LigRE: Reverse-Engineering of Control and Data Flow Models for Black-Box XSS Detection

Fabien Duchène
LIG Lab, Grenoble INP Ensimag
Grenoble F-38402, France
[lastname]@car-online.fr

Sanjay Rawat
LIG Lab, Grenoble INP Ensimag
Grenoble F-38402, France
sanjayr@ymail.com

Jean-Luc Richier, Roland Groz
LIG Lab, Grenoble INP Ensimag
Grenoble F-38402, France
{richier,groz}@imag.fr

***Abstract*—Fuzz testing consists of automatically generating and sending malicious inputs to an application in order to hopefully trigger a vulnerability. In order to be efficient, the fuzzing should answer questions such as: Where to send a malicious value? Where to observe its effects? How to position the system in such states? Answering such questions is a matter of understanding precisely enough the application. Reverse-engineering is a possible way to gain this knowledge, especially in a black-box harness. In fact, given the complexity of modern web applications, automated black-box scanners alternatively reverse-engineer and fuzz web applications to detect vulnerabilities.**

We present an approach, named as LigRE, which improves the reverse engineering to guide the fuzzing. We adapt a method to automatically learn a control flow model of web applications, and annotate this model with inferred data flows. Afterwards, we generate slices of the model for guiding the scope of a fuzzer.

Empirical experiments show that LigRE increases detection capabilities of Cross Site Scripting (XSS), a particular case of web command injection vulnerabilities.

***Index Terms*—Control Flow Inference, Data-Flow Inference, XSS, Web Application, Reverse-Engineering, Penetration Testing**

I. Introduction

A. Context

XSS is one of the most dangerous web attacks: it ranks third in the OWASP Top 10 vulnerabilities[1]. Criminals use XSS to spam social networks, spread malwares and steal money[2]. In 2013, XSS were found in Paypal, Facebook, and eBay[3, 4, 5].

Automatically detecting XSS is an open 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-filtered Type-2 XSS, and 1/3 do not detect any[6, 7]. This is due to an imprecise learned knowledge, imprecise test verdicts, and limited sets of attack values[8].

The automatic black-box detection of web vulnerabilities generally consists of two steps: "crawling" infers the *control flow* of the application, then "fuzzing" generates malicious inputs to exhibit vulnerabilities. Doupé et al. showed that precisely inferring the control flow increases vulnerability detection capabilities[9]. XSS involve both control and data flows, as they rely on an input value being partly copied to a transition output. Therefore we propose an approach that learns both flows, and uses them by driving the fuzzing.

B. High Level Overview

LigRE is a reverse-engineering approach which guides the fuzzing towards detecting XSS vulnerabilities. As illustrated in Figure 1, it first learns a data plus control flow model, and then generates slices of this model to guide the fuzzing.

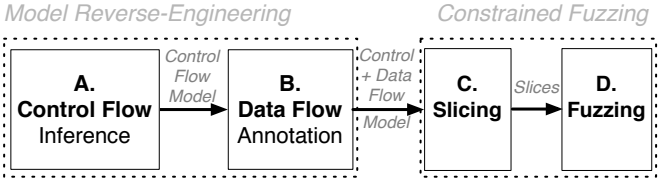

Fig. 1. High Level View of the LigRE Approach

1) Control and Data Flow Inference: Step A of Figure 1 learns the *control flow* of the application, using a state aware crawler, to maximize coverage. Step B annotates the inferred model with observable *data flows* of input values into outputs to produce a *control plus data flow model*. Annotations flow from a source t_{src} to a potential sink t_{dst}. A heuristic driven substring matching avoids false negatives.

2) Slicing and Fuzzing: The most promising annotations are prioritized. For each of them, we produce a slice of the model. Our particular slices are pruned models. They permit to drive the application to t_{src} origin, for sending a malicious value x_{src}, and then to guide the fuzzer to navigate towards t_{dst}, for observing the effects of x_{src}.

The paper is organized as follows. Section II provides a walk-through of LigRE over an example. Section III details the control flow inference. Section IV describes the data flows annotations. Section V describes how slices are generated and used to guide the fuzzing. Section VII measures the effectiveness of LigRE on typical applications. Finally, we discuss our approach in Section VIII, survey related work in Section IX, and conclude in Section X.

II. Illustrating Example

A. P0wnMe and XSS

P0wnMe is a voluntarily vulnerable web application containing several XSS. Once authenticated, a user can save a new message, view the saved ones, or logout. She saves a note, e.g., "egassem_", by filling and submitting the form, i.e., sending the abstract input `POST /?action=save_message&msg=egassem_` (transition 7 → 17) to the application. Later on, she lists the saved notes, by sending `GET /?action=get_messages` (transition 18 → 21). An extract of corresponding output is represented in Listing 1. Various server languages can produce this output.

```
1<H2>list of saved messages</H2>
2   egassem_<A href="./?action=delete&id=1">[X]</A>
```

Listing 1. Excerpt of P0wnMe Output for the Transition 18 → 21

The value of the input parameter `msg`, sent in the transition 7 → 17, is *reflected* in 18 → 21: we observe it in the output.

In this precise example, this *reflection* is not filtered: the exact value sent in 7 → 17 is copied in the output of 18 → 21. An attacker would attempt to send a malicious *msg* value to escape the confinement[10] (e.g., in Listing 1, one reflection is confined outside tags, before the `<A>` tag). An example of malicious input is 7 → 17(`POST /?action=save_message&msg=egassem_ <script> alert(1337)</script>`). An exert of the corresponding output for the subsequent transition 18 → 21 is `...of saved messages</h2> egassem_ <script> alert(1337)</script><a href="./?action=delete....`
When the victim's browser parses this output, it executes the *code* introduced by the attacker.

B. Limitations of Black-Box XSS Scanners

Most of considered open-source black box web scanners fail at detecting this XSS. The main reasons are imprecise application behavior awareness (some scanners do not navigate properly, and do not observe the reflections), imprecise test verdict (e.g., Skipfish considers a page model change to be a sufficient condition for XSS), and limited set of fuzzed values (unaware of the output structure or the filters). Our approach overcomes the first limitation using a combination of control flow inference, data flow inference and a guided fuzzing.

C. LigRE Execution on P0wnMe

LigRE infers a Control Flow Model (CFM) in the form of a colored automaton (nodes and continuous arrows of Figure 2), up to a tester defined precision. Then it walks through the model by generating HTTP requests and submitting them to the application. The corresponding responses (HTTP replies) are recorded. Data flows of sent input parameter values are inferred on the outputs, and annotated on the model (blue dashed lines on Figure 2).

