Skyfire: Data-Driven Seed Generation for Fuzzing

Junjie Wang, Bihuan Chen†, Lei Wei, and Yang Liu
Nanyang Technological University, Singapore
{wang1043, bhchen, l.wei, yangliu}@ntu.edu.sg
†Corresponding Author

***Abstract*—Programs that take highly-structured files as inputs normally process inputs in stages: syntax parsing, semantic checking, and application execution. Deep bugs are often hidden in the application execution stage, and it is non-trivial to automatically generate test inputs to trigger them. Mutation-based fuzzing generates test inputs by modifying well-formed seed inputs randomly or heuristically. Most inputs are rejected at the early syntax parsing stage. Differently, generation-based fuzzing generates inputs from a specification (e.g., grammar). They can quickly carry the fuzzing beyond the syntax parsing stage. However, most inputs fail to pass the semantic checking (e.g., violating semantic rules), which restricts their capability of discovering deep bugs.**

In this paper, we propose a novel data-driven seed generation approach, named Skyfire, which leverages the knowledge in the vast amount of existing samples to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. Skyfire takes as inputs a corpus and a grammar, and consists of two steps. The first step of Skyfire learns a probabilistic context-sensitive grammar (PCSG) to specify both syntax features and semantic rules, and then the second step leverages the learned PCSG to generate seed inputs.

We fed the collected samples and the inputs generated by Skyfire as seeds of AFL to fuzz several open-source XSLT and XML engines (i.e., Sablotron, libxslt, and libxml2). The results have demonstrated that Skyfire can generate well-distributed inputs and thus significantly improve the code coverage (i.e., 20% for line coverage and 15% for function coverage on average) and the bug-finding capability of fuzzers. We also used the inputs generated by Skyfire to fuzz the closed-source JavaScript and rendering engine of Internet Explorer 11. Altogether, we discovered 19 new memory corruption bugs (among which there are 16 new vulnerabilities) and 32 denial-of-service bugs.

I. Introduction

Fuzzing is an automatic random testing technique, which was first introduced in early 1990s to analyze the reliability of UNIX utilities [1]. Since then, it has become one of the most effective and scalable testing techniques to find vulnerabilities or crashes in commercial off-the-shelf (COTS) software, and hence has been widely used by mainstream software companies such as Microsoft [2], Google [3], and Adobe [4] to ensure the quality of their software products.

Fuzzing feeds a large amount of test inputs to the target program in the hope of triggering unintended program behaviors and finding bugs. The quality of the test inputs is one of the most important factors that influence the effectiveness and efficiency of fuzzers [5, 6]. Inputs are usually constructed in *mutation-based* and/or *generation-based* methods. Mutation-based method generates inputs by modifying well-formed seed inputs (e.g., bit flipping and token insertion). Such modifications can be totally random [1], or guided by code coverage [7], taint analysis [8, 9] that identifies those interesting bytes to mutate, symbolic execution [10, 11, 12] that relies on constraint solving to systematically explore execution paths, or a combination of taint analysis and symbolic execution [13, 14].

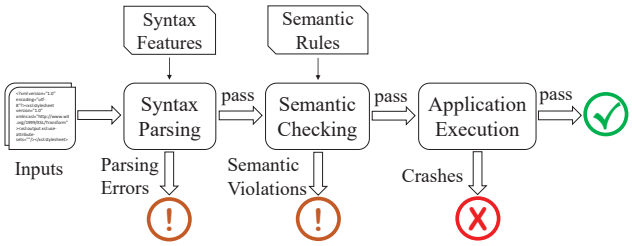


Fig. 1: Stages of Processing Highly-Structured Inputs

These guided mutation-based methods can effectively fuzz programs that process compact and unstructured data formats (e.g., images and videos). However, they are less suitable for programs that process highly-structured inputs (e.g., XSL and JavaScript). Such programs often process the inputs in stages, as shown in Fig. 1, i.e., *syntax parsing*, *semantic checking*, and *application execution*. Most malformed inputs generated by mutation-based fuzzing fail to pass the syntax parsing and thus rejected at an early stage of processing, which makes the fuzzers spend a large amount of time struggling with syntax correctness and heavily limits them to find deep bugs.

On the other hand, generation-based fuzzing constructs inputs from a specification, e.g., input models [15, 16, 17] that specify the format of data chunks and integrity constraints, and context-free grammars [18, 19, 20, 21] that describe the syntax features. Naturally, these model-based or grammar-based fuzzing can generate inputs that easily pass integrity checking (e.g., checksum) or grammatical checking, quickly carrying the fuzzing exploration beyond the syntax parsing stage. In that sense, these approaches can significantly narrow down the search space of fuzzing than those mutation-based fuzzing approaches.

However, for simple application of generic generational fuzzing, majority of the test cases are usually unable to pass the semantic checking. Meanwhile, it is often prohibitively expensive to systematically generate semantic valid test cases. This is a well-known problem of implementing a smart fuzzer. For example, an XSLT engine often checks whether an *attribute* can be applied on an *element*. If this semantic rule is violated, an "unexpected attribute name" message will be prompted, and the program will abort further execution. Such semantic rules are in fact implemented on a element-

DOI 10.1109/SP.2017.23

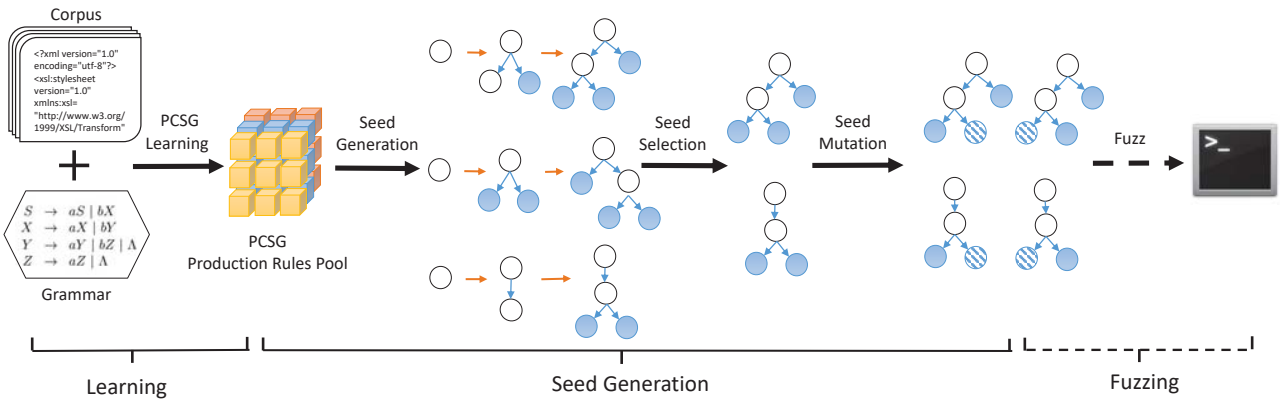


Fig. 2: The Overview of Our Approach

specific basis. As a result, only a small portion of the inputs generated from generic generation-based fuzzing can reach the application execution stage, where the deep bugs normally hide; and a large part of the application code is unreached.

To further generate semantically-valid inputs, some grammar-based fuzzing approaches [22, 23, 24] have been proposed to use hard-coded or manually-specified generation rules to express semantic rules the inputs generated should satisfy. However, different programs often implement different sets of semantic rules (e.g., for XSLT 1.0 and 2.0); and it is daunting and labor-intensive, or even impossible to manually express all the required semantic rules.

In this paper, we propose a novel data-driven seed generation approach, named Skyfire[1]. It leverages the vast amount of samples (i.e., corpus) to automatically extract the knowledge of grammar and semantic rules, and utilizes such knowledge to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. In that sense, Skyfire is orthogonal to mutation-based fuzzing approaches, providing high-quality seed inputs for them and improving their efficiency and effectiveness for programs that process highly-structured inputs. Besides, Skyfire advances the existing generation-based fuzzing approaches, i.e., carrying the fuzzing exploration beyond the semantic checking stage to reach the application execution stage to find deep bugs without any manual specification tasks.

Basically, Skyfire takes as inputs a *corpus* and a *grammar*, and generates seed inputs in two steps, as shown in Fig. 2. The first step parses the collected samples (i.e., the corpus) based on the grammar into abstract syntax trees (ASTs), and learns a probabilistic context-sensitive grammar (PCSG), which specifies both syntax features and semantic rules. Different from a context-free grammar that is widely used in generation-based fuzzing, each production rule in PCSG is associated with the *context* where the production rule can be applied as well as the *probability* that the production rule is applied under the context.

In the second step, Skyfire first generates seed inputs by iteratively selecting and applying a production rule, satisfying the context, on a non-terminal symbol until there is no non-terminal symbol in the resulting string. During this process, we prefer low-probability production rules to high-probability production rules for producing uncommon inputs with diverse grammar structures. Skyfire then conducts seed selection to filter out the seeds that have the same code coverage to reduce redundancy. Finally, Skyfire mutates the remaining seed inputs by randomly replacing a leaf-level node in the AST with the same type of nodes based on the production rules, which introduces semantic changes in a diverse way while maintaining grammar structures.

To evaluate the effectiveness and generality of our approach, we collected the samples of XSL, XML, and JavaScript. Then, we fed the collected samples and the inputs generated by Skyfire as seeds to the AFL [7] to fuzz several open-source XSLT and XML engines (i.e., Sablotron, libxslt, and libxml2). We also used the inputs generated by Skyfire to fuzz the closed-source JavaScript and rendering engine in Internet Explorer 11. Evaluation results have indicated that, i) Skyfire can generate well-distributed inputs, and hence effectively improve the code coverage of fuzzers (i.e., 20% for line coverage and 15% for function coverage on average); and ii) Skyfire can significantly improve the capability of fuzzers to find bugs. We found 19 new memory corruption bugs (among which we discovered 16 new vulnerabilities), and 32 new denial of service bugs (including stack exhaustion, NULL pointer dereference, and assertion failure).

This work makes the following main contributions.

- We propose to learn a probabilistic context-sensitive grammar (PCSG) from existing samples to describe syntax features and semantic rules for highly-structured inputs.
- We propose to leverage PCSG to generate seed inputs with diverse grammar structures, apply seed selection to reduce seed redundancy, and perform seed mutation to introduce semantic changes diversely, which aims to generate correct, diverse, and uncommon seeds for fuzzing programs that process highly-structured inputs.
- We evaluated Skyfire by using the inputs generated to fuzz several XSLT, XML, and JavaScript/rendering engines, and greatly improved the code coverage and discovered 16 new vulnerabilities.

