State of the Fuzz: An Analysis of Black-Box Vulnerability Testing

Mohammad Ghasemisharif
University of Illinois at Chicago
mghas2@uic.edu

ABSTRACT

Black-box vulnerability testing is a favored behavioral and functionality testing for finding vulnerabilities when no internal information regarding the system is available to the tester. Traditional black-box fuzzers are oblivious to the state changes generated by random and unexpected inputs which makes them not only inefficient, but inept in testing stateful applications. On the other hand, modern automated scanners have shifted towards guided input generation using state-aware testing which aims to create input samples more efficiently and estimate the state changes by utilizing the outputs as a feedback mechanism. Similarly, differential black-box testing techniques have taken state-aware approaches and evolutionary input generation into account to limit the number of generated inputs and increase the code-coverage.

This paper surveys three different black-box testing techniques, aiming to present an organized overview of the approaches which systematically improve automated black-box testing and differential fuzzing. It first provides an overview of required preliminaries and terminologies which are used throughout the paper. Next, it highlights the challenges of each technique, the problems they are aiming to solve as well as the proposed solutions and their evaluation. Finally a discussion of current issues, limitations, and a summary of future research direction are discussed.

1 INTRODUCTION

Modern web applications are built on top of intricate web technologies which are developed separately and pieced together. The added complexity has largely increased the burden on developers to prevent security bugs that are hard to detect, especially vulnerabilities stem from semantic bugs. Due to the burden of manual testing, automated testing tools are becoming an inevitable part of software testing to prevent unwanted outcomes and associated consequences resulting from unexpected inputs. In the realms of software testing, automated black-box fuzzing has advantages over white-box testing and automated code analysis, even though its limited perspective of application's internal is restrictive. In particular, black-box testing not only enables emulating the attackers point of view, it is a requisite tool when the application source code is not available. Moreover, white-box testing requires additional skill and knowledge to analyze the code which makes it domain specific and not adaptable to other applications (with different programming languages) as opposed to black-box testing which only relies on input/output analysis. Therefore, an efficient black-box fuzzer is applicable to variety of applications as it is invariant to a particular programming language or system.

While black-box testing has a simpler perspective and separates users point of view from developers, they are generally inefficient [17]. Their inefficiency mainly originates from lack of knowledge regarding the application's internal state which leads to generating a considerable amount of input samples for finding a single bug. In fact, black-box scanners do not perform well in detecting *stored* XSS [2] and modeling application semantic as well as increasing observability might improve the detection of such vulnerabilities [8]. The lack of knowledge encompasses the absence of an oracle which defines the expected behavior (output) of the application. Differential testing [18] is a well-known method for filling the void of such oracle by testing functionality of similar programs as cross-referencing oracles. While numerous studies have been conducted to increase observability of a black-box through model inference techniques [12] [20] [7] [22] and differential testing [21] [6], there is a trade off between domain dependency and efficacy.

This paper aims to provide an insight into three different techniques of improving black-box fuzzing. While each approach tackles the problem differently, they are similar in terms of attempting to increase observability (knowledge) by means of building a model in a black-box setting. The crux of this paper revolves around black-box testing, with a primary emphasis on providing details of how each method infers a model. However, additional information regarding grey-box testing is provided in Section 3.3 for comparison.

The rest of the paper is organized as follows: Section 2 introduces terminology used throughout the paper. Section 3 describes three methods of improving black-box fuzzers, followed by evaluation of each technique. Section 4 provides a comparison of described techniques and discusses their limitations. Finally, Section 5 concludes the paper.

2 DEFINITIONS

This section aims to provide required preliminaries and necessary terms for the discussion of subsequent sections.

Testing strategies. Three main type of strategies exist in application testing and their use case can differ according to the testing condition. In *black-box* testing, the system is viewed as a black-box which information regarding the system internals and code structure is not available to the tester. However, the tester has enough access to probe the system with queries and observe the outputs. On the other hand, the application's internal structure, processes and functionalities are fully transparent in *white-box* testing. Even though transparency yields a more accurate characterization of bugs, it requires the tool to be domain-dependent and specific to one framework. Finally, *grey-box* testing is a combination of both prior techniques in which the tester is not completely oblivious to application's internal, but her knowledge is limited to application's functionalities i.e. application's source code is not available but its functionality is known. Black-box and grey-box testing have

gained wide popularity due to the unavailability of application's source code in real world scenarios. This paper primarily focuses on black-box testing techniques.

Fuzz testing. Fuzzing is a software testing method which an automated program feeds the target application with a massive amount of malformed and unexpected inputs, and then the application is monitored in order to hopefully find potential software bugs. In the scope of this paper, software bugs are classified in two main categories: *memory-related* and *semantic* bugs. Typically, memory-related bugs display explicit output which makes them easier to find. In contrast, semantic bugs do not show clear incorrect behavior or message and are harder to detect. Section 3.3 provides further insight regarding semantic bugs and their corresponding testing technique.

Input generation. Fuzzers can fall into different categories based on their input generation method which can directly impact fuzzer's efficiency. *Mutation-based* [22] fuzzers generate inputs by mutating existing input samples. *Generation-based* [22] fuzzers learn the models of input format and generate new inputs based on the learned models. *Evolutionary* testing [19] follows a more efficient approach, also known as *guided testing* or *adaptive input generation*, that attempts to generate inputs using the output responses and application's behavior. In contrast, input generation in *unguided testing* does not depend on the received output, nor the changes in the application states.

Root cause analysis. Identifying the root cause of an observed bug for further mitigation or exploit development. The difficulty of such task depends on the information availability and testing strategy (e.g. crash dumps in white-box testing).

Differential tesing. A testing method that feeds the same set of inputs to multiple test applications, which have similar functionalities but different implementations, and looks for asymmetries between their behaviors to find bugs [18]. In part, differential testing tries to solve the problem of needing an oracle by using a series of similar applications as cross-referencing oracles. Section 3.2 and 3.3 provide approaches, challenges and applications of this method.