Model slices are computed (see Figure 3), and prioritized. Each slice is composed of a prefix and a suffix. For instance, the prefix $[0 \rightarrow 2, 2 \rightarrow 7]$ and the suffix $[7 \rightarrow 17, 17 \rightarrow 18, 18 \rightarrow 21]$. LigRE sends the prefix to the application, then pass the authentication credentials (e.g., cookie) to the w3af fuzzer[1][11] and limits its scope to the suffix.

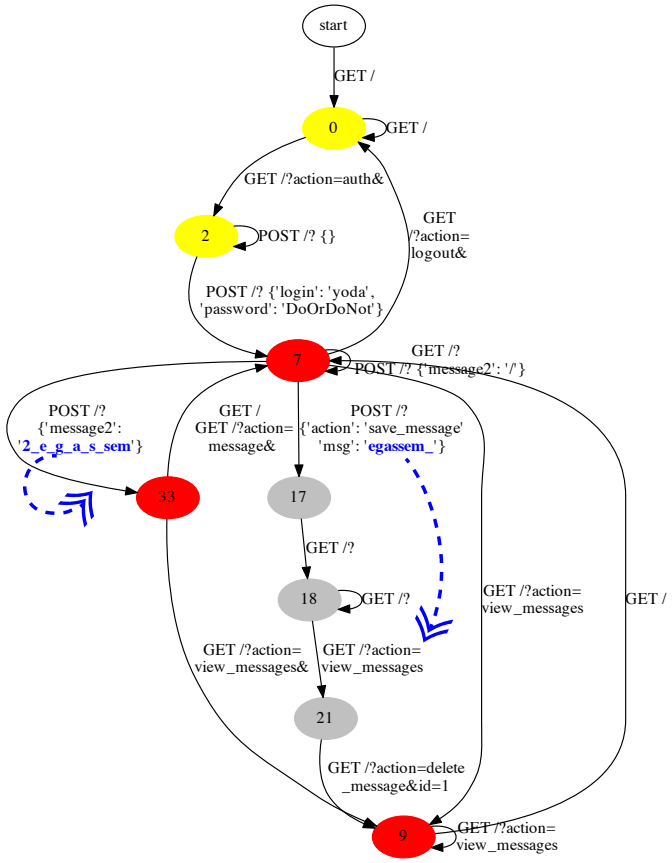

Fig. 2. Extract of the Control + Data Flow Model for the P0wnMe application

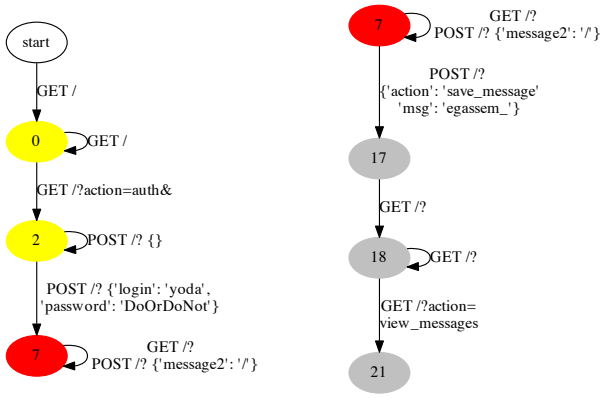

Fig. 3. A Slice produced by LigRE for the P0wnMe application (prefix on the left part, and suffix on the right)

When the fuzzing terminates, LigRE reports on the found XSS, if any, along with the corresponding HTTP requests.

[1]We use w3af as the fuzzer. Other configurable ones can also be used.

III. Heuristic based Control Flow Inference

Step A in Figure 1 takes as inputs parameters to interact with a remote web application (e.g., interface, authentication credentials), and outputs a *control flow model* (CFM). A CFM formalizes the observable behavior of a web application in a black-box harness. It is an Extended Finite-State-Machine (EFSM)[12] in which nodes (states) represent webpages and transitions represent requests and associated responses. As in [9], we color its nodes according to the *macro-state* of the application(e.g., logged-in, logged-out). A CFM is illustrated in Figure 2, if omitting the dashed blue lines. We formally define macro-state and CFM in Definition 1 and Definition 2.

The control flow inference step uses heuristics to identify which request changed the macro-state (Section III-C), choose the next request to be performed (Section III-D), and assess the degree of certainty in the model (Section III-E1). The model is iteratively built. In case of a non-coherent observation w.r.t. the model, backtracking undoes the recently built parts.

A. Abstraction Level

We define `concretize(i)` which produces an HTTP request *req* from a *spiderlink i*, and `abstract(resp)` which abstracts an HTTP response *resp* into a *page model p*, as shown in Figure 4. The inferred CFM (regular lines and nodes in Figure 2) corresponds to inputs and outputs at the *abstract level*. Those are defined as follows:

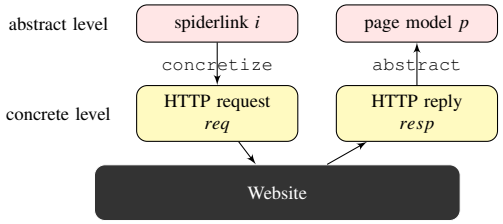

Fig. 4. Abstraction and Concretization Functions for Web Applications

1) Input, `concretize`(Spiderlink i): A *spiderlink* is a list of parameters (*name*, *value*, *method*), s.t. *method* ∈ {`GET`, `POST`, `COOKIE`, `HEADER`}. Spiderlinks are built from links and forms. A path from the root to a leaf in Figure 5 is a spiderlink.

In the process of `abstraction`, some parameters are omitted: Non Deterministic Values (NDV), or nonces[13], i.e., parameters whose values differ when sending twice an input sequence (and resetting the system in between). Examples of NDV include: anti-CSRF tokens, session_id, view_states. In the presence of NDV, crawlers achieve a limited coverage. We address this problem by requiring the user to identify NDV. NDV can be automatically detected[14].

2) Output, Abstraction to Page Model p: the website output, an HTTP Reply *resp*, is abstracted to a page model *p*(*resp*). The left side of Figure 5 represents the browser rendering of *resp*, while the right side represents the corresponding page model, which is a prefix tree. We build it by iteratively constructing a vector for each link or form in *resp*, and adding them to the tree. Each vector consists of:

- *dompath* is the shortest path in the Document Object Model from the root to the node
- *action* contains each part of the path split by /
- *params* is the list of *par* parameter names
- *values* are the *par* parameters values
- *methods* are the *par* parameters methods

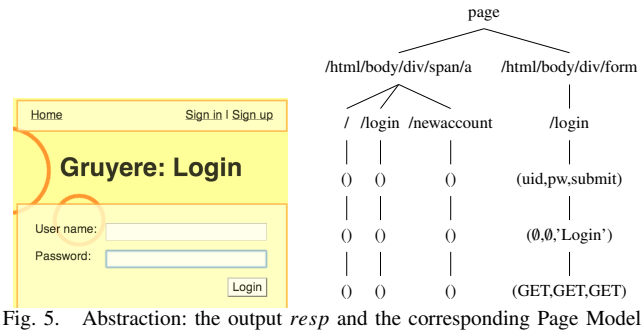

Fig. 5. Abstraction: the output *resp* and the corresponding Page Model *p*

Once a page model is built, spiderlinks can be generated by traversing it via a shortest path from the root to a leaf.

