SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning

George Argyros
Columbia University
argyros@cs.columbia.edu

Ioannis Stais
University of Athens
i.stais@di.uoa.gr

Suman Jana
Columbia University
suman@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Aggelos Kiayias
University of Edinburgh
Aggelos.Kiayias@ed.ac.uk

ABSTRACT

Finding differences between programs with similar functionality is an important security problem as such differences can be used for fingerprinting or creating evasion attacks against security software like Web Application Firewalls (WAFs) which are designed to detect malicious inputs to web applications. In this paper, we present SFADiff, a black-box differential testing framework based on Symbolic Finite Automata (SFA) learning. SFADiff can automatically find differences between a set of programs with comparable functionality. Unlike existing differential testing techniques, instead of searching for each difference individually, SFADiff infers SFA models of the target programs using black-box queries and systematically enumerates the differences between the inferred SFA models. All differences between the inferred models are checked against the corresponding programs. Any difference between the models, that does not result in a difference between the corresponding programs, is used as a counterexample for further refinement of the inferred models. SFADiff's model-based approach, unlike existing differential testing tools, also support fully automated root cause analysis in a domain-independent manner.

We evaluate SFADiff in three different settings for finding discrepancies between: (i) three TCP implementations, (ii) four WAFs, and (iii) HTML/JavaScript parsing implementations in WAFs and web browsers. Our results demonstrate that SFADiff is able to identify and enumerate the differences systematically and efficiently in all these settings. We show that SFADiff is able to find differences not only between different WAFs but also between different versions of the same WAF. SFADiff is also able to discover three previously-unknown differences between the HTML/JavaScript parsers of two popular WAFs (PHPIDS 0.7 and Expose 2.4.0) and the corresponding parsers of Google Chrome, Firefox, Safari, and Internet Explorer. We confirm that all these differences can be used to evade the WAFs and launch successful cross-site scripting attacks.

CCS'16 October 24-28, 2016, Vienna, Austria

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

Software developers often create different programs with similar functionality for various reasons like supporting different target platforms, resolving conflicting licenses, accommodating different hardware constraints and exploring diverse performance trade-offs. However, these programs often suffer from subtle discrepancies that cause them to produce different outputs for the same input due to either implementation bugs or vagueness of the underlying specifications. Besides hurting interoperability of the affected programs, these differences can also have serious security implications. An attacker can leverage these differences for fingerprinting: That is, to identify the exact version of a program running on a remote server. As different programs suffer from different vulnerabilities, such fingerprinting information is very useful to an attacker for choosing specific attack vectors. Besides fingerprinting, the behavioral discrepancies can also be used to launch evasion attacks against security software that detects potentially malicious input to a target program. In such cases, the security software must faithfully replicate the relevant parts of the input parsing logic of the target software in order to minimize false negatives. Any discrepancy between the input parsing logic of the security software and that of the target program can be used by an attacker to evade detection while still successfully delivering the malicious inputs. For example, Web Application Firewalls (WAFs) detect potentially malicious input to web applications such as cross-site scripting (XSS) attack vectors. Therefore, a WAF must parse HTML/JavaScript code in the same way as web browsers do. Any inconsistency between these two parsers can lead to an evasion attack against the WAF. However, making the WAF HTML/JavaScript parsing logic similar to that of the web browsers is an extremely challenging and errorprone task as most web browsers do not strictly follow the HTML standard.

For the reasons mentioned above, automated detection of the differences between a set of test programs providing similar functionality is a crucial component of security testing. Differential testing is a way for automatically finding such differences by generating a large number of inputs (either through black-box fuzzing or white-box techniques like symbolic execution) and comparing the outputs of the test programs against each other for each input. However, existing differential testing systems have several drawbacks that prevent them from scaling to real-world systems with large input space (e.g., WAFs, web browsers, and network pro-

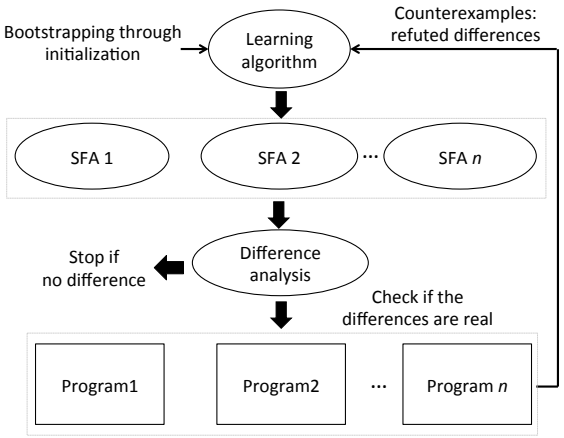


Figure 1: SFADiff **archtitecture**

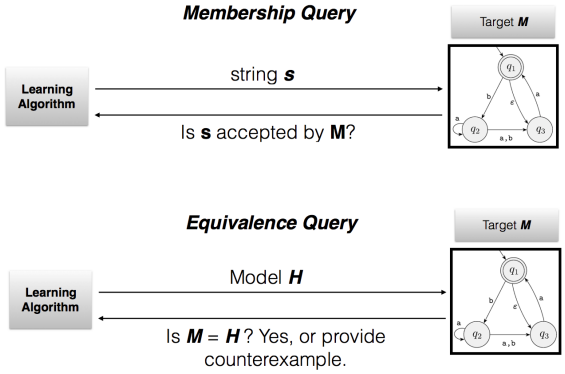


Figure 2: Types of queries that a learning algorithm can perform in our learning model.

tocol implementations). White-box techniques do not scale to such large systems mostly due to the overhead and complexity of the analysis process. Black-box fuzzing techniques try to brute-force through the vast input space without any form of guidance and therefore often fails to focus on the relevant parts of the input space.

In this paper, we present SFADiff, a black-box differential testing framework based on Symbolic Finite Automata (SFA) learning for automatically finding differences between comparable programs. Unlike existing differential testing techniques, instead of searching for each difference individually, SFADiff infers SFA models by querying the target programs in a black-box manner and checks for differences in the inferred models. SFADiff also verifies whether the candidate differences found from the inferred models indeed result in differences in the test programs. If a difference derived from the inferred models do not result in a difference in the actual programs, the corresponding input is reused as a counterexample to further refine the model.

Comparing two models in order to obtain counterexamples also provides a way to implement an *equivalence oracle* which checks the correctness of an inferred model and constitutes an essential component of the learning algorithm. In practice, simulating such an oracle is a challenging and computationally expensive task (cf. section 3). Nevertheless, our differential testing framework provides an efficient and elegant way to simulate an equivalence oracle by comparing the inferred models, thus the term "differential automata learning".

Figure 1 shows an overview of SFADiff architecture. SFADiff has several benefits over the existing approaches: (i) it explores the differences between similar programs in a systematic way and generalizes from the observations through SFA models; (ii) it can find and enumerate differences between SFA models efficiently; (iii) it can perform root cause analysis efficiently in a domain-independent manner by using the inferred models; and (iv) it also supports efficient bootstrapping mechanisms for incremental SFA learning for programs that only differ slightly (e.g., two versions of the same program).

We evaluated SFADiff in three different settings for finding differences between multiple TCP implementations, between different WAFs, and between the HTML/JavaScript parsers of WAFs and Web browsers. SFADiff was able to enumerate a large number of differences between the TCP implementations in Linux, FreeBSD, and Mac OSX. In the WAF setting, SFADiff found multiple differences between different WAFs as well as between different versions of the same WAF. Finally, SFADiff found three previously-unknown HTML/JavaScript parsing differences between two popular WAFs (PHPIDS 0.7 and Expose 2.4.0) and several major browsers like Google Chrome, Safari, Firefox, and Internet Explorer. Our experiments confirmed that all of these differences can be leveraged to launch successful cross-site scripting attacks while evading the vulnerable WAFs.

