ABSTRACT
Fuzz testing is an active testing technique which consists in automatically generating and sending malicious inputs to an application in order to hopefully trigger a vulnerability. Fuzzing entails such questions as: Where to fuzz? Which parameter to fuzz? What kind of anomaly to introduce? Where to observe its effects? etc.

Different test contexts depending on the degree of knowledge assumed about the target: recompiling the application (white-box), interacting only at the target interface (black-box), dynamically instrumenting a binary (grey-box).

In this paper, we focus on black-box test context, and specifically address the questions: How to obtain a notion of coverage on unstructured inputs? How to capture human testers intuitions and use it for the fuzzing? How to drive the search in various directions?

We specifically address the problems of detecting Memory Corruption in PDF interpreters and Cross Site Scripting (XSS) in web applications. We detail our approaches which use genetic algorithm, inference and anti-random testing. We empirically evaluate our implementations of XSS fuzzer KameleonFuzz and of PDF fuzzer ShiftMonkey.

Categories and Subject Descriptors
[Security and privacy]: Systems security — Vulnerability management, Vulnerability scanners

General Terms
Security Testing

Keywords
Fuzzing, Taint Inference, Anti-Random Testing, Evolutionary Algorithm, Black-Box Security Testing, Heuristic

1. INTRODUCTION

Software has bugs which can have security consequences. They are present because software development is hard. For instance, memory management and parsing are error prone, especially in a multi-threaded environment. Educating developers is not sufficient to eradicate all bugs before the products ship. Thus testing is a viable complementary approach to achieve this goal. Various security testing approaches exist: static analysis, concolic execution, model checking, fuzzing. Due to limited time and resources, the main criteria for choosing a testing method are: its capability of finding bugs within a given timespan and of finding bugs not found by other methods.

Test Harness. In case of access to the source code, white-box techniques range from static analysis to dynamic monitoring of instrumented code. If even the binary is inaccessible, black-box approaches generate inputs and observe responses. These are independent of the language used to create the application and avoid a harness setup. As they mimic the behaviours of external attackers, they are useful for offensive security purposes.

Fuzzing is an active stress testing technique, which consists in sending invalid inputs to the tested application, and observing its eventual failures. Due to its bugs finding capabilities, fuzzing is used at large scale. In black-box test context, most fuzzing approaches range from random data feeding, a modelization of the input to an inference of the input joined with anomaly detection. Fuzzing entails such questions as: Where to fuzz? Which parameter to fuzz? What kind of anomaly to introduce? Where to observe its effects? etc.

Memory corruption vulnerabilities arise due to confusion between code and data. Their impact ranges from attacker controlled code execution, privilege escalation to a complete control of the target. In black-box test context, most techniques range from random data feeding, a modelization of the input to an inference of the input joined with anomaly detection. The problem of black-box fuzzing mainly lays in its efficiency due to a very large search space. We present two black-box fuzzing techniques for respectively detecting XSS (Section 2) and Memory Corruption (Section 4) vulnerabilities.
2. BLACK-BOX XSS FUZZING

2.1 Introduction

Automatic black-box detection of web vulnerabilities generally consists in first “crawling” to infer the control flow of the application (hereafter referred as macro-state awareness), and then “fuzzing” to generate malicious inputs likely to exhibit vulnerabilities. As compared to scanners that are not macro-state aware, Doupé et al. increase vulnerability detection capabilities by inferring control flow models.\[1]\[1\]

In LigRE, Duchène et al. extend such models with taint flow inference and guides a fuzzer to improve detection capabilities one step further. XSS is a problem involving control+taint flows, and input sanitization. In presence of even basic sanitizers, many scanners have difficulties in creating appropriate inputs, and thus produce false negatives. In order to address aforementioned issues, we propose KameleonFuzz, a LigRE extension that mimics a human attacker by evolving and prioritizing the most promising malicious inputs and taint flows. We incorporate in KameleonFuzz a precise test verdict that relies on existing browser parsing and double taint inference.

Our Approach. KameleonFuzz is a black-box fuzzer which targets type-1 (reflected) and type-2 (stored) XSS and can generate full exploitation sequences. As illustrated in Figure 1, it consists of learning the model of the application and generating malicious inputs. We reuse the components A, B, C from \[28\]. The main contributions of our first approach are the blocks D1 and D2.

XSS involve a taint flow from a fuzzed value \(x_{src}\) on an HTTP request \(I_{src}\) to a vulnerable statement \(O_{dst}\) (HTML page). In a type-1 XSS, \(x_{src}\) directly appears (reflects) in the current output, whereas in a type-2, \(x_{src}\) is stored in an intermediate repository and reflected later.

Step A control flow inference learns how to navigate in the application. Given an interface and connection parameters(e.g., authentication credentials), a model is learnt in the form of an Extended Finite State Machine with instantiated parameter values, and a two level hierarchy (nodes and macro-states). The inferred model may not be complete.

Step B, approximate taint flow inference detects the possibility of XSS by observing reflections of a value \(x_{src}\), sent in the request \(I_{src}\) into an output \(O_{dst}\) (HTML page). It generates walks on the model, and approximatively infers the taint. A substring matching algorithm is used with a heuristic to avoid false negatives. Figure 2 illustrates a control+taint flow model.

Step C prunes the control+taint flow model by applying a specialized form of slicing, called chopping. This reduces the search space.

We first focus on the blocks D.1 (malicious input generation) and D.2 (precise taint flow inference). A genetic algorithm (GA), parameterized by an attack grammar, evolves malicious inputs. The attack grammar reduces the search space and mimics the behavior of a human attacker by constraining the mutation and crossover operators which generate next generation inputs. We define a fitness function that favors most suitable inputs for XSS attacks. Since server sanitizers may alter the observed value at the reflection point \(O_{dst}\), a naive substring match may not infer the taint precisely enough, which could lead to false negatives. To overcome such limitations, we perform a double taint inference. We detail these subcomponents in Section 3.

Contributions. Our first approach involves:

- the first black-box model-based GA driven fuzzer that detects type-1 and 2 XSS;
- a combination of model inference and fuzzing;
- an implementation of the approach and its evaluation.

Section 2.2 provides a walk-through of our approach over an example. Section 3 details how malicious inputs are generated and evolved. Section 4 evaluates KameleonFuzz on typical web applications. Finally, we discuss our approach in Section 5, survey related work in Section 6, and conclude in Section 10.

2.2 Illustrating Example

P0wnMe is a vulnerable application. Once logged-in, a user can save a new note, view the saved notes, or logout.

KameleonFuzz Execution on P0wnMe In steps A and B of Figure 1, LigRE infers a control+taint flow model of which a simplified extract\[15\] is shown in Figure 2. The control flow is represented by plain arrows (transitions) and nodes. A taint flow originates from a bold text \(x_{src}\), sent in \(I_{src}\), and reflects(dotted arrows) in \(O_{dst}\). For instance, the value \(egassem\), of the input parameter \(msg\) sent in the transition \(7 \rightarrow 17\) is reflected in the output of the transition \(18 \rightarrow 21\).