B. Vocabulary and Definitions

The history of inputs and outputs serves to build a *navigation graph*, which is used to build a *CFM*. Both are colored. Their coloration evolves to characterize the *macro-states*.

1) Macro-State: This important notion to understand the control flow of a web application designates "anything that influences the executed code at server side". Both nodes and macro-states represent the current execution context of the web application. They differ in their granularity. A node is characterized by a page (i.e., the last output). Whereas a macro-state is a set of nodes, i.e., at a higher level of abstraction, and is characterized by a common behavior of these nodes. For clarity, we refer to (micro-)states as nodes. Definition 1 formalizes this notion.

Definition 1 Macro-State

Let $M = (S, T)$ be a deterministic consistent Extended Finite-State-Machine (EFSM)[12]. A set of nodes $S_c = (s_\alpha, ..., s_\gamma, \gamma \geq 1) \subseteq S$ is a macro-state of color $c \in C \subset \mathbb{N} \iff$

- $\forall$ input $i \in X$, $\exists!$ (output $o_i \in Y$, node $s_i \in S$), $\forall \beta \in [\alpha...\gamma]$, $(s_\beta, i) \rightarrow (s_i, o_i)$ i.e., the execution of i from the node s_β produces the same output o_i and drives the system in the node s_i which only depends on i.
- S_C is a maximal connected component in M: i.e., either S_c has only one node, or $\forall s_\beta \in S_c, \exists s_\zeta \in S_c, s_\zeta \neq s_\beta, \exists t \in T, s_\beta \xrightarrow{t} s_\zeta$ or $s_\zeta \xrightarrow{t} s_\beta$.

2) Navigation Graph: This prefix tree contains the traces denoted as `history` in Listing 2. Figure 6 shows the evolution of nodes colors during the inference.

3) Control Flow Model: The notion of Control Flow Model (CFM) is formally defined in Definition 2. A CFM is illustrated by the nodes and continuous arrows of Figure 2.

Definition 2 Control Flow Model (CFM)

A CFM $M = (S, T, C)$, is a consistent, deterministic, colored EFSM, s.t. :

- macro-states partition S:
  - each node $n \in S$ has a color $c \in C$, i.e., the color of a node is a context variable in V ;
  - for each color $c \in C$, let S_c be the set of nodes in S having this color. Either S_c is empty, or S_c is a macro-state (Definition 1) ;
- transitions map to spiderlinks: for each transition $t \in T$,
  - the only operators used in guards are the *equality* (=) and the *logical and* ($\wedge$) ;
  - inputs are in $\{ req_0(method,action), req_1(method,action,param1_{name},param1_{value}), \dots \}$
- outputs are in $\{page\_model(num), num \in \mathbb{N}\}$

The inferred CFM are not necessarily total[15] nor completely specified for each pair of node and input. For the sake of simplicity, we minimize the representation in Figure 2. For instance, the transition $7 \rightarrow 33$ should be written as:
$req_1(method, action, param1_{name}, param1_{value}), ($
$req_1.method$ = "POST" $\wedge req_1.action$ = "/?"
$\wedge\ req_1.param1_{name}$ = "message2"
$\wedge\ req_1.param1_{value}$ = "2_e_g_a_sem"
$)/page\_model(6), c = 1$

C. Macro-State Change Detection

As shown in Listing 2, in order to precisely characterize the macro-state, we solve four sub-problems: Did the macro-state change? Which request changed the macro-state? What is the current macro-state? Which link to pick next?

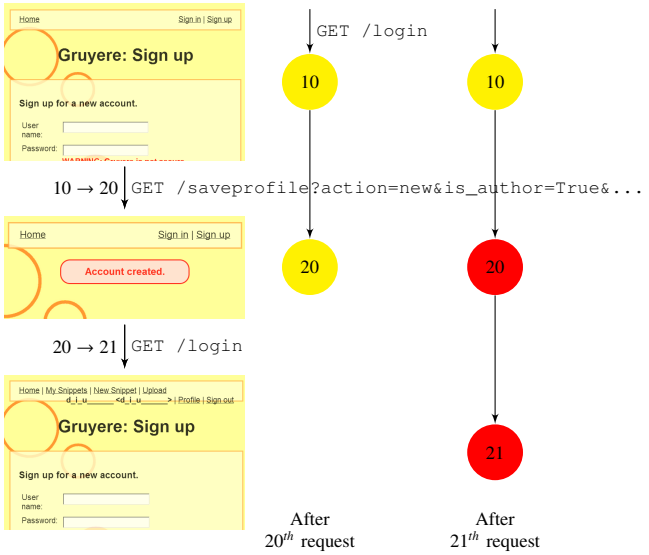

Fig. 6. Evolution of the Navigation Graph when the Macro-State Changes

1) Example: Figure 6 shows the evolution of an extract of the navigation graph when a macro-state change occurs. In this example, the request `GET /login` permits detecting the macro-state change, because the page model obtained in $\rightarrow 10$ is different from the one obtained in $20 \rightarrow 21$, and the same spiderlink `GET /login` was executed. $10 \rightarrow 20$ is selected as the cause of the state-change, because it has the highest `score` value among $[\rightarrow 10, 10 \rightarrow 20, 20 \rightarrow 21]$.

2) Control Flow Inference Algorithm: Listing 2 describes the inference of a control flow model from a web application.

```
1  while(not stopping_criterion):
2    if(curr_sequence_length>MAX_SEQUENCE_LENGTH):
3      webapp.reset()
4      curr_sequence_length = 0
5      spiderlink = start
6    output = webapp.send(spiderlink.concretize())
7    page = output.abstract()
8    if(macro_state.changed(page,history)):
9      curr_identifier += 1
10     k = index_changed_macro_state(page,history)
11     for i in range(k,len(history)):
12       history[i].identifier = curr_identifier
13     page.identifier = curr_identifier
14     model.identifiers.compute_colors()
15   else:
16     page.identifier = curr_identifier
17   history.append({output,page})
18   model.update(history)
19   spiderlink = page.pick_spiderlink()
20   curr_sequence_length += 1
21 return model
```

Listing 2. Control Flow Inference

3) Did the macro-state change: If a spiderlink i is sent twice to the application during the requests *prev* and *detect*, and the obtained page models are different (i.e., $\texttt{abstract}(resp(req(i_{prev}))) \neq \texttt{abstract}(resp(req'(i_{detect})))$), then the macro-state changed. This is the case for the spiderlinks i_{prev}=`GET /login`($\rightarrow 10$) and i_{detect}=`GET /login` ($20 \rightarrow 21$) in the navigation graph extract shown in Figure 6.