In summary, our main contributions are as follows:

- In section 4, we describe the design and implementation of SFADiff, the first differential testing framework based on automata learning techniques. We show that our framework can be used to perform several security critical tasks automatically such as finding evasion attacks, generating fingerprints, and identifying the root causes of the observed differences in a domain-independent manner.
- In section 3, we provide an efficient algorithm to bootstrap the SFA learning process from an initial model that allows for efficient incremental inference of similar programs.
- In section 5, we evaluate SFADiff on eleven applications from three different domains and show that it is able to find a large number of differences in all domains, including three previously-unknown evasion attacks against two popular WAFs, Expose and PHPIDS.

2. PRELIMINARIES

2.1 Definitions

A deterministic finite automaton (DFA) M over an alphabet Σ with set of states Q is specified by a transition function $\delta : Q \times \Sigma \to Q$. The subset $F \subseteq Q$ is called the set of accepting states. The language accepted by the automaton is denoted by $\mathcal{L}(M)$ and contains all those strings in Σ^* that, when parsed by the automaton starting from the initial state $q_0 \in Q$, lead to a state in F. Each DFA M induces a corresponding graph $G_M = (V, E)$ where $V = Q$ and $(q_i, q_j) \in E$ if and only if $\delta(q_i, \alpha) = q_j$ for some $a \in \Sigma$. We also denote an edge $(q_i, q_j) \in E$ as $q_i \to q_j$. We write $q_i \overset{*}{\to} q_j$ to denote that there exists a path in G_M between q_i

and q_j. We say that a path is *simple* if it does not contain any loops.

For a given automaton M, string $w \in \Sigma^*$ and state $q \in Q$ we denote by $M_q[w]$ the state that is reached when the automaton parses the string w, starting from state q. When the subscript is omitted the initial state q_0 is assumed. We also define the function $l : Q \to \{0, 1\}$ such that $l(q) = 1$ if and only if $q \in F$. It follows that $\mathcal{L}(M) = \{w \mid l(M_{q_0}[w]) = 1\}$. We denote by ϵ the empty string. For two strings s_1,s_2 and a set of strings W, we say that $s_1 \equiv s_2 \bmod W$ if, for every $w \in W$ it holds that $l(M[s_1 \cdot w]) = l(M[s_2 \cdot w])$. A predicate family $\mathcal{P}$ is a set of predicates. The following sets of strings defined for an automaton M play a fundamental role in learning algorithms:

- **Access strings.** We say that a string s access a state q if $M[s] = q$. The set of access strings for an automaton M is a set A such that, for each state q in M there exists $s \in A$ such that s access q.
- **Distinguishing strings.** The set of distinguishing strings is a set of strings D for which it holds that for each pair of states q, q' it holds that there is some $d \in D$ such that $l(M_q[d]) \neq l(M_{q'}[d])$.

Symbolic Finite Automata. Symbolic finite automata (SFA) are finite state machines that decide an input string by performing state transitions controlled by predicate membership. A DFA is a special case of an SFA where the predicate family is restricted to the forms "$x = a$" for $a \in \Sigma$. We will adopt the following definition that has been used to formally describe this class of machines [5] :

Definition 1. A symbolic finite automaton (SFA) is a tuple $(Q, q_0, F, \mathcal{P}, \Delta)$, where Q is a finite set of states, $q_0 \in Q$ the *initial state*, $F \subseteq Q$ is the set of *final states*, $\mathcal{P}$ is a predicate family and $\Delta \subseteq Q \times \mathcal{P} \times Q$ is the *move* relation. For each state q, we define the guard predicate set as follows $\text{guard}(q) := \{\phi : \exists p \in Q, (q, \phi, p) \in \Delta\}$.

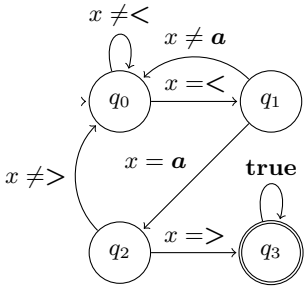
Extension to programs with non-binary output. Due to space constraints, we describe our algorithms for the case of programs with binary output. Nevertheless, to model programs with general output, SFAs can be replaced with symbolic finite state transducers (SFTs) [30], and the corresponding learning algorithms for transducers [5] can be used. All of our algorithms can be easily extended to transducers.

2.2 Learning Model

The learning algorithms used in this paper work in an active learning model called *exact learning from membership and equivalence queries*. Contrary to the traditional supervised machine learning setting, where the models are trained on a given dataset, active learning algorithms are able to query the target machine with any input of their choice and obtain the correct label for that input from the target. Specifically, in our learning model, we assume that a learner, who is trying to learn an unknown automaton M, has access to an oracle answering two types of queries: (i) *membership queries* through which the learner can submit a string s and obtain whether $s \in \mathcal{L}(M)$ or not and (ii) *equivalence queries* through which the learner can submit a model H and obtain whether $\mathcal{L}(H) = \mathcal{L}(M)$. Figure 2 shows a pictorial presentation of these queries.

2.3 SFA Learning Algorithm

For learning SFAs, we use the ASKK algorithm proposed



S \ W	ϵ	a>	>
ϵ	0	0	0
<	0	1	0
<a	0	0	1
<a>	1	0	0

Figure 3: A Symbolic Finite Automaton (SFA) for the regular expression .*<a>.* and the corresponding entries for the S, W sets from the observation table.

by Argyros et al. [5]. We present a brief overview of the algorithm below and encourage the interested readers to check [5] for more details. At a high level, the algorithm attempts to reconstruct the set of access and distinguishing strings for the target automaton, from which it is able to recover a correct model of the target machine. The transitions of the SFA are generated using a mechanism called the guardgen() algorithm that, given a sample set of transitions as input, generates a set of predicate guards for the SFA model.

The main data structure utilized by the algorithm is the special observation table $SOT = (S, W, \Lambda, T)$, where S and W are, possibly incomplete, sets of access and distinguishing strings for the target automaton, $\Lambda \subseteq S \cdot \Sigma$ is a set of sample transitions and T is a table with rows over $S \cup \Lambda$ and columns in W. Given a row s and column w, the table is populated with $T(s, w) = l(M[sw])$. Figure 3 shows a simple SFA along with the observation table entries for the S and W sets.

The algorithm initializes the table with $S = W = \{\epsilon\}$ and a set of sample transitions Λ (a single symbol suffices). The SOT is called *closed* if for every $\alpha \in \Lambda$, there exists $s \in S$ such that $\alpha \equiv s \mod W$. Once all entries in the table are populated using membership queries, the table is checked for closedness. If there exists an $\alpha \in \Lambda$ such that the closedness condition is not satisfied, then α is accessing a previously undiscovered state in the target automaton. Thus, we move α into the set S, fill the new entries in the table, and check again for closedness. Eventually, this process will produce a closed SOT if the target language is regular.

Updating models. Given a closed SOT, the learning algorithm constructs an SFA model. This model is then tested for equivalence with the target automaton. In the abstract learning model this is achieved using a single equivalence query, however, in practice, various testing methods are utilized to simulate an equivalence query. If the learned model is not equivalent to the target machine, the equivalence query returns a *counterexample* input s that causes the model to produce different output than the target machine. The learning algorithm uses the counterexample to refine the generated model by either adding a missing state or correcting an invalid transition.