The rest of this paper is structured as follows. Section II gives

[1]An Autobot scientist in the film *Transformers*.

N = { *document, prolog, content, element, reference, attribute, chardata, misc, entityRef, name, ...*}
Σ = { *comment, cdata, charRef, string, text, sea_ws, pi, ...*}
R = { *document* → *prolog? misc* element misc*,*
prolog → `'<?xml'` *attribute** `'?>'`,
element → `'<'` *name attribute** `'/>'` | `'<'` *name attribute** `'>'` *content* `'</'` *name* `'>'`,
attribute → *name = string,*
content → *chardata?* ((*element* | *reference* | *cdata* | *pi* | *comment*) *chardata?*)*,
reference → *entityRef* | *charRef,*
chardata → *text* | *sea_ws,*
misc → *comment* | *pi* | *sea_ws,*
... }
s = *document*

Fig. 3: Part of the Context-Free Grammar of XSL

the overview of our approach. Section III and IV respectively elaborate the details of PCSG learning and seed generation. Section V presents the implementation details and evaluates our approach. Section VI discusses the related work before Section VII draws the conclusions.

II. Approach Overview

In this section, we first clarify the target programs of our seed generation approach with a running example of XSL, and then present the overview of our approach.

A. Target Programs

In this work, we generate seeds for fuzzing programs that process highly-structured inputs such as XSL, XML, JavaScript, and HTML. Such programs usually exist in operating systems, system libraries, and web browsers, where certain inputs use a grammar to describe their syntax features. In particular, context-free grammars are widely used for this purpose.

Definition 1. A context-free grammar (CFG) is a 4-tuple $G_{cf} = (N, \Sigma, R, s)$, where

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,
- R is a finite set of production rules of the form $\alpha \rightarrow \beta_1\beta_2...\beta_n$, where $\alpha \in N, n \geq 1$, and $\beta_i \in (N \cup \Sigma)$ for $i = 1...n$,
- and $s \in N$ is a distinguished start symbol.

Example 1. Fig. 3 reports part of the context-free grammar of XSL. The start symbol is a $document$, which can be composed of a $prolog$ followed by an $element$. A $prolog$ contains zero or more $attributes$, which are pairs of attribute name and value (e.g., $version$ = "1.0"). An $element$ has an element $name$, zero or more $attributes$, and a $content$. The $content$ could be empty, $text$, $element$, $reference$, etc. Here we only introduce the basic grammars of XSL that will be used in the rest of the paper. A complete CFG of XSL can be found at [25].

On the other hand, highly-structured inputs need to satisfy semantic rules. Such rules check the semantic validity of those syntactically valid inputs, and prevent unexpected inputs from getting executed.

Example 2. An XSLT engine often checks whether an $attribute$ can be applied on an $element$. If this semantic rule is violated, an "unexpected attribute name" message will be prompted, and the program will abort further execution.

B. Overview of Skyfire

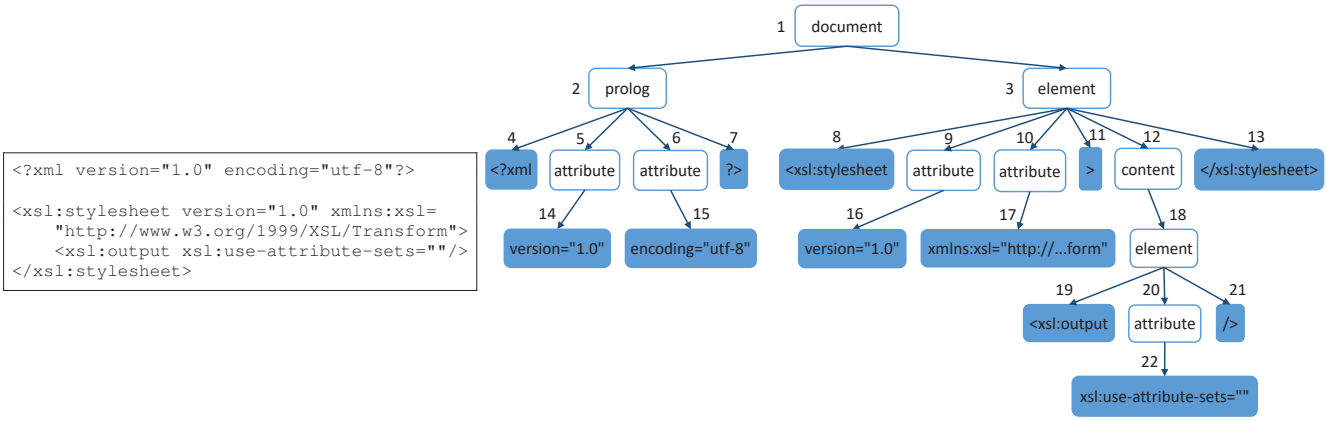
To ensure the breadth and depth of the fuzzing exploration on programs that process highly-structured inputs, our seed generation approach has three main goals. The first goal is to generate *correct* seeds, which guarantees that most inputs generated are syntactically and semantically valid. Such inputs can prevent the fuzzing exploration from being stuck at the syntax parsing and semantic checking stages and help to quickly make progress towards the application execution stage.

The second goal is to generate *diverse* seeds, which ensures the diversity of inputs with respect to grammars and semantic rules. Such diversity can help to quickly cover different function modules, which are often difficult to reach by traditional mutation-based fuzzing approaches.

The third goal is to generate *uncommon* seeds, which ensures that inputs are uncommon enough to reach less-fuzzed program code and trigger unintended program behaviors. Such inputs often have a higher possibility to reveal new bugs than common ones, and can be served as good seeds for fuzzers to mutate and explore unexplored program behaviors.

Following these three main goals, we propose a data-driven seed generation approach, named Skyfire. Fig. 2 presents the overview of our approach, which takes as inputs a *corpus* and a *grammar*, and generates seed inputs in two steps: *learning* and *generation*. The output of Skyfire can be fed as seeds to a fuzzer (e.g., AFL [7]) to start the *fuzzing* exploration. The corpus contains a large amount of samples crawled from the Internet. The grammar refers to a context-free grammar of samples, which is often publicly available (e.g., grammars of different languages can be found in the ANTLR community [26]).

The first step of Skyfire parses the samples according to the given grammar into abstract syntax trees (ASTs), and learns a *probabilistic context-sensitive grammar* (PCSG), which has a pool of production rules. We propose PCSG to specify both syntax features and semantic rules, while a CFG only describes syntax features. In particular, each production rule in a PCSG is associated with a *context*, where the production rule can be



(a) A Sample XSL File

(b) The Abstract Syntax Tree of the Sample XSL File

Fig. 4: A Running Example: A Sample XSL File (4a) and its Corresponding Abstract Syntax Tree (4b)

applied, to express semantic rules. Each production rule is also associated with the *probability* that the production rule is applied under the given context.

Based on the pool of extracted production rules, in the second step, Skyfire first generates seed inputs in an iterative process, where a production rule, whose context is satisfied, is applied on a non-terminal symbol until there is no non-terminal symbol in the resulting string. In this process, we prefer low-probability production rules to high-probability ones to produce uncommon inputs with diverse grammar structures. Then Skyfire performs seed selection to filter out the seeds that have the same code coverage to reduce seed redundancy. Finally, Skyfire mutates the remaining seed inputs by randomly replacing a leaf-level node in the AST with the same type of nodes according to the production rules, which aims at introducing semantic changes in a diverse way while maintaining the grammar structures.

In the following two sections, we will first elaborate how to learn a PCSG to describe both syntax features and semantic rules (Section III), and then we will explain how to generate seed inputs based on the learned PCSG (Section IV).

III. PCSG Learning

In this section, we first introduce the probabilistic context-sensitive grammar (PCSG), and then elaborate how to learn a PCSG.

A. Probabilistic Context-Sensitive Grammar

Programs that process highly-structured inputs usually implement a set of semantic rules to check the semantic validity of the inputs. Normally, due to the complexity of various semantic rules, only a subset of the complete semantic rules are implemented. In that sense, we need to model those implemented semantic rules, and generate inputs most of which satisfy the implemented semantic rules such that these inputs can have the possibility to further challenge the application execution implementation and the unimplemented semantic rules.

Example 3. Fig. 4a shows a sample XSL file, which is used to illustrate some semantic rules. For clarity, we also give its AST in Fig. 4b.

TABLE I: Examples of Semantic Rules

#	Error Messages of Violating Semantic Rules	Context
1.	XML declaration not well-formed	parent
2.	The root element that declares the document to be an XSL style sheet is xsl:stylesheet or xsl:transform	parent and first sibling
3.	Unexpected attribute {...}	first sibling
4.	Unbound prefix	first sibling
5.	XSL element xsl:stylesheet can only contain XSL elements	great-grandparent
6.	Required attribute {...} is missing	first sibling and all mandatory attributes
7.	Duplicate attribute	all siblings

- The two $attribute$ nodes 5 and 6 in Fig. 4b can be either $version$ or $encoding$ as their parent is a $prolog$ node. If they are other attributes, the error message "XML declaration not well-formed" will be thrown by XSLT engines.
- The $element$ node 3 can only be $xsl:stylesheet$ or $xsl:transform$ because its parent is a $document$ node and its sibling is a $prolog$ node. It reflects the semantic rule that "the root element that declares the document to be an XSL style sheet is $xsl:stylesheet$ or $xsl:transform$".
- The two $attribute$ nodes 9 and 10 can only be $version$ or $xmlns:xsl$ since their parent is an $element$ node of type $xsl:stylesheet$. Otherwise, the error message "unexpected attribute {...}" will be reported.
- The $attribute$ node 10 can only be set to $xmlns:xsl =$ "$http://www.w3.org/1999/XSL/Transform$" once the $attribute$ node 9 is set to $version =$ "1.0" since its parent is an $element$ node of type $xsl:stylesheet$ and these two attributes are mandatory. Otherwise, the error message "required attribute {...} is missing" is reported.