4) Which request changed the macro-state: If a macro-state change is detected between i_{prev} and i_{detect}, then the question "which request in the history between those changed the macro-state?" arises. To answer it, the heuristic function `score` represents the likelihood of a request having changed the macro-state. For a spiderlink $i \in [i_{prev}, \dots, i_{detect}]$ the higher the value of `score(i)`, the more likely i changed the macro-state. The dimensions of `score` are listed in Table I. If there is a +, resp. −, in front of the dimension, then `score` is increasing, resp. decreasing, w.r.t. this dimension. `score` is used in `index_changed_macro_state` in Listing 2.

Due to its effectiveness, we use the PQ-gram distance as a metric for similarity between page models[16].

5) What is the macro-state of the current execution context: The current execution context is the current node in the model. It results of the submission of the spiderlinks since the last reset. In order to know if the current node is one previously encountered, it is necessary to merge macro-states. For this purpose, an *identifier* is associated to each

TABLE I
DIMENSIONS OF THE `SCORE(SPIDERLINK I)` HEURISTIC

+ or - weight	dimension name
++	number of input parameters
+	distance between page models ($p_{prev} \underrightarrow{i} p_i$)
+	get_or_post
−	number of times performed (total)
−	number of times it changed the state
−	number of requests between i_{prev} and i
−−	number of potential contradictions (approx.)

TABLE II
DIMENSIONS OF THE `NAVIGATING(SPIDERLINK I)` HEURISTIC

+ or - weight	dimension name
+	number of times it changed the macro-state
+	get_or_post
−	number of times performed
−	number of artificially generated parameter values
−	(1+consecutive_contradictions)*num_state_change
−	num recently performed
− − −	request never executed

node. If the macro-state changes, then the current identifier increases, it is unchanged otherwise (see Listing 2). Doupé et al. reduced this macro-state collapsing problem to the coloring of an undirected graph of identifiers[9]. If there is an edge between two identifiers, then they will have different colors. `compute_colors()` of Listing 2 is explained in [9]. Backtracking may occur during coloring(see Section III-E3).

D. Navigation Strategy

After each output abstraction, LigRE chooses the next spiderlink to explore. It generates spiderlinks for facilitating future data flows inference, and permits to prune the CFM.

1) Choosing the Next Spiderlink to Explore: After obtaining a page model p, LigRE must decide what is the next spiderlink in $spiderlinks(p)$ to explore. The heuristic function `navigating` represents the likelihood of a spiderlink to be picked up to be executed on the application: for a given spiderlink i, the lower the value of `navigating(i)`, the more likely i will be picked up. Table II lists its dimensions. `pick_spiderlink` in Listing 2 uses it.

Either the current node contains unexplored spiderlinks and one of them is chosen according to their `navigating` score, or the shortest path in the model to nodes containing non explored spiderlinks is computed using Dijkstra's algorithm[17]. The stopping criterion evaluates to true when for each node of the CFM, the outgoing transitions have been explored a tester defined number of times. The tester can limit the number of requests and the execution time.

2) Pruning: Testers may want to prune the model for readability, or speed. This process is known to reverse engineers of binary executables[18]. LigRE permits to specify page model patterns in order not to explore matching spiderlinks.

3) Automatic Form Filling: In case forms are found in the web application, LigRE creates spiderlinks that may contain automatically generated values. The value generation aims at limiting the false positives during the later data flow inference.

We want a function *artif* that receives an input parameter *name* (e.g., *msg* in Figure 2) and produces a value s.t. the following properties hold for "most different" input parameter:

TABLE III
DIMENSIONS OF THE `CONFIDENCE(SPIDERLINK I)` HEURISTIC

weight	dimension name
−	number of nodes in the shortest path from root
−	number of unexplored spiderlinks in the page model
−	... that have same hash as one which permit determining a macro-state change

- it is easy to compute *artif(name)*
- it is infeasible to modify an input parameter name *name* without changing *artif(name)*

Those properties are two of the four of ideal cryptographic hash functions. Some web fuzzers use hash functions[19].

E. Backtracking

Backtracking consists of undoing parts of the model and recomputing them with an additional constraint. It occurs when either a *contradiction* is observed on part of the model with a low *confidence*, or the *abstract execution* leads to a different page model than the concrete execution. We here describe a special case of `model.update()` in Listing 2.

1) Confidence: This metric is applicable to a node or a transition. It expresses the level of trust in a part of the model. The higher its value, the more confident we are in the coloring and the positioning of the element. Table III contains the dimensions used in this function.

2) Potential Contradiction: This is defined in Definition 3. Let us assume that the node b is the current state. If there exists

Definition 3 Potential Contradiction

Let a and b be two nodes in the model. A contradiction between a and b is defined as follows:

$$contradiction(a,b) = \begin{cases} True & \text{if } ((confidence(a) > confidence(b)) \\ & \wedge (page\_model(a) == page\_model(b)) \\ & \wedge (color(a) \neq color(b))) \\ False & \text{otherwise} \end{cases}$$

a node a, s.t. $contradiction(a,b)$ is *True*, then we *may* have missed detecting a state change. Thus contradictions are inputs for `navigating`(see Table II) and `score`(see Table I).

3) Backtracking: We hypothesize that the web application is deterministic at the abstract level: if an input sequence of spiderlinks from the start node is executed several times, the sequences of obtained page models are the same.

Each sent spiderlink is executed on the application, and on the currently inferred CFM. It may happen that the CFM execution leads to a different page model than the application one. This is a non-determinism: either the application is not deterministic, or the current CFM is not correct. We assume it is the second case.

In such a situation, we rely on a heuristic stating that the ultimate macro-state change was not correct: we considered the identifiers α and β to map to the same macro-state, but this turned out to be wrong. Thus, we add an edge between α and β, redo the coloring and update of the model, reset the application, and start a new sequence from the initial node.

IV. Data Flow Model Annotation

The data flow model annotation corresponds to step B in Figure 1. It consumes a control flow model, to which it adds inferred data flows, thus producing a model, such as the one represented in Figure 2. In such a model, the blue/dotted text represents the source of a reflection t_{src}, and the blue/dotted arrow edges designates the reflection destination t_{dst}.

This step consists of first generating paths to navigate in the CFM, and then actively submitting those paths to the web application while inferring observable data flows.

A. Definitions

Reflection and Control + Data Flow Model (CDFM) are defined respectively in Definition 4 and Definition 5. The data-flow computation is explained in Section IV-C.

Definition 4 Reflection / Data Flow Annotation

Let $M = (S, T, C)$ be an EFSM[12]. A reflection $(x_{src}, t_{src}, t_{dst})$, $\in (D_{inp_x} \times T \times T)$ is an inferred data-flow from the value of an abstract input parameter x_{src} sent in the transition t_{src} to the concrete output of the transition t_{dst}.