3. BOOTSTRAPPING SFA LEARNING

Motivation. Consider a user that has invested a significant time budget to infer an SFA model for a specific version of a program. When a new version of the program is released, one can expect it to be, in many aspects, similar with the previous version. In such settings, the ability to

incrementally learn the SFA model for the new version can be a very useful feature. The learning process will become significantly faster if SFADIFF can somehow utilize the old model for learning the new model. In this section, we provide an efficient algorithm in order to bootstrap the SFA learning algorithm by initializing it with an existing model. Our method ensures that, if the system we are trying to infer is the same as the model used for initializing the learning algorithm, only a single equivalence query will be made by the learning algorithm in order to verify the equivalence of the model with the system. Since simulating equivalence queries is usually the most expensive part in learning, being able to save equivalence queries provide a significant overall optimization in the learning process.

Notice that, most popular algorithms for simulating equivalence queries are intractable for large alphabets. For example, consider the case of Chow's W-method [12], that is used by popular automata inference frameworks like LearnLib [24] for simulating equivalence queries. The W-method accepts as input a model automaton M with m states and an upper bound n on the number of states of the target automaton. The W-method compiles a set of test cases to verify that, if the target automaton has at most n states, then it is equivalent to the model automaton. Unfortunately, in order to verify equivalence, the W-method performs $O(n^2 m|\Sigma|^{n-m+1})$ membership queries to the target system. The exponential term in the alphabet size makes the method prohibitive for usage in models with large alphabets (e.g. all printable characters or even larger sets if we include Unicode symbols).

Our algorithm. Given an initial SFA model M_{init} we bootstrap the ASKK algorithm by creating a special observation table $SOT = (S, W, \Lambda, T)$ with the S, W, Λ sets initialized from M_{init}, as described below, while the entries of the table are filled using membership queries to the target automaton. This technique allows us to build a correct model if the initial model and the target system are equivalent. If the two systems are not equivalent but similar, i.e. they share certain access and distinguishing strings, then our initialization algorithm will recover those without performing any equivalence queries. We will now describe how to initialize each component of the special observation table.

3.1 Initializing the SOT

Initializing S. Initializing S corresponds to the recovery of all access strings of M_{init}. This is a straightforward procedure using a DFS search in the graph induced by M_{init}. The procedure starts with an empty access string for the initial state of the automaton. Every time we exercise a transition (q_s, ϕ, q_t), we check if an access string for q_t is already in S. If no access string exists for q_t then, we select a witness $\alpha \in \phi$ from the predicate guard of the transition and we assign the access string $s_{q_s}\alpha$ for state q_t where $s_{q_s} \in S$ is an access string for q_s. Once all states are covered, we return the set of access strings.

Initializing W. Initializing the W set corresponds to the creation of a set of distinguishing strings for M_{init}. Algorithms for creating distinguishing sets for DFAs date back to the development of Chow's W-method [12]. Adapting these algorithms in the SFA setting is straightforward by adapting the SFA minimization algorithms developed recently by D'Antoni and Veanes [14]. We note that these algorithms are the most efficient known algorithms for SFA minimization and the adaptation for generating a set of distinguishing strings will produce a set of distinguishing strings of size $n-1$ for an SFA with n states.

Initializing Λ. In order to correctly initialize the Λ component of the SOT, we have to provide, for every state q of M_{init} a set of sample transitions that, when given as input into the guardgen() algorithm will produce the correct set of predicate guards for q.

The predicate guards used by the SFA learning algorithm in [5] are simply sets of symbols from the alphabet. Given a set of sample transitions for a state q, the guardgen() algorithm from [5] works as follows: All transitions for symbols from state q already in the Λ set are grouped into predicate guards based on the target of the transition which is determined as in the original L^* algorithm [6]. The transitions for symbols which are not part of the Λ set are merged into the predicate guard with the largest size, i.e. the transition containing most symbols. The intuition behind this algorithm is that in most parsers, only a small numbers of symbols is advancing the automaton towards an accepting state, while most other symbols are grouped together in a single transition leading to a rejecting state.

Therefore, given a state q in M_{init}, in order to construct a sample set of transitions that will result in producing the correct predicate guards with the aforementioned guardgen() algorithm, we proceed as follows: Let $\{\phi_1, \phi_2, \ldots, \phi_k\}$ be the set of predicate guards for the state q such that $i < j \implies |\phi_i| \geq |\phi_j|$. Moreover, let s_q be the access string for q and $T = \cup_{i \in \{2,\ldots,k\}} \phi_i$. Then, for each $\alpha \in T$, we add the string $s_q\alpha$ in Λ. This will ensure that the predicate guards for $\phi_2, \ldots, \phi_k$ will be produced correctly by the guardgen() algorithm. Finally, we have to ensure that enough sample transitions from ϕ_1 are added in Λ in order for ϕ_1 to get all implicit transitions which are not part of Λ. To achieve that, we select $l_2 = |\phi_2| + 1$ elements $\alpha_j \in \phi_1, j \in \{1, \ldots, l_2\}$ and add the strings $s_q\alpha_j$ in Λ. This operation ensures that if the transitions of the target automaton are the same as in M_{init}, they will be generated correctly by the guardgen() algorithm. Repeating this procedure for all states of M_{init} completes the initialization of the Λ set.

4. DIFFERENTIAL SFA LEARNING

4.1 Basic Algorithm

The main idea behind our differential testing algorithm is to leverage automata learning in order to infer SFA-based models for the test programs and then compare the resulting models for equivalence as shown in Figure 1. As mentioned above, this technique has a number of advantages such as being able to generalize from comparing individual input/output pairs and build models for the programs that are examined.

Algorithm 1 provides the basic algorithmic framework for differential testing using automata learning. The algorithm takes two program implementations as input. The first function calls, to the `GetInitialModel` function, are responsible for bootstrapping the models for the two programs. In our case this function is implemented using the observation table initialization algorithm described in Section 3. The initialized models are then checked for differences using the `RCADiff` function call. The internals of this function are described in detail in Section 4.2. This function is responsible for categorizing the differences in the two models and

Algorithm 1 Differential SFA Testing Algorithm

Require: P_1, P_2 are two programs
  function GetDifferences(P_1, P_2)
    $M_1 \leftarrow$ GetInitialModel(P_1)
    $M_2 \leftarrow$ GetInitialModel(P_2)
    while true **do**
      $S \leftarrow$ RCADiff(M_1, M_2)
      if $S = \emptyset$ **then**
        return $\emptyset$
      end if
      modelUpdated $\leftarrow False$
      for $s \in S$ **do**
        if $P_1(s) \neq M_1(s)$ **then**
          $M_1 \leftarrow$ UpdateModel(M_1, s)
          modelUpdated $\leftarrow True$
        end if
        if $P_2(s) \neq M_2(s)$ **then**
          $M_2 \leftarrow$ UpdateModel(M_2, s)
          modelUpdated $\leftarrow True$
        end if
      end for
      if modelUpdated $= False$ **then**
        return S
      end if
    end while
  end function

return a sample set of inputs covering all categories that can cause the two programs to produce different outputs. The algorithm stops if the two models are equivalent. Otherwise, `RCADiff` returns a set of inputs that cause the two SFA models to produce different output.

However, since these differences are obtained by comparing the program models and not the actual programs, they might contain false positives resulting from inaccurate models. To detect such cases, we verify all differences obtained from the `RCADiff` call using the actual test programs. If any input is found not to produce a difference in the implementations, then that input is used as a counterexample in order to refine the model through the `UpdateModel` call. Finally, when a set of differences in the two models is verified to contain only true positives, the algorithm returns the set of corresponding inputs back to the user.

The astute reader may notice that, if no candidate differences are found between the two models, the algorithm terminates. For this reason, model initialization plays a significant role in our algorithm, since the initialized models should be expressive enough in order to provide candidate differences. It is interesting to point out that the candidate differences do not have to be real differences.