As shown in Example 3, semantic rules determine whether a production rule can be applied on a non-terminal symbol, i.e., the application *context* of a production rule; and the context is usually related to the non-terminal symbol's parent, grandparent, great-grandparent, and siblings. Table I reports several common semantic rules (i.e., the error message of their violation and the required context information), including the semantic rules il-

TABLE II: Part of the Learned Production Rules of XSL

ID	Context	Production Rule		Probability
1	<null, null, null, null>	document	→ prolog element	0.82
2			→ element	0.18
3	<null, null, document, null>	prolog	→ <?xml attribute attribute?>	0.646
4			→ <?xml attribute?>	0.347
5			→ ...	
6	<null, null, document, prolog>	element	→ <xsl:stylesheet attribute attribute attribute>content</xsl:stylesheet>	0.0034
7			→ <xsl:transform attribute attribute>content</xsl:transform>	0.0001
8			→ ...	
9	<document, element, content, element>	element	→ <xsl:template attribute>content</xsl:template>	0.0282
10			→ <xsl:variable attribute>content</xsl:variable>	0.0035
11			→ <xsl:include attribute/>	0.0026
12			→ ...	
13	<null, document, prolog, <?xml >	attribute	→ version="1.0"	0.0056
14			→ encoding="utf-8"	0.0021
15			→ ...	
16	<null, document, element, <xsl:stylesheet>	attribute	→ xmlns:xsl="http://www.w3.org/1999/XSL/Transform"	0.0068
17			→ version="1.0"	0.0052
18			→ ...	
19	<element, content, element, <xsl:text>	content	→ chardata	0.0750
20			→ reference	0.0073
21			→ ...	

lustrated in Example 3. Based on this understanding, we define a *context-sensitive grammar* (CSG) to incorporate semantic rules in the grammar.

Definition 2. Similar to CFG, a context-sensitive grammar (CSG) is a 4-tuple $G_{cs} = (N, \Sigma, R, s)$, where

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,
- R is a finite set of context-aware production rules of the form $[c]\alpha \rightarrow \beta_1\beta_2...\beta_n$, where c is the context in the form of $\langle$type of α's great-grandparent, type of α's grandparent, type of α's parent, value of α's first sibling or type of α's first sibling if the value is null$\rangle$, $\alpha \in N, n \geq 1$, and $\beta_i \in (N \cup \Sigma)$ for $i = 1...n$,
- and $s \in N$ is a distinguished start symbol.

Instead of capturing all kinds of semantic rules, we define the context in such a way that most semantic rules can be expressed and they can be efficiently learned in the learning step and efficiently checked in the generation step. Notice that the first five semantic rules in Table I can be captured, but the last two cannot be expressed due to the large amount of information required to model the context. Currently we think such hierarchical relations are sufficient in capturing the diversity of the semantic rules, while keeping redundancy low. However, the context can be extended (e.g., by using more than one sibling and/or by using more or fewer ancestors) to incorporate such semantic rules.

On the other hand, we need to generate inputs that are uncommon enough to trigger exceptional program behaviors. Thus, we define a probabilistic context-sensitive grammar (PCSG) to capture the frequency of a production rule in the corpus.

Definition 3. A probabilistic context-sensitive grammar (PCSG) is a tuple $G_p = (G_{cs}, q)$, where

- $G_{cs} = (N, \Sigma, R, s)$ is a CSG,
- and $q : R \rightarrow \mathbb{R}^+$ assigns each production rule with a probability so that $\forall \alpha \in N$:

$$\sum_{[c]\alpha \rightarrow \beta_1\beta_2...\beta_n \in R} q([c]\alpha \rightarrow \beta_1\beta_2...\beta_n) = 1.$$

Here $\mathbb{R}^+$ denotes the set of non-negative real numbers. Definition 3 ensures that for a given non-terminal symbol, the probability of the applicable production rules for all contexts sums to one. In summary, by integrating both context and probability of a production rule, a PCSG is more expressive than a CFG or a CSG, which summarizes the knowledge of the samples in a single data structure.

B. Learning a Probabilistic Context-Sensitive Grammar

Considering the vast amount of available samples on the Internet, we propose to extract the knowledge inside the samples to automatically and efficiently learn a PCSG. Our learning step requires two inputs: samples and their grammar.

Given the grammar, we parse the samples into abstract syntax trees (ASTs). Then, we traverse the ASTs to extract every parent-children pair and the corresponding context information (as defined in Definition 2), which correspond to the application of a production rule $\alpha \rightarrow \beta_1\beta_2...\beta_n$ under a context c.

Example 4. In Fig. 4b, node 1 is the parent of nodes 2 and 3, which forms a parent-children pair. This pair corresponds to the application of $document \rightarrow prolog\ element$ under the context $\langle null, null, null, null\rangle$ as $document$ is the root node. Nodes 4, 5, 6, and 7 are the children of node 2, which corresponds to the application of $prolog \rightarrow <?xml\ attribute\ attribute\ ?>$ under the context $\langle null, null, document, null\rangle$. Besides, node

9 and node 16 correspond to the application of $attribute \rightarrow version$ = "1.0" under context $\langle null, document, element, <xsl{:}stylesheet\rangle$ where $<xsl{:}stylesheet$ is the value of node 9's first sibling.

Through counting the occurrence of each parent-children pair under different contexts in the corpus, we can get the maximum likelihood estimation for the probability of each production rule under a context, i.e.,

$$q([c]\alpha \rightarrow \beta_1\beta_2...\beta_n) = \frac{count([c]\alpha \rightarrow \beta_1\beta_2...\beta_n)}{count(\alpha)}$$

where $count([c]\alpha \rightarrow \beta_1\beta_2...\beta_n)$ is the number of times that the rule $\alpha \rightarrow \beta_1\beta_2...\beta_n$ is seen in all ASTs under the context c, and $count(\alpha)$ is the number of times that α is seen in all ASTs.

Example 5. Table II shows part of the learned production rules of XSL. We can see that, there are only two production rules (1 and 2) whose left-side is a $document$. Their context can only be $\langle null, null, null, null\rangle$, which is consistent with the fact that $document$ is the root node; and their probability is respectively 0.82 and 0.18. Besides, production rules 13 and 14 capture the first semantic rule in Table I, while production rules 6, 7, and 8 reflect the second semantic rule in Table I. It is worth noting that the probability of the production rules for $element$, $attribute$, and $content$ is very low because there are many different types of $elements$, $attributes$, and $contents$ in the corpus.

All the learned production rules are saved in a pool. They will be used during the generation step to generate well-distributed inputs, as will be introduced in Section IV. Notice that here we adopt a simple learning method for the sake of efficiency, and we will investigate the possibility to apply advanced learning algorithms (e.g., deep learning [27]).

IV. Seed Generation

In this section, we elaborate how to generate seeds based on the learned PCSG.

A. Seed Generation

Given the learned PCSG, we can generate a set of seed inputs through left-most derivation (see Definition 4). Specifically, for generating an input t, we can initially set t to the start symbol of PCSG, and iteratively apply the following steps until there is no non-terminal symbol in t: i) get the left-most non-terminal symbol l in t and the corresponding context c, ii) *randomly* choose a production rule r from R_l whose left-side is l given c, and iii) apply r on l in t. This generation method, based on *random left-most derivation*, is terminated when the time budget is reached, or a predefined number of seed inputs are generated. Due to the nature of PCSG, most inputs generated are correct with respect to grammars and semantic rules. Notice that t is discarded if it still contains non-terminals when the time budget is hit.

Definition 4. Given a PCSG $G_p = ((N, \Sigma, R, s), q)$, a left-most derivation is a sequence of symbols $t_0, ..., t_n$, where

- $t_0 = s$, i.e., t_0 contains only the start symbol,
- $t_n \in \Sigma^*$, i.e., t_n is made up of terminal symbols, and Σ^* denotes the set of all possible strings made up of sequences of words taken from Σ,
- t_i for $i = 1, ..., n$ is derived from t_{i-1} by replacing the left-most non-terminal symbol α (with context c) in t_{i-1} with certain β where $[c]\alpha \rightarrow \beta$ is a production rule in R.

Algorithm 1 Generating Seed Inputs from a PCSG

Input: the PCSG $G_p = ((N, \Sigma, R, s), q)$
Output: the set of seeds generated T
1: $T := \emptyset$
2: **repeat**
3: $t := s$ // set the seed input to the start symbol
4: $c := null$ // set the context to null
5: $num := 0$ // set the times of applying rules to 0
6: **repeat**
7: $l :=$ the left-most non-terminal symbol in t
8: update c according to l
9: $R_l :=$ the set of rules in R whose left-side is l given c
10: **if** $random() < 0.9$ **then**
11: heuristically choose a less-frequently applied and less-complexity rule r from low-probability rules in R_l
12: **else**
13: heuristically choose a less-frequently applied and less-complexity rule r from high-probability rules in R_l
14: **end if**
15: replace l in t with the right side of r
16: $num := num + 1$
17: **until** there is no non-terminal symbol in t, or $num == 200$
18: $T := T \cup \{t\}$
19: **until** time budget is reached, or enough seed inputs are generated

However, due to the randomness of choosing production rules in the above generation method, some problems may arise. First, the generation might become non-terminating since production rules can be recursive. Hence, expanding a non-terminal symbol by applying a recursive production rule can lead to a string that includes the same non-terminal symbol again. For instance, an $element$ in XSL can contain a $content$, and the $content$ can contain another $element$. Further, this problem becomes amplified when a production rule can contain many non-terminal symbols. For example, a $content$ can contain hundreds of $elements$.

Second, the inputs generated might become unnecessarily complex because of the production rules that are recursive (i.e., increasing the complexity in depth), or have many non-terminal symbols (i.e., increasing the complexity in breadth). As a result, the inputs generated can be very large in size, which might be directly rejected by fuzzers (e.g., AFL [7] refuses to take inputs that are larger than 1 MB by default).

To address these problems and follow our three main goals, we propose a generation method based on *heuristic left-most derivation*. Algorithm 1 describes the procedure of our method. The key difference from the previous random generation method is that, we design several heuristics into the left-most derivation (Line 10, 11, 13, and 17) to heuristically choose a rule r from R_l, whose left-side is l given the context c (Line 7–9).
Heuristic 1: Favor low-probability production rules. To generate uncommon inputs that have a relatively high chance to trigger unintended program behaviors, we leverage the probability

Fig. 5: An Example of the Seed Generating Procedure by Heuristically Applying the Left-Most Derivation

```
<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform">
    <xsl:output omit-xml-declaration="yes"/>
</xsl:stylesheet>
```

Fig. 6: The Input Mutated from Fig. 4a

of each rule to first partition R_l into high-probability rules R_l^H and low-probability rules R_l^L, and then choose a rule according to Heuristics 2 and 3 from R_l^L with higher probability (i.e., 0.9) than from R_l^H (Line 10–13). Here we established 0.9 as a good threshold value empirically. Notice that the partition is achieved by sorting the rules based on the probability and finding the two neighbor rules that have the maximum probability difference.

Heuristic 2: Favor low-frequency production rules, and restrict the application number of the same production rule. To prevent rules from being densely or sparsely applied and thus generate expressive inputs that have diverse grammar structures, we record the history information about the application of a rule, i.e., the frequency that a rule has been applied in the generation of an input. In particular, we first trim those rules that have been applied for 3 times (which is empirically established), and then prefer the low-frequency rules to high-frequency rules.