Definition 5 Control and Data Flow Model (CDFM)

A control and data flow model is a CFM(Definition 2) in which reflections $refl = (x_{src}, t_{src}, t_{dst})$ are annotated in the form of a data-flow function df: $(D_{inp_x} \times T \times T) \rightarrow \{True, False\}$

$$df(refl) = \begin{cases} True & \text{if } refl \text{ has been observed} \\ False & \text{otherwise} \end{cases}$$

B. Generating Paths

Random walk and breadth-first exploration are the implemented strategies for generating input sequences from a model. They limit the length of the input sequences, and the number of times they traverse each node. If a sequence is a prefix of another one, then we only keep the latter. We analyzed XSS on fifteen applications of various complexity, and observed that the longest shortest path between t_{src} and t_{dst}, both included, is 4 transitions, and the shortest path to reach the deepest t_{dst} was 8, thus we arbitrarily limit the length of the generated sequences to 8 (prefix+suffix).

C. Computing Data Flows

For each sequence $I = (t_1, ..., t_k)$, for each concrete output o_j, $j \in [1..k]$, for each previously sent input parameter value x_{m_n}, $m \in [1..j]$, a distance between x_{m_n} and o_j is computed.

Specifically, the data flow inference consists of first searching in the output o_j for exact substrings of x_{m_n} of a minimal length, marking those found substrings, clustering them, and then computing the edit distance[20] from x_{m_n} to the clusters. If this distance is lower than an empirically determined threshold, then a data flow is annotated on the model (see Figure 2).

V. Control+Data Flow Aware Fuzzing

A. Overview

Control+Data Flow Aware Fuzzing encompasses steps C and D in Figure 1: first prioritizing the considered data flows (Section V-B), producing slices(Section V-C), and then using those slices to guide a fuzzer. Its pseudo-code is in Listing 3. `get_reflections` returns the observed reflections by decreasing priority. It uses the `prioritization(reflection)` heuristic function whose dimensions are described in Table IV. The higher its value, the more likely this reflection will be tested first. For each reflection, LigRE positions the application in the node from which t_{src} originates by sending the `prefix` sequence. Then LigRE feeds the fuzzer an authentication context and a suffix ($CH(t_{src}, t_{dst})$) for the fuzzer to navigate from t_{src} to t_{dst}.

```
1  def control_data_aware_fuzzing(webapp, fuzzer):
2    vulns = []
3    for refl in model.get_reflections():
4      webapp.reset()
5      prefix=shortest_path(from=root,to=refl.src)
6      webapp.execute(prefix)
7      fuzzer.config.auth = webapp.context
8      suffix = shortest_path(refl.src,refl.dst)
9      fuzzer.config.urls = suffix
10     vulns += fuzzer.do()
11   return vulns
```

Listing 3. Control+Data Flow-aware Fuzzing

B. Reflection Prioritization

Table V is an extract of the prioritization table in its initial state. dim_k corresponds to the dimension k of Table IV. Each line of Table V designates a reflection. Initially, *chosen*, the set of yet chosen reflections, is empty. The first reflection to be chosen is a, since it has the highest prioritization value. Dimensions are updated: $dim_{4,5}(a)$+=1. Then b is chosen, similarly: $dim_{4,5}(b)$+=1. Later on, either c or d will be chosen. Let us assume c is chosen first. Then $dim_{4,5}(c)+ = 1$ and $dim_4(d)+ = 1$ are updated, since c and d have the same x_{src}.

TABLE IV
Dimensions of `prioritization(reflection,chosen)`

#	weight	dimension name
1	–	number of reflections having the same parameter name x_{src}
2	–	number of reflections having the same (t_{src}, t_{dst})
3	–	number of macro-states from t_{src} to t_{dst}
4	–	number of *yet chosen* reflections having the same x_{src}
5	–	number of *yet chosen* reflections having the same (t_{src}, t_{dst})

TABLE V
Prioritization of Reflections (extract, initial state)

id	t_{src}	x_{src}	t_{dst}	dim_1	dim_2	dim_3	dim_4	dim_5
a	$7 \rightarrow 33$	message2	$7 \rightarrow 33$	1	1	0	0	0
b	$7 \rightarrow 17$	msg	$18 \rightarrow 21$	1	1	1	0	0
c	$33 \rightarrow 9$	action	$33 \rightarrow 9$	5	1	0	0	0
d	$18 \rightarrow 21$	action	$21 \rightarrow 9$	5	1	0	0	0

C. Chopping

Slicing permits to limit the state space exploration. This technique focuses on parts of the applications w.r.t. a *slicing criterion*. The notion of slicing has been extended to model-based languages. Various techniques are proposed in the literature[21]. In LigRE, we are interested in finding paths between a source t_{src} and a destination t_{dst} on the model. Thus we use a compressed form of slicing called *chopping*[22], which captures this relation, as described in Definition 6.

Definition 6 Chopping on the Model

Let $M = (S, T, C, refl)$ be a CDFM (Definition 5). Given $(t_{src}, t_{dst}) \in T^2$, a source and a destination transitions, a chopping $CH(t_{src}, t_{dst})$ consists of all the nodes in S on which t_{dst} (transitively) is control dependent. Such a set is also called a path p from t_{src} to t_{dst}. We consider one of the shortest paths according to Dijkstra's algorithm.
$CH(t_{src}, t_{dst}) = \{n \in S \,|\, p \in t_{src} \rightarrow^* t_{dst} \wedge p = \langle n_1, \cdots, n_k \rangle \wedge \exists i : n = n_i\}$; where $t_{src} \rightarrow^* t_{dst}$ is a (transitive) relation i.e. $s \rightarrow d$ if exists a transition from s to d and $\rightarrow^*$ is its transitive closure.

VI. Implementation

The approach is implemented as a tool "*LigRE*" containing 8000 SLOC of Python3.2. Figure 7 represents its architecture. KameleonFuzz[8] extends LigRE by improving the fuzzer.

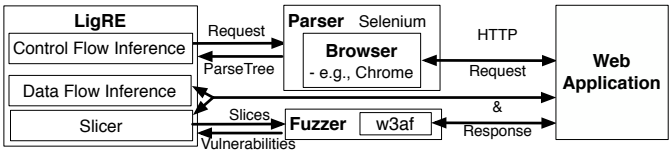

Fig. 7. Architecture of LigRE

During the control flow inference, the parse tree (approximated by a subset of the Document Object Model) is obtained using the selenium library[23] which instruments the Google Chrome browser to parse HTTP replies. During the data flow inference, requests are performed directly to the web application. During the fuzzing, LigRE drives the application in the source via the prefix slice ; it then parameterizes the suffix slice for a fuzzer (w3af[11]).