Application fingerprinting. Uniquely identifying an application (in a black-box experiment) based on its observed behavior. First, the application's unique feature set (behavior) is learned and mapped to a fingerprint, and then during the identification process, if a similar behavior is seen, the corresponding fingerprint is returned as the potential application. Successful fingerprinting can reveal sensitive information such as device type (e.g. firewall) or known vulnerabilities.

Symbolic Finite Automata. Finite state machines where several transitions from one state to the target state are combined and replaced with a symbolic move (predicate). Predicates act as transition guards. An example of a Symbolic Finite Automaton is shown in Figure 3. If the application can be modeled with SFA and the alphabet size is very large (such as UTF-16), SFA outperform classical automata. The main SFA applications are regex processing and sanitizer analysis [25]. While the output of SFA symbolic moves are binary, for application with non-binary output, SFA can be extended to *Symbolic Finite Transducers* [11]. Formal definitions of automata are provided in the Appendix.

Cross-site Scripting (XSS). A type of web application vulnerability which enables an attacker to inject malicious scripts into a (trusted) web application. The malicious script is then run in another user's browser once she visits the trusted web application without raising suspicion [1]. One defense mechanism against XSS attacks is using Web Application Firewall (WAF) [4]. As it is discussed in Section 3.2, preventing evasion attacks against WAF is crucial since a single logic error in the filter can eventually lead to code execution. Generally, the alphabet size in regex based filter (WAF) is large which is problematic for a typical fuzzer. A solution to this problem is explored in Section 3.2.

3 TESTING METHODOLOGIES

This section explores various applications of automated black-box testing in web vulnerability scanners and differential testing. Each subsection outlines a brief overview of the problem and its systematic improvement technique.

3.1 State-Aware Web Vulnerability Scanner

A web application is a stack of multi-layer technologies that are developed separately and assembled. The complex nature of the web has made the testing process more cumbersome, and the application's code less understandable. In fact, vulnerable applications are more complex and have a higher code churn than neutral ones [23]. Thus, automated testing as a complementary technique has become a crucial part of software testing. To facilitate the testing process, a number of automated web vulnerability scanners have been developed by academic community [14] [15] [16]. Nevertheless, due to the black-box nature of the server-side applications, the proposed solutions suffer from low accuracy and partial code-coverage [12] [13]. Proposed technique by Doupé et al. [12] overcomes this shortcoming by automatically inferring web application's internal state machine and understanding state changes to maximize code-coverage.

3.1.1 ***Problem Description***. A typical web application vulnerability scanner has two main parts, a crawler which interacts and navigates between different pages/parts of the application and a fuzzer that tests those visited parts. Interactions with a web application can move its current state onto a different state and such transitions cannot be captured by a navigation-based crawler, and thus part of the web application remains completely untouched by a navigation-based fuzzer. Consequently, it is necessary for the crawler to be aware of application's state given the opaqueness of application's internal structure. Such inference must be done by only using requests and responses to and from the application. In this section, application's *state* refers to server-side code execution. Figure 1 illustrates difference coverage in navigation-based and state-aware crawler.

3.1.2 ***Inferring State Machine***. Since the server-side code is viewed as black-box, any modeled state machine must be inferred by making HTTP requests and observing corresponding responses. The proposed solution by [12] uses four components to determine

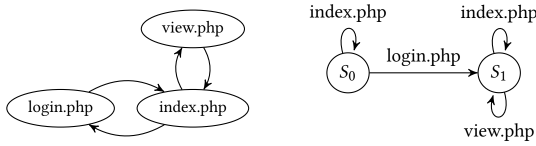

Figure 1: Example of navigation graph (left) and application's state machine (right) [12]. As opposed to the navigation graph, state machine captures the state transition caused by `login.php` request.

state changes and learn a minimized state machine for a web application.

Clustering similar pages. HTML pages are clustered based on their *link structure similarity* in order to prevent infinite scanning (e.g. calendar pages) as well as detect state changes. Each page contains a set of links (anchors and forms) which acts as an interface for interacting with the web application, hence navigating between page clusters provides further information regarding the state change. First, page links are constructed in form of ⟨*dompath, action, params, values*⟩ vectors and stored in a prefix tree. Each prefix tree represents *one page* with multiple links which each link (vector) starts from the root and ends in a leaf node. Next, prefix trees are merged together based on their similarities to form an *Abstract Page Tree (APT)*. Similarity measurement in APT is based on the number of shared elements from the beginning of link vectors (at each level of prefix tree). Finally, APT's subtrees are merged together to form an *Abstract Page*. Page links in each *Abstract Page* (subtree of depth *n*) share similar *dompaths*, have greater number of leaves than the median of their siblings, and have at least $8 \times \left(1 + \frac{1}{n+1}\right)$ leaves. In case of HTTP redirection, a special redirect element is considered for HTTP redirects and the location (target) URL is assigned for its value. An example vector representation of an anchor tag with `href` of `/user/profile.php?id=0&page` is ⟨`/html/body/div/span/a`, `(/user,profile.php)`, `(id,page)`, `(0)`⟩. Figure 2 shows the corresponding prefix tree and Abstract Page Tree.

Detecting state changing request. To an external observer of a black-box web application, if two identical requests generate different responses, the state of the application must have changed. In order to locate the request which has altered the state, a heuristic approach rates HTTP requests between these two identical requests. Suppose R and R' are two identical requests which $R' \prec R$. The following score function calculates the score of each request (i) between R and R':

$$Score(n_{i,transition}, n_{i,seen}, distance_i) = 1 - \left(1 - \frac{n_{i,transition}+1}{n_{i,seen}+1}\right)^2 + \frac{BOOST_i}{distance_i+1}$$

The score function operates based on the number of times request i: caused a state transition ($n_{i,transition}$), has been seen ($n_{i,seen}$), and the aggregate number of requests between i and R ($distance_i$). The variable $BOOST_i$ is `0.2` and `0.1` for `POST` and `GET` requests respectively because `POST` requests are more probable to alter a state. Finally, the candidate request that maximizes the score function is returned as the request which has changed the state.

