lancrenon, J., Skrobot, M., Tang, Q.: Two More Efficient Variants of the J-PAKE Protocol. In: Manulis, M., Sadeghi, A., Schneider, S. (eds.) Applied Cryptography and Network Security – ACNS 2016. LNCS, vol. 9696, pp. 58–76. Springer (2016)
19. MacKenzie, P.: The PAK Suite: Protocols for Password-Authenticated Key Exchange. DIMACS Technical Report 2002-46 (2002)
20. Shoup, V.: On Formal Models for Secure Key Exchange. Cryptology ePrint Archive, Report 1999/012 (1999)
21. Williams, S.C.: On the Security of Key Exchange Protocols. Ph.D. thesis, University of Bristol, UK (2011)

On the Content Security Policy Violations due to the Same-Origin Policy

Dolière Francis Some
Université Côte d'Azur
Inria, France
doliere.some@inria.fr

Nataliia Bielova
Université Côte d'Azur
Inria, France
nataliia.bielova@inria.fr

Tamara Rezk
Université Côte d'Azur
Inria, France
tamara.rezk@inria.fr

ABSTRACT

Modern browsers implement different security policies such as the Content Security Policy (CSP), a mechanism designed to mitigate popular web vulnerabilities, and the Same Origin Policy (SOP), a mechanism that governs interactions between resources of web pages.

In this work, we describe how CSP may be violated due to the SOP when a page contains an embedded iframe from the same origin. We analyse 1 million pages from 10,000 top Alexa sites and report that at least 31.1% of current CSP-enabled pages are potentially vulnerable to CSP violations. Further considering real-world situations where those pages are involved in same-origin nested browsing contexts, we found that in at least 23.5% of the cases, CSP violations are possible.

During our study, we also identified a divergence among browsers implementations in the enforcement of CSP in srcdoc sandboxed iframes, which actually reveals a problem in Gecko-based browsers CSP implementation. To ameliorate the problematic conflicts of the security mechanisms, we discuss measures to avoid CSP violations.

1. INTRODUCTION

Modern browsers implement different specifications to securely fetch and integrate content. One widely used specification to protect content is the Same Origin Policy (SOP)[3]. SOP allows developers to isolate untrusted content from a different origin. An origin here is defined as scheme, host, and port number. If an iframe's content is loaded from a different origin, SOP controls the access to the embedder resources. In particular, no script inside the iframe can access content of the embedder page. However, if the iframe's content is loaded from the same origin as the embedder page, there are no privilege restrictions w.r.t. the embedder resources. In such a case, a script executing inside the iframe can access content of the embedder webpage. Scripts are considered trusted and the *iframe becomes transparent* from a developer view point. A more recent specification to protect content in web pages is the Content Security Policy (CSP)[19]. The primary goal of CSP is to mitigate cross site scripting attacks (XSS), data leaks attacks, and other types of attacks. CSP allows developers to specify, among other features, trusted domain sources from which to fetch content. One of the most important features of CSP, is to allow a web application developer to specify trusted JavaScript sources. This kind of restriction is meant to permit execution of only trusted code and thus prevent untrusted code to access content of the page.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3038914

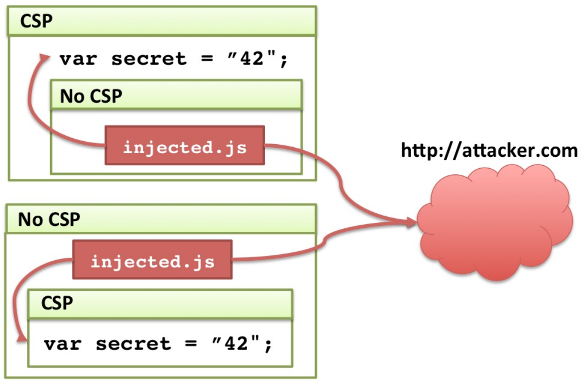

Figure 1: An XSS attack despite CSP.

In this work, we report on a fundamental problem of CSP. CSP[28] defines how to protect content in an isolated page. However, it does not take into consideration the page's context, that is its embedder or embedded iframes. In particular, CSP is unable to protect content of its corresponding page if the page embeds (using the *src* attribute) an iframe of the same origin. The CSP policy of a page will not be applied to an embedded iframe. However, due to SOP, the iframe has complete access to the content of its embedder. Because same origin iframes are transparent due to SOP, this opens loopholes to attackers whenever the CSP policy of an iframe and that of its embedder page are not compatible (see Fig. 1).

We analysed 1 million pages from the top 10,000 Alexa sites and found that 5.29% of sites contain some pages with CSPs (as opposed to 2% of home pages in previous studies[5]). We have identified that in 94% of cases, CSP may be violated in presence of the document.domain API and in 23.5% of cases CSP may be violated without any assumptions (see Table 3).

During our study, we also identified a divergence among browsers implementations in the enforcement of CSP[28] in sandboxed iframes embedded with *srcdoc*. This actually reveals an inconsistency between the CSP and HTML5 sandbox attribute specification for iframes.

We identify and discuss possible solutions from the developer point of view as well as new security specifications that can help prevent this kind of CSP violations. We have made publicly available the dataset that we used for our results in[2]. We have installed an automatic crawler to recover the same dataset every month to repeat the experiment taking into account the time variable. An accompanying technical report with a complete account of our analyses can be found at[18].

In summary, our contributions are: (i) We describe a new class of vulnerabilities that lead to CSP violations. (Section 1). (ii) We perform a large and depth scale crawl of top sites, highlighting CSP adoption at sites-level, as well as sites origins levels. Using this dataset, we report on the possibilities of CSP violations between the SOP and CSP in the wild. (Section 3). (iii) We propose guidelines in the design and deployment of CSP. (Section 3.4). (iv) We reveal an inconsistency between the CSP specification and HTML5 sandbox attribute specification for iframes. Different browsers choose to follow different specifications, and we explain how any of these choices can lead to new vulnerabilities. (Section 5).

2. CONTENT SECURITY POLICY AND SOP

The Content Security Policy (CSP)[19] is a mechanism that allows programmers to control which client-side resources can be loaded and executed by the browser. CSP (version 2) is an official W3C candidate recommendation[28], and is currently supported by major web browsers. CSP is delivered in the `Content-Security-Policy` HTTP response header, or in a `<meta>` element of HTML.

CSP applicability A CSP delivered with a page controls the resources of the page. However it does not apply to the page's embedding resources[28]. As such, CSP does not control the content of an iframe even if the iframe is from the same origin as the main page according to SOP. Instead, the content of the iframe is controlled by the CSP delivered with it, that can be different from the CSP of the main page.

CSP directives CSP allows a programmer to specify which resources are allowed to be loaded and executed in the page. These resources are defined as a set of origins and known as a *source list*. Additionally to controlling resources, CSP allows to specify allowed destinations of the AJAX requests by the `connect-src` directive. A special header `Content-Security-Policy-Report-Only` configures a CSP in a report-only mode: violations are recorded, but not enforced. The directive `default-src` is a special fallback directive that is used when some directive is not defined. The directive `frame-ancestors` (meant to supplant the HTTP `X-Frame-Options` header[28]), controls in which pages the current page may be included as an iframe, to prevent clickjacking attacks[16]. See Table 1 for the most commonly used CSP directives[22].