4.2 Difference Analysis

Assume that we found and verified a number of inputs that cause the two programs under test to produce different outputs. One fundamental question is whether we can classify these inputs in certain equivalence classes based on the cause of the deviant behavior. We will now describe how we can use the inferred SFAs in order to compute such a classification. Ideally, we would like to assign in two inputs that cause a difference the same root cause if they follow the same execution paths in the target programs. Since the program source is unavailable, we trace the execution path of the inputs in the respective SFA models.

RCADiff algorithm. Given two SFAs M_1 and M_2, it is straightforward to compute their intersection by adapting the classic DFA intersection algorithm [28]. Let $M_{prod} = (Q_1 \times Q_2, (q_0, q_0), \{(q_i, q_j) : q_i \in F_1 \wedge q_j \in F_2\}, \mathcal{P}, \Delta)$ be the, minimal, product automaton of M_1, M_2. Notice initially, that the reason a difference is observed in the output after processing an input in both SFAs is that the labels of the states reached in the two machines are different. This motivates our definition of *points of exposure*.

Algorithm 2 Difference Categorization Algorithm

Require: M_1, M_2 are two SFA Models
  function RCADiff(M_1, M_2)
    $M_{prod} \leftarrow$ ProductSFA(M_1, M_2)
    $S \leftarrow \emptyset$
    for $(q_i, q_j) \in Q_{prod} \mid l(q_i) \neq l(q_j)$ **do**
      $S \leftarrow S \cup$ SimplePaths($M_{prod}, (q_i, q_j)$)
    end for
    return Path2Input(S)
  end function

Definition 2. Let M_{prod} be the intersection SFA of M_1, M_2 as defined above. We define the set $\{(q_i, q_j) | (q_i, q_j) \in Q_{prod} \wedge q_i \in Q_1 \wedge q_j \in Q_2 \wedge l(q_1) \neq l(q_2)\}$ to be the *points of exposure* for the differences between M_1, M_2.

Intuitively, the points of exposure are the reasons the differences in the programs are observed through the output of programs. The path to a point of exposure encodes two different execution paths in machines M_1 and M_2 respectively which, under the same input, end up in states producing different output. Thus, we say that any simple path to a point of exposure is a *root cause* of a difference.

Definition 3. Let M_1, M_2 be two SFAs and M_{prod} be the intersection of M_1, M_2. Let $Q_p \subseteq Q_{prod}$ be the points of exposure for M_{prod}. We say that the set of simple paths $S = \{q_0 \xrightarrow{*} q_p | q_p \in Q_p\}$ is the set of *root causes* for the differences between M_1 and M_2.

Equipped with the set of paths our classification algorithm works as follows: Given two inputs causing a difference, we first reduce the path followed by each input into a simple path, i.e. we remove all loops from the path. For example, an input following the path $q_0 \rightarrow q_4 \rightarrow q_5 \rightarrow q_4 \rightarrow q_{10}$ will be reduced to the path $q_0 \rightarrow q_4 \rightarrow q_{10}$. Afterwards, we classify the two inputs in the same root cause if the simple paths followed by the inputs are the same.

Algorithm 2 shows the pseudocode for the `RCADiff` algorithm. The algorithm works by collecting all the distinct root causes from the product automaton using the the `SimplePaths` function call. This function accepts an SFA and a target state and returns all simple paths from the initial state to the target state using a BFS search. Afterwards, each path is converted into a sample input through the function `Path2Input`. This function works by selecting, for each edge $q_i \rightarrow q_j$ in the path, a symbol $\alpha \in \Sigma$ such that $(q_i, \phi, q_j) \in \Delta \wedge \phi(\alpha) = 1$. Finally, these symbols are concatenated in order to form an input that exercise the given path in the SFA.

4.3 Differentiating Program Sets

In this section, we describe how our original differential testing framework can be generalized into a `GetSetDifferences` algorithm which works as follows: Instead of getting two programs as input, the `GetSetDifferences` algorithm receives two sets of programs $\mathcal{I}_1 = \{P_1, \ldots, P_n\}$ and

$\mathcal{I}_2 = \{P_1, \ldots, P_m\}$. Assume that the output of each program is a bit $b \in \{0, 1\}$. The goal of the algorithm is to find a set of inputs S such that, the following condition holds:

$$\exists b \ \forall P_1 \in \mathcal{I}_1, P_1(s) = b \wedge \forall P_2 \in \mathcal{I}_2, P_2(s) = 1 - b$$

While conceptually simple, this extension provides a number of nice applications. For example, consider the problem of finding differences between the HTML/JavaScript parsers of browsers and those of WAFs. While finding such differences between a single browser and a WAF will provide us with an evasion attack against the WAF, the `GetSetDifferences` algorithm allows us to answer more sophisticated questions such as: (i) Is there an evasion attack that will bypass multiple different WAFs? and (ii) Is there an evasion attack that will work across different browsers? Also, as we describe in Section 4.4, this extension allows us to produce succinct fingerprints for distinguishing between multiple similar programs.

GetSetDifferences Algorithm. We extend our basic `GetDifferences` algorithm as follows: First, instead of initializing two program models as before, we initialize the SFA models for all programs in both sets accordingly. Similarly, when we verify the candidate differences obtained from the inferred models, all programs in both sets should be checked. Besides these changes, the skeleton of the GetDifferences algorithm remains the same.

The most crucial and time-consuming part of our extension is the extension to the `RCADiff` functionality in order to detect differences between two sets of models. Recall that RCADiff utilizes the product construction and then finds the simple paths leading to the points of exposure. Given two sets of models, we compute the intersection between all the models in the two sets. Afterwards, we set the points of exposure as follows. Let $q = (q_0, \ldots, q_{m+n})$ be a state in the product automaton. Furthermore, assume that state q_i corresponds to automata M_i from one of the input sets $\mathcal{I}_1, \mathcal{I}_2$. Then, q is a point of exposure if

$$\forall M_i \in \mathcal{I}_1, M_j \in \mathcal{I}_2 \implies l(q_i) \neq l(q_j)$$

With this new definition of the points of exposure, the modified `RCADiff` algorithm proceeds as in the original case to find all simple paths in the product automaton that lead to the points of exposure.

One potential downside of this algorithm is that, its complexity increases exponentially as we add more models in the sets. For example, computing the intersection of m DFA with n states each, requires time $O(n^m)$ while, in general, the problem is PSPACE-complete [21]. That being said, we stress that the number of programs we have to check in practice will likely be small and many additional heuristics can be used to reduce the complexity of the intersection computation.

4.4 Program Fingerprints

Formally, the fingerprinting problem can be described as follows: given a set $\mathcal{I}$ of m different programs and black-box access to a server T which runs a program $P_T \in \mathcal{I}$, how can one find out which program is running in the server T by simply querying the program in a black-box manner, i.e. find $P \in \mathcal{I}$ such that $P = P_T$.

In this section, we present two different fingerprinting algorithms that provide different trade-offs between computational and query complexity. Both these algorithms build

Algorithm 3 Fingerprint Tree Building Algorithm

Require: $\mathcal{I}$ is a set of Programs

```
function BuildFingerprintTree(I)
    if |I| = 1 then
        root.data ← P ∈ I
        return root
    end if
    P_i, P_j ← I
    s ← GetDifferences(P_i, P_j)
    root.data ← s
    root.left ← BuildFingerprintTree(I \ P_i)
    root.right ← BuildFingerprintTree(I \ P_j)
    return root
end function
```

a binary tree called fingerprint tree that stores strings that can distinguish between any two programs in $\mathcal{I}$. Given a fingerprinting tree, our first algorithm requires $|\mathcal{I}|$ queries to the target program. If the user is willing to perform extra off-line computation, our second algorithm demonstrates how the number of queries can be brought down to $\log m$.