Collapsing of similar states. Similar states in the inferred model must be collapsed to not only minimize the state machine, but also detect transition to a previously visited state. This problem can be addressed using a graph coloring approach. At first, all states are unique. Next, edges are placed between separate states using the information of previously collected state changing requests and page clusters. Two separate states are colored differently (not collapsed) if they do not share at least one page or they have identical requests which land in different page clusters. Finally, if a matching request R occurs in two different states (a,b) and both land to other separate states (c,d), an edge will be added between initial states (a,b) and graph coloring will be repeated.

Navigating the crawler. Modeling the state machine not only depends on *what*, but *how* the requests are sent. Since each request can cause a state change, requests need to be sent *sequentially* and not *concurrently*. Requests are selected from the pool of links in the last visited page. In case there are no unvisited links left, the crawler selects a path to another page which has unexplored links. The path selection method aims to maximize exploration of each state before moving to another one.

3.1.3 ***Evaluation***. While *false positive rate* is a main performance metric in vulnerability scanners, assessing performance of state-aware crawler primarily hinges on *code coverage* percentage. To evaluate false positive rate, the authors modify `w3af` [3], a well-known open-source vulnerability scanner, to make it state-aware. The performance of state-aware `w3af` is then measured against its original *state-unaware* version. This measures the improvement, if any, of added *awareness*. Moreover, since the state-aware crawler uses both `POST` and `GET` requests, the recursive mode of `wget` [5] (only `GET`) is used as the baseline for *unaware* crawling. Although the experiment reports substantial code coverage increase (up to 140.71 percent over `wget`) and less false positive (compared to state-unaware `w3af`), the proposed solution can not perform well in *Single Page Application (SPA)* where Ajax is the predominant request type and contents are dynamically updated with each interaction. Finally, the sequential crawling assumes that the application is under influence of one user, thus, as authors note, the proposed paradigm cannot perform correctly in a multi user application where different users can impact the states simultaneously.

3.2 Differential Automata Learning

While previous technique improves code-coverage and their method of inferring state machine can be extended to applications other than web apps, the approach is not scalable in circumstances where the number of possible inputs is huge. Increasing the scalability requires another form of finite state machines called *Symbolic Finite Automata (SFA)* [26] which multiple state transitions are combined and replaced by symbolic moves (predicates). Furthermore, the emphasis of previous method was increasing code coverage rather than detecting incorrect behavior in response to random inputs. One popular setting for black-box fuzzing is differential testing. In differential testing, the fuzzer leverages differences in a series of similar application to find semantic bugs. Since these applications are the same in terms of functionalities but have different implementations, they can be

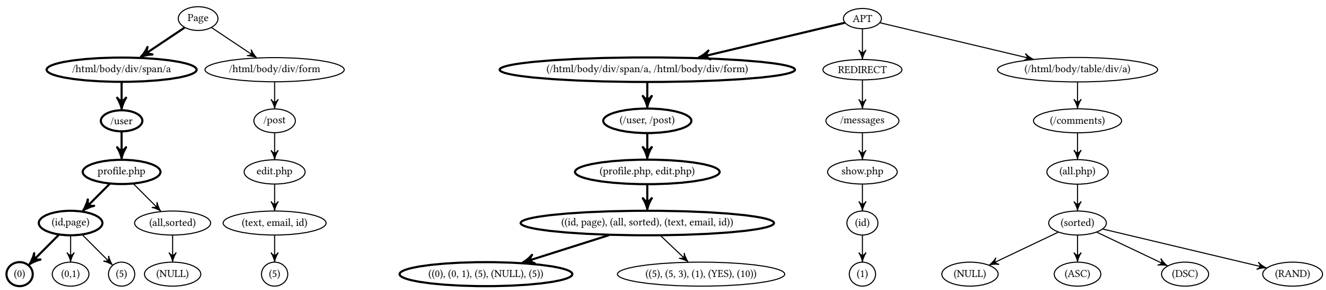

Figure 2: Example of prefix tree (left) and corresponding Abstract Page Tree (right) [12]. Moving from the root to a leaf in prefix tree specifies a link in a page, while in APT, it represents a page.

used as cross-referencing oracles for what is considered as deviation of correct behavior. This section presents another technique proposed by Argyros et al. [6], called SFADiff, to improve black-box testing using differential automata learning. Furthermore, it provides two attacks by leveraging discrepancies found in differential testing.

3.2.1 ***Problem Description.*** In testing a regex-based filter or a string sanitizer, where the number of possible inputs (alphabet size) is large and fuzzing purpose is to find a set of inputs that bypasses certain rules, testing all input variations is not feasible. For instance, the alphabet size of practical regular expression which its characters are represented by 16-bit bit-vectors (UTF-16) is 2^{16}, and classical automata are unable to efficiently model over such alphabet size [11]. In contrast, discussed state-aware crawler (Section 3.1) uses a small number of page links to build application's state machine and such approach clearly is not adaptable to the current problem (e.g. testing string sanitizer). Therefore, classical automata are not applicable to infer application's state machine where the alphabet size is considerable and another approach must be taken into account. Moreover, having a ground truth enables the fuzzer to guide the input generation towards samples that are more likely to find bugs (Section 3.3). Unfortunately, such ground truth is not available in a typical black-box setting. Nevertheless, as discussed in Section 2, one way of overcoming this difficulty is differential testing where the ground truth comes from the majority of observed behaviors (or outputs) in testing similar applications. The following subsections explain SFADiff's solution to the described problem.