Source lists CSP source list is traditionally defined as a *whitelist* indicating which domains are trusted to load the content, or to communicate. For example, a CSP from Listing 1 allows to include scripts only from `third.com`, requires to load frames only over HTTPS, while other resource types can only be loaded from the same hosting domain.

```
Content-Security-Policy: default-src 'self
    ';
script-src third.com; child-src https:
```

Listing 1: Example of a CSP policy.

Directive	Controlled content
`script-src`	Scripts
`default-src`	All resources (fallback)
`style-src`	Stylesheets
`img-src`	Images
`font-src`	Fonts
`connect-src`	XMLHttpRequest, WebSocket or EventSource
`object-src`	Plug-in formats (object, embed)
`report-uri`	URL where to report CSP violations
`media-src`	Media (audio, video)
`child-src`	Documents (frames), [Shared] Workers
`frame-ancestors`	Embedding context

Table 1: Most common CSP directives[22].

A whitelist can be composed of concrete hostnames (`third.com`), may include a wildcard `*` to extend the policy to subdomains (`*.third.com`), a special keyword `'self'` for the same hosting domain, or `'none'` to prohibit any resource loading.

Restrictions on scripts Directive `script-src` is the most used feature of CSP in today's web applications[22]. It allows a programmer to control the origin of scripts in his application using source lists. When the `script-src` directive is present in CSP, it blocks the execution of any inline script, JavaScript event handlers and APIs that execute string data code, such as `eval()` and other related APIs. To relax the CSP, by allowing the execution of inline `<script>` and JavaScript event handlers, a `script-src` whitelist should contain a keyword `'unsafe-inline'`. To allow `eval()`-like APIs, the CSP should contain a `'unsafe-eval'` keyword. Because `'unsafe-inline'` allows execution of *any* inlined script, it effectively removes any protection against XSS. Therefore, nonces and hashes were introduced in CSP version 2[28], allowing to control which inline scripts can be loaded and executed.

Sandboxing iframes Directive `sandbox` allows to load resources but execute them in a separate environment. It applies to all the iframes and other content present on the page. An empty `sandbox` value creates completely isolated iframes. One can selectively enable specific features via `allow-*` flags in the directive's value. For example, `allow-scripts` will allow executions of scripts in an iframe, and `allow-same-origin` will allow iframes to be treated as being from their normal origins.

Same-Site and Same-Origin Definitions.

In our terminology, we distinguish the web pages that belong to the same site from the pages that belong to the same origin. By *page* we refer to any HTML document – for example, the content of an iframe we call *iframe page*. In this case, the page that embeds an iframe is called a *parent page* or *embedder*.

By *site* we refer to the highest level domain that we extract from Alexa top 10,000 sites, usually containing the domain name and a TLD, for example `main.com`. All the pages that belong to a site, and to any of its subdomains as `sub.main.com`, are considered *same-site* pages.

According to the Same Origin Policy, an *origin* of a page is scheme, host and port of its URL. For example, in `http:`

`//main.com:81/dir/p.html`, the scheme is "http", the host is "main.com" and the port is 81.

2.1 CSP violations due to SOP

Consider a web application, where the main page `A.html` and its iframe `B.html` are located at `http://main.com`, and therefore belong to the same origin according to the same-origin policy. `A.html`, shown in Listing 2, contains a script and an iframe from `main.com`. The local script `secret.js` contains sensitive information given in Listing 3. To protect against XSS, the developer behind `http://main.com` have installed the CSP for its main page `A.html`, shown in Listing 4.

```html
<html>
  <script src="secret.js"></script>
   ...
  <iframe src="B.html"></iframe>
</html>
```

Listing 2: Source code of `http://main.com/A.html`.

```js
  var secret = "42";
```

Listing 3: Source code of `secret.js`.

```
Content-Security-Policy: default-src 'none
    ';
script-src 'self'; child-src 'self'
```

Listing 4: CSP of `http://main.com/A.html`.

This CSP provides an effective protection against XSS:

2.1.1 Only parent page has CSP

According to the latest version of CSP[1], only the CSP of the iframe applies to its content, and it ignores completely the CSP of the including page. In our case, if there is no CSP in `B.html` then its resource loading is not restricted. As a result, an iframe `B.html` without CSP is potentially vulnerable to XSS, since any injected code may be executed within `B.html` with no restrictions. Assume `B.html` was exploited by an attacker injecting a script `injected.js`. Besides taking control over `B.html`, this attack now propagates to the including page `A.html`, as we show in Fig. 1. The XSS attack extends to the including parent page because of the inconsistency between the CSP and SOP. When a parent page and an iframe are from the same origin according to SOP, a parent and an iframe share the same privileges and can access each other's code and resources.

For our example, `injected.js` is shown in Listing 5.

This script executed in `B.html` retrieves the secret value from its parent page (`parent.secret`) and transmits it to an attacker's server `http://attacker.com` via XMLHttpRequest[2].

```js
function sendData(obj, url){
  var req = new XMLHttpRequest();
  req.open('POST', url, true);
  req.send(JSON.stringify(obj));
}
sendData({secret: parent.secret}, 'http://
    attacker.com/send.php');
```

Listing 5: Source code of `injected.js`.

[1]`https://www.w3.org/TR/CSP2/#which-policy-applies`

[2]The XMLHttpRequest is not forbidden by the SOP for `B.html` because an attacker has activated the Cross-Origin Resource Sharing mechanism[21] on her server `http://attacker.com`.

A straightforward solution to this problem is to ensure that the protection mechanism for the parent page also propagates to the iframes from the same domain. Technically, it means that the CSP of the iframe should be the same or more restrictive than the CSP of the parent. In the next example we show that this requirement does not necessarily prevent possible CSP violations due to SOP.

2.1.2 Only iframe page has CSP

Consider a different web application, where the including parent page `A.html` does not have a CSP, while its iframe `B.html` contains a CSP from Listing 4. In this example, `B.html`, shown in Listing 6 now contains some sensitive information stored in `secret.js` (see Listing 3).

```html
<html>
   ...
  <script src="secret.js"></script>
</html>
```

Listing 6: Source code of `http://main.com/B.html`.

Since the including page `A.html` now has no CSP, it is potentially vulnerable to XSS, and therefore may have a malicious script `injected.js`. The iframe `B.html` has a restrictive CSP, that effectively contributes to protection against XSS. Since `A.html` and `B.html` are from the same origin, the malicious injected script can profit from this and steal sensitive information from `B.html`. For example, the script may call the `sendData` function with the secret information:

```js
sendData({secret: children[0].secret}, '
    http://attacker.com/send.php');
```

Thanks to SOP, the script `injected.js` fetches the secret from it's child iframe `B.html` and sends it to `http://attacker.com`.

2.1.3 CSP violations due to origin relaxation

A page may change its own origin with some limitations. By using the `document.domain` API, the script can change its current domain to a superdomain. As a result, a shorter domain is used for the subsequent origin checks[3].

Consider a slightly modified scenario, where the main page `A.html` from `http://main.com` includes an iframe `B.html` from its sub-domain `http://sub.main.com`. Any script in `B.html` is able to change the origin to `http://main.com` by executing the following line:

```js
document.domain = "main.com";
```

If `A.com` is willing to communicate with this iframe, it should also execute the above-written code so that the communication with `B.html` will be possible. The content of `B.html` is now treated by the web browser as the same-origin content with `A.html`, and therefore any of the previously described attacks become possible.

2.1.4 Categories of CSP violations due to SOP

We distinguish three different cases when the CSP violation might occur because of SOP:

Only parent page or only iframe has CSP A parent page and an iframe page are from the same origin, but only one of them contains a CSP. The CSP may be violated due to the unrestricted access of a page without CSP

[3]`https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Changing_origin`

to the content of the page with CSP. We demonstrated this example in Sections 2.1.1 and 2.1.2.

