Fuzzing with Code Fragments

Christian Holler
Mozilla Corporation[*]
choller@mozilla.com

Kim Herzig
Saarland University
herzig@cs.uni-saarland.de

Andreas Zeller
Saarland University
zeller@cs.uni-saarland.de

Abstract

Fuzz testing is an automated technique providing random data as input to a software system in the hope to expose a vulnerability. In order to be effective, the fuzzed input must be *common enough* to pass elementary consistency checks; a JavaScript interpreter, for instance, would only accept a semantically valid program. On the other hand, the fuzzed input must be *uncommon enough* to trigger exceptional behavior, such as a crash of the interpreter. The *LangFuzz* approach resolves this conflict by using a *grammar* to randomly generate valid programs; the code fragments, however, partially stem from *programs known to have caused invalid behavior before.* LangFuzz is an effective tool for security testing: Applied on the Mozilla JavaScript interpreter, it discovered a total of 105 new severe vulnerabilities within three months of operation (and thus became one of the top security bug bounty collectors within this period); applied on the PHP interpreter, it discovered 18 new defects causing crashes.

1 Introduction

Software security issues are risky and expensive. In 2008, the annual CSI Computer Crime & Security survey reported an average loss of 289,000 US$ for a single security incident. Security testing employs a mix of techniques to find vulnerabilities in software. One of these techniques is *fuzz testing*—a process that automatically generates random data input. Crashes or unexpected behavior point to potential software vulnerabilities.

In web browsers, the JavaScript interpreter is particularly prone to security issues; in Mozilla Firefox, for instance, it encompasses the majority of vulnerability fixes [13]. Hence, one could assume the JavaScript interpreter would make a rewarding target for fuzz testing. The problem, however, is that fuzzed input to a JavaScript interpreter must follow the syntactic rules of JavaScript. Otherwise, the JavaScript interpreter will reject the input as invalid, and effectively restrict the testing to its lexical and syntactic analysis, never reaching areas like code transformation, in-time compilation, or actual execution. To address this issue, fuzzing frameworks include strategies to model the structure of the desired input data; for fuzz testing a JavaScript interpreter, this would require a built-in JavaScript grammar.

Surprisingly, the number of fuzzing frameworks that generate test inputs on grammar basis is very limited [7, 17, 22]. For JavaScript, *jsfunfuzz* [17] is amongst the most popular fuzzing tools, having discovered more that 1,000 defects in the Mozilla JavaScript engine. jsfunfuzz is effective because it is hardcoded to target a specific interpreter making use of specific knowledge about past and common vulnerabilities. The question is: Can we devise a *generic* fuzz testing approach that nonetheless can exploit *project-specific* knowledge?

In this paper, we introduce a framework called *LangFuzz* that allows black-box fuzz testing of engines based on a context-free grammar. LangFuzz is not bound against a specific test target in the sense that it takes the *grammar* as its input: given a JavaScript grammar, it will generate JavaScript programs; given a PHP grammar, it will generate PHP programs. To adapt to *specific* targets, LangFuzz can use its grammar to learn *code fragments* from a given *code base*. Given a suite of previously failing programs, for instance, LangFuzz will use and recombine fragments of the provided test suite to generate new programs—assuming that a recombination of previously problematic inputs has a higher chance to cause new problems than random input.

The combination of fuzz testing based on a language grammar and reusing project-specific issue-related code fragments makes LangFuzz an effective tool for security testing. Applied on the Mozilla JavaScript engine, it discovered a total of 105 new vulnerabilities within three months of operation. These bugs are serious and

[*]At the time of this study, Christan Holler was writing his master thesis at Saarland University. He is now employed at Mozilla.

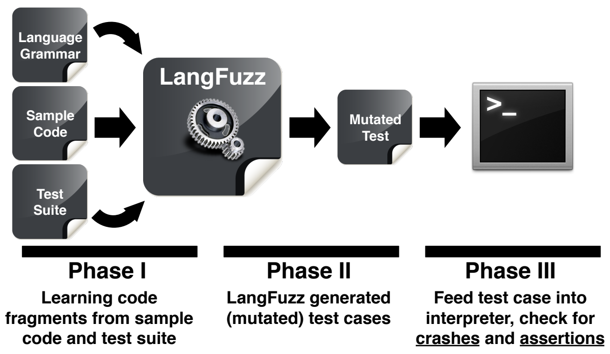
Language Grammar
Sample Code
Test Suite
LangFuzz
Mutated Test
>_
Phase I
Learning code fragments from sample code and test suite
Phase II
LangFuzz generated (mutated) test cases
Phase III
Feed test case into interpreter, check for crashes and assertions

Figure 1: LangFuzz workflow. Using a language grammar, LangFuzz parses code fragments from sample code and test cases from a test suite, and mutates the test cases to incorporate these fragments. The resulting code is then passed to the interpreter for execution.

valuable, as expressed by the 50.000$ bug bounties they raised. Nearly all the detected bugs are memory safety issues. At the same time, the approach can generically handle arbitrary grammars, as long as they are weakly typed: applied on the PHP interpreter, it discovered 18 new defects. All generated inputs are semantically correct and can be executed by the respective interpreters.