Basic fingerprinting algorithm. The `BuildFingerprintTree` algorithm (shown in Algorithm 3) constructs a binary tree that we call a *fingerprint tree* where each internal node is labeled by a string and each leaf by a program identifier. In order to build the fingerprint tree recursively, we start with the set of all programs $\mathcal{I}$, choose any two arbitrary programs P_i, P_j from $\mathcal{I}$, and use the differential testing framework to find differences between these programs. We label the current node with the differences, remove P_i and P_j from $\mathcal{I}$, and call `BuildFingerprintTree` recursively until a single program is left in $\mathcal{I}$. If $\mathcal{I}$ has only one program, we label the leaf node with the program and return.

Given a fingerprint tree, we solve the fingerprinting problem as follows: Initially, we start at the root node and query the target program with a string from the set that labels the root node of the tree. If the string is accepted (resp. rejected), we recursively repeat the process along the left subtree (resp. right subtree), until we reach a leaf node that identifies the target program.

Time/query complexity. For the following we assume an input set of programs $\mathcal{I}$ of size $|\mathcal{I}| = m$. Our algorithm has to find differences between all $\binom{m}{2}$ different program pairs. The fingerprint tree resulting from the algorithm will be a full binary of height m. Assuming that the complexity of the differential testing algorithm is D, we get that the overall time complexity of the algorithm is $O(2^{m-1} + \binom{m}{2}D)$. Finally, the query complexity of the algorithm is $|\mathcal{I}|$-1 queries, since each query will discard one candidate program from the list.

Reducing queries using shallow fingerprint trees. Notice that, in the previous algorithm, we need m queries to the target program in order to find the correct program because we discard only one program at each step. We can cut down the number of queries by shallower fingerprint trees at the cost of higher off-line computational complexity for building such trees.

Consider the following modification in the `BuildFingerprintTree` algorithm: First, we partition $\mathcal{I}$ into k subsets $\mathcal{I}_1, \ldots \mathcal{I}_k$ of size m/k each. Next, we call `BuildFingerprintTree` algorithm with the set $\mathcal{I}_S = \{\mathcal{I}_1, \ldots, \mathcal{I}_k\}$ as input programs and replace the call to `GetDifferences` with `Get-`

`SetDifferences`. This algorithm will generate a full binary tree of height k that can distinguish between the programs in the different subsets of $\mathcal{I}$. We can recursively apply the same algorithm on each of the leafs of the resulting fingerprinting tree, further splitting the subsets of $\mathcal{I}$ until each leaf contains a single program.

Time/query complexity. It is evident that the algorithm will eventually terminate since each subset is successively portioned into smaller sets. Let us assume that $D_{set}(k)$ the complexity of the `GetSetDifferences` algorithm when the input program sets are of size k (see section 4.3 for a complexity analysis of $D_{set}(k)$). The number of queries required for fingerprinting an application with this algorithm will be equal to the height of the resulting fingerprint tree. Note that each subset is of size m/k and to distinguish between the k subsets using our basic algorithm we need $k-1$ queries. Therefore we get the equation $T(m) = T(m/k) + (k-1)$ describing the query complexity of the algorithm. Solving the equation we get that $T(m) = (k-1)\log_k m$ which is the query complexity for a given k. When $k = 2$ we will need $\log m$ queries to identify the target program. Since each program provides one bit of information per query (accept/reject), a straightforward decision tree argument [13] provides a matching lower bound on the query complexity of the problem.

Regarding the time complexity of the problem, we notice that, at the i-th recursive call to the modified `BuildFingerprintTree` algorithm, we will have an input set of size m/k^i since the initial set is repeatedly partitioned into k subsets. the overall time complexity of building the tree is $\sum_{i=1}^{\log_k m}(2^{m/k^i} + \binom{m/k^i}{2} D_{set}(m/k^i))$. We omit further details here as the complexity analysis is a straightforward adaptation of the original analysis.

5. EVALUATION

5.1 Initialization evaluation

Our first goal is to evaluate the efficiency of our observation table initialization algorithm as a method to reduce the number of equivalence queries while inferring similar models. The experimental setup is motivated by our assumptions that the initialization model and the target model would be similar. For that purpose, we utilized 9 regular expression filters from two different versions of ModSecurity (versions 3.0.0 and 2.2.7) and PHPIDS WAFs (versions 0.7.0 and 0.6.3). The filters in the newer versions of the systems have been refined to either patch evasions or possibly to reduce false positive rate.

For our first experiment we used an alphabet of 92 symbols, the same one used in our next experiments, which contains most printable ASCII characters. Since, in this experiment, we would like to measure the reduction offered by our initialization algorithm in terms of equivalence queries, we simulated a complete equivalence oracle by comparing each inferred model with the target regular expression.

Results. Table 1 shows the results of our experiments. First, notice that in most cases the updated filters contain more states than their previous versions. This is expected, since most of the times the filters are patched to cover additional attacks, which requires the addition of more states for covering these extra cases. We can see that, in general, our algorithm offers a massive reduction of approximately 50× in the number of equivalence queries utilized in order

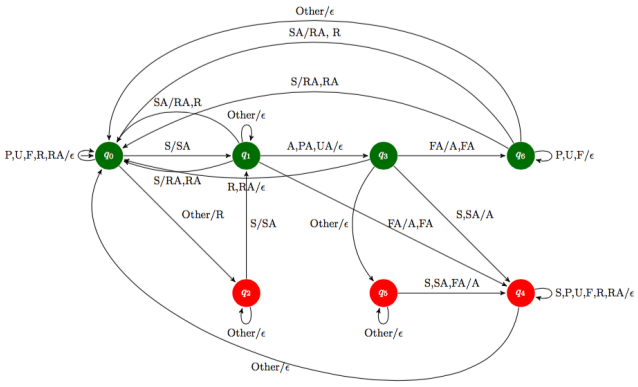


Figure 4: State machine inferred by SFADIFF for Mac OSX TCP implementation. The TCP flags that are set for the input packets are abbreviated as follows: SYN(S), ACK(A), FIN(F), PSH(P), URG(U), and RST(R).

to infer a correct model. This comes with a trade-off since the number of membership queries are increased by a factor of 1.15×, on average. However, equivalence queries are usually orders of magnitude slower than membership queries. Therefore, the initialization algorithm results in significant overall performance gain. We notice that 2/3 cases where we observed a large increase (more than 1.2×) in membership queries (filters PHPIDS 50 & PHPIDS 56) are filters for which states were removed in the new version of the system. This is expected since, in that case, SFADIFF makes redundant queries for an entry in the observation table that does not correspond to an access string. Another possible reason for an increase in the number of the membership queries is the chance that the distinguishing set obtained by the SFA learning algorithm is smaller than the one obtained by the initialization algorithm which is always of size $n-1$ where n is the number of states in a filter. Exploring ways to obtain a distinguishing set of minimum size is an interesting direction in order to further develop our initialization algorithm. Nevertheless, in all cases, the new versions of the filters were similar in structure with the older versions and thus, our initialization algorithm was able to reconstruct a large part of the filter and massively reduce the number of equivalence queries required to obtain the correct model.

5.2 TCP state machines

For our experiments with TCP state machines, we run a simple TCP server on the test machine while the learning algorithm runs as a client on another machine in the same LAN. Because the TCP protocol will, possibly, emit output for each packet sent, the ASKK algorithm is not suited for this case. Thus, we used the algorithm from [5] for learning deterministic transducers in order to infer models of the TCP state machines.

Alphabet. For this set of experiments, we focus on the effect of TCP flags on the TCP protocol state transitions. More specifically, we select an alphabet with 11 symbols including 6 TCP flags: SYN(S), ACK(A), FIN(F), PSH(P), URG(U), and RST(R) along with all possible combinations of these flags with the ACK flag, i.e., SA, FA, PA, UA, and RA.