Parent and iframe have different CSPs A parent page and an iframe page are from the same origin, but they have different CSPs. Due to SOP, the scripts from one page can interfere with the content of another page thus violating the CSP.

CSP violation due to origin relaxation A parent page and an iframe page have the same higher level domain, port and scheme, but however they are not from the same origin. Either CSP is absent in one of them, or they have different CSPs – in both cases CSP may be violated because the pages can relax their origin to the high level domain by using `document.domain` API, as we have shown in Section 2.1.3.

3. EMPIRICAL STUDY OF CSP VIOLATIONS

We have performed a large-scale study on the top 10,000 Alexa sites to detect whether CSP may be violated due to an inconsistency between CSP and SOP. For collecting the data, we have used CasperJS[15] on top of PhantomJS headless browser[8]. The User-Agent HTTP header was instantiated as a recent Google Chrome browser.

3.1 Methodology

The overview of our data collection and CSP comparison process is given in Figure 2. The main difference in our data collection process from previous works on CSP measurements in the wild[22, 5] is that we crawl not only the main pages of each site, but also other pages. First, we collect pages accessible through links of the main page and pointing to the same site. Second, to detect possible CSP violations due to SOP, we have collected all the iframes present on the home pages and linked pages.

3.1.1 Data Collection

We run PhantomJS using as user agent *Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome51.0.2704.63 Safari/537.36*. The study was performed on an internal cluster of 200 cores, using OpenMP to benefit from parallelization.

Home Page Crawler For each site in top 10,000 Alexa list, we crawl the home page, parse its source code and extract three elements: (1) a CSP of the site's home page stored in HTTP header as well as in `<meta>` HTML tag; we denote the CSPs of the home page by $\mathcal{C}$; (2) to extract more pages from the same site, we analyse the source of the links via `<a href=...>` tag and extract URLs that point to the same site, we denote this list by L. (3) we collect URLs of iframes present on the home page via `<iframe src=...>` tag and record only those belonging to the same site, we denote this set by $\mathcal{F}$.

Page Crawler We crawl all the URLs from the list of pages L, and for each page we repeat the process of extraction of CSP and relevant iframes, similar to the steps (1) and (3) of the home page crawler. As a result, we get a set of CSPs of linked pages $\mathcal{C}_L$ and a set of iframes URLs $\mathcal{F}_L$ that we have extracted from the linked pages in L.

Iframe Crawler

For every iframe URL present in the list of home page iframes $\mathcal{F}_H$, and in the list of linked pages iframes $\mathcal{F}_L$, we extract their corresponding CSPs and store in two sets: $\mathcal{C}_F$ for home page iframes and $\mathcal{C}_{LF}$ for linked page iframes.

3.1.2 CSP adoption analysis

Since CSP is considered an effective countermeasure for a number of web attacks, programmers often use it to mitigate such attacks on the main pages of their sites. However, if CSP is not installed on some pages of the same site, this can potentially leak to CSP violations due to the inconsistency with SOP when another page from the same origin is included as an iframe (see Figure 1). In our database, for each site, we recorded its home page, a number of linked pages and iframes from the same site. This allows us to analyse how CSP is adopted at every popular site by checking the presence of CSP on every crawled page and iframe of each site. To do so, we analyse the extracted CSPs: $\mathcal{C}$ for the home page, $\mathcal{C}_L$ for linked pages, $\mathcal{C}_F$ for home page iframes, and $\mathcal{C}_{LF}$ for linked pages iframes.

3.1.3 CSP violations detection

To detect possible CSP violations due to SOP, we have analysed home pages and linked pages from the same site, as well as iframes embedded into them.

CSP Selection

To detect CSP violations, we first remove all the sites where no parent page and no iframe page contains a CSP. For the remaining sites, we pointwise compare (1) the CSPs of the home pages $\mathcal{C}$ and CSPs of iframes present on these pages $\mathcal{C}_F$; (2) the CSPs of the linked pages $\mathcal{C}_L$ and CSPs of their iframes $\mathcal{C}_{LF}$. To check whether a parent page CSP and an iframe CSP are equivalent, we have applied the CSP comparison algorithm (Figure 2)

CSP Preprocessing We first normalise each CSP policy, by splitting it into its directives.

- If **default-src** directive is present (**default-src** is a fallback for most of the other directives), then we extract the source list s of **default-src**. We analyse which directives are missing in the CSP, and explicitly add them with the source list s.
- If **default-src** directive is absent, we extract missing directives from the CSP. In this case, there are no restrictions in CSP for every absent directive. We therefore explicitly add them with the most permissive source list. A missing **script-src** is assigned *** 'unsafe-inline' 'unsafe-eval'** as the most permissive source list [28].
- In each source list, we modify the special keywords: (i) **'self'** is replaced with the origin of the page containing the CSP; (ii) in case of **'unsafe-inline'** with hash or nonce, we remove **'unsafe-inline'** from the directive since it will be ignored by the CSP2. (iii) **'none'** keywords are removed from all the directives; (iv) nonces and hashes are removed from all the directives since they cannot be compared; (iv) each whitelisted domain is extended with a list of schemes and port numbers from the URL of the page includes the CSP[4].

[4]For example, according to CSP2, if the page scheme is `https`, and a CSP contains a source `example.com`, then the user agent should allow content only from `https://example.com`, while if the current scheme is `http`, it would allow both `http://example.com` and `https://example.com`.

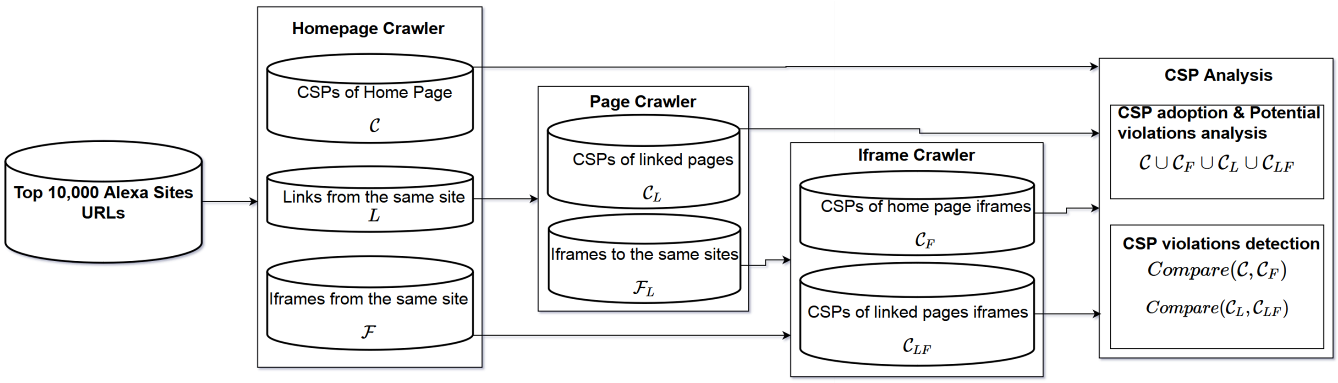
Top 10,000 Alexa Sites URLs
Homepage Crawler
CSPs of Home Page
$\mathcal{C}$
Links from the same site
L
Iframes from the same site
$\mathcal{F}$
Page Crawler
CSPs of linked pages
$\mathcal{C}_L$
Iframes to the same sites
$\mathcal{F}_L$
Iframe Crawler
CSPs of home page iframes
$\mathcal{C}_F$
CSPs of linked pages iframes
$\mathcal{C}_{LF}$
CSP Analysis
CSP adoption & Potential violations analysis
$\mathcal{C} \cup \mathcal{C}_F \cup \mathcal{C}_L \cup \mathcal{C}_{LF}$
CSP violations detection
$Compare(\mathcal{C}, \mathcal{C}_F)$
$Compare(\mathcal{C}_L, \mathcal{C}_{LF})$