Heuristic 3: Favor low-complexity production rules. To reduce the unnecessary complexity in breadth of inputs generated, we use the number of non-terminal symbols in a rule to measure the complexity of a rule, and favor low-complexity rules. With Heuristic 2, we prefer low-frequency and low-complexity rules when choosing a rule from R_l^L or R_l^H at Line 11 or 13.

Heuristic 4: Restrict the total number of rule applications. To reduce the unnecessary complexity in depth of inputs generated, we empirically limit the total number of rule applications to be 200 in the generation of an input (Line 17). With Heuristic 3, we also partially resolve the non-terminating problem so that the generation step can quickly converge.

Example 6. Fig. 5 shows an example of our input generation by heuristically applying left-most derivation to XSL. t_i represents the result string after applying a production rule. t_0 is initialized with the start symbol $document$. The generation is conducted until no more non-terminal symbol is left, which is successfully completed in 10 rule applications. In fact, this simple input is the same one to Fig. 4; and it crashed the XSLT engine Sablotron 1.0.3 [28]. Specifically, it triggers a buffer underflow that can be exploited to code execution.

TABLE III: Statistics of Samples Crawled and Generated

Language	XSL	XML	JavaScript
# of Unique Samples Crawled	18,686	19,324	525,647
# of Distinct Samples Crawled	671	732	NA
# of Distinct Seeds Generated by Skyfire	5,017	5,923	NA

B. Seed Selection

Our seed generation method can generate many inputs. However, not all inputs generated are unique and important. To select unique inputs and reduce the seed redundancy, we use the code coverage as the criterion to perform seed selection, which is also the commonly-used criterion in fuzzers to keep a seed. Instead of relying on fuzzers to reactively keep unique seeds through a time-consuming procedure of mutation and execution, we select the seeds in a proactive way before feeding them to fuzzers. In particular, we use *gcov* [29] to obtain line coverage and function coverage for open-source programs, and static analysis and coverage tools built on *PIN* [30] to get basic block coverage for closed-source programs. If one input triggers new block coverage, we keep it as a seed.

C. Seed Mutation

After seed generation and selection, the inputs generated have the diversity of grammar structures. To further ensure their semantic diversity, we slightly mutate each input generated by randomly replacing a leaf-level node in the AST with the same type of nodes according to the production rules, to provide an ad-hoc disturbance on leaf nodes distribution. In other words, only the production rules whose right sides contain only terminal symbols will be used in this mutation procedure. Compared to those small-step mutations (e.g., byte flipping) in fuzzers, ours can be seen as big-step mutations that can hardly be achieved by fuzzers' small-step mutations. It is very difficult to get from $version =$ "1.0" to $encoding =$ "utf-8" by randomly flipping bits. Finally, the inputs after mutation can be fed to a fuzzer.

Example 7. Considering the input as shown in Fig. 4, we can replace node 22 with another value of $attribute$ type from the pool of production rules. Fig. 6 shows such an input, mutated from Fig. 4a by replacing node 22 with $omit\text{-}xml\text{-}declaration$ = "yes".

TABLE IV: Unique Bugs Found in Different XSLT and XML Engines

Unique Bugs (#)	XSL						XML		
	Sablotron 1.0.3			libxslt 1.1.29			libxml2 2.9.2/2.9.3/2.9.4		
	Crawl+AFL	Skyfire	Skyfire+AFL	Crawl+AFL	Skyfire	Skyfire+AFL	Crawl+AFL	Skyfire	Skyfire+AFL
Memory Corruptions (New)	1	5	8[§]	0	0	0	6	3	11[¶]
Memory Corruptions (Known)	0	1	2[†]	0	0	0	4	0	4[‡]
Denial of Service (New)	8	7	15	0	2	3	2	1	3[⊕]
Total	9	13	25	0	2	3	12	4	18

[§] CVE-2016-6969, CVE-2016-6978, CVE-2017-2949, CVE-2017-2970, and one pending report.

[¶] CVE-2015-7115, CVE-2015-7116, CVE-2016-1835, CVE-2016-1836, CVE-2016-1837, CVE-2016-1762, and CVE-2016-4447; pending reports include GNOME bugzilla 766956, 769185, 769186, and 769187. After Feb 9, 2017, it was communicated via GNOME bugzilla by Apple libxml2 maintainer that the latter three reports will be addressed by a 'combined patch' to report 764615 and 765468.

[†] CVE-2012-1530, CVE-2012-1525.

[‡] CVE-2015-7497, CVE-2015-7941, CVE-2016-1839, and CVE-2016-2073.

[⊕] GNOME bugzilla 759579, 759495, and 759675.

V. Implementation and Evaluation

We implemented Skyfire in Java with 3.1k lines of code. In particular, we use Heritrix [31], an open-source, extensible, and web-scale crawler project, to collect samples and establish the corpus. For a given grammar, we use ANTLR [32] to automatically generate the corresponding lexer and parser. The lexer and parser generated can parse the collected samples into ASTs to facilitate the learning and generation steps.

A. Evaluation Setup

To evaluate the effectiveness and generality of our approach, we choose XSL, XML, as well as JavaScript, as the target languages for seed generation. The grammars of these languages are available in the ANTLR community [26]. Note that XSL and XML share the same grammar but have different semantic rules.

Sample Collection. To collect samples, we conducted a three-month crawling. During this period, we downloaded 5.3 TB of 21,507,025 web pages in total, including XSL files, XML files, JavaScript files, HTML files, images, etc. We only kept XSL, XML, and JavaScript files. After crawling, we removed duplicated samples through computing their hash values and removed invalid samples through parsing them using ANTLR [32]. As reported in the first row in Table III, we collected 18,686, 19,324, and 525,647 unique XSL, XML, and JavaScript samples. Compared with JavaScript, XSL, and XML are relatively rare resources, and thus the number of XSL and XML samples are relatively small.

Target Programs. We used two open-source XSLT engines Sablotron 1.0.3 [28] and libxslt 1.1.29 [33], the open-source XML engine libxml2 2.9.2, 2.9.3, and 2.9.4 [34], and the closed-source JavaScript/rendering engine in Internet Explorer 11 as the target programs for fuzzing. For open-source projects, we compiled them with AddressSanitizer [35], and for closed-source project, we enabled Page Heap and Application Verifier [36].

- Sablotron is a efficient, compact, and portable XSL toolkit, which implements XSLT 1.0, DOM Level 2, and XPath 1.0. It is an open-source project and subject to the Mozilla Public License or the General Public License, and uses Expat by James Clark as the XML parser. It is adopted into Adobe PDF Reader and Adobe Acrobat.
- libxslt is the XSLT C library developed for the GNOME project, which is based on libxml2. libxslt is used in a variety of products such as Chrome browser, Safari browser, and PHP 5.
- libxml2 is the XML C parser and toolkit developed for the GNOME project, available open-source under the MIT License. It is also widely used outside GNOME due to its excellent portability and existence of various bindings. It is used in Linux, Apple iOS/OS X, and tvOS. The Google Patch Reward Program lists libxml2 as a core infrastructure data parser [37].
- Internet Explorer is a discontinued series of graphical web browsers developed by Microsoft and included as part of the Microsoft Windows line of operating systems, starting from 1995. Internet Explorer is one of the most widely used web browsers.

AFL. In the experiments, we used AFL [7] as the fuzzer as it is designed to be practical: it has modest performance overhead, applies a variety of highly-effective fuzzing strategies and effort minimization tricks, requires almost no configuration and scales well to real-world programs. In particular, it uses a novel type of compile-time instrumentation to discover interesting test inputs that can trigger new internal states in the fuzzed program, which substantially improves the coverage of the fuzzed program.

Fuzzing Approaches. First, we directly applied the samples crawled as well as the inputs generated by Skyfire to fuzz several XSLT and XML engines. These approaches are respectively referred to as *Crawl* and *Skyfire*. Then, we fed the samples crawled and the inputs generated by Skyfire as two sets of seeds to AFL [7] to fuzz the same XSLT and XML engines. These approaches are referred as *Crawl+AFL* and *Skyfire+AFL*. In the experiments, we compared these four approaches. Note that, for JavaScript, we could not use AFL since it is closed-source, and it is too large and slow for our native port of Windows-AFL to fuzz. Instead, we applied the inputs generated by Skyfire directly to fuzz Internet Explorer.

TABLE V: New Vulnerabilities and Types

Vulnerability	Type
CVE-2016-6978	Out-of-bound read
CVE-2016-6969	Use-after-free
Pending advisory 1	Double-free / UAF
CVE-2017-2949	Out-of-bound write
CVE-2017-2970	Out-of-bound write
CVE-2015-7115	Out-of-bound read
CVE-2015-7116	Out-of-bound read
CVE-2016-1762	Out-of-bound read
CVE-2016-1835	Use-after-free
CVE-2016-1836	Use-after-free
CVE-2016-1837	Use-after-free
CVE-2016-4447	Out-of-bound read
Pending advisory 2	OOB read / UAF
Pending advisory 3	Out-of-bound read
Pending advisory 4	Use-after-free
Pending advisory 5	Out-of-bound read

Seed Generation. Based on the guidelines in AFL, we need to use the *afl-cmin* utility to identify a set of functionally distinct seeds that exercise different code paths in the target program when a large amount of seeds are available, which was the case in our experiments. After using *afl-cmin* on the samples crawled, we had 671 and 732 distinct XSL and XML samples, as shown in the second row in Table III. For a fair comparison, we used Skyfire to generate the same number of inputs to the samples crawled for XSL, XML, and JavaScript. After using *afl-cmin* on these inputs generated, we had 5,017 and 5,923 distinct XSL and XML seeds, as shown in the last row in Table III.

Research Questions. Using the previous experiments setup, we aim to answer the following research questions through the experiments:

- **RQ1**: Can Skyfire generate good seeds for a fuzzer to find bugs? (Section V-B)
- **RQ2**: Can Skyfire improve the code coverage of target programs? (Section V-C)
- **RQ3**: Are the proposed PCSG and the four heuristics used in Skyfire effective in generating seeds? (Section V-D)
- **RQ4**: How efficient is Skyfire? (Section V-E)

We ran the experiments with eight VirtualBox 5.0.4 [38] virtual machines, each of which was configured with 8 CPUs and 3GB RAM. Six of them ran Ubuntu 14.04, which were used for fuzzing Sablotron, libxslt, and libxml2; and two of them ran Windows 7, which were used for fuzzing Internet Explorer 11. We have fuzzed XSLT and XML engines for a span of 15 months, but we have just started to fuzz the JavaScript and rendering engine in Internet Explorer 11 for two months. For the ease of presentation, we will separately discuss the preliminary results for JavaScript in Section V-G.

