KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection

Fabien Duchene*
[lastname]@car-online.fr

Sanjay Rawat*
IIIT
Hyderabad, India
sanjayr@ymail.com

Jean-Luc Richier,
Roland Groz
LIG Lab
Grenoble F-38402, France
[first].[last]@imag.fr

ABSTRACT

Fuzz testing 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? Where to observe its effects? etc.

In this paper, we specifically address the questions: *How* to fuzz a parameter? *How* to observe its effects? To address these questions, we propose *KameleonFuzz*, a black-box Cross Site Scripting (XSS) fuzzer for web applications. KameleonFuzz can not only generate malicious inputs to exploit XSS, but also detect how close it is revealing a vulnerability. The malicious inputs generation and evolution is achieved with a genetic algorithm, guided by an attack grammar. A double taint inference, up to the browser parse tree, permits to detect precisely whether an exploitation attempt succeeded.

Our evaluation demonstrates no false positives and high XSS revealing capabilities: KameleonFuzz detects several vulnerabilities missed by other black-box scanners.

Categories and Subject Descriptors

[**Security and privacy**]: Systems security — *Vulnerability management*, *Vulnerability scanners*

General Terms

Security Testing

Keywords

Cross-Site Scripting, Fuzzing, Evolutionary Algorithm, Black-Box Security Testing, Taint Inference, Model Inference

*At the time of this work, Fabien and Sanjay were working at LIG, INP Grenoble, France.

CODASPY'14, March 3–5, 2014, San Antonio, Texas, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2278-2/14/03 $15.00.
http://dx.doi.org/10.1145/2557547.2557550.

1. INTRODUCTION

Context. Over the past years, XSS has, infamously, maintained its position in the top vulnerabilities[31]. Criminals use XSS for performing malicious activities e.g., spam, malware carrier, or user impersonation on websites like Paypal, Facebook, and eBay [25, 29, 53]. Due to the complexity and code size of such websites, automatic detection of XSS is a non-trivial problem. In case of access to the source code, white-box techniques range from static analysis to dynamic monitoring of instrumented code. If the binary or the code are inaccessible, black-box approaches generate inputs and observe responses. Such approaches are independent of the language used to create the application, and avoid a harness setup. As they mimic the behaviors of external attackers, they are useful for offensive security purposes, and may test defenses such as web application firewalls. Automated black-box security testing tools for web applications have long been around. However, even in 2012, the fault detection capability of such tools is low: the best ones only detect 40% of non-sanitized Type-2 XSS, and 1/3 do not detect any[3, 2]. This is due to an imprecise learned knowledge, approximate verdicts, and limited sets of attack values.

Automatic black-box detection of web vulnerabilities generally consists in first "crawling" to infer the *control flow* of the application (hereinafter 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[10]. In LigRE, Duchène et al. extends such models with taint flow inference and guides a fuzzer to improve detection capabilities one step further[16]. 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 LigREextension 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 [16]. The main contributions of this paper are the blocks *D1* and *D2*.

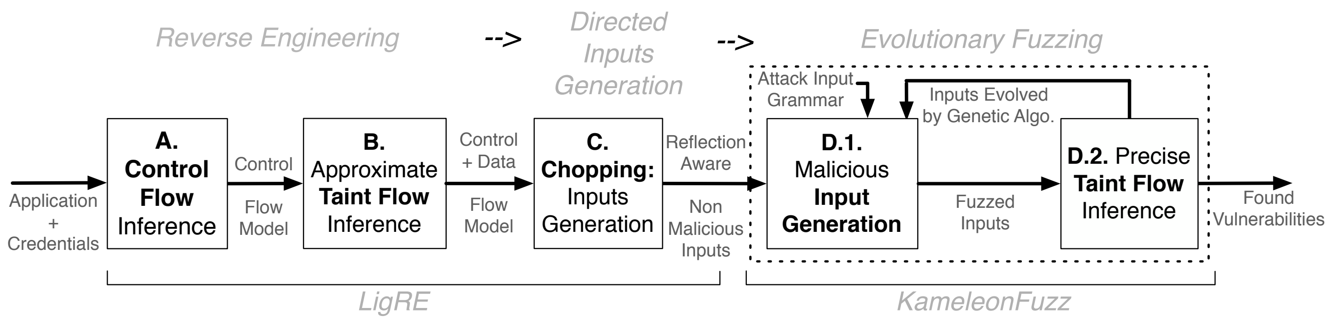

Reverse Engineering
-->
Directed Inputs Generation
-->
Evolutionary Fuzzing
Application + Credentials
A. Control Flow Inference
Control Flow Model
B. Approximate Taint Flow Inference
Control + Data Flow Model
C. Chopping: Inputs Generation
Reflection Aware Non Malicious Inputs
Attack Input Grammar
Inputs Evolved by Genetic Algo.
D.1. Malicious Input Generation
Fuzzed Inputs
D.2. Precise Taint Flow Inference
Found Vulnerabilities
LigRE
KameleonFuzz

Figure 1: High Level Approach Overview

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.