Figure 2: Data Collection and Analysis Process

Sites successfully crawled	9,885
Pages visited	1,090,226
Pages with iframe(s) from the same site	648,324
Pages with same-origin iframe(s)	92,430
Pages with same-origin iframe(s) where page and/or iframe has CSP	692
Pages with CSP	21,961 (2.00%)
Sites with CSP on home page	228 (2.3%)
Sites with CSP on some pages	523 (5.29%)

Table 2: Crawling statistics

CSP Comparison We compare all the directives present in the two CSPs to identify whether the two policies require the same restrictions. Whenever the two CSPs are different, our algorithm returns the names of directives that do not match. The demonstration of the comparison is accessible on[2]. For each directive in the policies we compare the source lists and the algorithm proceeds if the elements of the lists are identical in the normalised CSPs.

3.1.4 Limitations

Our methodology and results have two(2) limitations that we explain here.

User interactions The automatic crawling process did not include any real-user-like interactions with top sites. As such the set of iframes and links URLs we have analysed is an underestimate of all links and iframes a site may contain.

Pairs of (parent-iframe) In this study, we consider CSP violations in same origin (parent, iframe) couples only. Their are though further combinations such as couples of sibling iframes in a parent page that we could have considered. Overall, our results are conservative, since the problem might have been worst without those limitations.

3.2 Results on CSP Adoption

The crawling of Alexa top 10,000 sites was performed in the end of August, 2016. To extract several pages from the same site, we have also crawled all the links and iframes on a page that point to the same site. In total, we have gathered 1,090,226 from 9,885 different sites. On median, from each site we extracted 45 pages, with a maximum number of 9,055 pages found on `tuberel.com`. Our crawling statistics is presented in Table 2. More than half of the pages contain an iframe, and 13% of pages do contain an iframe from the same site. This indicates the potential surface for

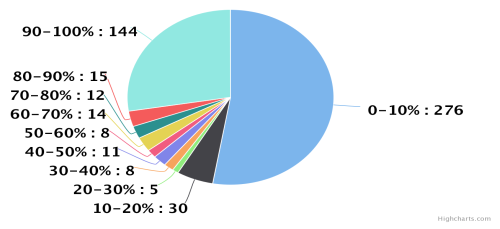

Figure 3: Percentage of pages with CSP per site

the CSP violations, when at least one page on the site has a CSP installed. We discuss such potential CSP violation in details in Section 3.3.3. Similarly to previous works on CSP adoption[22, 5], we have found that CSP is present on only 228 out of 9,885 home pages (2.31%). While extending this analysis to almost a million pages, we have found a similar rate of CSP adoption (2.00%).

Differently from previous studies that anlaysed only home pages, or only pages in separation, we have analysed how many sites have at least some pages that adopted CSP. We have grouped all pages by sites, and found that 5.29% of sites contain some pages with CSPs. It means that CSP is more known by the website developers, but for some reason is not widely adopted on all the pages of the site.

We have then analysed how many pages on each site have adopted CSPs. For each of 523 sites, we have counted how many pages (including home page, linked pages and iframes) have CSPs. Figure 3 shows that more than half of the sites have a very low CSP adoption on their pages: on 276 sites out of 529, CSP is installed on only 0-10% of their pages. This becomes problematic if other pages without CSP are not XSS-free. However, it is interesting to note that around a quarter of sites do profit from CSP by installing it on 90-100% of their pages.

3.3 Results on CSP violations due to SOP

As described in Section 2.1.4, we distinguish several categories of CSP violations when a parent page and an iframe on this page are from the same origin according to SOP. To account for possible CSP violations, we only consider cases when either parent, or iframe, or both have a CSP installed. From all the 21,961 pages that have CSP installed, we have removed the pages, where CSPs are in report-only mode, having left 18,035 pages with CSPs in enforcement mode.

Table 3 presents possible CSP violations due to SOP.

	Same-origin parent-iframe	Possible to relax origin	Total
Only parent page has CSP	**83**	**1388**	1471
Only iframe has CSP	**16**	**240**	256
Different CSPs in parent page and iframe	**70**	**44**	114
No CSP violations	551	109	660
CSP violations total	**169 (23.5%)**	**1672 (94%)**	1841

Table 3: Statistics CSP violations due to Same-Origin Policy

	Same-origin parent-iframe	Possible to relax origin
Only parent page CSP	yandex.ru	twitter.com, yandex.ru, mail.ru
Only iframe CSP	amazon.com, imdb.com	–*
Different CSP	twitter.com	–*

*Not found in top 100 Alexa sites.

Table 4: Sample of sites with CSP violations due to Same-Origin Policy

We have extracted the parent-iframe couples that might cause a CSP violation because either (1) only parent or only iframe installed a CSP, or (2) both installed different CSPs. First, to account for direct violations because of SOP, we distinguish couples where parent and iframe are from the same origin (columns 2,3), we have found 720 cases of such couples. Second, we analyse possible CSP violations due to origin relaxation: we have collected 1781 couples that are from different origins but their origins can be relaxed by `document.domain` API (see more in Section 2.1.3) – these results are shown in column 3.

In Table 4 we present the names of the domains out of top 100 Alexa sites, where we have found different CSP violations. Each company in this table have been notified about the possible CSP violation. Concrete examples of the page and iframe URLs and their corresponding CSPs for each such violation can be found in the corresponding technical report[18]. All the collected data is available online[2].

CSP violations in presence of `document.domain` According to our results, in presence of `document.domain`, 94% of (parent, iframe) pages can have their CSP violated. Those violations can occur only if both parent and iframes pages execute `document.domain` to the same top level domain. Thus, our result is an over-approximation, assuming that `document.domain` is used in all of those pages and iframes. According to[1], `document.domain` is used in less than 3% of web pages.

3.3.1 *Only parent page or only iframe has CSP*

We first consider a scenario when a parent page and an iframe are from the same origin, but only one of them contains a CSP. Intuitively, if only a parent page has CSP, then an iframe can violate CSP by executing any code and accessing the parent page's DOM, inserting content, access cookies etc. Among 720 parent-iframe couples from *the same origin*, we have found 83 cases (11.5%) when only parent has a CSP, and 16 cases (2.2%) when only iframe has a CSP. These CSP violations originate from 13 (for parent) and 4 (for iframe) sites. For example, such possible violations are found on some pages of amazon.com, yandex.ru and imdb.com (see Table 4). CSP of a parent or iframe may also be violated because of *origin relaxation*. We have identified 1388 cases (78%) of parent-iframe couples where such violation may occur because CSP is present only in the parent page. This was observed on 20 different sites, including twitter.com, yandex.ru and others. Finally, in 240 cases (13.5%) only iframe has CSP installed, which was found on 11 different sites. We manually checked the parent and iframes involved in CSP violations for sites in Table 4. In all of those sites, either the parent or the iframe page is a login page[2]. We furthermore checked how effective are the CSP of those pages, using CSPEvaluator[5], proposed by Lukas et al.[22]. and found out that the CSP policies involved in these are moreover all bypassable.