B. Vulnerabilities and Bugs Discovered (RQ1)

Table IV shows the unique bugs that were found in Sablotron, libxslt, and libxml2 by Crawl+AFL, Skyfire, and Skyfire+AFL

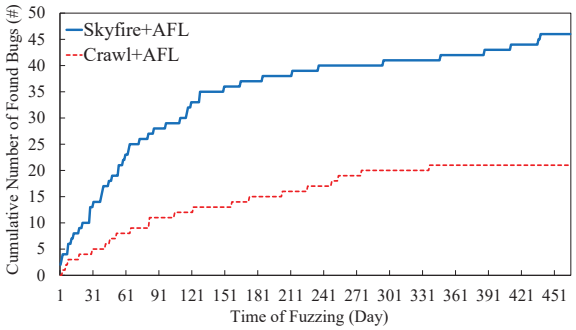


Fig. 7: The Cumulative Number of Unique Bugs over Time

respectively. Notice that the Crawl approach used the well-formed samples crawled and thus did not trigger any bugs.

Bug Results. Skyfire+AFL discovered 19 previously unknown memory corruption bugs. However, Skyfire only found eight of them, while Crawl+AFL only found seven of them. Moreover, Skyfire+AFL discovered six known memory corruption bugs, but using Skyfire alone, we only found one of them and Crawl+AFL found four of them. Further, Skyfire+AFL found 21 new denial of service bugs (including stack exhaustion, NULL pointer dereference, and assertion failure). Among them, both Skyfire and Crawl+AFL only found ten. Notice that all the bugs were found at the application execution stage (see Fig. 1). To sum up, all the bugs discovered by Skyfire and Crawl+AFL were also found by Skyfire+AFL, while Skyfire+AFL found bugs that were not found by Skyfire or Crawl+AFL. These results indicates that, Skyfire can provide good seed inputs for a fuzzer, hence releasing the burden of finding good seeds through continuous execution and mutation from the fuzzer, and significantly improving the capability of the fuzzer to discover bugs for programs that process highly-structured inputs.

Vulnerability Results. From the 19 previously unknown memory corruption bugs, we successfully discovered 16 new vulnerabilities whose types are shown in Table V. Ten of them are out-of-bound read/write vulnerabilities, while seven of them are user-after-free vulnerabilities; In total 11 CVE identifiers are assigned, and five reports are still unresolved. For the other three previously unknown memory corruption bugs in Sablotron, they are not reproducible in the latest Adobe Reader. Two of them have been internally fixed by Adobe since the adoption of Sablotron 1.0.2, and one is no longer exploitable due to code refactoring by Adobe. In addition, there are 21 new denial of service bugs, which are not exploitable. For Sablotron, we did not send the bug reports to Adobe due to low severity; for libxml2, we submitted three stack overflow (i.e., stack exhaustion due to infinite recursion) reports at the time of writing.

Capability to Keep Finding Bugs. Fig. 7 presents the cumulative number of unique bugs over the whole fuzzing time. The X axis denotes the number of days fuzzing was conducted, and the Y axis denotes the cumulative number of unique bugs. It can be seen that, both Skyfire+AFL and Crawl+AFL kept detecting unique bugs in the first eight months, but Skyfire+AFL found

TABLE VI: Line and Function Coverage of Sablotron, libxslt, and libxml2

Program			Line Coverage (%)				Function Coverage (%)			
Name	Lines	Functions	Crawl	Crawl+AFL	Skyfire	Skyfire+AFL	Crawl	Crawl+AFL	Skyfire	Skyfire+AFL
Sablotron 1.0.3	10,561	2,230	34.0	39.0	65.2	69.8	29.8	32.6	48.1	50.1
libxslt 1.1.29	14,418	778	29.6	38.1	57.4	62.5	30.0	34.2	51.9	53.1
libxml2 2.9.4	67,420	3,235	13.5	15.3	22.0	23.8	15.7	16.3	24.1	25.9

TABLE VII: Detailed Code Coverage of Sablotron 1.0.3

File (.cpp)	#Bug	Line Coverage (%)				Function Coverage (%)			
	25	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL
arena	0	86.8	92.1	92.1	92.1	85.7	85.7	85.7	85.7
base	0	45.7	45.7	92.6	92.6	61.5	61.5	84.6	84.6
context	**1**	**31.8**	**44.9**	**79.6**	**90.6**	**41.0**	**51.3**	**84.6**	**87.2**
datastr	**2**	**69.5**	**70.3**	**79.1**	**82.4**	**68.5**	**69.4**	**79.3**	**81.1**
decimal	**3**	**8.9**	**8.9**	**66.5**	**91.1**	**28.0**	**28.0**	**72.0**	**72.0**
domprovider	0	14.1	19.8	40.2	40.8	20.7	27.6	50.0	50.0
encoding	0	38.1	50.4	52.2	53.1	61.5	69.2	76.9	76.9
error	0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
expr	**1**	**31,4**	**52.6**	**87.0**	**90.3**	**38.1**	**61.9**	**79.4**	**80.4**
hash	0	60.5	60.5	83.2	83.2	61.5	61.5	69.2	69.2
key	0	4.3	4.3	78.4	79.9	12.0	12.0	72.0	72.0
numbering	0	0.0	0.0	65.2	93.5	0.0	0.0	84.6	100.0
output	**6**	**51.8**	**52.0**	**86.9**	**89.3**	**53.8**	**53.8**	**80.8**	**80.8**
parser	0	67.8	69.1	81.8	94.9	45.7	45.7	82.9	97.1
platform	0	68.4	89.5	100.0	100.0	50.0	83.3	100.0	100.0
proc	0	41.2	42.7	67.9	68.7	36.2	40.0	66.2	66.2
sablot	0	23.3	23.3	23.3	23.3	19.0	19.0	19.0	19.0
sdom	0	0.2	0.2	0.2	0.2	1.4	1.4	1.4	1.4
situa	0	59.5	60.7	64.6	65.4	45.0	45.0	52.5	55.0
sxpath	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
tree	**4**	**36.5**	**38.4**	**72.4**	**83.6**	**41.7**	**42.7**	**68.0**	**75.7**
uri	0	48.3	48.3	70.8	72.9	60.0	60.0	75.0	75.0
utf8	**4**	**34.3**	**34.3**	**62.7**	**64.9**	**38.9**	**38.9**	**66.7**	**66.7**
vars	**1**	**11.5**	**15.3**	**86.6**	**89.8**	**22.2**	**29.6**	**92.6**	**92.6**
verts	**3**	**30.9**	**31.7**	**72.0**	**76.5**	**38.3**	**39.5**	**57.5**	**62.3**

TABLE VIII: Detailed Code Coverage of libxslt 1.1.29

File (.c)	#Bug	Line Coverage (%)				Function Coverage (%)			
	3	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL
common	0	9.6	9.6	80.8	82.7	33.3	33.3	100.0	100.0
date	0	2.7	2.7	21.7	26.6	1.4	1.4	35.7	40.0
dynamic	0	3.1	3.1	3.1	3.1	33.3	33.3	33.3	33.3
exslt	0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
functions	**2**	**3.6**	**9.1**	**73.8**	**78.2**	**23.1**	**38.5**	**92.3**	**92.3**
math	0	5.4	5.4	30.8	33.5	2.8	2.8	44.4	44.4
saxon	0	9.3	9.3	38.4	53.5	12.5	12.5	62.5	75.0
sets	0	7.2	7.2	56.8	58.6	12.5	12.5	62.5	62.5
strings	0	2.7	2.7	43.0	49.2	9.1	9.1	63.6	63.6
attributes	0	3.2	3.2	83.1	85.7	13.3	13.3	86.7	86.7
attrvt	0	65.2	91.3	93.8	95.0	83.3	100.0	100.0	100.0
documents	0	48.4	71.1	75.0	77.3	66.7	77.8	77.8	77.8
extensions	0	46.5	56.2	64.3	64.3	56.5	64.5	71.0	71.0
extra	0	11.5	38.5	47.9	47.9	20.0	40.0	60.0	60.0
functions	0	12.9	30.8	67.8	74.0	33.3	50.0	91.7	91.7
imports	0	60.9	89.9	89.1	89.9	85.7	85.7	85.7	85.7
keys	0	24.0	38.3	86.8	88.3	35.7	57.1	92.9	92.9
namespaces	0	37.1	50.6	74.7	82.9	57.1	57.1	71.4	71.4
numbers	0	0.0	0.0	52.3	87.6	0.0	0.0	52.5	93.8
pattern	0	46.2	62.5	81.4	86.7	57.6	63.6	75.8	78.8
preproc	0	70.7	78.4	94.7	96.5	89.7	89.7	100.0	100.0
security	0	23.5	26.1	53.0	55.7	46.2	46.2	69.2	69.2
templates	0	22.7	43.0	72.2	75.9	18.2	45.5	72.7	72.7
transform	**1**	**41.1**	**55.7**	**72.2**	**77.8**	**49.3**	**62.7**	**74.6**	**76.1**
variables	0	51.4	63.3	68.6	70.8	66.7	69.4	77.8	77.8
xslt	0	68.5	77.9	86.1	87.2	78.4	78.4	89.2	89.2
xsltlocale	0	1.7	1.7	29.7	89.0	16.7	16.7	66.7	66.7
xsltutils	0	22.6	29.5	48.3	56.7	34.1	34.1	48.8	48.8
libxslt-py	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
libxslt	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
testThreads	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
xsltproc	0	28.4	30.0	30.5	30.5	42.9	42.9	42.9	42.9

bugs more effectively and more efficiently than Crawl+AFL. In the next seven months, Crawl+AFL became extremely slow in detecting unique bugs. Differently, Skyfire+AFL, on the other hand, still continuously found unique bugs, although becoming less efficient.

> Based on the bug and vulnerability results from Table IV and V and Fig. 7, we can positively answer **RQ1** that Skyfire can generate high-quality seed inputs for a fuzzer, hence making the fuzzer keep detecting bugs and significantly improving the fuzzer's capability to discover bugs and vulnerabilities.

C. Code Coverage (RQ2)

Table VI reports the line and function coverage of Sablotron, libxslt, and libxml2 before and after the 15-month fuzzing. The first three columns report the lines of code and the number of functions for each program. The other columns show the line and function coverage achieved by the four approaches. Here we only show the results for libxml2 2.9.4 and do not explicitly show results for libxml2 2.9.2 and 2.9.3, since they have the similar results.