The blocks D.1 (malicious input generation) and D.2 (precise taint flow inference) are the main focus of this paper. 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. The contributions of this paper are:

- 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.

The rest of the paper is organized as follows. Section 2 provides a walk-through of our approach over an example. Section 3 details how malicious inputs are generated and evolved. Section 5 evaluates KameleonFuzz on typical web applications. Finally, we discuss our approach in Section 6, survey related work in Section 7, and conclude in Section 8.

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, LigREinfers a control+taint flow model of which a simplified extract[1] 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 $18 \rightarrow 21$.

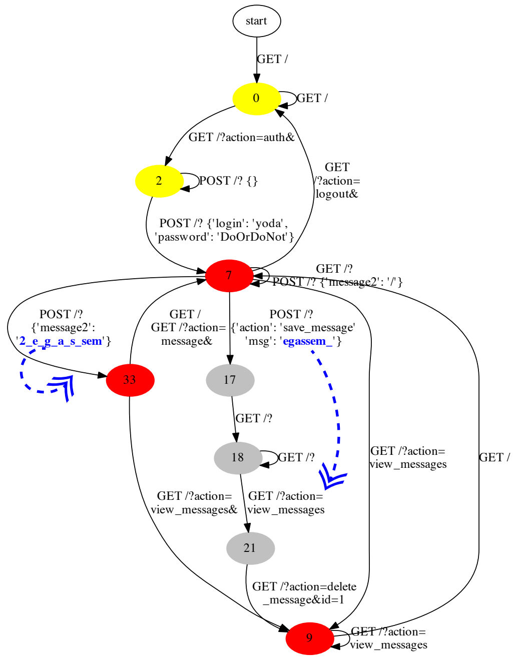

start
GET /
0
GET /
GET /?action=auth&
2
POST /? {}
GET /?action= logout&
POST /? {'login': 'yoda', 'password': 'DoOrDoNot'}
7
GET /? POST /? {'message2': '/'}
POST /? {'message2': '2_e_g_a_s_sem'}
GET / GET /?action= message&
POST /? {'action': 'save_message' 'msg': 'egassem_'}
33
17
GET /?
18
GET /?
GET /?action= view_messages
GET /
GET /?action= view_messages&
GET /?action= view_messages
21
GET /?action=delete _message&id=1
9
GET /?action= view_messages

Figure 2: Inferred control+taint model (extract)

[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'}`). [16] formalizes a control+taint flow model.

Figure 2 contains a reflection for the value `2_e_g_a_s_sem` of the parameter *message*2 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 1 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[45] i.e., it spans over different levels in the parse tree.

```php
<?php function webapp_filter($str) {
    if(eregi('"|`|>|<|;|/',$str)) {
        $filtered_str = "XSS attempt!";
    } else {
        $filtered_str = str_replace(" ","",$str);
    }
    return $filtered_str;
} ?>
```

Listing 1: A vulnerable sanitizer in P0wnMe

Table 1 shows fuzzed values sent by w3af[36], 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.

Fuzzed Value (x_{src})	**Reflection**
`SySlw`	`SySlw`
`uI<hf>hf"hf'hf(hf)uI`	`XSS attempt!`
`</A/style="xss:exp/**/ression(fake_alert('XSS'))">`	
`";!-"<klqn>=&{()}`	
`<IFRAME SRC="javascript:fake_alert('klqn');"></IFRAME>`	

Table 1: w3af fuzzed values (extract)

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 an input 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). Table 2 illustrates this evolution.

An extract of the output O_{dst} for the last individual is `<input name="message2" value=' WUkp'\t onload='alert(94478) '/>`
Since the sanitizer in Listing 1 removes the space ␣, and not \t,\r or \n, the individual is a successful XSS exploit, as the syntactic confinement of the reflection of x_{src} is violated.

This example illustrates how evolutionary input generation can adapt to sanitizers. In the next section, we elaborate on the evolutionary nature of our fuzzing technique.