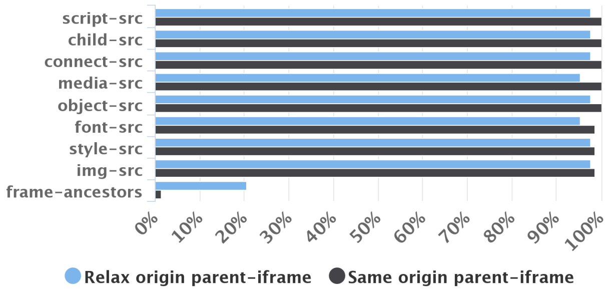

Figure 4: Differences in CSP directives for parent and iframe pages

3.3.2 *Parent and iframe have different CSPs*

In a case when a page and iframe are from the same origin, but their corresponding CSPs are different, may also cause a violation of CSP. From the 720 *same-origin* parent-iframe couples, we have found 70 cases (9.7%) (from 3 sites) when their CSPs differ, and for *an origin relaxation* (from 6 sites) case, we have identified only 44 such cases (2.5%). This setting was found on some pages of twitter.com for instance.

We have further analysed the differences in CSPs found on parent and iframe pages. For all the 114 pairs of parent-iframe (either same-origin or possible origin relaxation), we have compared CSPs they installed, directive-by-directive. Figure 4 shows that every parent CSP and iframe CSP differ on almost every directive – between 90% and 100%. The only exception is **frame-ancestors** directive, which is almost the same in different parent pages and iframes. If properly set, this directive gives a strong protection against clickjacking attacks, therefore all the pages of the same origin are equally protected.

[5]https://csp-evaluator.withgoogle.com/

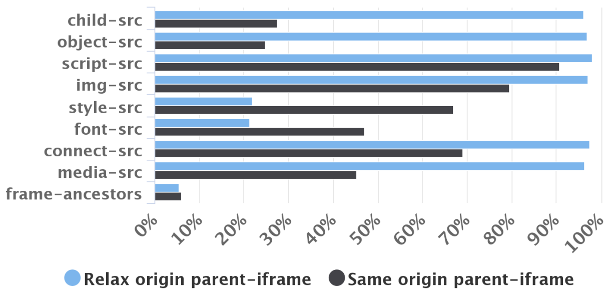

Figure 5: Differences in CSP directives for same-origin and relaxed origin pages

3.3.3 Potential CSP violations

A *potential CSP violation* may happen when in a site, either some pages have CSP and some others do not, or pages have different CSP. When those pages get nested as parent-iframe, we can run into CSP violations, just like in the direct CSP violations cases we have just reported above. To analyse how often such violations may occur, we have analysed the 18,035 pages that have CSP in enforcement mode. These pages originate from 729 different origins spread over 442 sites. Table 5 shows that 72% of CSPs (12,899 pages) can be potentially violated, and these CSPs originate from pages of 379 different sites (85.75%). To detect these violations, for each page with a CSP in our database, we have analysed whether there exists another page from the same origin, that does not have CSP. This page could embed the page with CSP and violate it because of SOP. We have detected 4381 such pages (24%) from 197 origins. Similarly, we detected 1223 pages (7%) when there are same-origin pages with a different CSP. Similarly, we have analysed when potential CSP violations may happen due to origin relaxation. We have detected 4728 pages (26%), whose CSP may be violated because of other pages with no CSP, and 2567 pages (14%), whose CSP may be violated because of different CSP on other relaxed-origin pages.

For the pages that have different CSPs, we have compared how much CSPs differ. Figure 5 shows that CSPs mostly differ in `script-src` directive, which protects pages from XSS attacks. This means, that if one page in the origin does whitelist an attacker's domain or an insecure endpoints [22], all the other pages in the same origin become vulnerable because they may be inserted as an iframe to the vulnerable page and their CSPs can be easily violated.

3.4 Responses of websites owners

We have reported those issues to a sample of sites owners, using either HackerOne[6], or contact forms when available. Here are some selected quotes from our discussions with them.

> "*Yes, of course we understand the risk that under some circumstances XSS on one domain can be used to bypass CSP on another domain, but it's simply impossible to implement CSP across all (few hundreds) domains at once on the same level. We are implementing strongest CSP currently possible for different pages on different domains and keep going with this process to protect all pages, after that we will strengthen the CSP. We believe it's better to have stronger CSP policy where possible rather than have same weak CSP on all pages or not having CSP at all. Having in mind there are hundreds of domains within mail.ru, at least few years are required before all pages on all domains can have strong CSP.*" – `Mail.ru`

> "*[...]the sandbox is a defense in depth mitigation[...]. We definitely don't allow relaxing document.domain on www.dropbox.com[...]*" – `Dropbox.com`

> "*While this is an interesting area of research, are you able to demonstrate that this behavior is currently exploitable on Twitter? It appears that the behavior you have described can increase the severity of other vulnerabilities but does not pose a security risk by itself. Is our understanding correct? [...]We consider this to be more of a defensive in depth and will take into account with our continual effort to improve our CSP policy*" – `Twitter.com`

> "*I believe we understand the risk as you've described it.*" – `Imdb.com`

[6]https://hackerone.com

4. AVOIDING CSP VIOLATIONS

Preventing CSP violations due to SOP can be achieved by having the **same** effective CSP for **all** same-origin pages in a site, and prevent origin relaxation.

Origin-wide CSP: Using CSP for all same-origin pages can be manually done but this solution is error-prone. A more effective solution is the use of a specification such as Origin Policy[27] in order to set a header for the whole origin.

Preventing Origin Relaxation: Having an origin-wide CSP is not enough to prevent CSP violations. By using origin relaxation, pages from different origins can bypass the SOP[17]. Many authors provide guidelines on how to design an effective CSP[22]. Nonetheless, even with an effective CSP, an embedded page from a different origin in the same site can use `document.domain` to relax its origin. Preventing origin relaxation is trickier.

Programmatically, one could prevent other scripts from modifying `document.domain` by making a script run first in a page[20]. The first script that runs on the page would be:

```
Object.defineProperty(document, "domain",
    { __proto__: null, writable: false,
    configurable: false});
```

A parent page can also indirectly disable origin relaxation in iframes by sandboxing them. This can be achieved by using **sandbox** as an attribute for iframes or as directive for the parent page CSP. Unfortunately, an iframe cannot indirectly disable origin relaxation in the page that embeds it. However, the **frame-ancestors** directive of CSP gives an iframe control over the hosts that can embed it. Finally, a more robust solution is the use of a policy to deprecate `document.domain` as proposed in the draft of Feature policy[29]. The feature policy defines a mechanism that allows developers to selectively enable and disable the use of various browser features and APIs.

Iframe sandboxing: Combining attribute **allow-scripts** and **allow-same-origin** as values for **sandbox** successfully

	Pages	Origins	Sites
A same origin page has no CSP	4381	197	197
A same origin page has a different CSP	1223	23	23
Total Potential violations due to same origin pages	**5604 (31.1%)**	**-**	**-**
A same origin (after relaxation) page has no CSP	4728	340	183
A same origin (after relaxation) has a different CSP	2567	135	44
Total Potential violations due to same origin (after relaxation	**7295(40.4%)**	**-**	**-**
Potential violations total	12899 (72%)	591 (81%)	379 (52%)

Table 5: Potential CSP violations in pages with CSP

disables `document.domain` in an iframe[7]. We recommend the use of **sandbox** as a CSP directive, instead of an HTML iframe attribute. The first reason is that **sandbox** as a CSP directive, automatically applies to all iframes that are in a page, avoiding the need to manually modify all HTML iframe tags. Second, the **sandbox** directive is not programmatically accessible to potentially malicious scripts in the page, as is the case for the **sandbox** attribute (which can be removed from an iframe programmatically, replacing the sandboxed iframe with another identical iframe but without the **sandbox** attribute).