3.2.2 ***SFA Learning.*** *Symbolic Finite Automata* (SFA) solve the alphabet size problem by combining multiple state transitions and replacing them with a predicate, which drastically reduces the number of transitions between states. SFADiff uses automata learning which entails active learning through querying a program (as a black-box) and comparing inputs and outputs to derive a SFA model of the program. Automata learning employs two type of queries to learn an exact model of the automaton: *membership* and *equivalence query*. Suppose an unknown automaton M belongs to a program and its accepted language is $\mathcal{L}(M)$. The learning algorithm is allowed to submit a string s as a membership query and obtain the result whether $s \in \mathcal{L}(M)$. Similarly, the algorithm can submit a hypothesis H as a finite automaton through a single equivalence query and check whether $\mathcal{L}(H) = \mathcal{L}(M)$. In case that $\mathcal{L}(H) \neq \mathcal{L}(M)$, a counterexample will be returned in which the algorithm uses to refine the obtained model. Even though equivalence query, in abstract, is a single membership query, in practice these queries are implemented by exhaustive search for finding counterexamples. Thus, reducing equivalence queries can significantly decrease the overhead of automata learning. SFADiff leverages model bootstrapping to speed up the learning process. The intuition behind bootstrapping lies in the assumption that different versions of a program share certain parts in the model, thereby by learning an old model and incrementally improving it, the learning algorithm is able to derive the correct model with less equivalence queries. SFADiff uses SFA learning algorithm presented in [7] to efficiently implement equivalence queries and bootstrapping to attain the SFA-based model of a (black-box) program.

Differential SFA testing. While having a model representation for a program improves understanding of the program internals to a certain extent, detecting bugs requires an oracle to define the expected correct behavior. In absence of such oracle, SFADiff tests multiple programs in a differential testing setting. In this setting, programs with similar functionalities but different implementations are selected and their models are learned using the SFA learning technique. As opposed to a typical black-box differential testing method which looks for differences in the inputs/outputs, SFADiff looks for model discrepancies. After learning SFA model of each program, intersection of models are computed and differences between models are extracted using the following intuition: if processing an input by product automaton (intersection) of two models results in a state in which its corresponding states in each model have different labels (e.g. input in one automaton reaches a final state, in the other does not), the input is a candidate for the difference and the corresponding state in product automaton is *a point of exposure*. However, before reaching such conclusion, the input must be tested with the actual program for its correctness to ensure that it is not caused by model inaccuracy. If the input is a false positive, which means that the program's model and the program does not agree on the same output, it is then utilized as a counterexample for refining the model; otherwise, it is collected for analyzing the root cause.

Root cause analysis. After collecting points of exposure, those with matching simple paths (no loops) are classified as the root

cause of a discrepancy. In other words, if multiple inputs are detected that cause discrepancies, their simple (execution) path is traced in the corresponding SFA model and the ones that start from the same initial state and end in the same target state are grouped and labeled as the root cause. This categorization allows better understanding of the root cause considering the circumstance which the source code of the program is not given. This section further explores how discrepancies between models can be exploited in two forms of attacks.

SFADiff covers both cases where two programs or two sets of programs are differentially tested. Suppose two sets of programs $\mathcal{I}_1 = \{P_1, P_2, ..., P_n\}$ and $\mathcal{I}_2 = \{P_1, P_2, ..., P_m\}$ are given and each program's output is a bit $b \in \{0, 1\}$. Differential analysis attempts to find (a set of) inputs s such that:

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

Such set of inputs can be exploited in the following attacks:

a *Evasion Attack.* In a simple setting where two programs such as a Web Application Firewall (WAF) and an HTML/Javascript parser of a browser are differentially tested, each application decides whether an input string is an executable Javascript: the WAF must block Javascript execution to prevent XSS attacks (Section 2), but the browser's HTML/Javascript parser function is to accept and execute a (Javascript) string. Regardless of the outputs, the underlying parsing logic is the same in both applications since they both determine the *executability* of an input, thereby input strings that are accepted by browser's HTML/Javascript parser but are not flagged as malicious by WAF can eventually *evade* the firewall. Figure 3 depicts inferred SFA models of PHPIDS 0.7 parser (WAF) and Google Chrome's HTML/Javascript parser by SFADiff. As an illustration, processing substring "=-a" of string "<p onclick=-a()></p>" reaches the final state (q_3^c) in Google Chrome's parser (executable) while it returns to the initial state (q_0^p) in PHPIDS 0.7 (not malicious).

b *Application Fingerprinting.* Given a set of known applications $\mathcal{I} = \{P_1, P_2, ..., P_n\}$ and an unknown (black-box) target application P_t, application fingerprinting aims to pinpoint $P \in \mathcal{I}$ where $P_t = P$ by only querying P_t. SFADiff utilizes differences of programs ($\mathcal{I}$) as their unique fingerprints which later serve as distinguishing factors in application fingerprinting. SFADiff's fingerprinting technique has two main steps: 1) building a fingerprint tree and 2) searching for a program in the fingerprint tree. Two versions of a fingerprint tree are proposed by SFADiff. The simpler version selects two arbitrary programs from $\mathcal{I}$ at each iteration, obtains their distinguishing factor (strings), and stores the strings in a tree node. Next, it passes ($\mathcal{I} - \{rejected\, program\}$) to the left subtree and ($\mathcal{I} - \{accepted\, program\}$) to the right subtree and recursively does the same steps on the remaining programs till only one program is in each leaf. The complex version follows the same approach, but instead of comparing two programs, it compares and stores the differences of two program subsets at each node where the k program subsets are defined as $\mathcal{I}_s = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_k\}$. Therefore during the fingerprint search, the simple version eliminates one program at each level and the target application can be identified by sending $|\mathcal{I}| - 1$ queries, whereas the complex version can locate the target program with $(k-1)\log_k |\mathcal{I}|$ queries (e.g. $\log |\mathcal{I}|$ for $k = 2$). While the number of queries is reduced, the complex fingerprint tree requires more computation time for building the tree, and as authors suggest, it is suitable for an attacker who can invest more time in offline computation.