Fuzzed Value (x_{src})	**Reflection**	XSS Fit.	Gen.
`T9nj1'><script>alert(18138)</script>`	`XSS␣Attempt!`	3.1	1
`oH1eqL'␣onload="document.body.innerHTML+='<div␣id=90480></div>'"␣fakeattr='`	`XSS␣Attempt!`	3.2	1
`ZuIa2'␣onload=alert(94478)`	`ZuIa2'onload=alert(94478)`	13.3	1
`WUkp'\tLgpRa`	`WUkp'\tLgpRa`	9.1	1
`WUkp'\t␣onload='alert(94478)`	`WUkp'\tonload='alert(94478)` ✓	18.5	2

Table 2: KameleonFuzz fuzzed values (extract)

3. EVOLUTIONARY 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).

```
                              ▷ Create the first generation
for $l \in [1..n]$ do
    Popul[l] ← newIndividual(Chopping, Attack_Grammar)
end for
                              ▷ Evolve the population
repeat
    for all individual I(x) in Popul do
        RESET the Application
        O = SEND I(x) to Application
        T = precise TAINT_INFERENCE(x,O,Parser)
        Compute VERDICT(x,T,Patterns)
        Compute FITNESS(I,x,O,T,Model)
    end for
    CROSSOVER: $f$ fittest individuals to produce $m$ Children
    for all C in Children do
        if random(0,1) ≤ MutationRate then
            MUTATE(C,Attack_Grammar)
        end if
    end for
    Popul ← $(n - m)$ fittest parents + $m$ children
until stopCondition
```

Algorithm 1: Genetic Algorithm pseudo-code

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}.

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[49], string transformations in case of context change [51], known attacks vectors [38, 21].

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[32]).

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[40] 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 C contains an excerpt of the attack grammar.

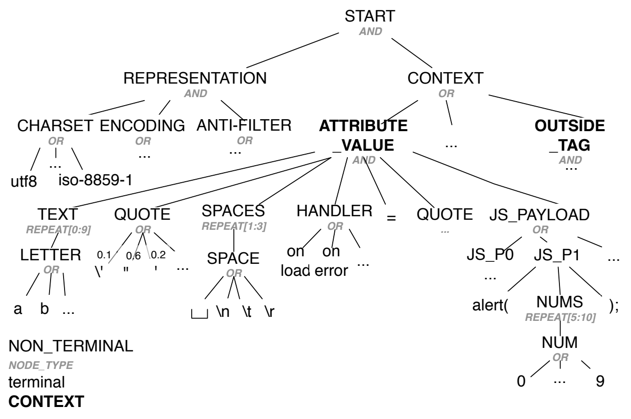


Figure 3: Structure of the attack grammar (extract)

Generating a fuzzed value consists in walking through its production rules and, if applicable, performing choices. Producing the corresponding string from a fuzzed value consists in concatenating the strings obtained by a depth-first exploration of the context subtree, representing this string in a given charset, applying the anti-filter function, and applying an encoding function. For instance, the string that results from the fuzzed value of Figure 4 is `WUkp'␣\t onload='alert(94478)`, in `UTF-8` charset, on which the `identity` function is applied as an anti-filter, and with no final encoding change (node `plain`).

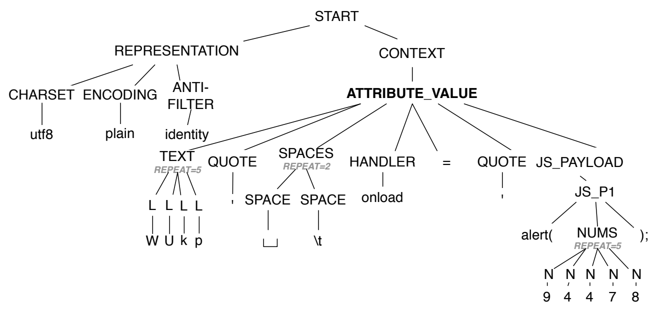


Figure 4: The Production Tree of a Fuzzed Value

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 [26]) is applied between the fuzzed value x_{src} and the output