Figure 1 describes the structure of LangFuzz. The framework requires three basic input sources: a *language grammar* to be able to parse and generate code artifacts, *sample code* used to learn language fragments, and a *test suite* used for code mutation. Many test cases contain code fragments that triggered past bugs. The test suite can be used as sample code as well as mutation basis. LangFuzz then generates new test cases using code mutation and code generation strategies before passing the generated test cases to a test driver executing the test case—e.g. passing the generated code to an interpreter.

As an example of a generated test case exposing a security violation, consider Figure 2 that shows a security issue in Mozzila's JavaScript engine. RegExp.$1 (Line 8) is a pointer to the first grouped regular expression match. This memory area can be altered by setting a new input (Line 7). An attacker could use the pointer to arbitrarily access memory contents. In this test case, Lines 7 and 8 are newly generated by LangFuzz, whereas Lines 1–6 stem from an existing test case.

The remainder of this paper is organized as follows. Section 2 discusses the state of the art in fuzz testing and provides fundamental definitions. Section 3 presents how LangFuzz works, from code generation to actual test execution; Section 4 details the actual implementation. Section 5 discusses our evaluation setup, where we compare LangFuzz against jsfunfuzz and show that LangFuzz detects several issues which jsfunfuzz misses. Section 6 describes the application of LangFuzz on PHP.

```
1 var haystack = "foo";
2 var re_text = "^foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);
```

Figure 2: Test case generated by LangFuzz, crashing the JavaScript interpreter when executing Line 8. The static access of RegExp is deprecated but valid. Reported as Mozilla bug 610223 [1].

Section 7 discusses threats to validity, and Section 8 closes with conclusion and future work.

2 Background

2.1 Previous Work

"Fuzz testing" was introduced in 1972 by Purdom [16]. It is one of the first attempts to automatically test a parser using the grammar it is based on. We especially adapted Purdom's idea of the "Shortest Terminal String Algorithm" for LangFuzz. In 1990, Miller et al. [10] were among the first to apply fuzz testing to real world applications. In their study, the authors used random generated program inputs to test various UNIX utilities. Since then, the technique of fuzz testing has been used in many different areas such as protocol testing [6, 18], file format testing [19, 20], or mutation of valid input [14, 20].

Most relevant for this paper are earlier studies on grammar-based fuzz testing and test generations for compiler and interpreters. In 2005, Lindig [8] generated code to specifically stress the C calling convention and check the results later. In his work, the generator also uses recursion on a small grammar combined with a fixed test generation scheme. Molnar et al. [12] presented a tool called *SmartFuzz* which uses symbolic execution to trigger integer related problems (overflows, wrong conversion, signedness problems, etc.) in x86 binaries. In 2011, Yang et al. [22] presented *CSmith*—a language-specific fuzzer operating on the C programming language grammar. *CSmith* is a pure generator-based fuzzer generating C programs for testing compilers and is based on earlier work of the same authors and on the random C program generator published by Turner [21]. In contrast to LangFuzz, *CSmith* aims to target correctness bugs instead of security bugs. Similar to our work, *CSmith* randomly uses productions from its built-in C grammar to create a program. In contrast to LangFuzz, their grammar has non-uniform probability annotations. Furthermore, they already introduce semantic rules during their generation process by using *filter functions*, which allow or disallow certain productions depending on the context. This is reasonable when constructing a fuzzer for a specific language, but very difficult for a language independent approach as we are aiming for.

Fuzzing web browsers and their components is a promising field. The most related work in this field is the work by Ruderman and his tool *jsfunfuzz* [17]. *Jsfunfuzz* is a black-box fuzzing tool for the JavaScript engine that had a large impact when written in 2007. *Jsfunfuzz* not only searches for crashes but can also detect certain correctness errors by differential testing. Since the tool was released, it has found over 1,000 defects in the Mozilla JavaScript Engine and was quickly adopted by browser developers. jsfunfuzz was the first JavaScript fuzzer that was publicly available (it has since been withdrawn) and thus inspired LangFuzz. In contrast, LangFuzz does not specifically aim at a single language, although this paper uses JavaScript for evaluation and experiments. Instead, our approaches aim to be solely based on grammar and general language assumptions and to combine random input generation with code mutation.