Limitations An origin-wide CSP (the same CSP for all same origin pages) can become very liberal if all same origin pages do not require the same restrictions. In order to implement the solution we propose, one needs to consider the intended relation between a parent page and an iframe page, in presence of CSP. In the case where the two(2) pages should be allowed direct access to each other content, then, since same origin pages can bypass page-specific security characteristics [9], the solution is to have the same CSP for both the page and the iframe. However, if direct access to each other content is not a required feature, one can keep different CSPs in parent and iframe, or have no CSP at all in one of the parties, but their contents should be isolated from each other. The solution here is to use sandboxing. Nonetheless, there are other means (such as `postMessage`) by which one can securely achieve communication between the pages.

5. INCONSISTENT IMPLEMENTATIONS

Combining origin-wide CSP with **allow-scripts sandbox** directive would have been sufficient at preventing the inconsistencies between CSP and the same origin policy. Unfortunately, we have discovered that for some browsers, this solution is not sufficient. Starting from HTML5, major browsers, apart from Internet Explorer, supports the new **srcdoc** attribute for iframes. Instead of providing a URL which content will be loaded in an iframe, one provides directly the HTML content of the iframe in the **srcdoc** attribute. According to CSP2 [28], §5.2, the CSP of a page should apply to an iframe which content is supplied in a **srcdoc** attribute. This is actually the case for all majors browsers, which support the **srcdoc** attribute. However, there is a problem when the **sandbox** attribute is set to an **srcdoc** iframe.

[7]We found out that `dropbox.com` actually puts **sandbox** attribute for all its iframes, and therefore avoids the possible CSP violations. We have had a very interesting discussion on `Hackerone.com` with Devdatta Akhawe, a Security Engineer at Dropbox, who told us more about their security practices regarding CSP in particular.

Webkit-based[8] and **Blink**-based[9] browsers (Chrome, Chromium, Opera) always comply with CSP. The CSP of a page will apply to all **srcdoc** iframes, even in those iframes which have a different origin than that of the page, because they are sandboxed without **allow-same-origin** .

In contrast, we noticed that in Gecko-based browsers (Mozilla Firefox), the CSP of the page applies to that of the srcdoc iframe if and only if allow-same-origin is present as value for the attribute. Otherwise it does not apply. The problem with this choice is the following. A third party script, whitelisted by the CSP of the page, can create a **srcdoc** iframe, sandboxing it with **allow-scripts** only, and load any resource that would normally be blocked by the CSP of the page if applied in this iframe. This way, the third party script successfully bypasses the restrictions of the CSP of the page. Even though loading additional scripts is considered harmless in the upcoming version 3 [26, 22] of CSP, this specification says nothing about violations that could occur due to the loading of other resources inside a **srcdoc** sandboxed iframe, like resources whitelisted by **object-src** directive for instance, additional iframes etc.

We have notified the W3C, and the Mozilla Security Group. Daniel Veditz, a lead at Mozilla Security Group, recognizes this as a bug and explains:

> "*Our internal model only inherits CSP into same-origin frames (because in theory you're otherwise leaking info across origin boundaries) and iframe sandbox creates a unique origin. Obviously we need to make an exception here (I think we manage to do the same thing for src=data: sandboxed frames).*"

CSP specification and srcdoc iframes The problem of imposing a CSP to an unknown page is illustrated by the following example[25]. If a trusted third party library, whitelisted by the CSP of the page, uses security libraries inside an isolated context (by sandboxing them in a **srcdoc** iframe, setting **allow-scripts** as sole value for the **sandbox**) then, the page's CSP will block the security libraries and possibly introduce new vulnerabilities. Because of this, it was unclear to us the intent of CSP designers regarding srcdoc iframes. Mike West, one of the CSP editors at the W3C and also Developer Advocate in Google Chrome's team, clarified this to us:

> "*I think your objection rests on the notion of the same-origin policy preventing the top-level document from reaching into it's sandboxed child. That seems accurate, but it neglects the bigger*

[8]https://en.wikipedia.org/wiki/WebKit
[9]https://en.wikipedia.org/wiki/Blink_(web_engine)

*picture: **srcdoc** documents are produced entirely from the top-level document context. Since those kinds of documents are not delivered over the network, they don't have the opportunity to deliver headers which might configure their settings. We impose the parent's policy in these cases, because for all intents and purposes, the **srcdoc** document* is *the parent document.*"

6. RELATED WORK

CSP has been proposed by Stamm et al.[19] as a refinement of SOP[3], in order to help mitigate Cross-Site-Scripting[30] and data exfiltration attacks. The second version[28] of the specification is supported by all major browsers, and the third version [26] is under active development. Even though CSP is well supported [5], its endorsement by web sites is rather slow. Weissbacher et al.[24] performed the first large scale study of CSP deployment in top Alexa sites, and found that around 1% of sites were using CSP at the time. A more recent study by Calzavara et al.[5], show that nearly 8% of Alexa top sites now have CSP deployed in their front pages. Another recent study, by Weichselbaum et al.[22] come with similar results to the study of Weissbacher et al.[24]. Our work extends previous results by analysing the adoption of CSP by site not only considering front pages but all the pages in a site. Almost all authors agree that CSP adoption is not a straightforward task, and lots of (manual) effort are needed in order to reorganize and modify web pages to support CSP.

Therefore, in order to help web sites developers in adopting CSP, Javed proposed CSP Aider, [10] that automatically crawl a set of pages from a site and propose a site-wide CSP. Patil and Frederik[14] proposed UserCSP, a framework that monitors the browser internal events in order to automatically infer a CSP for a web page based on the loaded resources. Pan et al.[13] propose CSPAutoGen, to enforce CSP in real-time on web pages, by rewriting them on the fly client-side. Weissbacher et al.[24] have evaluated the feasibility of using CSP in report-only mode in order to generate a CSP based on reported violations, or semi-automatically inferring a CSP policy based on the resources that are loaded in web pages. They concluded that automatically generating a CSP is ineffective. A difficulty which remains is the use of inline scripts in many pages. The first solution is to externalize inline scripts, as can be done by systems like deDacota[6]. Kerschbaumer et al.[12] find that too many pages are still using **'unsafe-inline'** in their CSPs. They propose a system to automatically identify legitimate inline scripts in a page, thereby whitelisting them in the CSP of the underlying page, using script hashes.