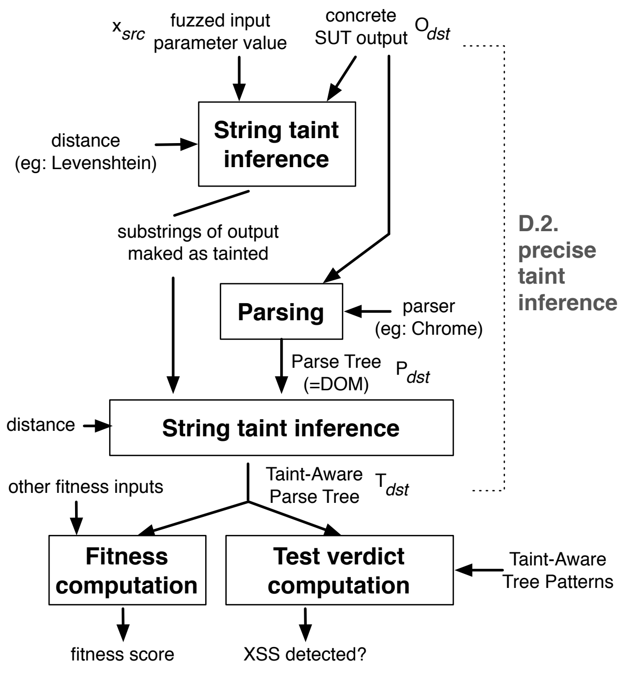


Figure 5: Precise Taint Inference ($I_{src} \to O_{dst} \to T_{dst}$)

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 [41], 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 7). 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.

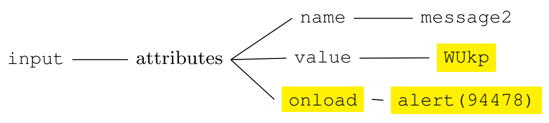


Figure 6: A Taint-Aware Parse Tree T_{dst} (extract). The payload is a message box that displays 94478 (harmless).

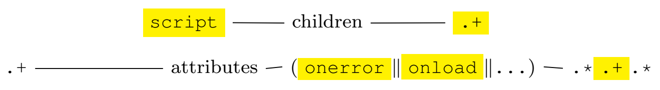


Figure 7: Two Taint-Aware Tree Patterns, represented in a Linear Syntax (resp. a tainted script tag content and a tainted event handler attribute)

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}, sent in the transition I_{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 [17, 15], we drew a sketch of the currently used fitness function.

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

Table 3: Dimensions of the `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 important, 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 [8]. 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_{dst}).

- 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 wether 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 leafs. 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 consists in performing a different choice for the HANDLER non terminal (e.g., `onmousover` 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. 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 8, we instrument Google Chrome[18] with the Selenium library[24]. We use LigRE, a control+taint flow model inference tool and slicer.

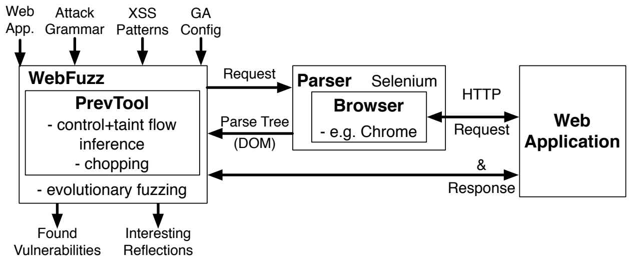


Figure 8: Architecture of KameleonFuzz

5. 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.

5.1 Test Subjects

As described in Table 4, we select seven **web applications** of various complexity. In Appendix A, 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 B contains the configuration we used during the experiments. It is important to note that only LigREand KameleonFuzz are macro-state aware.

Application	Description	Version	Plugins
P0wnMe	Intentionally Vulnerable	0.3	
WebGoat	Intentionally Vulnerable	5.4	
Gruyere	Intentionally Vulnerable	1.0	
WordPress	Blog	3.2.1	Count-Per-Day 3.2.3
Elgg	Social Network	1.8.13	
phpBB	Forum	2.0	
e-Health	Medical	04/16/2013	

Table 4: Tested Web Applications

5.2 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.

5.3 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 loose 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:

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

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

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

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 LigREand KameleonFuzz, are not macro-state aware we configure them to exclude requests that would irreversibly 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 9 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 $\frac{32}{35} = 91.4\%$. 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.