Overall Coverage Results. For line coverage, our samples crawled covered 34.0% lines of Sablotron, 29.6% lines of libxslt, and 13.5% lines of libxml2. After the 15-month fuzzing, AFL respectively increased their line coverage to 39.0%, 38.1%, and 15.3%. On average, through AFL, 5.1% of the code was further covered. For Skyfire, the inputs generated covered 65.2% lines of Sablotron, 57.4% lines of libxslt, and 22.0% lines of libxml2. We can see that the inputs generated already had a much higher coverage than Crawl+AFL. After fuzzing, AFL improved their line coverage to 69.8%, 62.5%, and 23.8% respectively; and further covered 3.8% of the code. On the

TABLE IX: Detailed Code Coverage of libxml2 2.9.4

File (.c)	#Bug	Line Coverage (%)				Function Coverage (%)			
	18	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL	Crawl	Crawl +AFL	Skyfire	Skyfire +AFL
HTMLparser	**3**	**0.4**	**0.4**	**33.6**	**46.7**	**1.0**	**1.0**	**49.0**	**57.3**
HTMLtree	0	1.0	1.0	41.9	54.0	4.2	4.2	33.3	41.7
SAX	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SAX2	**1**	**38.1**	**39.7**	**53.3**	**53.7**	**55.1**	**57.1**	**69.4**	**69.4**
buf	0	48.4	55.9	63.0	63.5	59.5	62.2	67.6	67.6
c14n	0	0.0	0.0	37.5	40.3	0.0	0.0	55.0	57.5
catalog	0	0.4	0.4	23.6	23.9	2.8	2.8	33.3	33.3
chvalid	0	0.0	35.9	35.9	35.9	0.0	11.1	11.1	11.1
debugXML	0	0.0	0.0	0.0	14.8	0.0	0.0	0.0	21.1
dict	**6**	**65.4**	**78.9**	**79.4**	**79.4**	**70.8**	**75.0**	**75.0**	**75.0**
encoding	0	49.2	60.0	65.0	66.1	70.6	73.5	79.4	79.4
entities	0	35.0	38.9	52.1	54.5	43.5	43.5	60.9	60.9
error	**1**	**48.2**	**52.0**	**63.7**	**63.7**	**40.0**	**45.0**	**60.0**	**60.0**
globals	0	19.5	19.5	19.5	26.2	44.2	44.2	44.2	55.8
hash	0	62.1	69.0	70.3	70.3	70.0	70.0	73.3	73.3
legacy	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
list	0	24.2	24.2	34.3	34.3	21.2	21.2	30.3	30.3
parser	**5**	**48.3**	**55.9**	**64.4**	**65.2**	**60.4**	**62.6**	**72.0**	**73.1**
parserInternals	**2**	**56.6**	**66.4**	**72.0**	**72.1**	**57.9**	**60.5**	**73.7**	**73.7**
pattern	0	0.0	0.0	9.1	9.1	0.0	0.0	18.4	18.4
relaxng	0	0.1	0.1	0.1	0.1	0.7	0.7	0.7	0.7
runsuite	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
runtest	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
threads	0	33.3	33.3	33.3	33.3	57.1	57.1	57.1	57.1
tree	0	20.0	21.5	26.8	26.9	28.6	29.8	38.1	38.1
trionan	0	40.6	40.6	40.6	40.6	25.0	25.0	25.0	25.0
uri	0	39.6	59.2	62.6	63.2	71.4	80.0	82.9	82.9
valid	0	27.9	28.5	35.9	37.3	36.4	36.4	47.5	49.2
xinclude	0	0.0	0.0	0.0	10.2	0.0	0.0	0.0	24.3
xmlIO	0	44.5	45.6	55.6	63.2	47.5	48.8	56.2	61.2
xmlcatalog	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
xmllint	0	9.4	9.4	38.9	54.7	5.1	5.1	32.2	42.4
xmlmemory	0	7.0	7.0	7.0	7.0	13.6	13.6	13.6	13.6
xmlreader	0	0.0	0.0	22.8	22.8	0.0	0.0	14.4	14.4
xmlsave	0	57.1	59.9	62.6	67.7	46.6	46.6	48.3	53.4
xmlschemastypes	0	0.1	0.1	0.1	0.1	1.5	1.5	1.5	1.5
xmlstring	0	23.1	35.1	44.3	44.9	35.5	41.9	54.8	54.8
xpath	0	0.1	0.1	5.7	5.7	0.4	0.4	13.1	13.1

whole, Skyfire+AFL outperformed Crawl+AFL by around 20% in line coverage.

On the other hand, in terms of function coverage, the samples crawled covered 25.2% functions on average; and AFL further covered 2.5% functions. Instead, the inputs generated by Skyfire covered 41.1% functions on average; and AFL further covered 1.7%. Generally, Skyfire+AFL outperformed Crawl+AFL by about 15% in function coverage.

From these results, we can see that, Skyfire can provide well-distributed seed inputs for a fuzzer such that the fuzzer's code coverage can be greatly improved. Another interesting finding is that, given well-distributed seed inputs, the increased coverage through AFL was reduced instead, i.e., from Crawl+AFL's 5.1% to Skyfire+AFL's 3.8% in line coverage and from Crawl+AFL's 2.5% to Skyfire+AFL's 1.7% in function coverage. This can be explained that, Skyfire releases the burden of finding seeds to reach interesting portions of the fuzzed program from the fuzzer such that the fuzzer can focus attention on triggering unintended behaviors in these interesting portions.

Detailed Coverage Results. We analyzed the line and function coverage of Sablotron, libxslt, and libxml2 at the source code file level in detail. Table VII, VIII, and IX report the coverage of all the `.c` or `.cpp` files. The first column shows the file name, and the second column lists the number of unique bugs found in a file. Similar to Table VI, the next four columns present the line coverage of each file, and the last four columns list the function coverage of each file.

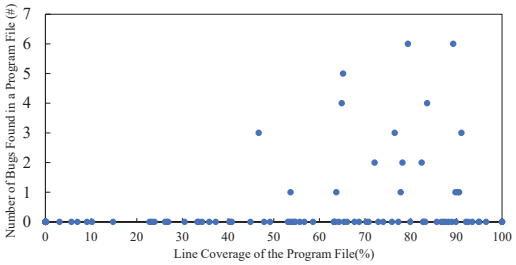
We can see from these tables that, Skyfire+AFL improved the coverage of 21 of 25 files for Sablotron, 26 of 32 files for libxslt, and 28 of 38 files for libxml2 over Crawl+AFL. It indicates that the coverage of most files was increased. Besides, we manually looked into the files that had extremely low code coverage, and analyzed the reasons for such low coverage. First, some files are for testing and thus will not be executed (e.g., `testThreads` in libxslt, and `runsuite`, `runtest`, and `runxmlconf` in libxml2). Second, some files will only be executed with specific configurations. For example, we ran libxslt through command line, but `libxslt-py`, `libxslt`, and `types` can be covered only when we run libxslt via Python interface. Similarly, we ran libxml2 with the default command line configuration, and thus we did not cover many functionalities (e.g., `xmlcatalog` and `c14n`) in libxml2 and had low overall coverage. However, such functionalities can be touched by setting various command line configurations before fuzzing. Third, some files contain code for processing deprecated features (e.g., `sdom` in Sablotron and `SAX` in libxml2), and thus will not be executed.

Code Coverage vs. Bugs. We also analyzed the relationship between code coverage of a file and the number of bugs found in that file, as described in Fig. 8. We can see that, all the bugs were found in those files that had more than 45% line coverage and more than 55% function coverage. It shows that improving code coverage can help to find bugs, and fuzzers can utilize the coverage information to guide their fuzzing process such that they can focus on the less-covered program files.

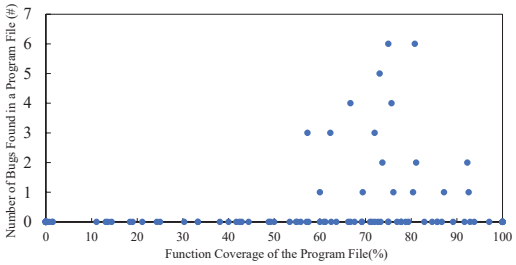
> Based on these observations from Table VI-IX and Fig. 8, we can positively answer **RQ2** that Skyfire can generate well-distributed inputs for a fuzzer, and thus significantly improves the code coverage of a fuzzer (e.g., 20% in line coverage and 15% in function coverage).

D. Effectiveness of Context and Heuristics (RQ3)

Context. To evaluate the effectiveness of context in PCSG, we implemented a CFG-based input generation approach that used Heuristics 3 and 4 to address the non-termination problem (see Section IV-A), and compared the percentage of inputs generated that passed the semantic checking stage (see Fig. 1). For a fair comparison, we used the two approaches to generate the same number of inputs (i.e., 10,000) for XSL and XML. Notice that by looking at the exit code, we can easily determine if an input fails to pass the semantic checking.

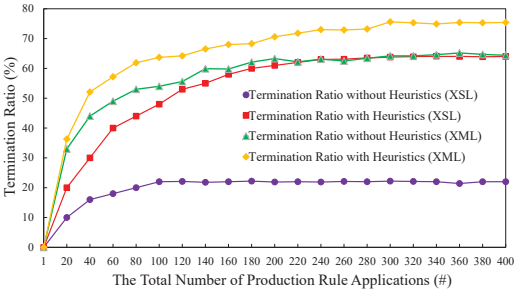


(a) Line Coverage vs. Bugs

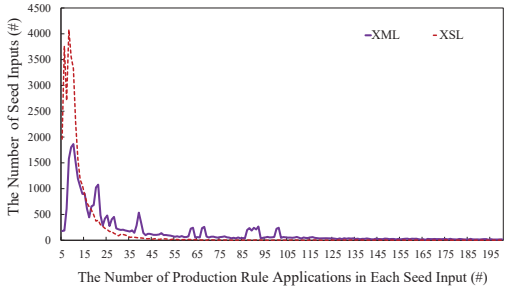


(b) Function Coverage vs. Bugs

Fig. 8: The Relationship between Code Coverage of a File and the Number of Bugs Found in that File



(a) Termination Ratio with and without Heuristics



(b) Distribution of the Number of Rule Applications

Fig. 9: Evaluation Results for the Used Heuristics in our Seed Generation Approach

The results showed that, none of the XSL inputs generated by the CFG-based approach could pass the semantic checking since they all failed to pass the first two semantic rules in Table I. Similarly, only 34% of the XML inputs generated by the CFG-based approach could pass the semantic checking. Instead, by considering context, 85% and 63% of the XSL and XML inputs generated by Skyfire passed the semantic checking and reached the application execution stage. It indicates that, by considering context, PCSG is more effective than CFG in generating inputs that can pass the semantic checking.