Miller and Peterson [11] evaluated these two approaches—random test generation and modifying existing valid inputs—on PNG image formats showing that mutation testing alone can miss a large amount of code due to missing variety in the original inputs. Still, we believe that mutating code snippets is an important step that adds regression detection capabilities. Code that has been shown to detect defects helps to detect incomplete fixes when changing their context or fragments, especially when combined with a generative approach.

LangFuzz is a pure black-box approach, requiring no source code or other knowledge of the tested interpreter. As shown by Godefroid et al. [7] in 2008, a grammar-based fuzzing framework that produces JavaScript engine input (Internet Explorer 7) can increase coverage when linked to a constraint solver and coverage measurement tools. While we consider coverage to be an insufficient indicator for test quality in interpreters (just-in-time compilation and the execution itself heavily depend on the global engine state), such an extension may also prove valuable for LangFuzz.

In 2011, Zalewski [23] presented the *crossfuzz* tool that is specialized in DOM fuzzing and revealed some problems in many popular browsers. The same author has published even more fuzzers for specific purposes like ref fuzz, mangleme, Canvas fuzzer or transfuzz. They all target different functionality in browsers and have found severe vulnerabilities.

2.2 Definitions

Throughout this paper, we will make use of the following terminology and assumptions.

Defect. Within this paper, the term "defect" refers to errors in code that cause abnormal termination only (e.g. crash due to memory violation or an assertion violation). All other software defects (e.g. defect that produce false output without abnormal termination) will be disregarded, although such defects might be detected under certain circumstances. We think that this limitation is reasonable due to the fact that detecting other types of defects using fuzz-testing generally requires strong assumptions about the target software under test.

Grammar. In this paper, the term "grammar" refers to context-free grammars (Type-2 in the Chomsky hierarchy) unless stated otherwise.

Interpreter. An "interpreter" in the sense of this paper is any software system that receives a program in source code form and then executes it. This also includes just-in-time compilers which translate the source to byte code before or during runtime of the program. The main motivation to use grammar-based fuzz testing is the fact that such interpreter systems consist of lexer and parser stages that detect malformed input which causes the system to reject the input without even executing it.

3 How LangFuzz works

In fuzz testing, we can roughly distinguish between two techniques: *Generative* approaches try to create new random input, possibly using certain constraints or rules. *Mutative* approaches try to derive new testing inputs from existing data by randomly modifying it. For example, both jsfunfuzz [17] and *CSmith* [22] use generative approaches. LangFuzz makes use of both approaches, but mutation is the primary technique. A purely generative design would likely fail due to certain semantic rules not being respected (e.g. a variable must be defined before it is used). Introducing semantic rules to solve such problems would tie LangFuzz to certain language semantics. Mutation, however, allows us to learn and reuse existing semantic context. This way, LangFuzz stays language-independent without losing the ability to generate powerful semantic context to embed generated or mutated code fragments.

3.1 Code mutation

The mutation process consists of two phases, a *learning phase* in the beginning and the main *mutation phase.*

In the learning phase, we process a certain set of sample input files using a parser for the given language (derived from the language grammar). The parser will allow us to separate the input file into *code fragments* which are essentially examples for non-terminals in the grammar. Of course, these fragments may overlap (e.g. an `expression` might be contained in an `ifStatement` which is a `statement` according to the grammar). Given a large codebase, we can build up a `fragment pool` consisting of expansions for all kinds of non-terminal symbols. Once we have learned all of our input, the mutation phase starts. For mutation, a single target file is processed again using the parser. This time, we randomly pick some of the fragments we saw during parsing and replace them with other fragments of the same type. These code fragments might of course be semantically invalid or less useful without the context that surrounded them originally, but we accept this trade-off for being independent of the language semantics. In Section 3.3, we discuss one important semantic improvement performed during fragment replacement.

As our primary target is to trigger defects in the target program, it is reasonable to assume that existing test cases (especially regressions) written in the target language should be helpful for this purpose; building and maintaining such test suites is standard practice for developers of interpreters and compilers. Using the mutation process described in the previous section, we can process the whole test suite file by file, first learning fragments from it and then creating executable mutants based on the original tests.

3.2 Code generation

With our mutation approach, we can only use those code fragments as replacements that we have learned from our code base before. Intuitively, it would also be useful if we could generate fragments on our own, possibly yielding constructs that cannot or can only hardly be produced using the pure mutation approach.