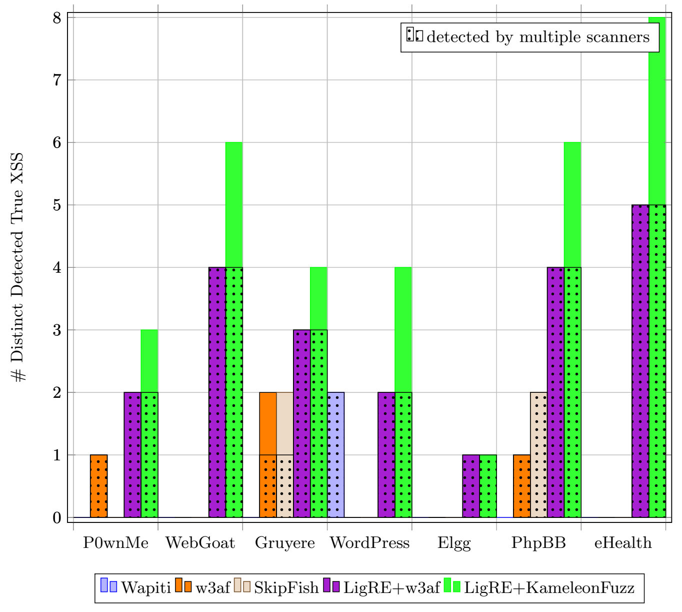
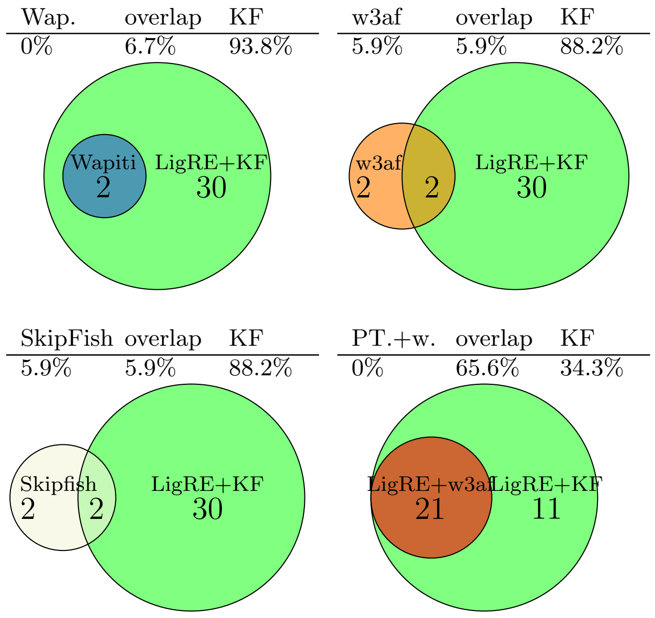


Figure 9: **Detection Capabilities of Black-Box XSS Scanners**

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.



Wap.	overlap	KF
0%	6.7%	93.8%

w3af	overlap	KF
5.9%	5.9%	88.2%

SkipFish	overlap	KF
5.9%	5.9%	88.2%

PT.+w.	overlap	KF
0%	65.6%	34.3%

Figure 10: Number of True XSS found, only_by, and overlap of LigRE+KameleonFuzz and other scanners

LigRE+KameleonFuzz detects more true XSS than other scanners. It has no false positive.
KameleonFuzz increases XSS detection capabilities.
The non null only_by of w3af and Skipfish suggest they are complementary to KameleonFuzz.

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

Application	Potential Reflections	Generations to detect the found XSS	Transitions $start \rightarrow O_{dst}$			True XSS Found	False Positive
			1st	2nd	3rd		
P0wnMe	37	3	4	6	7	**3**	0
WebGoat	134	2	6	7	7	**6**	0
Gruyere	23	4	2	2	7	**4**	0
WordPress	52	2	2	2	5	**4**	0
Elgg	59	1	6			**1**	0
PhpBB	213	4	5	5	6	**6**	0
e-Health	12	1	4	4	4	**8**	0

Table 5: KameleonFuzz detection capabilities on the considered applications

HTTP requests. Thus, we set up a proxy between the scanner 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.

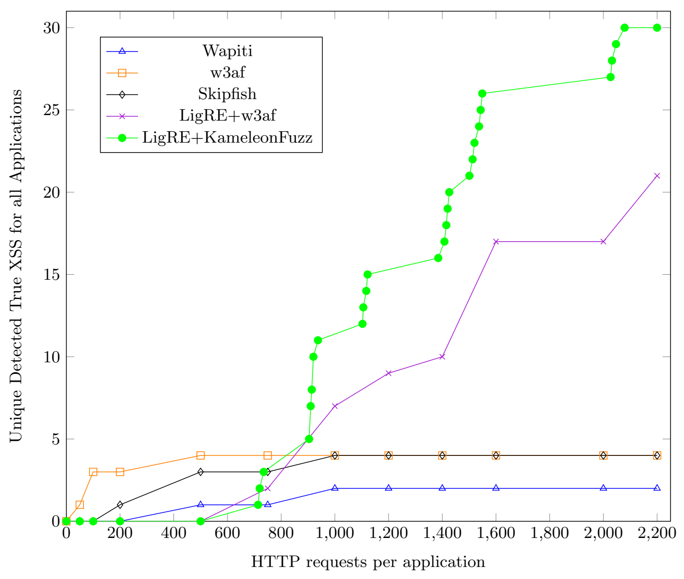


Figure 11: Detection Efficiency of Black-Box XSS Scanners

On considered applications, below approximately 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, LigREimproves 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 KameleonFuzz detects 42.9% more XSS than fuzzing with w3af. On the LigRE+w3af and LigRE+KameleonFuzz curves, we can observe several landings, which mostly correspond to the end of the LigREcontrol+taint flow inference for a given application.