Membership queries. Once our learning algorithm for-

IDS Rules	Without Init Member	Without Init Equiv	With Init Member	With Init Equiv	Learned States	Init Filter States	States Diff	Member Overhead	Equiv Speedup
MODSEC 973323	2367	97	2400	2	25	25	0	1.01	48.50
MODSEC 973324	768	55	892	19	15	12	3	1.16	2.89
MODSEC 973330	887	62	941	21	15	12	3	1.06	2.95
PHPIDS 22	17195	252	17330	105	70	45	25	1.01	2.40
PHPIDS 27	144759	2618	149159	437	66	59	7	1.03	5.99
PHPIDS 40	11119	337	11152	68	35	25	10	1.00	4.96
PHPIDS 41	6635	318	8535	137	25	21	4	1.29	2.32
PHPIDS 50	6206	255	9829	1	25	27	-2	1.58	255.00
PHPIDS 56	38768	840	46732	7	60	62	-2	1.21	120.00
		Avg= 537.11×		Avg= 88.56×				Avg= 1.15×	Avg= 49.45×

Table 1: The performance (no. of equivalence and membership queries) of the SFA learning algorithm with and without initialization for different rules from two WAFs (ModSecurity OWASP CRS and PHPIDS).

OS	States	Queries
OSX Yosemite (version 14.5.0)	7	858
Debian Linux (Kernel v3.2.0)	9	1100
FreeBSD 10.3	9	1100

Table 2: Results for different TCP implementations: Number of states in each model and number of membership queries required to infer the model.

Input	Linux	OSX	FreeBSD
S, S	SA, RA	SA, RA, RA	SA
S, A, F	SA, A, FA	SA	SA
S, RA, A	SA, R	SA, R	SA

Table 3: Some example fingerprinting packet sequences found by SFADiff across different TCP implementations. The TCP flags that are set for the input packets are abbreviated as follows: SYN(S), ACK(A), FIN(F), and RST(R).

mulates a membership query, our client implementation creates a sequence of TCP packets corresponding to the symbols and sends them to the server.

Our server module is a simple python script which works as follows: The script is listening for new connections on a predefined port. Once a connection is established our server module makes a single `recv` call and then actively close the connection. In addition, for each different membership query we spawn a new server process on a different port to ensure that packets belonging to different membership queries will not be mixed together.

The learning algorithm handles the sequence and acknowledgement numbers in the outgoing TCP packets in the following way: a random sequence number is used as long as no SYN packet is part of a membership query; otherwise, after sending a SYN packet we set the sequence and acknowledgement numbers of the following packets in manner consistent with the TCP protocol specification. In case the learning algorithm receives a RST packet during the execution of a membership query, we also reset the state of the sequence numbers, i.e. we start sending random sequence numbers again until the next SYN packet is send.

After sending each packet from a membership query, the learning algorithm waits for the response for each packet using a time window. If the learning algorithm receives any re-transmitted packets during that time, it ignores those packets. We detect re-transmitted packets by checking for duplicate sequence/acknowledgement numbers. Ignoring the re-transmitted packets is crucial for the convergence of the learning algorithm as it helps us avoid any non-determinism caused by the timing of the packets.

Initialization. As TCP membership queries usually outputs more information in terms of packets than one bit, our algorithm worked efficiently for the TCP implementations even without any initialization. Therefore, for the TCP experiments, we start the learning algorithm without any initial model.

Results. We used SFADiff in order to infer models for the TCP implementations of three different operating systems: Debian Linux, Mac OSX and FreeBSD. The inferred models contain all state transitions that are necessary to capture a full TCP session. Figure 4 shows the inferred state machine for Mac OSX. States in green color are part of a normal TCP session while states in red color are reached when an invalid TCP packet sequence is sent by the client. The path $q_0 \rightarrow q_1 \rightarrow q_3$ is where the TCP three-way handshake takes place and it is leading to state q_3 where the connection is established, while the path $q_3 \rightarrow q_6 \rightarrow q_0$ close the connection and returns to the initial state (q_0). Table 2 shows that the inferred model for Mac OSX contain fewer states than the respective FreeBSD and Linux models. Manual inspection of the models revealed that these additional states are due to different handling of invalid TCP packet sequences. Finally, in Table 3, we present some sample differences found by SFADiff. Note that, even though the state machines of Linux and FreeBSD contain the same number of states, they are not equivalent, as we can see in Table 3, since the two implementations produce different outputs for all three inputs.

5.3 Web Application Firewalls and Browsers

In this setting, we perform two sets of experiments: (i) we use SFADiff to explore differences in HTML/JavaScript signatures used by different WAFs for detecting XSS attacks; and (ii) we use SFADiff to find differences in the JavaScript parsing implementation of the browsers and the WAFs that can be exploited to launch XSS attacks while bypassing the WAFs.

For these tests, we configure the WAFs to run as a server and the learning algorithm executes as a client on the same machine. The browser instance is also running on the same machine. The learning algorithm communicates with the browser instance through WebSockets. The learning algo-

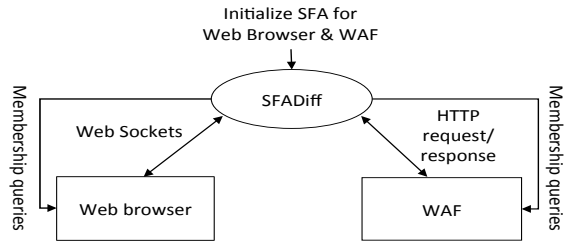


Figure 5: The setup for SFADiff finding differences between the HTML/JavaScript parsing in Web browsers and WAFs.

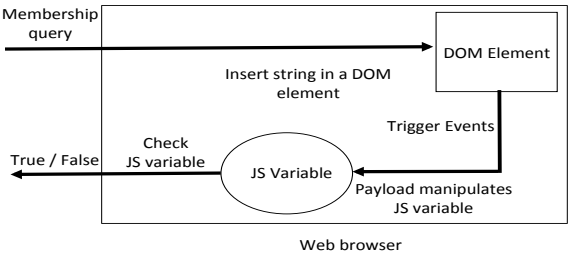


Figure 6: The implementation of membership queries for Web browsers.

rithm can test whether an HTML page with some JavaScript code is correctly parsed by the browser and if the embedded JavaScript is executed or not by exchanging messages with the browser instance. The overall setup is shown in Figure 5.

Alphabet. We used an alphabet of 92 symbols containing most printable ASCII characters. This allows us to encode a wide range of Javscript attack vectors.

Membership queries to the browser. In order to allow the learning algorithm to drive the browser, we make the browser connect to a web server controlled by the learning algorithm. Next, the learning algorithm sends a message to the browser over WebSockets with the HTML/JavaScript content corresponding to a membership query as the message's payload. Upon receiving such a message, the browser sets the query payload as the `innerHTML` of a DOM element and waits for the DOM element to be loaded. The user's browser dispatches a number of events (such as "click") on the DOM element and examines if the provided string led to JavaScript execution. These events are necessary for triggering the JavaScript execution in certain payloads. In order to examine if the JavaScript execution was successful, the browser monitors for any change in the value of a JavaScript variable located in the page. The payload, when executed, changes the variable value in order to notify that the execution was successful. Furthermore, in order to cover more cases of JavaScript execution, the user's browser also monitors for any JavaScript errors that indicate JavaScript execution. After testing the provided string, the user's browser sends back a response message containing a boolean value that indicates the result. The results of the membership queries are cached by the learning algorithm in order to be reused in the future. The details of our implementation of membership queries for the browsers is shown in Figure 3.