Figure 2 contains a reflection for the value \(2_e_g_a_s\_sem\) of the parameter \(message2\) sent in the transition \(7 \rightarrow 33\). An extract of the output \(O_{dst}\) is \(<input name="message2" value='2_e_g_a_s\_sem'/>>\ where we highlight the reflection. Here, the reflection context is inside a tag attribute value. The context influences how an attacker generates fuzzed values. Listing 3 shows the server sanitizer for this reflection. It blocks simple attacks. Attackers search a fuzzed value s.t. if passed through the sanitizer, then its reflection is not syntactically confined in the context\[17\], i.e., it spans over different levels in the parse tree.

Listing 1: A vulnerable sanitizer in P0wnMe

Table 1 shows fuzzed values sent by \w3af\[33\], a black-box open source scanner, when testing WebApp. \w3af iterates over a list of fuzzed values. It does not learn from previous requests, nor considers the reflection context.

In step D, KameleonFuzz generates individuals, i.e., normal input sequences in which it fuzzes the reflected value. The chopping (step C of LigRE) produces the input sequences. The attack grammar produces the fuzzed values. For each individual, the taint is precisely inferred. It is

\[1\] For clarity sake, we only represent the inputs on the transitions, and the outputs correspond to colored nodes. Each color corresponds to a macro-state. The inputs are composed of an HTTP method(e.g., POST), a part of the URL (e.g., ?) and POST parameters (e.g., \{'message2': '2_e_g_a_s\_sem'\}). \[28\] formalizes a control+taint flow model.
Figure 1: High Level XSS Fuzzing Approach Overview

Figure 2: Inferred control+taint model (extract)

Table 1: w3af fuzzed values (extract)

Table 2: KameleonFuzz fuzzed values (extract)

3. EVOLUTIONARY WEB FUZZING

The fuzzing (step D in Figure 1) generates a population of individuals (GA terminology). An individual is an input sequence generated by LigRE in which KameleonFuzz generates a fuzzed value $x_{src}$ according to the attack grammar for the reflected parameter. As described in Algorithm 1, this population is evolved via the mutation and crossover operators (Section 3.6) w.r.t. the attack grammar (Section 3.2) and according to their fitness score (Section 3.5).