Using a language grammar, it is natural to generate fragments by *random walk* over the tree of possible expansion series. But performing a random walk with uniform probabilities is not guaranteed to terminate. However, terminating the walk without completing all expansions might result in a syntactically invalid input.

Usually, this problem can be mitigated by restructuring the grammar, adding non-uniform probabilities to the edges and/or imposing additional semantic restrictions during the production, as in the *CSmith* work [22].

Restructuring or annotating the grammar with probabilities is not straightforward and requires additional work for every single language. It is even reasonable to assume that using fixed probabilities can only yield a coarse approximation as the real probabilities are conditional, depending on the surrounding context. To overcome these problems, we will use an algorithm that performs the generation in a *breadth-first* manner:

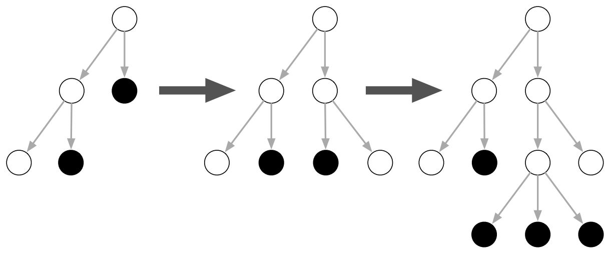
Figure 3: Example of a stepwise expansion on the syntax tree: Dark nodes are unexpanded non-terminals (can be expanded) while the other nodes have already been expanded before.

```
1. Set current expansion $e_{cur}$ to the start symbol $S$
2. Loop $num$ iterations:
    (a) Choose a random non-terminal $n$ in $e_{cur}$:
        i. Find the set of productions $P_n \subseteq P$ that
           can be applied to $n$.
        ii. Pick one production $p$ from $P_n$ randomly
            and apply it to $n$, yielding $p(n)$.
        iii. Replace that occurrence of $n$ in $e_{cur}$ by
             $p(n)$.
```

Figure 3 gives an example of such a stepwise expansion, considering the code as a syntax tree. Dark nodes are unexpanded non-terminals that can be considered for expansion while the remaining nodes have already been expanded before. This algorithm does not yield a valid expansion after *num* iterations. We need to replace the remaining non-terminal symbols by sequences of terminal symbols. In the learning phase of the mutation approach we are equipped with many different examples for different types of non-terminals. We randomly select any of these code fragments to replace our remaining non-terminals. In the unlikely situation that there is no example available, we can use the *minimal expansion* of the non-terminal instead. During mutation, we can use learned and generated code fragments.

3.3 Adjusting Fragments to Environment

When a fragment is replaced by a different fragment, the new fragment might not fit with respect to the semantics of the remaining program. As LangFuzz does not aim to semantically understand a specific language, we can only perform corrections based on *generic* semantic assumptions. One example with a large impact are *identifiers*.

Many programming languages use identifiers to refer to variables and functions, and some of them will throw an error if an identifier has not been declared prior to using it (e.g. in JavaScript, using an identifier that is never declared is considered to be a runtime error).

We can reduce the chances to have undeclared identifiers within the new fragment by replacing all identifiers in the fragment with identifiers that occur somewhere in the rest of the program. Note that this can be done purely at the syntactic level. LangFuzz only needs to know which non-terminal in the grammar constitutes an identifier in order to be able to statically extract known identifiers from the program and replace identifiers in the new fragment. Thus, it is still possible that identifiers are unknown at the time of executing a certain statement (e.g. because the identifier is declared afterwards), but the chances of *identifier reuse* are increased.

Some languages contain identifiers that can be used without declaring them (usually *built-in objects/globals*). The adjustment approach can be even more effective if LangFuzz is aware of these global objects in order to ignore them during the replacement process. The only way to identify such global objects within LangFuzz is to require a list of these objects as (optional) argument. Such global object lists are usually found in the specification of the respective language.

4 The LangFuzz Implementation

Based on the methods described so far, we now assemble the different parts to get a proof-of-concept fuzzer implementation that works as described in the overview diagram (Figure 1) in the introduction.

Typically, LangFuzz starts with a *learning phase* where the given sample code is parsed using the supplied language grammar, thereby learning code fragments (Section 4.1). The input of this learning phase can be either a sample code base or the test suite itself. Once the learning step is complete, LangFuzz starts to process the test suite. All tests are parsed and the results are cached for performance reasons.

Then the tool starts the actual working phase:

1. From the next test to be mutated, several fragments (determined by an adjustable parameter, typically 1–3) are randomly selected for replacement.

2. As a single fragment can be considered as multiple types (e.g. `if (true) {...}` can be seen as an if-statement but also more generally as a statement), we randomly pick one of the possible interpretations for each of those fragments.