Heuristics. Our evaluation in Section V-B and V-C has indicated that Skyfire can generate well-distributed seeds, which also reflects the effectiveness of the Heuristics 1 and 2 (see Section IV-A). To evaluate the effectiveness of the Heuristics 3 and 4, for Skyfire with and without the heuristics, we measured the *termination ratio*, i.e., the number of inputs that are generated within a specified number of production rule applications. Here we varied the total number of production rule applications from 20 to 400, and 1,000 inputs generated. As shown in Fig. 9a, only 20% of the XSL inputs were generated in 100 applications of production rules when the heuristics were not used; and this termination ratio increased to 50% when the heuristics were used. For XML inputs, 55% and 65% of them were generated in 100 applications of production rules when the heuristics were not and were used. This indicates that the Heuristics 3 and 4 effectively resolve the non-termination problem. Notice that, in our implementation, we empirically set the total number of production rule applications (see Heuristic 4) to 200 for a balance between termination and complexity based on Fig. 9a.

Moreover, to evaluate the complexity of the inputs generated, we respectively generated 200,000 XSL and XML inputs, and computed the distribution of the number of rule applications to generate an input. As shown in Fig. 9b, we can see that most of the inputs were generated within 45 applications of production rules, which reflects that the inputs generated are mostly not complex. This indicates that the Heuristics 3 and 4 effectively reduce the unnecessary complexity of the seeds generated.

> Based on these observations, we can positively answer **RQ3** that both context and heuristics used in Skyfire are effective in generating seeds. In particular, context can help generate semantically valid seeds so that they can reach the application execution stage. Heuristics can help generate well-distributed seeds efficiently and reduce the unnecessary complexity.

E. Performance Overhead (RQ4)

To evaluate the efficiency of Skyfire, we measured the execution time to learn a PCSG from the samples crawled (18,686, 19,324, and 525,647 XSL, XML, and JavaScript samples) as well as the execution time to generate 18,686, 19,324, and 525,647 XSL, XML, and JavaScript seeds. As reported in Table X, the PCSGs of XSL and XML were learned in around 1.6 hours, and their generation step only took around 21 seconds. The PCSG of JavaScript was learned in 41.0 hours due to the large number of samples. We can see that, both the learning and generation steps have linear computation time with respect to the number of samples crawled and seeds generated, which is scalable.

> From Table X, we can positively answer **RQ4** that Skyfire is scalable with respect to the learning and generation steps.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
    <xsl:value-of select="substring-before('F','-')"/>
    </xsl:template>
</xsl:stylesheet>
```

(a) Sample Generated that did not Trigger any Bugs

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
    <xsl:value-of select="substring-before('F')"/>
    </xsl:template>
</xsl:stylesheet>
```

(b) Sample Mutated from Fig. 10a by AFL that Triggered a Bug

Fig. 10: A Sample Generated by Skyfire and then Mutated by AFL to Trigger a New Bug that has been Fixed

TABLE X: Performance of Learning and Generation

Time	XSL	XML	JavaScript
Learning (h)	1.5	1.6	41.0
Generation (s)	20.3	20.6	521.2

F. Case Study and Discussion

We can see from Table IV that Skyfire can generate inputs that directly trigger bugs. For example, Fig. 4a was generated by Skyfire, and triggered a buffer underflow bug in Sablotron 1.0.3. However, some bugs cannot be found by our inputs generated; and instead, a fuzzer's capability has to be used to detect them. For example, the code fragment in Fig. 10a was generated by Skyfire and did not trigger any bugs. However, when it was fed to AFL, and mutated by AFL into the code fragment shown in Fig. 10b, an out-of-bound access vulnerability was triggered in Sablotron, which also exists in the XSLT engine of the latest Adobe Reader and Adobe Reader DC.

In particular, this vulnerability is triggered when the *value-of* element is supplied with an XPath expression that has an incorrect call to *substring-after()* or *substring-before()*. When the vulnerability is triggered, an uninitialized Expression Atom object can be fetched via an out-of-bound read, then used for subsequent virtual function call to *tostring()*, allowing straightforward remote exploitation as long as the attacker controls the uninitialized memory content.

The root cause to this vulnerability is the incorrect check for *substring-before()* and *substring-after()* in `expr.cpp` as given in Fig. 11, i.e., the argument count check at Line 941 is after the two *tostring()* calls that use the arguments (at Line 939 and 940). Therefore, when only one argument is provided, as shown in Fig. 10b, *atoms[1]* is an uninitialized Expression atom pointer that leads to code execution via the *tostring()* call.

In this example, AFL's mutation removed the second parameter of function call *substring-before*, which cannot be achieved by Skyfire's generation and mutation. This is because Skyfire involves big-step mutations (e.g., replacing an attribute), and AFL employs small-step mutations (e.g., byte flipping). This indicates that Skyfire is a good complementary to fuzzers. By combining them together, we can greatly improve the bug-finding capability.

G. Preliminary Results for JavaScript

We used the seed JavaScript samples generated with variable names normalized, in conjunction with a relatively fixed HTML template, to fuzz the Trident rendering engine (`mshtml.dll`)

```
934  case EXFF_SUBSTRING_BEFORE:
935  case EXFF_SUBSTRING_AFTER:
936  {
937    Str strg;
938    Str theBigger, theSmaller;
939    E( atoms[0] -> tostring(S, theBigger) );
940    E( atoms[1] -> tostring(S, theSmaller) );
941    checkArgsCount(2);
942    checkIsString2(0,1);
943    int where = firstOccurence(theBigger,theSmaller);
944    if (where == -1)strg.empty();
945    else{
946      if (functor == EXFF_SUBSTRING_BEFORE){
947        if (where == 0)strg.empty();
948        else getBetween(strg, theBigger, 0, where-1);
949      }
950      else getBetween(strg, theBigger,
951      where + utf8StrLength(theSmaller), -1);
952    };
953    retxpr.setAtom(strg);
954  }; break;
```

Fig. 11: The Vulnerable Code Fragement for CVE-2016-6978

of Internet Explorer 11 for around two months. The result has shown that JavaScript, as a more complex language than XML and XSL, has more semantic rules. On the other hand, the JavaScript and rendering engine for Internet Explorer usually manipulate a vast variety of objects that interact in a more complex way, where code coverage may not be a good indicator to characterize such interactions. Therefore, a better indicator is needed to capture such interactions and be used as the feedback during fuzzing.

There are also other subtle issues from the intrinsic design and implementation complexity of JavaScript. One example is variables and their scopes, to generate truly semantic valid JavaScript segments, existing segments with semantic rules for variable definition, assignment and reference have to be parsed and annotated separately; when chaining them together, variable names may be normalized and their initialization and references may have evolutionary cross-overs where these semantic rules are respected.

We leave it as our future work to keep extending our approach to better support more complex languages such as JavaScript and SQL. Specifically, for JavaScript, we plan to target Webkit JavaScriptCore, which is open-source and more standalone; and for the SQL language, we plan to fuzz sqlite3.

Quantitatively, at the time of writing, we have run 1,591,263 JavaScript seed inputs on Internet Explorer 11, and found 11 NULL pointer dereference bugs and achieved 85% basic block coverage as measured by PIN [30]. However, such results still demonstrate the promising seed generation capability of Skyfire.

We will continue enriching the strategies of Skyfire for fuzzing Internet Explorer (or other browsers) as our future work.

VI. Related Work

Instead of listing all related work, we focus our discussion on the most related ones: mutation-based fuzzing, generation-based fuzzing, and fuzzing boosting.

A. Mutation-Based Fuzzing

Mutation-based fuzzing often generates inputs via modifying well-formed seed inputs through mutation operators such as bit flipping and token insertion. Such modifications can be totally random [1], or guided by different heuristics.

AFL [7] applies a novel type of compile-time instrumentation and genetic algorithm to automatically discover interesting test inputs that can trigger new internal states in the fuzzed program. AFL regards a new transition from a basic block to another as a new internal state. Directed by such coverage information, AFL substantially improves the code coverage for the program, and has been widely used in the security community.

BuzzFuzz [8] and TaintScope [9] use taint analysis to locate the interesting byte for guiding the mutation. Specifically, BuzzFuzz [8] applies dynamic taint analysis to automatically locate the regions in the original seed inputs that influence values used at vulnerable points (i.e., points where the program might contain an error). BuzzFuzz then automatically generates new test inputs by fuzzing these identified regions in the original seed inputs. Since these new test inputs typically preserve the underlying syntactic structure of the original seed inputs, they can often pass the initial input parsing components to exercise code deep to those semantic checking components. Similarly, TaintScope [9] first locates the checksum-based integrity checks by branch profiling techniques and bypasses such checks by control flow alteration techniques. Then it uses taint analysis to identify those bytes in a well-formed input that are used in security-sensitive operations and focuses the attention on modifying such bytes.

SAGE [11, 12] and the approach in [10] leverage symbolic execution to perform fuzzing. In particular, SAGE [11, 12] implements a new search algorithm that maximizes the number of new test inputs generated from each run of symbolic execution. Given a path condition, all the constraints rather than one of the constraints in that path are systematically negated one by one, conjuncted with the prefix of the path condition leading to it, and then solved by a constraint solver. In this way, a single run of symbolic execution can generate a set of new test inputs. Babić et al. [10] propose a three-stage processing approach to generate test inputs that can reach the potential vulnerabilities in the program. It first runs dynamic analysis with a small number of seed inputs to resolve indirect jumps in the binary code and builds a visibly pushdown automaton to reflect the global program control-flow. Then it uses static analysis to the inferred automaton to find potential vulnerabilities. Finally, it uses the results of the prior phases to assign weights to automaton edges and then uses symbolic execution to generate test inputs and direct its exploration to the target potential vulnerabilities.

Dowser [13] and BORG [14] combine taint analysis and symbolic execution to guide the fuzzing. Dowser [13] targets buffer overflow and underflow vulnerabilities. It only considers the code that accesses an array in a loop, rather than all the possible instructions in the program. It uses taint analysis to determine the input bytes that influence the array index. After finding all such candidate sets of instructions, it ranks them according to an estimation of how likely they contain interesting vulnerabilities, and then symbolically executes most promising sets to generate the test inputs that trigger the vulnerability. Similarly, BORG [14] targets buffer over-read bugs. It works by first using taint analysis to select buffer accesses that could lead to an over-read bug and then guiding symbolic execution towards those accesses along program paths that could actually lead to an over-read.