3.1 Individual

An individual is an input sequence targeting a specific reflection. It contains a non-malicious input sequence extracted from the chopped model, and fuzzed value $x_{src}$. This sequence encompasses the originating transition $I_{src}$, and the transition where to observe the reflection $O_{dst}$. An extract of the output $O_{dst}$ for the last individual is

```html
<T9nj1>'<script>alert(18138)</script> XSS Attempt!

ohIeqL' onload="document.body.innerHTML='<div id=90480>
</div>'" fakeattr='XSS Attempt!'</A></script>

ZuTa2' onload=alert(94478)" ZuTa2' onload=alert(94478)

WUkp\tLgpRa WUkp\tLgpRa 9.1 1

WUkp\t onload='alert(94478)'

Table 2 illustrates this evolution.

---

An extract for the test verdict (did this individual trigger an XSS?) and the fitness score (how close is this individual of triggering an XSS?). The best individuals are mutually recombined according to the attack grammar to create the next generation: e.g., the individuals 3 and 4 of generation 1 produce the individual 1 of generation 2. This process is iterated until a tester defined stopping condition is satisfied (e.g., one XSS is found).
3.2 Attack Grammar

In order to constrain the search space (subset of $A^*$, $A$ being the alphabet for the targeted encoding), we use an attack grammar for generating fuzzed values. This grammar also constrains mutation and crossover operators (lines 3, 14, 16 of Algorithm 1). Attackers would attempt to send such fuzzed values to the application. As compared to a list of payloads as in w3af and skipfish, an attack grammar can generate more values, and is easier to maintain thanks to its hierarchical structure.

The knowledge used to build the attack grammar consists of the HTML grammar [34], string transformations in case of context change [37], known attacks vectors [38].

We give a taste of how to build the attack grammar, as it is yet manually written and its automatic generation is a research direction. Figure 3 illustrates its structure. The first production rule consists of representation and context information. Inside an attribute value (<input value="reflection"/>), and outside a tag (<h1>reflection), are examples of reflection contexts. The representation consists of encoding, charset, and special string transformation functions that we name anti-filter (e.g., PHP addslashes [34]).

In order to create the attack grammar, we assume the availability of $S$, a representative set of vulnerable web applications (different from the tested applications) and corresponding XSS exploits. For each reflection context, the analyst writes a generalization of the XSS exploits in the form of production rules with terminals and non-terminals. In case of production rules including the OR or REPEAT operators, she assigns weights on choices, depending on their frequency of use in the exploits of $S$. If no weights are assigned, all choices weigh equally. Once created, we use this attack grammar for fuzzing the tested applications.

We represent the grammar in an Extended Backus–Naur Form [39] with bounded number of repetitions. By construction, the attack grammar is acyclic. Thus it unfolds to a finite number of possibilities. Listing 3 of Appendix F contains an excerpt of the attack grammar.

![Figure 3: Structure of the attack grammar (extract)](image)

![Figure 4: The Production Tree of a Fuzzed Value](image)

3.3 Precise Taint Flow Inference (D.2)

The precise taint flow inference permits obtaining information about the context of a reflection. This later serves for computing a precise test verdict, and is an input for the fitness function.

The flow for producing the taint aware parse tree $T_{dst}$ is illustrated in Figure 5. First, a string to string taint inference algorithm (e.g., with Levenshtein edit distance [13]) is applied between the fuzzed value $x_{src}$ and the output $O_{dst}$ in which it is reflected. In parallel, a parser (e.g., from
Google Chrome) evaluates the application output $O_{dst}$ and produces a parse tree $P_{dst}$ (e.g., Document Object Model (DOM)). Then the taint is inferred between $x_{src}$ and each node of $P_{dst}$ to produce $T_{dst}$, a taint aware parse tree (see Figure 6), as follows.

For each node of an output parse tree $P_{dst}$, we compute a string distance between each tainted substring and the node textual value. Then we only keep the lowest distance score. If this score is lower than a tester defined threshold, then this node is marked as tainted. This taint condition may be slightly relaxed in case a cluster of neighbors nodes has a distance “close to the threshold”. The inferred taint aware parse tree $T_{dst}$ is an input for the fitness function and test verdict.

It is important to note that, instead of writing our own parser, as done in [2], we rely on a real-world parser. This has two advantages. First, we are flexible with respect to the parser (e.g., for XSS: Chrome, Firefox, IE ; for other vulnerabilities such as SQL injections, we could rely on a SQL parser). Secondly, we are certain about the real-world applicability of the detected vulnerabilities.

### 3.4 Test Verdict

The test verdict answers to the question “Did this individual trigger an XSS vulnerability?” . The taint-aware parse tree $T_{dst}$ (Figure 6) is matched against a set of taint-aware tree patterns (e.g., Figure 5). If at least one pattern matches, then the individual is an XSS exploit (i.e., the test verdict will output “yes, vulnerability detected”). A taint tree pattern is a tree containing regular expressions on its nodes. Those regular expressions may contain strings (e.g., `script`), taint markers, repetition operators `*`, or the match-all character `.`. The tester can provide its own patterns. We incorporate in KameleonFuzz default patterns for XSS vulnerabilities. Those all violate the syntactic confinement of tainted values. The second pattern illustrated in Figure 7 matches the parse tree represented in Figure 6.

![Figure 5: Precise Taint Inference ($I_{src} \rightarrow O_{dst} \rightarrow T_{dst}$)](image)

<table>
<thead>
<tr>
<th>weight</th>
<th>id</th>
<th>dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ + + 1</td>
<td>successfully injected character classes</td>
<td></td>
</tr>
<tr>
<td>+ + + 2</td>
<td>tainted nodes in the parse tree $T_{dst}$</td>
<td></td>
</tr>
<tr>
<td>++ 3</td>
<td>singularity</td>
<td></td>
</tr>
<tr>
<td>++ 4</td>
<td>transitions from source $I_{src}$ to reflection $O_{dst}$</td>
<td></td>
</tr>
<tr>
<td>++ 5</td>
<td>new page discovered</td>
<td></td>
</tr>
<tr>
<td>++ 6</td>
<td>new macro-state discovered</td>
<td></td>
</tr>
<tr>
<td>+ 7</td>
<td>unexpected page seen</td>
<td></td>
</tr>
<tr>
<td>+ 8</td>
<td>page correctly formed w.r.t. output grammar</td>
<td></td>
</tr>
<tr>
<td>+ 9</td>
<td>unique nodes from the start node</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: Dimensions of the fitness function**

XSS vulnerabilities. Those all violate the syntactic confinement of tainted values. The second pattern illustrated in Figure 7 matches the parse tree represented in Figure 6.

![Figure 6: A Taint-Aware Parse Tree $T_{dst}$ (extract). The payload is a message box that displays 94478 harmless.](image)

![Figure 7: Two Taint-Aware Tree Patterns, represented in a Linear Syntax (resp. a tainted script content and a tainted event handler attribute)](image)

### 3.5 Fitness

The fitness function assesses “how close” is an individual to finding an XSS vulnerability. The higher its value, the more likely the GA evolution process will pick the genes of this individual for creating the next generation. The inputs of the fitness function are the individual $I$, the concrete output $O_{dst}$ in which the fuzzed value $x_{src}$ is reflected, $T_{dst} = taint(parse(O_{dst}), x_{src})$ the taint-aware parse tree, and the application model $M$. The fitness dimensions are related to properties we observed between the fuzzed value and the reflection in case of successful XSS attacks. Those dimensions are listed in Table 3. In [2], we drew a sketch of the currently used fitness function.

Those dimensions model several intuitions that a human penetration tester may have. The most significant ones are:

- **1: Percentage of Successfully Injected Character Classes.** Characters that compose leaves of individual fuzzed value tree (see Figure 4) are categorized into classes depending on their meaning in the grammar. This metric expresses the “injection power” for the considered reflection.
- **2: Number of Tainted Nodes in the Parse Tree.** Whereas injecting several character classes is impor-