Membership queries to the WAF. SFADiff sends an HTTP request to the WAFs containing the corresponding HTML/JavaScript string as payload to perform a membership request, The WAF analyzes the request, decides whether to allow/block the payload, and communicates the decision back to SFADiff. SFADiff caches the results of the membership queries in order to be reused in the future.

Equivalence queries. We perform equivalence queries in two ways: first, whenever an equivalence query is sent either to the browser or to a WAF, we check that the model complies to the answers of all membership queries made so far. This ensures that simple model errors will be corrected before we perform more expensive operations such as cross-checking the two models against each other. Afterwards, we proceed to collect candidate differences and verify them against the actual test programs as described in Section 4.

Initialization. We initialize the observation tables for both the browser and the WAF using a small subset of filters that come bundled with PHPIDS and ModSecurity, two open-source WAFs in our test set. However, in the case of the browser we slightly modify the filters in order to execute our JavaScript function call if they are successfully parsed by the browser.

Fingerprinting WAFs. In order to evaluate the efficiency of our fingerprint generation algorithm we selected 4 different WAFs. Furthermore, To demonstrate the ability of our system to generate fine-grained fingerprints we also include 4 different versions of PHPIDS in our test set. As an additional way to avoid blowup in the fingerprint tree size we employ the following optimization: Whenever a fingerprint is found for a pair of firewalls, we check whether this fingerprint is able to distinguish any other firewalls in the set and thus further reduce the remaining possibilities. This simple heuristic significantly reduces the size of the tree: Our basic algorithm creates a full binary tree of height 8 while this heuristic reduced the size of the tree to just 4 levels.

Figure 5.3 presents the results of our experiment. The resulting fingerprinting tree also provides hints on how restrictive each firewall is compared to the others. An interesting observation is that we see the different versions of PHPIDS to be increasingly restrictive in newer versions, by rejecting more of the generated fingerprint strings. This is natural since newer versions are usually patching vulnerabilities in the older filters. Finally, we would like to point out that some of the fingerprints are also suggesting potential vulnerabilities in some filters. For example, the top level string, *union select from*, is accepted by all versions of PHPIDS up to 0.6.5, while being rejected by all other filters. This may raise suspicion since this string can be easily extended into a full SQL injection attack.

Evading WAFs through browser parser inference. For our last experiment we considered the setting of evaluating the robustness of WAFs against evasion attacks. Recall, that, in the context of XSS attacks, WAFs are attempting to reimplement the parsing logic of a browser in order to detect inputs that will trigger JavaScript execution. Thus, finding discrepancies between the browser parser and the WAF parser allows us to effectively construct XSS attacks that will bypass the WAF. In order to accomplish that, we used the setup described previously. However, instead of cross-checking the WAFs against each other, we cross-checked WAFs against the web browser in order to detect inputs which are successfully executing JavaScript in the browser, however they are not considered malicious by the WAF.

Table 4 shows the result of a sample execution of our system in the setting of detecting evasions. The execution time of our algorithm was about 6 minutes, in which 53 states

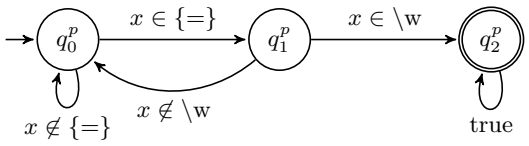


Figure 7: PHPIDS 0.7 parser (simplified version).

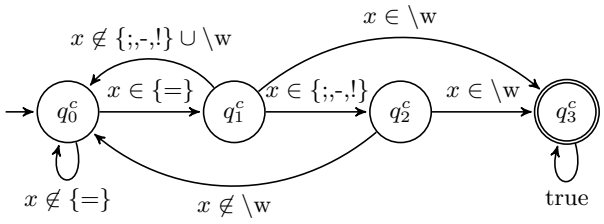


Figure 8: Google Chrome parser (simplified version).

were discovered in the browser parser and 36 states in PHPIDS. Our system converged fast into a vulnerability after improving the generated SFA models using the cached membership queries. This optimization was very important in order to correct invalid transitions generated by the learning algorithm in the inferred models. The number of invalid attacks that were attempted was 4. Each failed attack led to the refinement of the SFA models and the generation of new candidate differences. At some point the vector " " was reported as a difference by SFADIFF.

We were able to detect the same vulnerability using all major browsers and furthermore, the same problem was found to affect the continuation of PHPIDS, the Expose WAF. Finally, we point out that our algorithm also found three more variations of the same attack vector, using the characters "!", and ";".

Evasion analysis. Figures 7 and 8 shows simplified models of the parser implemented by the WAF and the browser respectively. These models contain a minimal number of states in order to demonstrate the aforementioned evasion attack. Notice that, intuitively, the cause for the vulnerability is the fact that from state q_1^p the parser of PHPIDS will return to the initial state with any non alphanumeric input, while the Google Chrome parser has the choice to first transition to q_2^c and then to an accepting state q_3^c using any alphanumeric character. For example, with an input "=!a" the product automaton will reach the point of exposure (q_0^p, q_3^c). Furthermore, using our root cause analysis, all different evasions we detected are grouped under a single root cause. This is intuitively correct, since a patch, which adds the missing state in the PHPIDS parser will address all evasion attacks at once.

5.4 Comparison with black-box fuzzing

To the best of our knowledge there is no publicly available black-box system which is capable of performing black-box differential testing like SFADIFF. A straightforward approach would be to use a black-box fuzzer (e.g. the PEACH fuzzing platform [1]) and send each input generated by the fuzzer to both programs. Afterwards, the outputs from both programs are compared to detect any differences. Note that, like SFADIFF, fuzzers also start with some initial inputs (seeds) which they subsequently mutate in order to generate more inputs for the target program. We argue that our approach is more effective in discovering differences for two reasons:

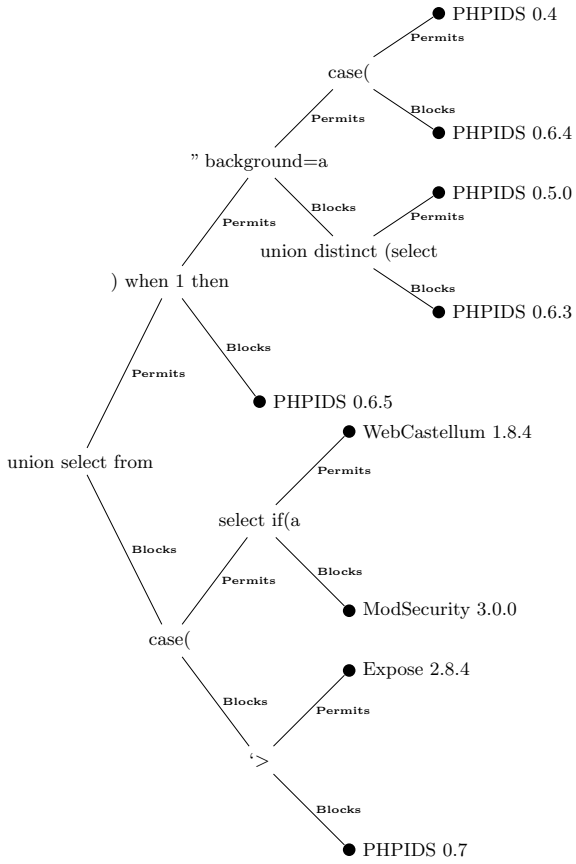


Figure 9: Fingerprint tree for different web application firewalls.