VII. Empirical Evaluation

We aim at determining if control plus data flow model aware XSS fuzzing is efficient enough to search for vulnerabilities in typical web applications. We also aim at comparing the fault detection capability of our prototype implementation LigRE against existing state of the art black-box vulnerability scanners. Relevant metrics include the number of distinct true XSS discovered, and the number only found by a given scanner. To measure the efficiency of the scanners, we compare the number of sent requests and of found XSS. In our experiments, LigRE detected XSS missed by other scanners, and most of the XSS found by those.

A. Test Subjects

1) Web Applications: We selected seven applications of various complexity, as described in Table VI. Our interest in them is expressed in Appendix A.

TABLE VI
Tested Web Applications

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	

2) BlackBox XSS Scanners: We consider the following open-source black-box XSS scanners to compare with our approach: Wapiti, w3af and SkipFish. They all infer the control flow and fuzz. We list their configuration in [24]. In addition to LigRE with all its components (A,B,C and D in Figure 1), we also include LigRE with only A(control flow inference) and D(w3af). We denote them as $LigRE_{ABC+D}$ and $LigRE_{A+D}$.

B. Research Questions

RQ1. (Fault Revealing): *Does control plus data flow aware fuzzing find more true vulnerabilities than other scanners?*

For each scanner and application, we sequentially configure the scanner, reset the application, set a random seed to the scanner, run it against the application, and retrieve the results. We repeat this process five times, using different seeds. If possible, scanners are configured s.t. they only target XSS. We configure them with the same information (e.g., credentials). When a scanner does not handle it, we perform two sub-runs: one with the cookie of a logged-on user and one without.

We adjust parameters for the runs to last at most five hours. Beyond this duration, we stop the scanner and manually analyze the results. The number of detected XSS is the union of distinct true XSS found during the different runs. An XSS is uniquely characterized by its source transition t_{src}, its fuzzed parameter name x_{src}, its destination output t_{dst} and the structure in which the value of x_{src} is reflected. For all scanners, we manually verify XSS. During our experiments, no scanner did report any false positive XSS (Skipfish reported other false positives).

TABLE VII
$LigRE_{ABC+D}$ detection capabilities on the tested applications

Application	Inferred Data-Flows	True XSS Detected	Nodes	Transitions
P0wnMe	28	2	13	51
WebGoat	134	4	20	80
Gruyere	23	3	30	130
WordPress	52	2	15	129
Elgg	59	1	49	214
PhpBB	213	4	63	279
e-Health	12	5	15	33

Table VII contains the numbers of annotated reflections, found XSS, inferred nodes and transitions. This illustrates

the practicality of LigRE to infer the control and data flow models of the evaluated applications. The number of nodes and transitions, may correspond to a partial application coverage.

Figure 8 represents the number of detected true XSS vulnerabilities for the considered scanners and applications. $LigRE_{ABC+D}$ detected the highest number of vulnerabilities for every application, and several vulnerabilities not detected by other scanners. These results confirm Doupé et al.'s experiments: improving the control flow inference($LigRE_{A+D}$) increases the vulnerability detection capabilities, as compared to non macro-state-aware scanners (Wapiti, w3af(D), Skipfish). Moreover, comparing $LigRE_{A+D}$ and $LigRE_{ABC+D}$ shows that data flow inference(B) and slices for flow aware fuzzing(C) also increase XSS detection capabilities. We notice that $LigRE_{ABC+D}$ founds vulnerabilities missed by other scanners, including $LigRE_{A+D}$: see the non-dotted part of Figure 8.

Most scanners achieve limited coverage due to their partial handling of basic forms, their inability to track the macro-state (beyond the classic logged in/out, and assuming the tester provides values). At times, they send requests regardless of the available links. The aggressive behavior of Skipfish is sometimes positive (e.g., in Gruyere, it found one XSS missed by others on 404 pages), sometimes not (e.g., in Wordpress, it submitted 150 times a form without detecting any XSS). For Elgg, both Skipfish and w3af loop between pages because they only consider the URL and not the page model.

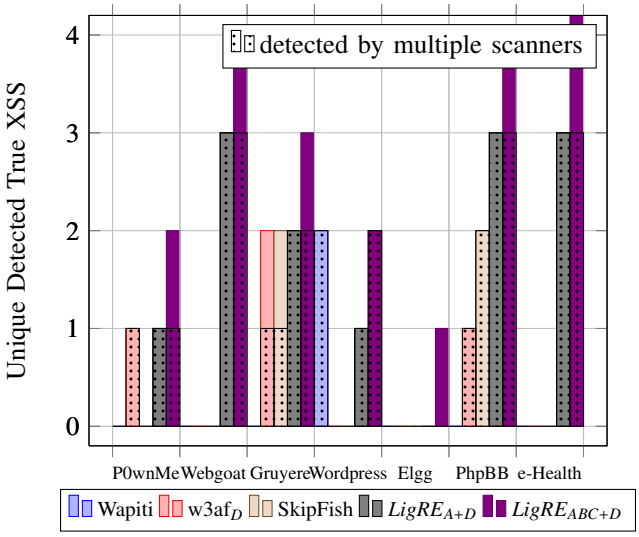

Fig. 8. XSS Detection Capabilities of Black-Box Scanners

On considered applications, a control plus data flow directed fuzzing increases XSS revealing capabilities.

RQ2. (Efficiency): *How efficient are the scanners in terms of vulnerability detection capabilities per number of tests?*

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 and 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 9.

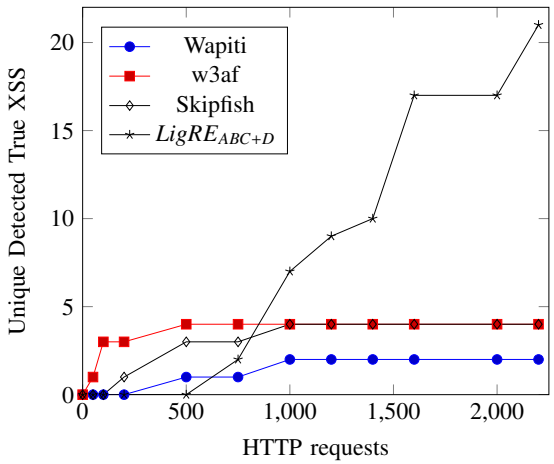

Fig. 9. XSS Detection Efficiency of Black-Box Scanners

Below approximatively 850 HTTP requests, w3af is the most efficient scanner. Thus we hypothesize that in applications with few macro-states, assuming it is able to navigate correctly, which in our experiments mainly happened in P0wnMe and Gruyere, then w3af is more efficient than other scanners at finding non filtered XSS.

Moreover, the LigRE proof-of-concept spends a significative number of requests in data flow inference (from 75 to 93%). There is room for improvement. An industrial implementation should consider additional heuristics to prune sequences: e.g., with a notion of achieved coverage of n long sub-sequences.