tant, it is however not a sufficient condition for an attacker to exert control on several parse tree nodes. Successful XSS injections are generally characterized by at least two neighbors tainted nodes (one which is supposed to confine the reflection, and the other(s) that contain the payload and a trigger for that payload). Thus, if an attacker is able to reflect on several nodes, we expect that it increases its chances to exploit a potential vulnerability.

- **3: Singularity of an individual w.r.t. its current generation.** A problem of GA is overspecialization that will limit the explored space and keep finding the same bugs [10]. To avoid this pitfall, we compute “how singular” an individual is from its current generation. This dimension uses the source transition $I_{src}$, the fuzzed value $x_{src}$, and the reflection context (i.e., the destination transition $O_{dst}$ and the tainted nodes in the parse tree $T_{det}$).

- **4: The higher the Number of Transitions between the source transition $I_{src}$ and its Reflection $O_{dst}$, the more difficult it is to detect that vulnerability, because it expands the search tree.**

- **a New Page (5) or Macro-State (6) discovered:** increases application coverage.

### 3.6 Mutation and Crossover Operators

A probability distribution decides whether an individual will be mutated or not. When a mutation will happen, an operator is applied either on the fuzzed value or on the input sequence.

The **fuzzed value** mutation operator works on the production tree of the fuzzed value $x_{src}$ (see Figure 4). We implemented several strategies for choosing which node to mutate and how to mutate (e.g., uniform distribution, Least Recently Used, _). The amplitude of the mutation is a decreasing function of the fitness score: if an individual has a high fitness score, the mutation will target nodes in the production tree that are close to leaves. Similarly, in case of low fitness score, the operator is more likely to mutate nodes close to the root. An example of fuzzed value mutation applied to Figure 4 is performed in a different choice for the HANDLER non terminal (e.g., onmouseover instead of onload).

The **input sequence** mutation operator works on the whole sequence $I$. It consists of either taking another path in the model from the source $I_{src}$ to the destination $O_{dst}$ or targeting a different reflection.

The **crossover operator** works at the fuzzed value level, i.e., on the production tree. Its inputs are two individuals of high fitness scores. It produces two children.

### 4. EMPIRICAL EVALUATION

Based on a prototype implementation, we evaluate KameleonFuzz against black-box open source XSS scanners, in terms of detection capabilities (RQ1) and detection efficiency (RQ2). In our experiments, KameleonFuzz detected most of the XSS detected by other scanners, several XSS missed by other scanners, and 3 previously unknown XSS.

#### 4.1 Implementation

KameleonFuzz is a python3 program which targets Type-1 and 2 XSS. It is composed of 4500 lines of code. As shown in Figure 4, we instrument Google Chrome with the Selenium library [17]. We use LigRE, a control+taint flow model inference tool and slicer.

![Figure 8: Architecture of KameleonFuzz](image)

#### 4.2 Test Subjects

As described in Table 4, we select seven web applications of various complexity. In Table 4, we detail our interest in them. KameleonFuzz detected at least one true XSS in all of them. We considered four black-box XSS scanners to compare with KameleonFuzz: Wapiti, w3af, SkipFish and LigRE+w3af. Appendix D contains the configuration we used during the experiments. It is important to note that only LigRE and KameleonFuzz are macro-state aware.

#### 4.3 Evaluation Setting

**XSS Uniqueness:** an XSS is uniquely characterized by its source transition $I_{src}$, its parameter name, its destination transition $O_{dst}$, and the tainted nodes in the parse tree $T(P(O_{dst}), I_{src})$. Hence if a fuzzed value is reflected two times in $O_{dst}$, e.g., in two different nodes in the parse tree, and for each node, the scanner generated an exploitation sequence, then we count two distinct XSS. In our experiments, the only time we had to distinguish two XSS using the nodes in the parse tree was in the Gruyere application.

We run the scanners on a Mac OS X 10.7.5 platform with a 64 Bit Intel Quad-Core i7 at 2.66GHz processor, and 4GB of RAM DDR3 at 1067MHz.

#### 4.4 Research Questions

RQ1. (Fault Revealing): Does evolutionary fuzzing find more true vulnerabilities than other scanners?
To answer this question, we consider the number of true positives, the number of false positives, and the overlap of true positives. For the first two metrics, we compare all tools, whereas for the overlap, we compare LigRE+KameleonFuzz against the others (Wapiti, w3af, SkipFish, LigRE+w3af). True positives are the number of XSS found by a scanner that actually are attacks, thus the higher, the better. If a scanner produces false positives, a tester will lose time, thus the lower the better. The overlap indicates vulnerabilities detected by several scanners. We denote as $T_A$ the number of True XSS vulnerabilities found by the scanner $A$. We define the overlap as:

$$\text{overlap}(A, B) = \frac{T_A \cap T_B}{T_A}$$

A low overlap indicates that scanners are complementary. We also consider the vulnerabilities only detected by one scanner:

$$\text{only}_\text{by}(A, B) = \frac{T_A}{T_A \cup T_B} - \text{overlap}(A, B)$$

A low only_by indicates that a given scanner does not find many XSS that the other missed.

For each scanner and application, we sequentially configure the scanner, reset the application, set a random seed to the scanner, run the scanner against the application, and retrieve the results. We repeat this process five times, using different seeds. Parameters have been adjusted so that each run lasts at most five hours. Beyond this period, we stop the scanner and analyze the produced results. The number of found vulnerabilities is the union of distinct true vulnerabilities found during the different runs. If possible, scanners are configured so that they only target XSS. We configure the scanners with the same information (e.g., authentication credentials). When a scanner does not handle this information correctly, we perform two sub-runs: one with the cookie of a logged-on user, and one without. Since all scanners, except LigRE and KameleonFuzz, are not macro-state aware we configure them to exclude requests that would irrevocably change the macro-state (e.g., logout when an authentication token is provided).

The practicality of LigRE+KameleonFuzz is illustrated in Table 5. This figure reports the number of potential reflections (i.e., potential sinks), found vulnerabilities (i.e., actual sinks for which a successful XSS exploit was generated), and generations to find all detected vulnerabilities during the fuzzing. The three columns in the middle report the length of created XSS exploits for the closest vulnerabilities from the start node.

**True and False XSS Positives.** We manually verify the XSS for each scanner. During our experiments, no scanner found a false positive XSS (Skipfish had other false positives). Figure 4 lists the results of the black-box scanners against each application. In our experiments, KameleonFuzz detected the highest number of XSS, and several XSS missed by others. The union of the distinct true XSS found by the scanners is 35. LigRE+w3af finds $\frac{23}{35} = 65.7\%$ of the known true XSS, whereas LigRE+KameleonFuzz finds $\frac{24}{35} = 91.4\%$. KameleonFuzz improves XSS detection capabilities. Since it is challenging to find reliable source (except our own human testing expertise) which provide an exhaustive list of the number of true XSS in the applications, we chose not to compute recall.

The overlap and only_by of true XSS found by LigRE+KameleonFuzz against other scanners are illustrated on Figure 10. KameleonFuzz finds the majority of known true XSS. W3af and SkipFish find the remaining ones. In the Gruyere application, Skipfish and w3af each found one vulnerability missed by all other scanners, including KameleonFuzz. Those consist of a not referenced 404 page containing a type-1 XSS, and of a type-2 XSS within the pseudo field when registering. It is harder to find the latter XSS than others: the application behaves differently as inferred when the scanner registers a new user with a fuzzed pseudo. Reusing the fuzzing learned knowledge in the inference may permit KameleonFuzz to detect this XSS. Additionally, Skipfish and w3af both detected one XSS in Gruyere that other scanners missed. Thus the only_by of SkipFish and w3af is two in Figure 10, whereas in Figure 9 one XSS is detected by both of them. Inferring the control flow for navigating to non-referenced pages may increase LigRE+KameleonFuzz XSS detection capabilities. If this is not an option, the tester should use LigRE+KameleonFuzz, SkipFish, and w3af.