If the cost of LigREinference is acceptable, then LigRE+KameleonFuzz is more efficient than LigRE+w3af. Otherwise, w3af alone is of interest.

6. DISCUSSION

6.1 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

Vulnerability	Output Grammar	Where to Parse?
Cross Site Scripting	HTML	HTML page
HPP Param. Pollution	HTTP	Reply Headers
PHP Code Injection	PHP	argument of `eval`
SQL Injection	SQL	arg. of `sql_query`
Shell Injection	Shell	...`exec`, `system`

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

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[20, 19].

6.2 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[50].

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[7]. To our knowledge, no scanner handles such cases.

Limitations due to the use of LigRE: KameleonFuzz supports Ajax applications if they offer similar functionality when the client does not interpret JavaScript. LigR-Erequires to identify non deterministic values in the applications [14]. Hossen et al. automated this identification[23].

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

6.3 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 those 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 B, 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.

7. RELATED WORK

7.1 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[41], 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 5). 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. [45] relies on non-lexical confinement as a sufficient fault detection measure, which is more efficient than [41], but requires a correctly formed output (which is not an always valid assumption on HTML webpages[20]) 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[36]. Moreover, most do not consider the reflection context, which can lead to false positive. IE8 [37] and NoScript [27] rely on regular expression on fuzzed values. XSSAuditor (Chrome XSS filter) performs exact string matching with JavaScript DOM nodes[1].

String Distance Based Approaches Sun[46] detects self-replicating XSS worms by computing a string distance between DOM nodes and requests performed at run-time by the browser.

IE8[37] and Chrome XSSAuditor[1] 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.

7.2 Learning and Security Testing

In its basic form, **fuzzing** is an undirected black-box active testing technique[28]. [52, 48, 22, 39] mainly targets memory corruption vulnerabilities. Stock et al.'s recent work fuzzes and detects Type-0 XSS in a white-box harness[44]. Heiderich et al. detect in black-box mutation-based XSS caused by browser parser quirks[19]. 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[30] and attacker scripts[6]. *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. [50, 47] and KameleonFuzz share this view. In their recent work[47], 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[50].

Model Inference for Security Testing Radamsa targets memory corruption vulnerabilities: it infers a grammar from known inputs then fuzzes to create new inputs[33]. Shu et al. passively infer a model from network traces, and actively fuzz inputs[42]. [34, 4, 5] 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[9]. Sotirov iterates between reverse-engineering of XSS filters, local fuzzing, and remote fuzzing[43]. Doupé et al. showed that inferring macro-state aware control flow models increases vulnerability detection capabilities[10]. With LigRE, Duchène et al. showed that enhancing such models with taint flows increases its capabilities even more[16].

KameleonFuzz extends LigREand 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.

8. CONCLUSION AND FUTURE WORK

In this paper, we present KameleonFuzz, the first black-box GA driven fuzzer targeting 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 [11].

We consider the following directions interesting for future work: How to automatically create an attack grammar? How to combine our approach and [47] 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? [13, 12, 35]

Acknowledgments

This work was supported by the projects ITEA2 №09018 DIAMONDS "Development and Industrial Application of Multi-Domain Security Testing Technologies", FP7-ICT №257876 SPaCIoS "Secure Provision and Consumption in the Internet of Services", ANR-INSE №0002-01 "BinSec BINary code analysis for SECurity", and the Rhône-Alpes region, INSA Lyon, CNRS Liris for the ARC6 grant.

References

[1] D. Bates, A. Barth, and C. Jackson. "Regular expressions considered harmful in client-side XSS filters". *WWW*. 2010, pp. 91–100.

[2] J. Bau et al. "State of the Art: Automated BlackBox Web Application Vulnerability Testing". *IEEE S&P*. 2010, pp. 332–345.

[3] J. Bau et al. *Vulnerability Factors in New Web Applications: Audit Tools, Developer Selection & Languages*. Tech. rep. Stanford, 2012.

[4] S. Bekrar et al. "A Taint Based Approach for Smart Fuzzing". *SECTEST with ICST*. 2012, pp. 818–825.

[5] S. Bekrar et al. "Finding Software Vulnerabilities by Smart Fuzzing". *International Conference on Software Testing, Verification, and Validation*. 2011, pp. 427–430.

[6] J. Budynek, E. Bonabeau, and B. Shargel. "Evolving computer intrusion scripts for vulnerability assessment and log analysis". *GECCO*. ACM, 2005.