Data flow inference is the main barrier to entry of LigRE. If acceptable, LigRE had the highest detection capabilities. Otherwise, traditional scanners are of interest.

RQ3. (Current Use by Testers): *What is the current use of control plus data flow models (CDFM) by testers?*

We conducted two surveys for evaluating the current level of use of CDFM by penetration testers[25], and how they obtained them. Figure 10 synthesize relevant knowledge.

Obtaining and using CDFM: Currently used open-source web scanners do not output CFDM. W3af[11] outputs CFM. It achieves a low coverage[9]. In our sample, no tester uses w3af to obtain a CFM. Those who make use of CFM rely on a manual crawling approach, using Burp[26] as a proxy, and manually draw CFM. However, since considered web fuzzers only accept a list of urls and an authentication context, they would achieve a low transition coverage.

Data Flow Tracking: 77% of testers do not perform white-box data flow tracking, mainly because they think not enough tools are available. 50% of those find this manual work tedious. Those who perform it rely on dynamic exact

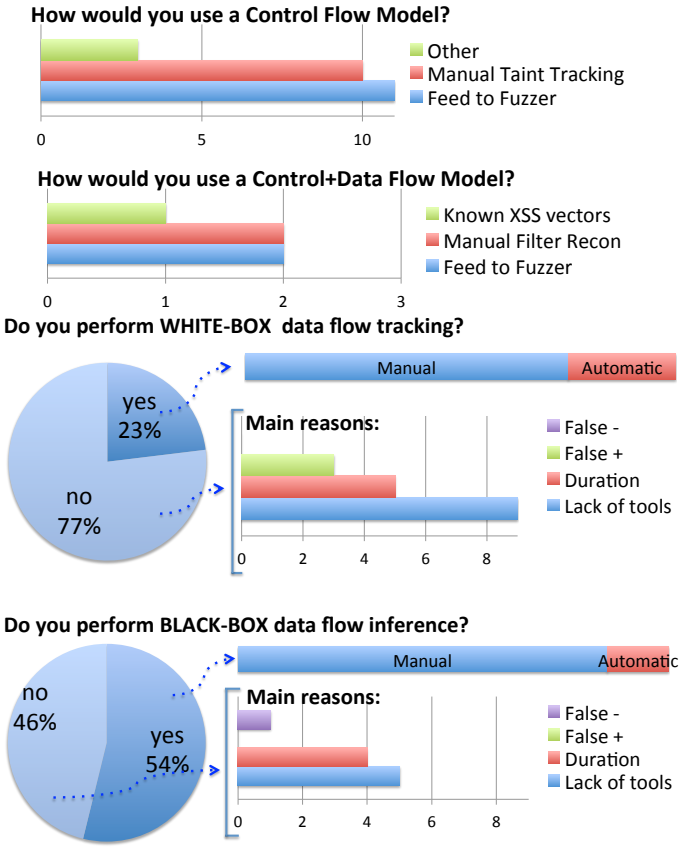

Fig. 10. Main Results of Poll for Web Applications Security Testers

string matching. 54% of testers perform black-box data flow inference. Most of them do it manually. 57% of those find it tedious. Data flow tracking aims at determining the exact composing of filters, in order to produce fuzzed inputs to bypass those filters. Performing it manually is time consuming, and limited to human expertise. In white-box, it may require knowledge of various server languages. Whereas in black-box, the ability to interpret few client side languages is enough.

Even skilled penetration testers largely rely on manual data flow tracking. Since such work is time consuming, and prone to false negatives, there is a need for tools producing hybrid control plus data flow models.

VIII. Approach Limitations

Reset: We assume the ability to reset the application in its initial node, which may not always be practical (e.g., when testing a live application having users connected, we have to work on a copy). However, this does not break the black-box harness assumption: we do not require to be aware of how the macro-state is stored (e.g., database). How to relax this assumption is a research direction.

Non-Deterministic Values (NDV): We assume the tester's ability to identify the NDV (e.g., anti-CSRF, viewstate, ...), or constantly changing pages[27]. Approaches to automate their detection have been investigated[14].

Dynamic Client-Side Content: Our implementation does not support Ajax applications, unless they gracefully downgrade (i.e., keep their functionality while navigating via HTTP requests instead of Ajax events). [28, 29] automatically infer Ajax applications. Flash and PDF files are not yet supported.

IX. Related Work

A. Control Flow Inference

Based on Angluin's L*, Shahbaz and Groz designed an algorithm for iteratively inferring the control flow of an I/O system. Cho et al. infer a botnet protocol[31] by adding a prediction heuristic to [30]. Hossen et al. automatically generate test drivers for non-Ajax web applications[32].

Doupé et al. showed that improving control flow inference increases vulnerability detection[9]. LigRE shares similarities with their macro-state-aware-crawler. Differences lay in the heuristics, the introduction of confidence, contradictions, backtracking, and data flow inference. Doupé et al. run experiments on a local cloud, whereas we run ours on a laptop.

Dessiatnikoff et al. cluster pages according a specially crafted distance for SQL injections[33]. Marchetto et al. dynamically infer the control flow of Ajax web applications[34]. They wrote abstraction functions for common Ajax primitives. Tonella et al. use genetic algorithm for finding the right balance between over and under-approximations of CFM[35].

LigRE does not make use of L* and is driven by heuristics. It clusters pages according to the notion of macro-state. The current implementation supports non-Ajax applications or Ajax applications which downgrade gracefully.

B. Data Flow Inference

W3af[11] and XSSAuditor[36](Chrome XSS filter) assume the fuzzed input value to be reflected without modification, and thus rely on exact string matching. This may lead to false negatives when input values are transformed[37, 8]. Skipfish generates three variants for a spiderlink, and assumes there is a data flow if the response varies[38, 33, 9]. This may lead to false positives, if the scanner is not aware of a macro-state change. Sun et al.[39] compute a string edit distance[20]. Sekar[40] proposed a filtering algorithm inspired from bioinformatics for improving the efficiency of Levenshtein's distance. LigRE relies on a filter-tolerant substring matching of a minimal length, and computes the edit distance on a smaller output. LigRE relies on the fuzzer test verdict.

C. Control plus Data Flow Models

Caselden et al. use similar models, named HI-CFG on basic blocs, to automatically generate exploits for memory corruption vulnerabilities in binary programs with a grey-box harness[41]. Netzob infers protocols implementations using L*, and enhance it with data flows w.r.t. equivalence, size, or repetition relations. Its test driver, abstraction, and concretization functions are written by an analyst[42]. With PRISMA, Krueger et al. infer control and data flow markov models of botnet protocols from traffic captures[43]. LigRE targets XSS, a command injection vulnerability, in web applications with a black-box harness, and produces CDFM to drive a fuzzer.