**RQ2.** (Efficiency): How efficient are the scanners in terms of found vulnerabilities per number of tests?

To answer this question, it is appropriate to observe the number of detected true XSS depending of the number of HTTP requests. Thus, we set up a proxy between the scan-
ner and the web application, and configure this proxy to limit the number of requests. We iteratively increase this limit, run the scanner, and retrieve the number of found distinct true XSS. We manually verify them. We run such a process five times per scanner, web application, and limit. For each number of requests, for each scanner, we sum the number of unique true XSS detected for all applications. The results are illustrated in Figure 11.

On considered applications, below approximatively 800 HTTP requests per application, w3af is the most efficient scanner. Thus we hypothesize that in applications with few macro-states, assuming it is able to navigate correctly, w3af is more efficient than other scanners at finding non filtered XSS. In our experiments, mainly happened in P0wnMe and Gruyere. In applications with more macro-states, assuming the cost of control+taint flow inference is acceptable, LigRE improves vulnerability detection. Starting from 900 HTTP requests, LigRE+KameleonFuzz detects more vulnerabilities per number of requests than LigRE+w3af. For instance, after 2200 requests per application, fuzzing with Kameleon-

4.5 Found 0-Day XSS

We list some of the 0-day XSS found using such techniques.

**SFR DSL Box.** SFR is the french Vodafone. KameleonFuzz automatically uncovered 39 Type-1 XSS in the web administration of the DSL boxes of SFR. The number of users impacted is estimated to 5.2 Million of customers. Such XSS permit for instance to remotely modify the routing table of such routers! We illustrate some of those XSS in Figure 12. These have been assigned the id CVE-2014-1599.
HITB 2014 CFP Website. HITB is a hacking conference which has a Call For Papers website in which authors submit their presentations and papers, and program committee members can review those. Our XSS Fuzzer helped us to find 2 Type-2 XSS in this web application. The impact is the ability to impersonate the administrators of the application, thus getting access to all the submissions! We illustrate some of our findings in Figure 13. One of the main organizers of HITB a.k.a. @l33tdawg quickly reacted positively to our findings!

5. DISCUSSION

5.1 Applicability to other Command Injection Vulnerabilities

Even though we only experimented with Type-1 and 2 XSS vulnerabilities, we are confident that the KameleonFuzz approach can be applied to other types of interpreter injection vulnerabilities, with proper adaptations (e.g., attack grammar), as shown in Table 6. Such adaptation still do not require access to the application source code, only the ability to intercept at run-time the arguments at the observation points. Thus for command injection vulnerabilities other than Type-1 and Type-2 XSS, one may consider our approach as having a grey-box harness. Using our approach for detecting Type-0 XSS and mutation-XSS is likely to require an adaptation of the attack grammar.

5.2 Approach Limitations

Reset: We assume the ability to reset the application in its initial state, which may not always be practical (e.g., when testing a live application on which there are users connected; we would work on a copy) or may take time. However, this does not break the black-box harness assumption: we do not need to be aware of how the macro-state is stored (e.g., database).

Generation of an Attack Grammar: Writing an attack grammar requires knowledge of the parameters mentioned in Section 3.2. This work is yet manual. The trade-off between the size of the language generated by this grammar and the fault detection capabilities is yet to be studied. A too narrow generated language (e.g., few produced fuzzed values for a given context, or very few contexts) may limit the fault detection capability, whereas a too important one may have limited efficiency. Moreover, the attack grammar is tied to the targeted injection sub-family (e.g., XSS, SQL injection, etc), thus the need for human input is a current limitation. There is room for research in automating this generation process.

XSS Model Hypothesis: We hypothesize that an XSS is the result of only one fuzzed value. Our current approach may have false negative on XSS involving the fuzzing of at least two fuzzed values at a time.

Limitations due to the use of LigRE: KameleonFuzz supports Ajax applications if they offer similar functionality when the client does not interpret JavaScript. LigRE requires to identify non deterministic values in the applications. Hossen et al. automated this identification.

Encoding: The precision and efficiency of the taint flow inference is dependent of the considered encoding transformations. Plain, url and base64 encodings are implemented. LigRE and KameleonFuzz can be extended to support more.

Table 6: Command Injections: Vulnerabilities, Output Grammars, and Observation Points

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Output Grammar</th>
<th>Where to Parse?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross Site Scripting</td>
<td>HTML</td>
<td>HTML page</td>
</tr>
<tr>
<td>HPP Param. Pollution</td>
<td>HTTP</td>
<td>Reply Headers</td>
</tr>
<tr>
<td>PHP Code Injection</td>
<td>PHP</td>
<td>argument of eval</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>SQL</td>
<td>arg. of sql_query</td>
</tr>
<tr>
<td>Shell Injection</td>
<td>Shell</td>
<td>...exec, system</td>
</tr>
</tbody>
</table>

Figure 13: 2 Type-2 XSS in the HITB 2014 CFP
5.3 Threats to Validity

**External Comparison:** We only compare to open source black-box web scanners and LigRE. We contacted several vendors of commercial products, but we did not receive a positive reply within a reasonable timeframe. Thus we were unaware to compare with commercial scanners. Those may obtain better results than the considered scanners.

**Randomness:** Scanners make extensive use of randomness. Since some XSS are not trivial to be found, their discovery may involve randomness and duration. We tried to limit such factors by running the scanners five times with different seeds and up to five hours. The chosen duration of the experiments may impact the results.

**Considered Applications:** Our comparison with other scanners is limited to the considered versions of scanners and applications. We cannot generalize results from these experiments. Running the scanners on other applications or scanners versions may produce different results.

**KameleonFuzz Parameters:** KameleonFuzz contains numerous adjustable parameters e.g., probabilities that drive the mutation and crossover operators during the fuzzing. In Appendix D we provide significative parameters and their default values. Those are chosen empirically. Because the value domain of each parameter is quite wide, and it is time consuming to run the whole test suite, it was not feasible to evaluate the combination of all parameters values and their impact. Thus, we cannot guarantee that the chosen default values achieve the best detection capabilities and efficiency.

5.4 Related Work

5.4.1 XSS Test Verdict in a Black-Box Approach

**Confinement Based Approaches** assume that malicious inputs break the structure at a given level (lexical or syntactical). As in Sekar’s work, we rely on non-syntactical confinement and we use detection policies that are both syntax and taint aware. A key difference is that Sekar wrote his own parser to propagate the taint, whereas we use the parser of a browser (e.g., Google Chrome). Thus we infer the taint twice (see Figure 3). By doing so, we are sure about the real-world applicability of the found XSS exploits, and our implementation is flexible w.r.t. the browser. It relies on non-lexical confinement as a sufficient fault detection measure, which is more efficient than [72], but requires a correctly formed output (which is not an always valid assumption on HTML webpages[41]) and is prone to false negatives.

**Regular-Expressions Based Approaches** assume that the fuzzed value is reflected “as such” in the application output i.e., that the sanitizer is the identity function. In case of sanitizers this may lead to false negatives [6]. Moreover, most do not consider the reflection context, which can lead to false positive. IES [57] and NoScript [51] rely on regular expression on fuzzed values. XSSAuditor (Chrome XSS filter) performs exact string matching with JavaScript DOM nodes.

**String Distance Based Approaches** Sun [54] detects self-replicating XSS worms by computing a string distance between DOM nodes and requests performed at run-time by the browser.

IES [54] and Chrome XSSAuditor [6] filters only work on Type-1 XSS. Whereas NoScript is able to block some Type-2 XSS, but is only available as a Firefox plugin.