Adaptive input generation. Fuzzers incorporate a number of different strategies in order to mutate previous inputs and generate new ones. For example, PEACH supports more than 20 different strategies for mutating an input. However, assuming that a new input does not cause a difference, no further information is extracted from it; the next inputs are unrelated to the previous ones. On the contrary, each input submitted by SFADIFF to the target program provides more information about the structure of the program and its output determines the next input that will be tested. For example, in the execution shown in table 4, SFADIFF utilized the initialization model and detected the additional state in Chrome's parser (cf. figures 7, 8). Notice that, the additional state in Chrome's parser was not part of the model used for initialization. This allowed SFADIFF to quickly discover an evasion attack after a few refinements in the generated models. Each refinement discarded a number of candidate differences and drove the generation of new inputs based on the output of previous ones.

Root cause analysis. In the presence of a large number of differences, black-box fuzzers are unable to categorize the differences without some form of white-box access to the program (e.g. crash dumps). On the other hand, as demonstrated in the evasion analysis paragraph of section 5.3, our root cause analysis algorithm provides a meaningful categorization of the differences based on the execution path they

Attributes	Browser Model	WAF Model
Membership	6672	4241
Cached Membership	448	780
Equivalence	0	3
Cached Equivalence	40	106
Learned States	53	36
Cross-Check Times	4	4
Provided Browser Model	(<(p|div|form|input) onclick=a()>) (</(p|div|form|input)>)	
Vulnerability Discovered		
Execution Time	382.12 seconds	

Table 4: A sample execution that found an evasion attack for PHPIDS 0.7 and Google Chrome on MAC OSX.

follow in the generated models.

6. RELATED WORK

Fingerprinting. Nmap [17] is a popular tool for OS fingerprinting that include mechanisms for fingerprinting of different TCP implementations among other things. However, unlike SFADiff, the signatures of different protocols in nmap are manually crated and tested. Similarly, in the WAF setting, Henrqiue et al. manually found several fingerprints for distinguishing popular WAFs.

Massicotte et al. [22] quantified the amount of signature overlap assuming direct white-box access to the signature database of the analyzed programs. They checked for duplication and intersection across different signatures. However, unlike our approach here, their analysis did not involve any learning mechanism.

Automated fingerprint generation. Caballero et al. [10] designed and evaluated an automated fingerprinting system for DNS implementations using simple machine learning classifiers like decision trees. They used targeted fuzzing to find differences between individual protocols. However, Richardson et al. [25] showed that such techniques do not tend to perform as good as the hand-crafted signatures for OS fingerprinting in realistic setting. Unlike these passive learning-based techniques, we use active learning along with automata inference for systematically finding and categorizing the differences. Moreover, unlike SFADiff, none of these techniques are capable of performing automated root cause analysis in a domain-independent way.

Shu et al. [27] explored the problem of automatically fingerprinting TCP implementations. However, instead of finding new differences, they reused the handcrafted Nmap signature set [17] to create parameterized extended finite state machine (PEFSM) models of these signatures for efficient fingerprinting. By contrast, our technique learns the model of the TCP implementations without depending on any handcrafted signatures. SFADiff is able to find such differences automatically, including multiple previously-unknown differences between TCP implementations.

Brumley et al. [9] describes how to find deviations in programs using symbolic execution that can be used for fingerprinting. However, such approaches suffer from the fundamental scalability challenges inherent in symbolic execution and thus cannot be readily applied in large scale software such as web browsers.

Differential testing. Differential testing is a way of testing a program without any manually crafted specifications by comparing its outputs to those of other comparable programs for the same set of inputs [23]. Differential testing has been used successfully for testing a diverse set of systems including C compilers [32], Java virtual machine implementations [11], SSL/TLS implementations [8], mobile applications for privacy leaks [20], PDF malware detectors [31], and space flight software [18]. However, unlike us, all these projects simply try to find individual differences in an ad hoc manner rather than inferring models of the tested programs and exploring the differences systematically.

Automata inference. The L^* algorithm for learning deterministic finite state automata from membership and equivalence queries was described by Angluin [4] and many variations and optimizations were developed in the following years. Balcazzar et al. [6] provide an overview of different algorithms under a unified notation. Initializing the L^* algorithm was originally described by Groce et al. [19]. Symbolic finite automata were introduced by Veanes et al. [29] as an efficient way to explore regular expression constraints, while algorithms for SFA minimization were developed recently by D'Antoni and Veanes [14]. The ASKK algorithm for inferring SFAs was developed recently by Argyros et al. [5]. When access to the source code is provided Botinčan and Babić [7] developed an algorithm for inferring SFT models of programs using symbolic execution. The L^* algorithm and variations has being used extensively for inferring models of protocols such as the TLS protocol [26], security protocols of EMV bank cards [2] and electronic passport protocols [3]. While some of these works note that differences in the models could be used for the purpose of fingerprinting, no systematic approach to develop and enumerate such fingerprints was described.

Fiterau-Brostean et al. [15, 16] used automata learning to infer TCP state machines and then used a model checker in order to check compliance with a manually created TCP specification. While similar in nature, our approach differs in the sense that our differential testing framework does not require a manual specification in order to check for discrepancies between two implementations.

ACKNOWLEDGMENTS

The first and fourth authors were supported by the Office of Naval Research (ONR) through contract N00014-12-1-0166. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or ONR. Second and fifth authors were supported by H2020 Project Panoramix # 653497 and ERC project CODAMODA, # 259152.

References

[1] Peach fuzzer. http://www.peachfuzzer.com/. (Accessed on 08/10/2016).

[2] F. Aarts, J. D. Ruiter, and E. Poll. Formal models of bank cards for free. In *Software Testing, Verifica-*

tion and Validation Workshops (ICSTW), IEEE International Conference on, 2013.

[3] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*. 2010.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[5] G. Argyros, I. Stais, A. Keromytis, and A. Kiayias. Back in black: Towards formal, black-box analysis of sanitizers and filters. In *Security and privacy (S&P), 2016 IEEE symposium on*, 2016.

[6] J. Balcázar, J. Díaz, R. Gavalda, and O. Watanabe. *Algorithms for learning finite automata from queries: A unified view*. Springer, 1997.

[7] M. Botinčan and D. Babić. Sigma*: Symbolic Learning of Input-Output Specifications. In *POPL*, 2013.

[8] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Security and privacy (S&P), 2016 IEEE symposium on*, 2014.

[9] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Symposium (USENIX Security)*, 2007.

[10] J. Caballero, S. Venkataraman, P. Poosankam, M. Kang, D. Song, and A. Blum. FiG: Automatic fingerprint generation. *Department of Electrical and Computing Engineering*, page 27, 2007.

[11] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99. ACM, 2016.

[12] T. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[13] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[14] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *ACM SIGPLAN Notices*, volume 49, pages 541–553. ACM, 2014.

[15] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning fragments of the TCP network protocol. In *Formal Methods for Industrial Critical Systems*. 2014.

[16] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer-Aided Verification (CAV)*. 2016.

[17] Fyodor. Remote OS detection via TCP/IP fingerprinting (2nd generation).

[18] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering (ICSE)*, 2007.

[19] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. 2002.

[20] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *CCS*, 2008.

[21] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, 1977.

[22] F. Massicotte and Y. Labiche. An analysis of signature overlaps in Intrusion Detection Systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2011.

[23] W. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[24] H. Raffelt, B. Steffen, and T. Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems (FMICS)*, 2005.

[25] D. Richardson, S. Gribble, and T. Kohno. The limits of automatic OS fingerprint generation. In *ACM workshop on Artificial intelligence and security (AISec)*, 2010.

[26] J. D. Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium (USENIX Security)*, 2015.

[27] G. Shu and D. Lee. Network Protocol System Fingerprinting-A Formal Approach. In *IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[28] M. Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[29] M. Veanes, P. D. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[30] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. *ACM SIGPLAN Notices*, 47, 2012.

[31] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers a case study on PDF malware classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium (NDSS)*, 2016.

[32] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.