Another direction of research on CSP, has been evaluating its effectiveness at successfully preventing content injection attacks. Calzavara et al.[5] found out that many CSP policies in real web sites have errors including typos, ill-formed or harsh policies. Even when the policies are well formed, they have found that almost all currently deployed CSP policies are bypassable because of a misunderstanding of the CSP language itself. Patil and Frederik found similar errors in their study[14]. Hausknecht et al.[7] found that some browser extensions, modified the CSP policy headers, in order to whitelist more resources and origins. Van Acker et al.[4] have shown that CSP fails at preventing data exfiltration specially when resources are prefetched, or in presence of a CSP policy in the HTML meta tag, because the order in which resources are loaded in a web application is hard to predict. Johns[11] proposed hashes for static scripts, and PreparedJS, an extension for CSP, in order to securely handle server-side dynamically generated scripts based on user input. Weichselbaum et al.[22] have extended nonces and hashes, introduced in CSP level 2[28], to remote scripts URLs, specially to tackle the high prevalence of insecure hosts in current CSP policies. Furthermore, they have introduced **strict-dynamic**. This new keyword states that any additional script loaded by a whitelisted remote script URL is considered a trusted script as well. They also provide guidelines on how to build an effective CSP. Jackson and Barth[9] have shown that same origin pages can bypass page-specific policies, like CSP. Though, their work predates CSP. To the best of our knowledge, we are the first to explore the interactions between CSP and SOP and report possible CSP violations.

7. CONCLUSIONS

In this work, we have revealed a new problem that can lead to violations of CSP. We have performed an in-depth analysis of the inconsistency that arises due to CSP and SOP and identified three cases when *CSP may be violated.*

To evaluate how often such violations happen, we performed a large-scale analysis of more than 1 million pages from 10,000 Alexa top sites. We have found that 5.29% of sites contain pages with CSPs (as opposed to 2% of home pages in previous studies).

We have also found out that 72% of current web pages with CSP, are potentially vulnerable to CSP violations. This concerns 379 (72.46%) sites that deploy CSP. Further analysing the contexts in which those web pages are used, our results show that when a parent page includes an iframe from the same origin according to SOP, in 23.5% of cases their CSPs may be violated. And in the cases where `document.domain` is required in both parent and iframes, we identified that such violations may occur in 94% of the cases.

We discussed measures to avoid CSP violations in web applications by installing an origin-wide CSP and using sandboxed iframes. Finally, our study reveals an inconsistency in browsers implementation of CSP for `srcdoc` iframes, that appeared to be a bug in Mozilla Firefox browsers.

Acknowledgments

The authors would like to thank the WebAppSec W3C Working Group for useful pointers to related resources at the early stage of this work, Mike West for very insightful discussions that considerably helped improve this work, Devdatta Akhawe for discussing some security practices at *Dropbox*, and anonymous reviewers and Stefano Calzavara for their valuable comments and suggestions.

8. REFERENCES

[1] Chrome Platform Status. `https://www.chromestatus.com/metrics/feature/popularity#DocumentSetDomain`.

[2] CSP violations online. `https://webstats.inria.fr?cspviolations`.

[3] Same Origin Policy. `https://www.w3.org/Security/wiki/Same_Origin_Policy`.

[4] S. V. Acker, D. Hausknecht, and A. Sabelfeld. Data Exfiltration in the Face of CSP. In X. Chen, X. Wang, and X. Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 853–864. ACM, 2016.

[5] S. Calzavara, A. Rabitti, and M. Bugliesi. Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild. In Weippl et al. [23], pages 1365–1375.

[6] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. deDacota: toward preventing server-side XSS via automatic code and data separation. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1205–1216. ACM, 2013.

[7] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I? - Content Security Policy Endorsement for Browser Extensions. In M. Almgren, V. Gulisano, and F. Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *Lecture Notes in Computer Science*, pages 261–281. Springer, 2015.

[8] A. Hidayat. PhantomJS Headless Browser, 2010-2016.

[9] C. Jackson and A. Barth. Beware of Finer-Grained Origins. In *Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[10] A. Javed. CSP Aider: An Automated Recommendation of Content Security Policy for Web Applications. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP'12)*, 2012.

[11] M. Johns. PreparedJS: Secure Script-Templates for JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings*, pages 102–121, 2013.

[12] C. Kerschbaumer, S. Stamm, and S. Brunthaler. Injecting CSP for Fun and Security. In O. Camp, S. Furnell, and P. Mori, editors, *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016), Rome, Italy, February 19-21, 2016.*, pages 15–25. SciTePress, 2016.

[13] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites. In Weippl et al. [23], pages 653–665.

[14] K. Patil and B. Frederik. A Measurement Study of the Content Security Policy on Real-World Applications. *I. J. Network Security*, 18(2):383–392, 2016.

[15] N. Perriault. CasperJS navigation and scripting tool for PhantomJS, 2011-2016.

[16] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[17] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 463–478, 2010.

[18] D. F. Some, N. Bielova, and T. Rezk. On the Content Security Policy violations due to the Same-Origin Policy. Technical report. `http://www-sop.inria.fr/members/Nataliia.Bielova/papers/CSP-SOP.pdf`.

[19] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 921–930. ACM, 2010.

[20] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman. Gradual typing embedded securely in JavaScript. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 425–438. ACM, 2014.

[21] A. van Kesteren. Cross Origin Resource Sharing. W3C Recommendation, 2014.

[22] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In Weippl et al. [23], pages 1376–1387.

[23] E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* ACM, 2016.

[24] M. Weissbacher, T. Lauinger, and W. K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 212–233, 2014.

[25] M. West. Content Security Policy: Embedded Enforcement, 2016.

[26] M. West. Content Security Policy Level 3. W3C Working Draft, 2016.

[27] M. West. Origin Policy. A Collection of Interesting Ideas, 2016.

[28] M. West, A. Barth, and D. Veditz. Content Security Policy Level 2. W3C Candidate Recommendation, 2015.

[29] M. West and I. Grigorik. Feature Policy. W3C Draft Community Group Report, 2016.

[30] I. Yusof and A. K. Pathan. Mitigating Cross-Site Scripting Attacks with a Content Security Policy. *IEEE Computer*, 49(3):56–63, 2016.

Securing Concurrent Lazy Programs
(Extended Abstract, Work-In-Progress)

Marco Vassena[1], Joachim Breitner[2], and Alejandro Russo[1]

[1] Chalmers University of Technology
{vassena,russo}@chalmers.se
[2] University of Pennsylvania
joachim@cis.upenn.edu

Information-Flow Control [11] (IFC) scrutinizes source code to track how data of different sensitivity levels (e.g., public or sensitive) flows within a program, and raises alarms when confidentiality might be at stake. There are several special-purpose compilers and interpreters which apply this technology: *Jif* [7] (based on Java), *FlowCaml* [9] (based on Caml and not developed anymore), *Paragon* [1] (based on Java), and *JS-Flow* [4] (based on JavaScript). Rather than writing compilers/interpreters, IFC can also be provided as a library in the functional programming language Haskell [5].

Haskell's type system enforces a disciplined separation of side-effect free code from side-effectful code, which makes it possible to introduce input and output (I/O) to the language without compromising on its *purity*. Computations performing side-effects are encoded as values of abstract types which have the structure of monads [6]. This distinctive feature of Haskell is exploited by state-of-the-art IFC libraries (e.g., **LIO** [15] and **MAC** [10]) to identify and restrict "leaky" side-effects without requiring changes to the language or runtime.

Another distinctive feature of Haskell is its *lazy evaluation* strategy. The evaluation is non-strict, as function arguments are not evaluated until required by the function, and it performs *sharing*, as the values of such arguments are stored for subsequent uses. In contrast, eager or *strict evaluation* reduces function arguments to its denoted values before executing the function's body.

There are many arguments for choosing between lazy and eager evaluation. From a security point of view it is unclear which evaluation strategy is more suitable to preserve secrets. To start addressing this subtlety, we need to consider the interaction between evaluation strategies and covert channels.