5.4.2 Learning and Security Testing

In its basic form, fuzzing is an undirected black-box active testing technique [44, 45, 46, 47]; mainly targets memory corruption vulnerabilities. Stock et al.’s recent work fuzzes and detects Type-0 XSS in a white-box harness [23]. Heiderich et al. detect in black-box mutation-based XSS caused by browser parser quirks [10]. LigRE+KameleonFuzz is a black-box fuzzer which targets Type-1 and 2 XSS.

**GA for black-box security testing** has been applied to evolve malwares [60] and attacker scripts [13]. KameleonFuzz is the first application of GA to the problem of black-box XSS search. Its fitness dimensions model the intuition of human security penetration testers.

**An Attack Grammar** produces fuzzed values for XSS as a composition of tokens. [83, 93] and KameleonFuzz share this view. In their recent work [82], Tripp et al. prune a grammar based on the test history to efficiently determine a valid XSS attack vector for a reflection. It would be interesting to compare KameleonFuzz to their approach, and to combine both. Wang et al. use a hidden Markov model to build a grammar from XSS vectors [80].

**Model Inference for Security Testing** Radamsa targets memory corruption vulnerabilities: it infers a grammar from known inputs then fuzzes to create new inputs [61]. Shu et al. passively infer a model from network traces, and actively fuzz inputs [74]. [24, 10] infer the likelihood for specific inputs parts of triggering failures.

For command injection vulnerabilities (XSS, SQL injection, ...), Dessiatnikoff et al. cluster pages according a specially crafted distance for SQL injections [27]. Sotirov iterates between reverse-engineering of XSS filters, local fuzzing, and remote fuzzing [73]. Doupé et al. showed that inferring macro-state aware control flow models increases vulnerability detection capabilities [21]. With LigRE, Duchène et al. showed that enhancing such models with taint flows increases its capabilities even more [28].

KameleonFuzz extends LigRE and is a black-box fully active testing approach. It generates and evolves fuzzed inputs on the obtained reflections using an attack grammar and the control+taint flow model.

In Section 4.3 we detail our second evolutionary fuzzing approach which drives the search in various directions in order to learn interesting combinations of anomaly operators for efficiently detecting memory corruption vulnerabilities in PDF interpreters.
6. EVOLUTIONARY BLACK-BOX FUZZING OF PDF INTERPRETERS

6.1 Introduction

During the early days of Fuzzing, using random inputs was sufficient to drive the applications into various states. However, as current applications perform more structural verifications on the inputs, it is necessary to use a more precise input knowledge. This is achieved using an initial set of valid inputs, and inputs models for control and/or data flows. If obtaining a valid set of inputs can generally be achieved at low cost, modeling the control plus data flows can be an expensive task. In fact corporations sell such models [14], and those are generally built by analysts. There exist algorithms to infer such models semi-automatically [2, 8, 12], assuming the availability of an abstraction. Abstractions are often written by analysts, and are specific to an input format family [1].

Interpreters are a particular class of applications which receive dynamic content such as scripts as inputs. Those inputs are analyzed, compiled and executed at run-time. Since interpreters perform verifications at each step, they may easily reject inputs [14]. Beizer described a fault model for messages: an application can reject a valid input, accept an invalid input, or fail due to an input (which can be valid or invalid) [8]. In our specific problem, a failure may be observed as a crash, which is often a sufficient condition for a memory corruption vulnerability. In such cases, program assertions may not be violated, but a program crashing has security implications, as the availability property does not hold for all inputs. Dynamic memory errors detectors permit a more precise test verdict by propagating the taint from the inputs to the data [12, 73].

When testing interpreters, as we rarely have access to the source code (e.g., Adobe Acrobat, Microsoft Internet Explorer), and instrumenting a binary program is a costly task, we hypothesize a black-box test context at the interface. During his XML fuzzing research which led to discovery of numerous 0-days [27], Grégoire asked to Radiumsa's authors which anomaly operators and parameters they “feel are the most appropriate”.

We leverage our experience in applying genetic algorithm for generating inputs [21, 24, 32], in order to automatically determine interesting combinations of operators and parameters. We aim at increasing the capacity of black-box fuzzers to explore a wide range of invalid accepted inputs. In particular, we apply genetic algorithm and anti-random testing techniques to the problem of fuzzing interpreters in black-box.

Contributions. In this second approach, we:
- present one of the first black-box evolutionary fuzzer targeting memory corruption vulnerabilities
- describe a fitness function for guiding black-box interpreter fuzzing
- empirically evaluate our implementation ShiftMonkey and observe that fitness driven black-box evolutionary fuzzing is more efficient than undirected fuzzing in detecting not deeply embedded vulnerabilities

6.2 Overview

This work consists in applying a Genetic Algorithm (GA) to the problem of fuzzing interpreters in a black-box context. GA iteratively evolve a population of individuals (i.e., a set of inputs), by selecting the best individuals (i.e., those having the highest fitness score) and evolving them using crossover and mutation operators to create the next generation. This process is iterated until a tester defined condition is met (e.g., number of found vulnerabilities, consumption of testing resources, etc.). A high level flow of GA is illustrated in Figure 14.

Figure 14: Evolutionary PDF Fuzzing via Genetic Algorithm

6.3 Individual

Representation An individual is composed of an initial input, and of an ordered list of anomaly operators and of their arguments. Figure 15 illustrates an individual representation and the process to decode its genes in order to concretize the genotype in phenotype (GA vocabulary). In this example, the first gene 3 corresponds to the file index in the input set. The next gene has the value 9, and corresponds to an anomaly operator having only one argument seed, thus the next gene value 138.4 is decoded as such.

Figure 15: Representation and Decoding of an Individual

The process of generating a new individual is illustrated in Figure 16. It occurs during the creation of the first generation (k = 0) and during the evolution of the population from the current generation (k) to the next one (k + 1).
Section 2 illustrates the approach to developing FAR. We start with an informal example to explain the basic approach, in general and the FAR method in particular. As illustrated in Figure 16, we convert the previously generated individuals to multi-dimensional vectors and apply the Fast Anti-Random (FAR) Test Generation technique [2]. It produces an orthogonal vector to the centroid of all the previous vectors. The new generated vector is then converted in its individual representation.

![Figure 16: Each New Individual is Anti-Random w.r.t. the Previous ones](image)

6.4 Fitness and Test Verdict

We submit each individual to the application, and then compute its test verdict and fitness score, as illustrated in Figure 18. We use a memory error detector as the oracle for computing the Test Verdict. Such oracles are commonly accepted in fuzzing [10, 11, 12]. Due to their specificities, we rely on detectors which are operating system specific (see Section 5).

When the test verdict evaluates to true i.e., the individual found a vulnerability, the individual is saved and discarded from the evolution. A new individual will replace it when computing the next generation.

The **Fitness Function** is a heuristic which assesses how close towards the goal - in our case, triggering a memory corruption vulnerability - an individual is. We designed a fitness function composed of several dimensions. Those are enumerated in Table 7, along with their meaning, and the reason behind their use. The higher the number of + or -, for a dimension, the more it influences the resulting fitness score. If there are +, resp. -, it means the fitness function is increasing, resp. decreasing w.r.t. this dimension.