[7] S. Dalili. *Browsers anti-XSS methods in ASP (classic) have been defeated!* 2012. URL: `http://soroush.secproject.com/downloadable/Browsers_Anti-XSS_methods_in_ASP_(classic)_have_been_defeated.pdf`.

[8] J. D. DeMott, R. J. Enbody, and W. F. Punch. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing". *Black Hat USA* (2007).

[9] A. Dessiatnikoff et al. "A clustering approach for web vulnerabilities detection". *17th PRDC*. IEEE. 2011, pp. 194–203.

[10] A. Doupé et al. "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner". *Usenix Sec* (2012).

[11] F. Duchene. *0-day XSS discovered with KameleonFuzz*. 2014. URL: `http://car-online.fr/0_day_xss_kameleonfuzz`.

[12] F. Duchene. "Harder, Better, Faster Fuzzer : Advances in Black-Box Evolutionary Fuzzing". *Hack In The Box (HITB)*. Amsterdam, Netherlands, 2014.

[13] F. Duchène. "Fuzz in the Dark: Genetic Algorithm for Black-Box Fuzzing". *Black-Hat*. São Paulo, Brazil, 2013.

[14] F. Duchène et al. "A Hesitation Step into the Black-box: Heuristic based Web Application Reverse Engineering". *NoSuchCon*. 2013.

[15] F. Duchène et al. "Fuzzing Intelligent de XSS Type-2 Filtrés selon Darwin: KameleonFuzz. Fuzzing Evolutionnaire de XSS Type-2 en Boîte Noire". *11th SSTIC*. 2013, pp. 289–311.

[16] F. Duchène et al. "LigRE: Reverse-Engineering of Control and Data Flow Models for Black-Box XSS Detection". *20th WCRE*. IEEE, 2013, pp. 252–261.

[17] F. Duchène et al. "XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing". *SECTEST with ICST*. 2012, pp. 815–817.

[18] Google. *Chrome*. URL: `https://www.google.com/chrome/`.

[19] M. Heiderich et al. "mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations". *CCS*. ACM, 2013.

[20] M. Heiderich et al. *Web Application Obfuscation:'-/WAFs.. Evasion.. Filters//alert (/Obfuscation/)-'*. Syngress, 2010.

[21] G. Heyes et al. *Shazzer - Shared XSS Fuzzer*. 2012. URL: `http://shazzer.co.uk`.

[22] C. Holler, K. Herzig, and A. Zeller. "Fuzzing with Code Fragments". *21st Usenix Security*. 2012.

[23] K. Hossen, R. Groz, and J.-L. Richier. "Security Vulnerabilities Detection Using Model Inference for Applications and Security Protocols". *SECTEST with ICST*. IEEE, 2011, pp. 534–536.

[24] J. Huggins, P. Hammant, et al. *Selenium, Browser Automation Framework*. URL: `http://code.google.com/p/selenium/`.

[25] R. Kugler. *PayPal.com XSS Vulnerability*. 2013. URL: `http://seclists.org/fulldisclosure/2013/May/163`.

[26] V. Levenshtein. "Binary coors capable of correcting deletions, insertions, and reversals". *Soviet Physics-Doklady*. Vol. 10. 1966.

[27] G. Maone. *NoScript, Firefox plug-in*. 2006. URL: `https://addons.mozilla.org/en-US/firefox/addon/noscript/`.

[28] B. Miller, L. Fredriksen, and B. So. "An empirical study of the reliability of operating system utilities". *Communications of The ACM* (1989).

[29] Nirgoldshlager. *Stored XSS In Facebook*. 2013. URL: `http://www.breaksec.com/?p=6129`.

[30] S. Noreen et al. "Evolvable malware". *GECCO*. ACM, 2009, pp. 1569–1576.

[31] OWASP. *Top Ten Project*. 2013.

[32] PHP. *addslashes function*. URL: http://php.net/manual/en/function.addslashes.php.

[33] P. Pietikäinen et al. “Security Testing of Web Browsers”. *Comm. of Cloud Software, vol. 1, no. 1, Dec. 23, ISSN 2242-5403* (2011).

[34] S. Rawat and L. Mounier. “An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A Case of Aiming in Dim Light”. *EC2ND*. 2010.

[35] S. Rawat et al. “Evolving Indigestible Codes: Fuzzing Interpreters with Genetic Programming”. *CICS, with SSCI*. IEEE, 2013, pp. 37–39.

[36] A. Riancho. *w3af - WebApp. Attack and Audit Framework*. 2011. URL: http://w3af.sourceforge.net.

[37] D. Ross. *IE 8 XSS Filter Implementation*. 2008. URL: http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[38] RSnake. *XSS Cheat Sheet Esp: for filter evasion*. 2007. URL: http://ha.ckers.org/xss.html.