Driller [39] combines fuzzing and concolic execution in a complementary way to find deep bugs. Inexpensive fuzzing is used to exercise compartments of an application, while concolic execution is used to generate inputs that satisfy the complex checks separating the compartments. When fuzzing is saturated and fails to trigger any new program behaviors, Driller switches to concolic execution to touch those hard-to-reach branches and generate test inputs, then switches back to fuzzing.

Kargén and Shahmehri [40] takes a different perspective to perform the fuzzing. They propose to perform mutations on the machine code of the generating programs instead of directly on a well-formed input such that they can leverage information about the input format encoded in the generating program to produce high-coverage test inputs.

In summary, these mutation-based fuzzing approaches can effectively fuzz programs that process unstructured or simply-structured inputs (e.g., images and multimedia). However, they become ineffective for programs that process highly-structured inputs (e.g., XSL and JavaScript). As a result, most malformed inputs from mutation-based fuzzing will be rejected at an early stage of program execution, failing to pass the syntax parsing, which makes the fuzzers waste a large amount of time dealing with syntax correctness while only finding trivial parsing errors, and heavily limits them to find deep bugs. Instead, our approach is orthogonal to mutation-based fuzzing techniques because we focus on seed generation and thus can provide well-distributed input seeds to the fuzzers.

B. Generation-Based Fuzzing

Generation-based fuzzing generates test inputs from a specification, and thus the inputs generated are adhere to the format required by applications.

Peach [15] and Spike [16] use the input models as the specification, and combine mutation to generate inputs. Input model specifies the format of data chunks and integrity constraints so that the inputs generated can pass integrity checking (e.g., checksum). Recently, Pham et al. [17] combine such input model-based approaches with symbolic execution. They identify the format constraint of a program using symbolic execution

and ensure the validity of the tests generated, which can help to swiftly carry the exploration beyond the parser code and improve the effectiveness of fuzzing.

CSmith [19], LangFuzz [20], IFuzzer [21], Radamsa [41], and the approach in [18] use the context-free grammar as the specification to generate inputs. Godefroid et al. [18] propose a dynamic test generation approach, where symbolic execution is involved to generate grammar-based constraints whose satisfiability is checked by a grammar-based constraint solver. CSmith [19] generates C programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that may destroy its ability to find bugs. It randomly selects an allowable production rule from the grammar to generate C programs. LangFuzz [20] uses the given grammar to learn code fragments from a given corpus (e.g., a suite of tests previously failed programs). Then it recombines fragments of the provided test suite to generate new programs, assuming that a recombination of previously problematic inputs has a high chance to cause new crashes. IFuzzer [21] uses a language's context-free grammar to extract code fragments from given test samples. Then it recomposes the code fragments in a biological evolutionary way to generate new samples. Radamsa [41] automatically builds a CFG describing the structure of given training samples, and uses the CFG to generate similar data for robustness testing. It strikes a practical balance between completely random and manual test design and proved to be very effective by finding hundreds of bugs.

These input model-based and grammar-based fuzzing approaches often easily pass the syntax parsing, but often fail to pass the semantic checking that checks the semantic validity of the inputs. As a result, only a small portion of the inputs generated from these approaches can reach the application execution stage, where the deep bugs normally hide. Differently, by directly considering semantic rules as contexts in the proposed PCSG, the inputs generated by our approach can mostly pass the semantic checking.

There are also some grammar-based fuzzing approaches [22, 23, 24] that use hard-coded or manually-specified generation rules to express semantic rules. mangleme[22] is an automated broken HTML generator and browser fuzzer, originally used to find dozens of security and reliability problems in all major Web browsers. Jsfunfuzz [23] is one of the most popular fuzzing tools, finding more than 1,000 bugs in the Mozilla JavaScript engine. It uses specific knowledge about past and common vulnerabilities and hard-codes rules to generate inputs. Dewey et al. [24] propose to use constraint logic programming (CLP) for program generation. Using CLP, testers can manually write declarative predicates to specify interesting program features, including syntactic features and semantic behaviors. However, it is daunting and labor-intensive, or even impossible to manually express the required semantic rules. Differently, we propose to directly learn such semantic rules from the existing samples, and automatically leverage such learned knowledge to generate inputs.

C. Fuzzing Boosting

Several boosting techniques [6, 42, 43, 44, 45] have been proposed to improve the efficiency of current fuzzing approaches.

Householder and Foote [42] introduce a machine learning-based algorithm for selecting two fuzzing parameters (i.e., the seed input and the proportion of the seed input for mutation) to maximize the number of unique application errors discovered during a fuzzing campaign. They greatly improve the efficiency of discovering unique application errors over basic parameter selection techniques.

Woo et al. [43] empirically investigate how to schedule the fuzzing of a given set of program-seed pairs in order to maximize the number of unique bugs found. To this end, they build a mathematical model for black-box mutational fuzzing, and use it to evaluate 26 existing and new randomized online scheduling algorithms. Similarly, Rebert et al. [6] empirically study how to pick seed files to maximize the total number of bugs found during a fuzz campaign. They evaluate six different algorithms and show that the choice of algorithm can greatly increase the number of discovered bugs. They also show that the current seed selection strategies in Peach [15] may fare no better than picking seeds at random.

Cha et al. [44] propose an algorithm to dynamically tune the mutation ratio in order to maximize the number of bugs found for black-box mutational fuzzing given a program and a seed input. They leverage symbolic analysis on the execution trace of a program-seed pair to detect dependencies among the bit positions of an input, and then use this dependency relation to compute a probabilistically optimal mutation ratio for this program-seed pair. The results showed an average of 38.6% more bugs than three previous fuzzers over eight programs within the same amount of fuzzing time.

AFLFast [45] boosts the AFL by several strategies to focus most of the fuzzing effort on low-frequency paths so as to explore more paths with the same amount of fuzzing time. During fuzzing, it chooses the seed i) that exercises lower frequency paths and ii) that have been chosen less often, which allows to fuzz the best seeds as early as possible.

Our seed generation approach transforms the generic seed selection problem for flat binary file formats into an automated generational approach, specifically for inputs with certain grammar structures and evaluated on a few programs that have undergone extensive fuzzing with traditional approaches; on the other hand, it is also orthogonal to these techniques and can be chained together as a first step. We plan to combine these boosting techniques with our approach to investigate whether the fuzzing efficiency can be further improved.

VII. Conclusions

In this paper, we have proposed a novel data-driven seed generation approach, named Skyfire, to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. We use the large corpus of samples and their grammar to automatically extract the semantic rules and the frequency of production rules, and holistically incorporate them by learning

a probabilistic context-sensitive grammar. The learned PCSG is then used to generate well-distributed seeds.

Using the seeds generated to fuzz several XSLT, XML, JavaScript and Rendering engines, we have empirically shown that Skyfire can generate well-distributed seeds and help to improve the code coverage and bug-finding capability of fuzzers. We discovered 19 new memory corruption bugs (among which we discovered 16 new vulnerabilities and received 33.5k USD bug bounty rewards) and 32 denial-of-service bugs.

In the future, we will continue applying and extending our seed generation approach to better support more different languages such as JavaScript, SQL, C, and Java. In addition to finding security bugs, we also hope to use the generated seed inputs to find compiler bugs.

ACKNOWLEDGMENTS

We would like to thank Michał Zalewski for the American Fuzzy Lop fuzzer.

This research is supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No.NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] Sdl process: Verification. [Online]. Available: https://www.microsoft.com/en-us/sdl/process/verification.aspx

[3] C. Evans, M. Moore, and T. Ormandy. (2011) Google online security blog – fuzzing at scale. [Online]. Available: https://security.googleblog.com/2011/08/fuzzing-at-scale.html

[4] B. Arkin. (2009) Adobe reader and acrobat security initiative. [Online]. Available: http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html

[5] C. Miller and Z. N. Peterson, "Analysis of mutation and generation-based fuzzing," Independent Security Evaluators, Baltimore, Maryland, Tech. Rep., 2007.

[6] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *USENIX Security*, 2014, pp. 861–875.

[7] American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[8] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE*, 2009, pp. 474–484.

[9] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *SP*, 2010, pp. 497–512.

[10] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA*, 2011, pp. 12–22.

[11] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[12] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[13] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *USENIX Security*, 2013, pp. 49–64.

[14] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The borg: Nanoprobing binaries for buffer overreads," in *CODASPY*, 2015, pp. 87–97.

[15] Peach fuzzer platform. [Online]. Available: http://www.peachfuzzer.com/products/peach-platform/

[16] Spike fuzzer platform. [Online]. Available: http://www.immunitysec.com/

[17] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *ASE*, 2016, pp. 543–553.

[18] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, 2008, pp. 206–215.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011, pp. 283–294.

[20] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security*, 2012, pp. 445–458.

[21] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *ESORICS*, 2016, pp. 581–601.

[22] mangleme. [Online]. Available: http://freecode.com/projects/mangleme/

[23] J. Ruderman. (2007) Introducing jsfunfuzz. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz

[24] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *ASE*, 2014, pp. 725–730.

[25] Xml grammar. [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/xml

[26] Antlr's grammar list for different languages. [Online]. Available: https://github.com/antlr/grammars-v4

[27] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[28] G. Alliance. (2006) Sablotron. [Online]. Available: http://freecode.com/projects/sablotron

[29] Gcov: Gnu coverage tool. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Gcov.html

[30] Pin - a dynamic binary instrumentation tool. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[31] G. Mohr, M. Stack, I. Rnitovic, D. Avery, and M. Kimpton, "An introduction to heritrix," in *4th International Web Archiving Workshop*, 2004.

[32] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[33] D. Veillard. (2003) Libxslt – the xslt c library for gnome. [Online]. Available: http://xmlsoft.org/libxslt/

[34] libxml2. [Online]. Available: http://www.xmlsoft.org/

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[36] Gflags and pageheap. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561(v=vs.85).aspx

[37] Google patch reward program rules. [Online]. Available: https://www.google.com.au/intl/iw/about/appsecurity/patch-rewards/index.html

[38] J. Watson, "Virtualbox: bits and bytes masquerading as machines," *Linux Journal*, vol. 2008, no. 166, 2008.

[39] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[40] U. Kargén and N. Shahmehri, "Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing," in *FSE*, 2015, pp. 782–792.

[41] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning, "Experiences with model inference assisted fuzzing." in *USENIX Security*, 2008.

[42] A. Householder and J. Foote, "Probability-based parameter selection for black-box fuzz testing," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2012-TN-019, 2012.

[43] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS*, 2013, pp. 511–522.

[44] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *SP*, 2015, pp. 725–741.

[45] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016, pp. 1032–1043.