<table>
<thead>
<tr>
<th>#</th>
<th>Weight</th>
<th>Dimension</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++</td>
<td>Depth of the Production Tree</td>
<td>Local Coverage</td>
</tr>
<tr>
<td>2</td>
<td>++</td>
<td>Used Production Rules</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>++</td>
<td>Distinct Anomaly Operators</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>++</td>
<td>Interpreter Warnings</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>- -</td>
<td>Individual Rejected</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>Duration to Load</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>+++</td>
<td>Singularity</td>
<td></td>
</tr>
</tbody>
</table>

![Table 7: Dimensions of the Fitness Function](image)

The most significative ones are:

- **The more Distinct Anomaly Operators are used to create an individual, the more constraints the processing of this individual is likely to violate, thus the more likely it may trigger a problem.

- **Input Rejected:** however, a too high quantity of anomaly operators may result in the individual being rejected, due to too many constraints violations. As a consequence, such an individual will achieve a low control flow path coverage of the tested application. Since we assume a black-box test harness, we do not measure e.g., coverage at the binary level. We approximate this dimension by observing if the application displays a message box when it processes the individual.

- **Interpreter Warnings:** During the parsing, the compilation, or execution stages, interpreters may output warnings. For instance, a JavaScript interpreter in a web browser may raise warnings. These indicate that the individual stimulates potentially interesting code.

- **Singularity:** A problem of GA is overspecialization that will limit the explored space and keep finding the same bugs [16]. To avoid this pitfall, we compute “how singular” an individual is from its generation. We use all individual dimensions for computing this metric.
6.5 Mutation and Crossover Operators

After having evaluated the test verdict and fitness score for each individual of the current population, the GA evolves this population for creating the next generation. The evolution consists in the application of GA operators: mutation and crossover.

A mutation consumes an individual and produces a new individual for the next generation. The higher the fitness score of an individual, the lower its number of mutated genes will be. Similar to the creation of individuals (see Figure 6.2), we abstract the individual, and apply an anti-random test generation technique.

A crossover consumes two “parents” individuals and produces two new ones, their “children”. The children inherit genes from both parents.

7. IMPLEMENTATION

We implemented this approach in a tool ShiftMonkey, which is composed of 2300 lines of Ruby code and 300 lines of Python code. Origami [65] and PyPDF2 [77] are the parsers for Portable Document Format (PDF) documents. Radamsa [58] is the anomaly generator. On Windows, AutoHotKey [50] permits detecting messageboxes, such as the ones represented in Figure 21. On Mac OS, we use AppleScript [3]. For the test verdict, we use both a crash occurrence and DrMemory [12]. DrMemory slowdown factor averages between two and fifteen times. It detects a wide range of memory corruption errors (e.g., invalid memory access, incorrect heap management, memory leaks, etc.).

8. EMPIRICAL EVALUATION

8.1 Test Subjects

Applications. Table 8 lists the tested applications. Those are widely used readers of PDF documents. The number of instructions and of basic blocks are indicators of the complexity of the tested applications. We detail our interest in those applications in Appendix A.

Fuzzers. Table 9 lists the black-box fuzzers that we considered during our experiments.

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>Mutation</th>
<th>Fitness</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShiftMonkey</td>
<td>Anti-Random</td>
<td></td>
<td>v1.0</td>
</tr>
<tr>
<td>ShiftMonkey</td>
<td>Random</td>
<td></td>
<td>v1.0</td>
</tr>
</tbody>
</table>

Table 9: Considered Black-Box Fuzzers

Platform. Each Windows 7 platform has a bi-core Intel i5 processor and 4GB of DDR3 1067MHz RAM. Each Mac OS 10.8 platform has either a dual-core Intel i5 or a quad-core Intel i7, and 8GB of DDR3 1067 MHz of RAM.

Methodology. We feed to the fuzzer the same initial input set, and let them run for the same duration. Once they are finished, we collect the inputs and count the unique vulnerabilities found. For discriminating whether the found individuals target the same vulnerability, we use the n latest function calls present in the stack-trace as an identifier. This is an approximation, since one vulnerability may be activated using different paths.

8.2 Research Questions

RQ3. (Fault Revealing): Does evolutionary fuzzing find more true vulnerabilities than undirected fuzzing?

We illustrate in Figure 19 the vulnerability detection capabilities of considered fuzzers on the tested applications.

Figure 19: RQ3: Vulnerability Detection Capabilities

Preliminary experimental results suggest that evolutionary fuzzing coupled with anti-random mutations has a higher fault detection capability than undirected fuzzing.

RQ4. (Coverage): What is the achieved basic block coverage?

We run the two fuzzers during the same duration and record the distinct traversed basic blocks. Results are illustrated in Figure 20. The grey color indicates that this basic block was executed, white indicates it was not executed.

Figure 20: RQ4: Basic Block Coverage

ShiftMonkey: Random ShiftMonkey: Evolutionary & Anti-Random Mutations
9. DISCUSSION

9.1 Limitations

Availability of a Parser. Our approach assumes the availability of a parser of the input format for providing several dimensions (e.g., Depth of the Production Tree and Used Grammar Production Rules). In case of non availability of such a parser, an alternative is to use another executable $e_2$ than the tested interpreter $e_1$, instrument the executable $e_2$ in grey-box, and approximate the dimensions by a coverage metrics (e.g., the number of different executed basic blocks).

Tuning of Dimensions Weights. In the process of designing the fitness function, the weight corresponding to each dimension has been setup according to our intuition of its importance. There may exist a better combination. Determining if such a combination exists is a research direction which requires to evaluate numerous combinations.

Probing for Complete Loading. Since we assume a black-box harness, ShiftMonkey is not necessarily immediately notified when the input has been completely processed. For this reason, ShiftMonkey may waste resources when periodically probing.

Reset Capability. We reset the tested applications by restarting them and deleting their state files (e.g., com.apple.Preview.savedState). Constructing a reset method is dependent of the tested application.

9.2 Threats to Validity

Randomness. Fuzzers make extensive use of randomness. Since some vulnerabilities are not trivial to be found, their discovery may involve randomness and duration. We tried to limit such factors by repeating our experiments several times. The chosen duration of the experiments may impact the results.

Considered Applications. Our comparison with other fuzzers is limited to the considered versions of fuzzers and applications. As of today, we cannot safely generalize results from those experiments. Running ShiftMonkey on other applications or fuzzers versions may produce different results.

ShiftMonkey Parameters. ShiftMonkey contains numerous adjustable parameters e.g., probabilities that drive the mutation and crossover operators during fuzzing campaigns. In Appendix E, we provide significative parameters and their default values. Those are chosen empirically because the value domain of each parameter is quite wide, and it is time consuming to run the whole test suite, it was not feasible to evaluate the combination of all parameters values and their impact. Thus, we cannot guarantee that the chosen default values achieve the best detection capabilities and efficiency.

Test Verdict. We used memory error detectors as test oracles. Since those are widely used in fuzz testing, this observation mitigates the threats to the external validity of our study.

9.3 Related Work

Black-Box Evolutionary Fuzzing. Noreen et al. evolve malwares using GA to avoid anti-viruses detection [93]. The fitness function is the singularity between an individual and the training set.