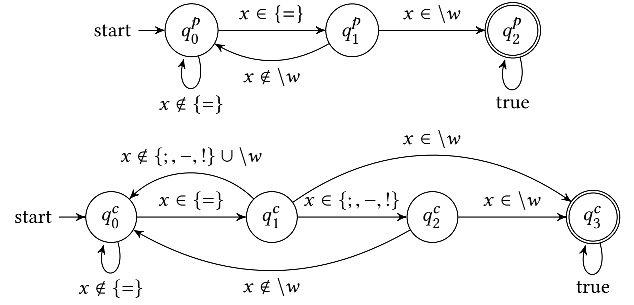

Figure 3: Simplified version of PHPIDS 0.7 parser (top) and Google Chrome parser (bottom) [6].

3.2.3 ***Evaluation.*** SFADiff's performance is measured in three criteria: 1) effectiveness of bootstrapping, success rate of 2) fingerprinting and 3) evasion attack. To assess bootstrapping performance, the number of equivalence queries are measured in learning the SFA model of 9 regular expression filters (2 versions of ModSecurity and 7 versions of PHPIDS) with and without bootstrapping technique. Even though the reported result illustrates 50× reduction in number of equivalence queries, the number of membership queries shows an increase of 1.15×. The authors claim that equivalence queries are much slower than membership queries in general, and thereby the result shows a significant improvement. In another experiment, the effectiveness of fingerprinting technique is evaluated by inferring and comparing the SFA model of TCP implementations in Linux, OSX and FreeBSD. Having extra number of states in Linux and FreeBSD (for handling erroneous TCP packets) compared to OSX as well as returning different outputs to the same input samples, act as distinguishing factors in operating system fingerprinting. To measure success rate of evasion attack, the Javascript parsing implementation of a WAF (PHPIDS) is differentially tested and cross-checked against a browser (Google Chrome). Figure 3 demonstrates the learned SFA models. As discussed previously, since Javascript parsers are structurally similar, if there is such a string which gets accepted by both models, it creates an evasion attack against the WAF. "=!a", "=-a" and "=;a" are reported as bypassing PHPIDS which are then classified under one root cause using the discussed automated root cause analysis technique.

Ordinarily, differential testing is a suboptimal approach since it relies on the majority vote of outputs (behavior) to fill the void of an oracle. In particular, a semantic bug caused by an incorrect design could prevail in the successive implementations without creating discrepancies. Nevertheless, SFADiff takes a different approach and looks for model differences rather than mere outputs. Moreover, automata learning to some extent converts black-box testing to a grey-box analysis which increases the amount of knowledge regarding the internals and changes the inherent lack of knowledge assumption in black-box fuzzing. While SFADiff's technique can

be defined as domain-independent, the fact that SFA learning is applicable to certain applications such as regex based filters and string sanitizer makes the approach limited to specific domains. The following section expounds a domain-independent method in black-box testing.

3.3 Domain Independent Differential Testing

As detailed, both previous approaches attempt to build a model of black-box application from inputs and outputs. Nonetheless, they are not completely domain agnostic. The state-aware crawler (Section 3.1) leverages prior knowledge of HTML page link structure and input-output-format (HTTP request/responses) to infer web application's state machine. While SFADiff (Section 3.2) is domain-independent, its learning technique is only pertinent to applications which can be modeled by SFA learning. Therefore, they are not adaptable to other domains. This section presents another approach suggested by Petsios et al. [21] that focuses on *evolutionary* input generation to push differential black-box testing further towards being domain agnostic. Although the proposed paradigm supports grey-box and black-box differential testing, the emphasis here is on the black-box aspect and the grey-box detail is provided for comparison.

3.3.1 ***Problem Description.*** Semantic bugs are the dominant root cause of software bugs compared to memory-related bugs [24]. They are very hard to detect since their deviation of correct behavior does not result in a crash. Moreover, merely expanding fuzzer's code-coverage does not guarantee an increase in bug detection. For instance, in *monolithic* code-coverage technique which code-coverage *percentage* is the main drive, inputs which generate different *behaviors*, but cover less code percentage will be ignored. Therefore, monolithic code-coverage is not an adequate metric, especially in semantic discrepancy testing where understanding application behavior plays a critical role. Essentially, finding semantic bugs requires domain specific knowledge, manual inspection of the program as well as an oracle which defines the correct behavior. This is where differential testing shines, since such oracle, or any knowledge about the application, is not available in black-box testing.

As seen in Section 3.2, differential testing uses the majority vote as the oracle's answer to *"What should be the program's behavior to a specific input?"*. However, existing differential testing methods face limitations in finding semantic bugs. First, they follow *unguided testing* method which does not include previous outputs in guiding future input generation. This results in probing the program with a large number of random inputs and hoping to find a single bug. Second, they rely on specific input format which cannot be adapted to other domains. Csmith [27] is an example of domain specific differential testing tool which uses randomized test-case to fuzz C compilers. The rest of this section is dedicated to describing NEZHA [21], a domain-independent differential testing technique, and its applications.

3.3.2 ***Evolutionary Input Generation.*** NEZHA uses a novel metric called *δ-diversity* to quantify behavior asymmetries caused by an input in differential testing. In other words, in differential testing where a series of similar applications is being tested and observed for behavior asymmetries, this metric aims to capture the diversity of those asymmetries. Such metric is further used to direct input mutation towards exploring parts of the application which exhibit more diverse behavior. As monolithic code-coverage lacks in capturing divergent behavior, δ-diversity is the key metric in NEZHA for refining input generation.