Sabelfeld and Sands [12] suggest that lazy evaluation might be intrinsically safer than eager evaluation for leaks produced by termination, as lazy evaluation could skip the execution of *unneeded* non-terminating computations that might involve secrets. In multi-threaded systems, where termination leaks are harmful [14], a lazy evaluation strategy seems to be the appropriate choice.

A lazy attack Unfortunately, although lazy evaluation could "save the day" when it comes to termination leaks, it is also vulnerable to leaks via another covert channel due to sharing. Recently [3], Buiras and Russo showed an attack against the **LIO** library [14] where lazy evaluation is exploited to leak information via the *internal timing covert channel* [13]. This covert channel manifests by the mere presence of concurrency and shared resources. It gets exploited by setting up threads to race for a public shared

```
let ℓ = [1 .. 10000000]
    r = sum ℓ
in do forkLIO    -- Secret thread
        (do s ← unlabel secret
            when (s ≡ 1 ∧ r ⩾ 10) return ())
      no_ops; no_ops
        -- Public threads
      forkLIO (do sendPublicMsg (r − r))
      forkLIO (do no_ops; sendPublicMsg 1)
```

Fig. 1: Lazy evaluation attack

resource in such a way that the secret value affects their timing and hence the winner of the race. **LIO** removes such leaks for those public shared-resources which can be identified by the library, such as references and file descriptors. Due to lazy evaluation, variables introduced by **let**-bindings and function applications—which are beyond **LIO**'s control[3]—become shared resources and their evaluation affects the threads' timing behavior.

Figure 1 shows the attack. In **LIO**, every thread has a current label which serves a role similar to the *program counter* in traditional IFC systems [16]. The first thread inspects a secret value ($s \leftarrow \mathit{unlabel\ secret}$), which sets the current label to secret. We refer to threads with such current label as *secret threads*. The other spawned threads have their current label set to public, therefore we call them *public threads*. Observe that the variable r hosts an expression that is somewhat expensive to calculate, as it first builds a list with ten million numbers ($\ell = [1\,..\,10000000]$) before summing up its elements ($r = \mathit{sum}\ \ell$). Importantly, it is referenced by both the secret and the public threads. Note that every thread is secure in isolation—the secret thread always returns $()$ and the public threads read no secret. Assume that the expression $\mathit{no\_ops}$ is some irrelevant computation that takes slightly longer than half the time it takes to sum up the ten million numbers. Then the public threads race to send a message on a shared-public channel using the function $\mathit{sendPublicMsg}$:

▷ If $s \equiv 1$ then the secret has by now evaluated the expression referenced by r, in order to check if $r \geqslant 10$ holds. Due to sharing, the first public thread will not have to re-calculate r and can output 0 almost imediately, while the other public thread is still occupied with $\mathit{no\_ops}$.

▷ If $s \equiv 0$ then the secret thread did not touch r. While the first public now has to evaluate r the second public thread has enough time to perform $\mathit{no\_ops}$ and output 1 first.

[3] As a shallow EDSL, **LIO** reuses much of the host language features to provide security (e.g., type-system and variable bindings). This design choice makes the code base small at the price of not fully controlling the features provided by the host language.

Thus, the last message on the public channel reveals the secret s. This attack can be magnified to a point where whole secrets are leaked systematically and efficiently [14]. Similar to **LIO**, other state-of-the-art concurrent IFC Haskell libraries [2, 10] suffer from this attack.

A naïve fix is to force the variable r to be fully evaluated before any public threads begin their execution. This works, but it defeats a main purpose of lazy evaluation, namely to avoid evaluating unneeded expressions. Furthermore, it is not always possible to evaluate expressions to their denoted value. Haskell programmers like to work with infinite structures, even though only finite approximation of them are actually used by programs. For example, if variable ℓ in Figure 1 were the list $[1 \mathbin{/} n \mid n \leftarrow [1\,..]]$ of reciprocals of all natural numbers and r the sum of those bigger than one millionth ($r = \mathit{sum}\ (\mathit{takeWhile}\ (\geqslant 1\mathrm{e}{-6})\ \ell)$). The evaluation of r uses only a finite portion of ℓ, so the modified program still terminates. But naïvely forcing ℓ to normal form would hang the program. This demonstrates that simply forcing evaluation as a security measure is unsatisfying, as it can introduce divergence and thus change the meaning of a program.

Contributions Instead, we present a novel approach to explicitly control sharing at the language level. We design a new primitive called $\mathit{lazyDup}$ which *lazily* duplicates unevaluated expressions. The attack in Figure 1 can then be neutralized by replacing r with $\mathit{lazyDup}\ r$ in the secret thread, which will then evaluate its own copy of r, without affecting the public threads. We present this primitive in the context of the security library **MAC** [10], which statically enforces IFC in Haskell. By injecting $\mathit{lazyDup}$ when spawning secret threads, we demonstrate that internal timing leaks via lazy evaluation are closed. Primitive $\mathit{lazyDup}$ is not only capable to secure **MAC** against lazy leaks, but also a wide range of other security Haskell libraries (e.g., **LIO** and **HLIO**). We are optimistic that a prototype implementation of $\mathit{lazyDup}$ is feasible without compiler or runtime modifications to the Glasgow Haskell Compiler[4]. To the best of our knowledge, we are the first ones to formally address the problem of internal timing leaks via lazy evaluation. We prove that well-typed programs satisfies progress-sensitive non-interference (PSNI) for a wide-range of deterministic schedulers. Our security are supported by mechanized proofs in the Agda proof assistant [8] and are parametric on the chosen (deterministic) scheduler[5]. As a by-prodcut of interest for the programming language community, we provide—to the best of our knowledge—the first operational semantics for lazy evaluation with mutable reference.

References

1. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: APLAS. LNCS, vol. 8301, pp. 217–232. Springer (2013)
2. Buiras, P., Vytiniotis, D., Russo, A.: HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In: ACM SIGPLAN International Conference on Functional Programming. ACM (2015)

[4] `https://www.haskell.org/ghc/`

[5] Available at `https://github.com/marco-vassena/lazy-mac`

3. Buiras, P., Russo, A.: Lazy programs leak secrets. In: Nordic Conference in Secure IT Systems. Springer-Verlag (2013)
4. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: ACM Symposium on Applied Computing. ACM (2014)
5. Li, P., Zdancewic, S.: Encoding information flow in Haskell. In: IEEE Workshop on Computer Security Foundations. IEEE Computer Society (2006)
6. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
7. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: ACM Symp. on Principles of Programming Languages. pp. 228–241 (1999)
8. Norell, U.: Dependently typed programming in agda. In: Kennedy, A., Ahmed, A. (eds.) Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009. pp. 1–2. ACM (2009), `http://doi.acm.org/10.1145/1481861.1481862`
9. Pottier, F., Simonet, V.: Information Flow Inference for ML. In: ACM Symp. on Principles of Programming Languages. pp. 319–330 (2002)
10. Russo, A.: Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In: ACM SIGPLAN International Conference on Functional Programming. ICFP, ACM (2015)
11. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE J. Selected Areas in Communications 21(1), 5–19 (2003)
12. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. Higher Order Symbol. Comput. 14(1) (Mar 2001)
13. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: ACM symposium on Principles of Programming Languages (1998)
14. Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J.C., Maziéres, D.: Addressing covert termination and timing channels in concurrent information flow systems. In: ACM SIGPLAN International Conference on Functional Programming. ACM (2012)
15. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: ACM SIGPLAN Haskell symposium (2011)
16. Volpano, D., Smith, G., Irvine, C.: A Sound Type System for Secure Flow Analysis. J. Computer Security 4(3), 167–187 (1996)