X. Conclusion

LigRE automatically reverse-engineers web applications as a control and data flow model. It prioritizes model slices to guide the scope of the fuzzing. Heuristics drive LigRE. Empirical experiments show that LigRE detects more XSS than open source and control flow aware scanners.

In addition of being an input for human penetration testers, the obtained models can be the first step for automated vulnerability detection: e.g., if provided to a model checker or a fuzzer. For instance, our evolutionary smart fuzzer KameleonFuzz[44, 8] can use such models, and improves the fuzzing step of LigRE to detect more complex filtered XSS.

Acknowledgments

This work was supported by the projects ITEA2 №09018 DIAMONDS "Development and Industrial Application of Multi-Domain Security Testing Technologies" and FP7-ICT №257876 SPaCIoS "Secure Provision and Consumption in the Internet of Services". We thank the anonymous reviewers for their helpful feedback on an earlier revision of this paper.

References

[1] OWASP, "Top ten project," 2013.
[2] W. Luo, J. Liu, J. Liu, and C. Fan, "An analysis of security in social networks," in *8th DASC*. IEEE, 2009, pp. 648–651.
[3] R. Kugler, "Paypal.com XSS vulnerability," 2013, http://seclists.org/fulldisclosure/2013/May/163.
[4] Nirgoldshlager, "Stored XSS in facebook," 2013, http://www.breaksec.com/?p=6129.
[5] ZentrixPlus, "ebay-security-researchers-hall-of-fame-hof," 2013, http://zentrixplus.net/blog/ebay-security-researchers-hall-of-fame-hof/.
[6] J. Bau, F. Wang, E. Bursztein, P. Mutchler, and J. C. Mitchell, "Vulnerability factors in new web applications: Audit tools, developer selection & languages," Stanford, Tech. Rep., 2012.
[7] J. Bau, E. Bursztein, D. Gupta, and J. C. Mitchell, "State of the art: Automated blackbox web application vulnerability testing," in *IEEE S&P*, 2010, pp. 332–345.
[8] F. Duchène, S. Rawat, J.-L. Richier, and R. Groz, "Fuzzing Intelligent de XSS Type-2 Filtrés selon Darwin: KameleonFuzz," in *11th SSTIC*, 2013, pp. 289–311.
[9] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," *USENIX Sec*, 2012.
[10] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *POPL*, 2006, pp. 372–382.
[11] A. Riancho, "w3af - web application attack and audit framework," http://w3af.sourceforge.net.
[12] A. Petrenko, S. Boroday, and R. Groz, "Confirming Configurations in EFSM Testing," *IEEE TSE*, vol. 30, pp. 29–42, 2004.
[13] Wikipedia, "Cryptographic nonce."
[14] K. Hossen, R. Groz, and J.-L. Richier, "Security Vulnerabilities Detection Using Model Inference for Applications and Security Protocols," in *SECTEST with ICST*. IEEE, 2011, pp. 534–536.
[15] I. Lee, "Code generation from extended finite state machines," 2010.
[16] N. Augsten, M. Böhlen, and J. Gamper, "Approximate matching of hierarchical data using pq-grams," in *31st VLDB*, 2005, pp. 301–312.
[17] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
[18] I. Guilfanov, "Decompilers and beyond," in *HITB*, 2008.
[19] epsylon, "XSSer," 2012, http://xsser.sourceforge.net/.
[20] V. Levenshtein, "Binary coors capable of correcting deletions, insertions, and reversals," in *Soviet Physics-Doklady*, vol. 10, 1966.
[21] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, "State-based model slicing: A survey," *ACM Computing Surveys*, 2013.
[22] D. Jackson and E. J. Rollins, "A new model of program dependences for reverse engineering," in *2nd SIGSOFT FSE*. ACM, 1994, pp. 2–10.
[23] J. Huggins, P. Hammant *et al.*, "Selenium, browser automation framework," http://code.google.com/p/selenium/.
[24] "Ligre: scanners configuration," http://car-online.fr/ligre_scan_conf.
[25] F. Duchène, S. Rawat, J.-L. Richier, and R. Groz, "A hesitation step into the black-box: Heuristic based web application reverse engineering," in *NoSuchCon*, 2013.
[26] "Burp suite," http://portswigger.net/burp/.
[27] SPaCIoS, "Deliverable 5.3: Final Proof of Concept," 2013.
[28] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *ICST*. IEEE, 2008, pp. 121–130.
[29] D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *15th WCRE*. IEEE, 2008, pp. 69–73.
[30] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *World Congress on Formal Methods*, 2009, pp. 207–222.
[31] C. Y. Cho, D. Babick, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *ACM CCS*, 2010, pp. 426–439.
[32] K. Hossen, J.-L. Richier, C. Oriat, and R. Groz, "Automatic generation of test drivers for model inference of web applications," in *SECTEST with ICST*. IEEE, 2013.
[33] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaaniche, and V. Nicomette, "A clustering approach for web vulnerabilities detection," in *17th PRDC*. IEEE, 2011, pp. 194–203.
[34] A. Marchetto, P. Tonella, and F. Ricca, "Reajax: a reverse engineering tool for ajax web applications," *Software*, pp. 33–49, 2012.
[35] P. Tonella, A. Marchetto, C. D. Nguyen, Y. Jia, K. Lakhotia, and M. Harman, "Finding the Optimal Balance between Over and Under Approximation of Models Inferred from Execution Logs," in *ICST*, 2012.
[36] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters," in *WWW*, 2010, pp. 91–100.
[37] M. Heiderich, E. Nava, G. Heyes, and D. Lindsay, *Web Application Obfuscation:'-/WAFs.. Evasion.. Filters//alert (/Obfuscation/)-'*. Syngress, 2010.
[38] M. Zalewski and N. Heinen, "Skipfish - web vulnerability scanner."
[39] F. Sun, L. Xu, and Z. Su, "Client-Side Detection of XSS Worms by Monitoring Payload Propagation," *ESORICS*, pp. 539–554, 2009.
[40] R. Sekar, "An Efficient Blackbox Technique for Defeating Web Application Attacks," in *NDSS*, 2009.
[41] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, "Transformation-aware Exploit Generation using a HI-CFG," UC Berkeley, 2013.
[42] G. Bossert and F. Guihéry, "Security evaluation of communication protocols in common criteria using netzob," in *ICCC*, 2013.
[43] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *AISEC*. ACM, 2012.
[44] F. Duchène, R. Groz, S. Rawat, and J.-L. Richier, "XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing," in *SECTEST with ICST*, 2012, pp. 815–817.

Appendix

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

PhpBB v2 is a forum platform. We include this version, as it is famous to contain several XSS[7].

e-Health 04/16/2013 is an extract of a medical platform used by patients and practitioners, developed by a company.