Algorithm 1 DiffTest takes applications $\mathcal{A}$, collection of inputs $\mathcal{I}$ and *n* generations as arguments and returns discrepancies of tested applications. *GlobalState* specifies guidance engine method [21].

```
1:  procedure DiffTest(I, A, n, GlobalState)
2:      discrepancies = 0 ;reported discrepancies
3:      while generation ≤ n do
4:          input = RANDOMCHOICE(I)
5:          mut_input = MUTATE(input)
6:          generation_paths = 0 ;for path δ-diversity
7:          generation_outputs = 0 ;for output δ-diversity
8:          for app ∈ A do
9:              app_path, app_outputs = RUN(app, mut_input)
10:             generation_paths ∪ = {app_path}
11:             generation_outputs ∪ = {app_outputs}
12:         end for
13:         if NEWPATTERN(generation_paths,
                          generation_outputs,
                          GlobalState)          then
14:             I ← I ∪ mut_input
15:         end if
16:         if IsDISCREPANCY(generation_outputs) then
17:             discrepancies ∪ = mut_input
18:         end if
19:         generation = generation + 1
20:     end while
21:     return discrepancies
22: end procedure
```

NEZHA's main algorithm rests on how inputs are generated. As discussed In Section 3.1, the state-aware crawler selects upcoming HTTP requests in such a way to maximize exploration of current state before leaving the state. Similarly, SFADiff (Section 3.2) drives input generation towards refining inferred automaton and minimizing the difference of the obtained model from its target application. In the context of differential fuzzing and NEZHA, semantic bugs are more likely to be found when the behavior asymmetries are maximized. Therefore, NEZHA's *guidance engine* generates upcoming inputs in the direction of maximizing δ-diversity. Computing δ-diversity can vary based on information availability (of testing strategy). The followings describes the role of δ-diversity in the guidance engine of grey-box and black-box testing.

Path δ-diversity (grey-box). In the context of Control Flow Graph (CFG), the execution path is defined as the sequence of accessed CFG edges as a result of executing a specific input. Since information regarding the execution path is accessible in a grey-box setting, the guidance engine primarily aims to generate inputs in a direction to explore as many execution paths as possible. Correspondingly, when multiple programs are examined in a differential testing setting, the execution path representation must be able to

capture unique execution paths for each program. As discussed in Section 3.3.1, global monolithic code-coverage lacks in quantifying behavior asymmetries. NEZHA has two representation methods with different granularity to tackle this problem: *coarse* and *fine*. In the first method, the *total number of unique accessed edges* for *each program* is stored in a tuple called *Path Cardinality*. The size of the tuple is equal to the total number of programs in the differential testing, and each entry belongs to one program. For instance, given a set of $\mathcal{P}$ programs and a set of $\mathcal{I}$ inputs, cardinality tuple $PC_{\mathcal{P},i}$ in response to input $i \in \mathcal{I}$ is $\langle |path_{p_1,i}|, |path_{p_2,i}|, ..., |path_{p_{|\mathcal{P}|},i}| \rangle$ which each program $p_k \in \mathcal{P}$. Finally, when all inputs are tested, *coarse* path δ-diversity is obtained from $PD_{Coarse} = |\bigcup_{i \in \mathcal{I}} \{PC_{\mathcal{P},i}\}|$. However, it is unclear which edges are accessed since the emphasis of *coarse* method is the total number of unique accessed edges. In contrast, *fine* technique stores *sets of unique edges* under execution of each input. Put differently, instead of counting total number of accessed edges, *fine* method stores accessed edges of each program as a set (with no duplicates). Thus, the final $PD_{\mathcal{P},i}$ tuple for *fine* method is $\langle path_set_{p_1,i}, path_set_{p_2,i}, ..., path_set_{p_{|\mathcal{P}|},i} \rangle$ which $path_set_{p,i}$ is a set of unique accessed edges of program $p_k \in \mathcal{P}$ under execution of input $i \in \mathcal{I}$. Consequently, *fine* path δ-diversity is computed from $PD_{Fine} = |\bigcup_{i \in \mathcal{I}} \{PD_{\mathcal{P},i}\}|$.

Output δ-diversity (black-box). A Black-box fuzzer has a limited perspective about the application's internal such as path execution, thereby δ-diversity must be computed from information other than the path execution. As discussed in Section 2, input and output information are available to the fuzzer. NEZHA measures δ-diversity using received errors under execution of a specific input. Given a set of programs $\mathcal{P}$ and a set of test inputs $\mathcal{I}$, executing input $i \in \mathcal{I}$ on each program $p_k \in \mathcal{P}$ results in a tuple of program outputs $OD_{\mathcal{P},i} = \langle o_{p_1,i}, o_{p_2,i}, ..., o_{p_{|\mathcal{P}|},i} \rangle$. Accordingly, input generation uses $|\bigcup_{i \in \mathcal{I}} \{OD_{\mathcal{P},i}\}|$ as the output δ-diversity for the guidance of creating upcoming input samples to maximize diversity of outputs. Using δ-diversity as a tunning parameter allows controlling input generation without taking input format into account. However, as the diversity quantification in output δ-diversity relies on the observed differences between the outputs (errors, messages, etc.), it is not useful in testing applications where outputs are not adequately expressive or diverse. Therefore, granularity of outputs must also be considered.

NEZHA's main algorithm is shown in Algorithm 1. It describes evolutionary testing and how inputs are evolved using the guidance engine. The algorithm runs for n generations and at each generation, an input is randomly selected from the input collection $\mathcal{I}$ and mutated. The mutated input is then run against all applications ($\mathcal{A}$) and corresponding outputs are collected. If the output is a new unobserved pattern (using δ-diversity concept), the mutated input will be added to the input collection for the upcoming tests. $GlobalState$ defines δ-diversity type (e.g. output δ-diversity is $GlobalState.UseOD$). Finally, the mutated input is added to the *discrepancy* set if it is accepted by at least one application and rejected by at least another application. When n generations of testing are done, the *discrepancy* set is returned as the final result.