[39] J. Ruderman. *Introducing jsfunfuzz*. 2007. URL: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz.

[40] R. S. Scowen. “Extended BNF — A generic base standard”. *SESP*. 1993.

[41] R. Sekar. “An Efficient Blackbox Technique for Defeating Web Application Attacks”. *NDSS*. 2009.

[42] G. Shu and D. Lee. “Testing Security Properties of Protocol Implementations - a Machine Learning Based Approach”. *ICDCS*. IEEE, 2007.

[43] A. Sotirov. “Blackbox Reversing of XSS Filters”. *ReCon*. 2008.

[44] B. Stock, S. Lekies, and M. Johns. “25 Million Flows Later - Large-scale Detection of DOM-based XSS”. *20th CCS*. ACM, 2013.

[45] Z. Su and G. Wassermann. “The essence of command injection attacks in web applications”. *POPL*. 2006.

[46] F. Sun, L. Xu, and Z. Su. “Client-Side Detection of XSS Worms by Monitoring Payload Propagation”. *ESORICS* (2009), pp. 539–554.

[47] O. Tripp, O. Weisman, and L. Guy. “Finding your way in the testing jungle: a learning approach to web security testing”. *ISSTA*. ACM, 2013, pp. 347–357.

[48] R. Valotta. “Fuzzing with DOM Level 2 and 3”. *DeepSec*. 2013.

[49] W3C. *HTML5 Content Model*. 2012. URL: http://www.w3.org/TR/html5/content-models.html.

[50] Y.-H. Wang, C.-H. Mao, and H.-M. Lee. “Structural Learning of Attack Vectors for Generating Mutated XSS Attacks”. *Computing Research Repository* (2010).

[51] J. Weinberger et al. “A systematic analysis of XSS sanitization in web application frameworks”. *ESORICS*. Springer, 2011, pp. 150–171.

[52] M. Zalewski. *Announcing CrossFuzz*. 2011. URL: http://lcamtuf.blogspot.fr/2011/01/announcing-crossfuzz-potential-0-day-in.html.

[53] ZentrixPlus. *eBay Sec. Hall of Fame*. 2013. URL: http://zentrixplus.net/blog/ebay-security-researchers-hall-of-fame-hof/.

APPENDIX

A. 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, update their profile. An XSS exists since several versions.

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.

PhpBB v2 is a forum platform. We include this version, as it is famous to contain several XSS[2].

e-Health 04/16/2013 is an extract of an industrial medical platform used by patients and practitioners.

B. 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
audit xss
audit config xss
    set numberOfChecks 3
    back
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 7.

Parameter	Default Value
LigRE.targeted_reflections – The percentage of reflections that KameleonFuzz will focus on. LigREorders them in descending order of potential interest[16].	0.8
GA.population_size – The size of the population i.e., the number of individuals. The actual amount is this value times the number of targeted LigREreflections.	5
GA.elitism – Number of individuals having the highest fitness score that are kept for the next generation.	4
GA.mutation_proba – The probability to apply a mutation operator on a new child.	0.5
GA.crossover_num_exchanges – Number of exchanges performed by the crossover operator. One exchange means a two points crossover i.e., for the whole sub-tree of the exchanged grammar (non)-terminal.	1

Table 7: Common parameters in KameleonFuzz and their default values. See Section 6.3 on how we chose those default values.

C. ATTACK GRAMMAR

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

```
START = REPRESENTATION CONTEXT
REPRESENTATION = CHARSET ENCODING
    ANTI_FILTER
CHARSET = ( "utf8" | "iso-8859-1" | ... )
ENCODING = ( "plain" | "base64_encode" |
    ... )
ANTI_FILTER = ( "identity" | "php_addslashes
    " | ... )
CONTEXT = ( ATTRIBUTE_VALUE | OUTSIDE_TAG |
    ... )
ATTRIBUTE_VALUE = TEXT QUOTE SPACES HANDLER
    "=" QUOTE JS_PAYLOAD QUOTE
HANDLER = ( "onload" | "onerror" | ... )
JS_PAYLOAD = ( JS_P0 | JS_P1 | ... )
JS_P1 = "alert(" NUMS ")"
NUMS = [5:10](NUM)
NUM = ("0" | "1" | "2" | ... | "9")
QUOTE = ("'" | "\"" | "" | "\\'" | ...)
SPACES = [1:3](SPACE)
SPACE = (" " | "\n" | "\t" | "\r")
TEXT = [0:9](LETTER)
LETTER = ("a" | "b" | ...)
```

Listing 3: attack grammar (excerpt)