Budynek et al. evolve attacker scripts that represent commands that “script kiddies” would type [3]. Their fitness function is the discretion of the attack in terms of traces left behind and the number of achieved goals. It has a notion of attack grammar, but only for generating individuals. 1-point crossover is performed at a random point without respect of the grammar.

Duchêne et al. combines inference and evolutionary fuzzing of web applications for detecting cross-site scripting vulnerabilities. Their fitness function dimensions model the intuition of security analysts.

Black-Box Interpreters Fuzzing.

With $radamsa$ [11], Pietikäinen et al. use three families of techniques: random data enumeration, anomaly operators on samples, and model inference plus anomalization plus generation [42].

$L90, 46$ are grammar based generative fuzzers which target JavaScript interpreters. The former targets the whole language, while the latter targets a subset specifically related to operators manipulating sets of Document Object Model (DOM) nodes. These fuzzers are specific to one input format. Similarly $CSmith$ generates C programs for testing compilers [90].

We apply a statistical theory algorithm to the problem of selection of parameters (randomization seed, file seed, range). $89$ evaluate 26 scheduling algorithms and parameters in order to maximize both the vulnerability detection capabilities and efficiency.

$88$ generate programs in two phases: first they generate abstract scripts, and then they replace abstract labels with concrete identifiers (e.g., variable name, function call, etc.). They target ActionScript. They exposed previously unreported vulnerabilities in the Adobe Flash ActionScript virtual machine.

In the $LangFuzz$ approach, Holler et al. require a target grammar and code fragments known of having triggered vulnerabilities in the past. They mutate code fragments w.r.t. the targeted grammar with a limited length expansion. They exposed previously unreported JavaScript and PHP vulnerabilities [14]. $94$ infer the grammar from a training set and fuzzed w.r.t. this grammar around known accepted invalided inputs which previously exposed a vulnerability.

ShiftMonkey is a black-box fuzzer which does not require a grammar, and mutates inputs in a directed way.

Evolutionary Fuzzing in Grey-Box for Memory Corruption. $88, 16, 71, 76, 63$ target memory corruption vulnerabilities in a grey-box harness. $88, 82$. Mamba is a GA fuzzer which specifically targets Mac OS X file parsers. $63$ evolve inputs using GA towards previously statically detected potential sinks. $16$. fitness function is the number of executed basic blocks and the singularity of inputs. It performs random 1-point crossover and 2-points mutation.
fitness function is the distance between the executed basic blocks and the targeted sinks.

Moreover, there exists a variety of frameworks for building fuzzers: [34]. They support grey-box or black-box harness, require a model, and mutate the inputs in an undirected way.

Grey-box fuzzers benefit from additional knowledge (potential sinks, number of traversed basic blocks). Thus they generally achieve higher fault detection capabilities and efficiency than black-box fuzzers. However, they require the tester to put more effort into harnessing or recompiling (for white-box fuzzers) the application.

10. CONCLUSION

We presented techniques of interest for security testers: model inference, evolutionary fuzzing, taint inference, anti-random testing, heuristic-driven fuzzing. We specifically addressed the two following problems:

KameleonFuzz is the first black-box GA driven fuzzer to target Type-1 and 2 XSS. As compared to previous work, our precise double taint inference can reuse real-world parsers, our evolution is conformant to a tester defined attack grammar, and a fitness function drives the process by focusing on the most promising potential vulnerabilities. Our approach is of practical use to detect XSS, and outperforms state-of-the-art open source black-box scanners. It uncovered previously unknown XSS [21].

ShiftMonkey is one of the first black-box evolutionary fuzzer targeting memory corruption vulnerabilities. ShiftMonkey is driven by a genetic algorithm, which itself is guided by our fitness function. In our preliminary empirical evaluation, ShiftMonkey has shown to have a higher vulnerability detection efficiency and capabilities than traditional black-box fuzzers.

We consider the following directions interesting for future work: How to automatically create an attack grammar? How to combine our approach and [12] to increase the efficiency of XSS detection? How to improve the inferred model using additional knowledge gathered during the fuzzing? How to apply such a combination of model inference plus fuzzing to other class of vulnerabilities? [21, 24]. Finally, there is room for research on applying other evolutionary algorithms for guiding the fuzzing, but also on combining input format inference and evolutionary fuzzing [12, 24, 22].

References

[14] ShiftMonkey has shown to have a higher vulnerability detection efficiency than black-box fuzzers. However, they require the tester to put more effort into harnessing or recompiling (for white-box fuzzers) the application.


Finally, there is room for research on applying other evolutionary algorithms for guiding the fuzzing, but also on combining input format inference and evolutionary fuzzing [12, 24, 22].
A. TESTED PDF INTERPRETERS

Adobe Acrobat Reader is a PDF reader use at large scale in corporations.

Apple Preview is the default PDF reader shipped in the Apple Mac OS X operating system.

Apple Safari is the default browser shipped in Mac OS.

Fox-It Reader is a lightweight PDF reader mainly for Windows OS.

Sumatra PDF reader is an open-source PDF and e-book reader.

B. MESSAGEBOXES

Figure 21 contains examples of messageboxes we observed during the fuzzing.

C. TESTED WEB APPLICATIONS

P0wnMe v0.3 is an intentionally vulnerable web application for evaluating black-box XSS scanners. It contains XSS of various complexity (transitions, filters, structure).

WebGoat v5.4 is an intentionally vulnerable web application for educating developers and testers. Its multiple XSS lessons range from message book to human resources.

Gruyere v1.0 is an intentionally vulnerable web application for educating developers and testers. Users can update their profile, post and modify snippets, and view public ones.

Elgg v1.8.13 is a social network platform used by universities, governments. Users can post messages, create groups, and search. The count-per-day plugin contains XSS.

WordPress v3 is a blogging system: the blogger can create posts and tune parameters. Visitors can post comments, and search. The count-per-day plugin contains XSS.

D. WEB FUZZERS CONFIGURATION

We here list the main settings used during experiments. We also configure authentication credentials (cookie or username and login), but do not describe such settings here.

- Wapiti 2.20: -m "-all,xss"
- w3af 1.2 kali 1.0:
misc-settings
set maxThreads 1
set maxDepth 200
set maxDiscoveryTime 18000
back
plugins
discovery webSpider
discovery config webSpider
  set onlyForward True
back
plugins
discovery webSpider
discovery config webSpider
  set onlyForward True
back
start

Listing 2: w3af configuration

• SkipFish 2.10b: \(-Y \ -Z \ -m \ 10 \ -k \ 18000\)

• LigRE:
taint_flow.min_length = 6 characters
taint_flow.max_length = 8 HTTP requests

• common parameters in KameleonFuzz 2013-08-31
  are mentioned in Table 10.

Table 10: Common parameters in KameleonFuzz and their default values. See Section 5.3 on how we chose those default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LigRE.targeted_reflections</td>
<td>0.8</td>
</tr>
<tr>
<td>GA.population_size</td>
<td>5</td>
</tr>
<tr>
<td>GA.elitism</td>
<td>4</td>
</tr>
<tr>
<td>GA.mutation_proba</td>
<td>0.5</td>
</tr>
<tr>
<td>GA.crossover_n_exchanges</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 11: Parameters in ShiftMonkey and their default value. For explanations on the methodology for choosing them, see Section 9.2.

F. ATTACK GRAMMAR

Listing 3 contains an excerpt of the attack grammar. The fuzzed value in Figure 4 was generated using this grammar.

1 START = REPRESENTATION CONTEXT
2 REPRESENTATION = CHARSET ENCODING ANTI_FILTER
3 CHARSET = ( "utf8" | "iso-8859-1" | ... )
4 ENCODING = ( "plain" | "base64_encode" | ... )
5 ANTI_FILTER = ( "identity" | "php_addslashes " | ... )
6 CONTEXT = ( ATTRIBUTE_VALUE | OUTSIDE_TAG | ... )
7 ATTRIBUTE_VALUE = TEXT QUOTE SPACES HANDLER "=" QUOTE JS_PAYLOAD QUOTE
8 HANDLER = ( "onload" | "onerror" | ... )
9 JS_PAYLOAD = ( JS_P0 | JS_P1 | ... )
10 JS_P1 = "alert("NUMS ")"
11 NUMS = [5:10](NUM)
12 NUM = ("0" | "1" | "2" | ... | "9")
13 QUOTE = ("" | "\" | "\" | "\" | ... )
14 SPACES = [1:3](SPACE)
15 SPACE = (" " | "\n" | "\t" | "\r")
16 TEXT = [0:9](LETTER)
17 LETTER = ("a" | "b" | ... )