NEZHA uses delta-debugging [29] for analyzing root cause of discrepancies which requires keeping both the original input and the mutated one, as the comparison of inputs and their mutated version helps in identifying the cause. It should be noted that the mutation strategy selects predefined mutation techniques randomly (e.g. combining random substrings), and as the authors note, it is not fit for differential testing and requires improvement.

3.3.3 ***Evaluation***.

The experiments aim to evaluate NEZHA's performance in three parts. First, its ability to find discrepancies. Then, its performance compared to other domain-specific and domain-agnostic fuzzers, and third, to examine its guidance engine performance compared to each other. The first differential fuzzing is done in three areas: SSL libraries, file parsers, and PDF viewers. Overall, 778 unique discrepancies are found and the most significant portion belongs to testing six different SSL libraries (764) which, as authors explain, stems from finer granularity of error outputs as well as larger number of tested applications (compared to two file parsers and three PDF viewers). Although the authors identify the occurrence of 8 errors and crashes, it is not clear how many of the remaining discrepancies belong to semantic bugs. In the second set of experiments, the number of found discrepancies are 52, 27 and 6 times more than Frankcerts [9](domain-specific), Mucerts [10](domain-specific) and American Fuzzy Loop [28](domain-agnostic) respectively, which shows a promising improvement. Finally, in a similar setup of testing six different SSL libraries, output δ-diversity (black-box), path δ-diversity (grey-box), and global code-coverage are run separately and results are compared. Reported results show 30% improvement in black-box, and 22.75% improvement in grey-box over global code-coverage. Seemingly, in a differential testing context, if the outputs are sufficiently diverse and expressive, black-box testing with adaptive input generation can perform to the same extent as grey-box testing. This is illustrated in another experiment where a subset of error code is returned (where errors are less diverse) which shows a drastic reduction in performance of output δ-diversity.

As reported, tracking δ-diversity throughout testing allows driving input selection towards exploring parts of the application which would have been ignored had the fuzzer merely followed a monolithic code-coverage. However, the notion of differential diversity relies on the majority of tested programs to perform correctly as intended, otherwise discrepancies are erroneously labeled as bugs or bugs are simply overlooked and considered as correct behavior. Moreover, as authors clarify, mutation methods used in NEZHA are not optimized for differential testing and further optimization is required. Finally, while automatic bug localization and reporting are included in NEZHA, they may not scale well as the number of programs increases and manual analysis is still needed.

4 DISCUSSION

Taking a wider view on the matter, comparison of described techniques provides further evaluation on the subject as a whole. While discussed methodologies share a similar goal, the nuances of how models are derived and what information is required beforehand can differentiate techniques. Moreover, since the primary advantage of black-box fuzzing is it is not application specific, it is therefore imperative to juxtapose the techniques and discuss their adaptability in other domains.

Domain dependency. Even though the state-aware vulnerability scanner (Section 3.1) is (server-side) platform-agnostic, it relies on a specific input format to interact with the back end server. The input format is also tied to page link structure as the construction of Abstract Page Tree and clustering similar pages utilize link tuples and links which follow a specific standard. In contrast, NEZHA (Section 3.3), regardless of input format, quantifies a notion of diversity in the outputs (δ-diversity) and uses such measure as an input generation guidance. The achieved results by NEZHA illustrate that evolutionary testing can reduce input format dependency, though the context of differential testing and using cross-referencing oracle must also be taken into account. SFADiff (Section 3.2) takes an in-between approach; while the input structure is relatively domain-independent, the category of target programs that are *learnable* by SFA is specific and limited (e.g. regex based filters). Despite the differences of reliance on input format, the primary goal of all techniques is reducing code-dependency, increasing adaptability even in particular areas, while not losing performance.

On the contrary, output format dependency has less impact on adaptability since discrepancies in the output is considered as the distinguishing factor (of divergent behavior), especially when multiple programs are differentially tested. The state-aware crawler compares the responses of identical requests, regardless of the format, to infer when the state of the application changes. In a similar manner, NEZHA guides input generation to diversify outputs since heterogeneity is a key factor in its ability to increase code-coverage. Actually, as noted in Section 3.3, its performance depends on diverse and expressive outputs. While SFADiff's outputs, in particular the counterexamples, play an important role in SFA learning, its differential testing performance is determined by observed output differences (e.g. different states).

Root cause analysis. Locating root cause of observed discrepancies in a black-box differential testing setting is difficult. SFADiff's root cause analysis is able to group similar causes automatically, as seen in Section 3.2, whereas NEZHA's black-box guidance engine requires additional manual analysis. SFADiff's promising approach in bug localization is a result of learning an accurate model of a black-box. Thus, tracing paths in such model is analogous to following path execution in the application which is then used to locate the bug initiator. The state-aware vulnerability scanner does not specify a debugging strategy as it merely attempts to increase code-coverage.

Limitations. The following paragraph summarizes discussed limitations throughout the paper. The state-aware crawler underperforms in Single Page Applications and multi-user web applications where multiple users can affect application's internal state. In order for state-aware crawler to successfully infer a state machine of Ajax-based web application (or Single Page Application), the authors recommend converting Ajax responses (JSONs) into representative static pages. Furthermore, NEZHA's input mutation strategy, which uses up to five predefined operators, requires optimization for a differential testing setting. Finally, in general, bug localization is a difficult task in black-box differential testing due to the large number of generated discrepancies and opaqueness of execution paths, and as explained by authors, it needs further improvements in NEZHA.

Differential testing vs. state-aware vulnerability scanner. While NEZHA and SFADiff are not comparable to the state-aware vulnerability scanner as their contexts are not the same, few points are worth mentioning. Typically, differential testing is performed in a setting where the functionality of applications (that are being tested) are known, otherwise observed differences do not provide any additional information. However, such setting is not applicable while testing a target application on the web. As noted in Section 3.2, most gain in differential testing comes from offline computation, specially in generating fingerprints. Therefore, differential fuzzing techniques are intrinsically different than what is proposed by state-aware vulnerability scanner, though they may share a similar goal. Despite the differences, each mentioned technique considers a certain degree of knowledge (assumptions) about the application: the state-aware crawler takes input format into account, SFADiff's adaptability is dependent on the application type, and NEZHA's success (in black-box guidance engine) is tied to having diverse and expressive outputs. These assumptions play key parts in designing an efficient black-box fuzzer. Ultimately, all three methods follow the same objective: *increasing observability.* This not only indicates the imperative role of understanding application's internal states, but the viability of such inference by mere input/output analysis.

5 CONCLUSION

This paper presents a description of three different techniques for improving black-box testing. It describes how each method attempts to increase the observability and reduce code-dependency by means of building a representative model through analyzing input/output for a given black-box. It then provides a comparison of presented methods, their strength and weaknesses, followed by a discussion on their limitations and possible future optimization for interested researchers.

REFERENCES

[1] Cross-site Scripting. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). (Accessed: 2018-02-22).
[2] Stored Cross site scripting. https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_(OTG-INPVAL-002). (Accessed: 2018-02-22).
[3] w3af. https://github.com/andresriancho/w3af/. (Accessed: 2018-02-22).
[4] Web Application Firewall. https://www.owasp.org/index.php/Web_Application_Firewall. (Accessed: 2018-02-22).
[5] Wget. https://www.gnu.org/software/wget/. (Accessed: 2018-02-22).
[6] Argyros, G., Stais, I., Jana, S., Keromytis, A. D., and Kiayias, A. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1690–1701.
[7] Argyros, G., Stais, I., Kiayias, A., and Keromytis, A. D. Back in black: towards formal, black box analysis of sanitizers and filters. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 91–109.
[8] Bau, J., Bursztein, E., Gupta, D., and Mitchell, J. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 332–345.
[9] Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 114–129.
[10] Chen, Y., and Su, Z. Guided differential testing of certificate validation in ssl/tls implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 793–804.
[11] D'Antoni, L., and Veanes, M. The power of symbolic automata and transducers. In *CAV'17* (July 2017), Springer.
[12] Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. Enemy of the state: A state-aware black-box web vulnerability scanner.

[13] Doupé, A., Cova, M., and Vigna, G. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2010), Springer, pp. 111–131.
[14] Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX conference on Security* (2010), USENIX Association, pp. 10–10.
[15] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 40–52.
[16] Jovanovic, N., Kruegel, C., and Kirda, E. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security 18*, 5 (2010), 861–907.
[17] Khan, M. E., Khan, F., et al. A comparative study of white box, black box and grey box testing techniques. *Int. J. Adv. Comput. Sci. Appl 3*, 6 (2012).
[18] McKeeman, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.
[19] Pargas, R. P., Harrold, M. J., and Peck, R. R. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability 9*, 4 (1999), 263–282.
[20] Pellegrino, G., and Balzarotti, D. Toward black-box detection of logic flaws in web applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).
[21] Petsios, T., Tang, A., Stolfo, S., Keromytis, A. D., and Jana, S. Nezha: Efficient domain-independent differential testing. In *Security and Privacy (SP), 2017 IEEE Symposium on* (2017), IEEE, pp. 615–632.
[22] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).
[23] Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering 37*, 6 (2011), 772–787.
[24] Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., and Zhai, C. Bug characteristics in open source software. *Empirical Software Engineering 19*, 6 (2014), 1665–1705.
[25] Veanes, M. Applications of symbolic finite automata. In *International Conference on Implementation and Application of Automata* (2013), Springer, pp. 16–23.
[26] Veanes, M., De Halleux, P., and Tillmann, N. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on* (2010), IEEE, pp. 498–507.
[27] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 283–294.
[28] Zalewski, M. AFL. http://lcamtuf.coredump.cx/afl/. (Accessed: 2018-02-22).
[29] Zeller, A. Yesterday, my program worked. today, it does not. why? In *ACM SIGSOFT Software engineering notes* (1999), vol. 24, Springer-Verlag, pp. 253–267.

APPENDIX

This section provides the formal definitions of Deterministic Finite Automata, Symbolic Finite Automata, and Symbolic Finite Transducers.

Deterministic Finite Automata. A *Deterministic Finite Automaton* M is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite set called alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accept states. The accepted language over M is denoted by $\mathcal{L}(M)$ and contains all the strings which each starts from q_0 and ends in a state in F [6].

Symbolic Finite Automata. A *Symbolic Finite Automaton* M is a tuple $(Q, q_0, F, \mathcal{P}, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, $\mathcal{P}$ is the predicate family, and $\Delta \subseteq Q \times \mathcal{P} \times Q$ is a finite set of transitions [6].

Symbolic Finite Transducers. A *Symbolic Finite Transducer* is a a tuple $(Q, q_0, F, \mathcal{A}, \Delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, $\mathcal{A}$ is an effective label algebra, and $\Delta \subseteq Q \times \Psi \times \Lambda^* \times Q$ is called transitions. A transition $(p, \varphi, \bar{f}, q)$ in Δ which can be shown as $p \xrightarrow{\varphi/\bar{f}} q$ where $\bar{f}$ is the output function, indicates if an input symbol a in a state p satisfies guard φ, it will produce a sequence of outputs using $\bar{f}(a)$ and move onto state q [11].