3. Finally, the mutated test is executed and its result is checked (Section 4.3).

4.1 Code Parsing

In the learning and mutation phase, we parse the given source code. For this purpose, LangFuzz contains a parser subsystem such that concrete parsers for different languages can be added. We decided to use the ANTLR parser generator framework [15] because it is widespread and several grammars for different languages exist in the community. The parser is first used to learn fragments from the given code base which LangFuzz then memorizes as a token stream. When producing a mutated test, the cached token stream is used to find all fragments in the test that could be replaced and to determine which code can be replaced according to the syntax—we can mutate directly on the cached token stream.

4.2 Code Generation

The code generation step uses the stepwise expansion (Section 3.2) algorithm to generate a code fragment. As this algorithm works on the language grammar, LangFuzz also includes an ANTLR parser for ANTLR grammars. However, because LangFuzz is a proof-of-concept, this subsystem only understands a subset of the ANTLR grammar syntax and certain features that are only required for parsing (e.g. implications) are not supported. It is therefore necessary to simplify the language grammar slightly before feeding it into LangFuzz. LangFuzz uses further simplifications internally to make the algorithm easier: Rules containing quantifiers ('*', '+') and optionals ('?') are de-sugared to remove these operators by introducing additional rules according to the following patterns:

$$\begin{aligned} X* &\rightsquigarrow (R \to \varepsilon \,|\, XR) && \text{(zero or more)} \\ X+ &\rightsquigarrow (R \to X \,|\, XR) && \text{(one or more)} \\ X? &\rightsquigarrow (R \to \varepsilon \,|\, X) && \text{(zero or one)} \end{aligned}$$

where X can be any complex expression. Furthermore, sub-alternatives (e.g. $R \to ((A|B)C|D)$), are split up into separate rules as well. With these simplifications done, the grammar only consists of rules for which each alternative is only a sequence of terminals and non-terminals. While we can now skip special handling of quantifiers and nested alternatives, these simplifications also introduce a new problem: The additional rules (*synthesized rules*) created for these simplifications have no counterpart in the parser grammar and hence there are no code examples available for them. In case our stepwise expansion contains one or more synthesized rules, we replace those by their minimal expansion as described in Section 3.2. All other remaining non-terminals are replaced by learned code fragments as described earlier. In our implementation, we introduced a size limitation on these fragments to avoid placing huge code fragments into small generated code fragments.

After code generation, the fragment replacement code adjusts the new fragment to fit its new environment as described in Section 3.3. For this purpose, LangFuzz searches the remaining test for available identifiers and maps the identifiers in the new fragment to existing ones. The mapping is done based on the identifier name, not its occurrence, i.e. when identifier "a" is mapped to "b", all occurrences of "a" are replaced by "b". Identifiers that are on the built-in identifier list (e.g. global objects) are not replaced. LangFuzz can also actively map an identifier to a built-in identifier with a certain probability.

4.3 Running Tests

In order to be able to run a mutated test, LangFuzz must be able to run the test with its proper *test harness* which contains definitions required for the test. A good example is the Mozilla test suite: The top level directory contains a file *shell.js* with definitions required for all tests. Every subdirectory may contain an additional shell.js with further definitions that might only be required for the tests in that directory. To run a test, the JavaScript engine must execute all shell files in the correct order, followed by the test itself. LangFuzz implements this logic in a test suite class which can be derived and adjusted easily for different test frameworks.

The simplest method to run a mutated test is to start the JavaScript engine binary with the appropriate test harness files and the mutated test. But starting the JavaScript engine is slow and starting it over and over again would cost enormous computation time. To solve this problem, LangFuzz uses a *persistent* shell: A small JavaScript program called the *driver* is started together with the test harness. This way, we reduce the number of required JavaScript engines to be started drastically. The driver runs a set of tests within one single JavaScript engine and signals completion when done. LangFuzz monitors each persistent shell and records all input to it for later reproduction. Of course the shell may not only be terminated because of a crash, but also because of timeouts or after a certain number of tests being run. The test driver is language dependent and needs to be adapted for other languages (see Section 6); such a test driver would also be required if one implemented a new fuzzer from scratch.

Although the original motivation to use persistent shells was to increase test throughput it has an important side-effect. It increased the number of defects detected. Running multiple tests within a single shell allows individual tests to influence each other. Different tests may use the same variables or functions and cause crashes that would not occur when running the individual tests alone. In fact, most of the defects found in our experiments required multiple tests to be executed in a row to be triggered. This is especially the case for memory corruptions (e.g. garbage collector problems) that require longer runs and a more complex setup than a single test could provide.

Running multiple tests in one shell has the side effect that it increases the number of source code lines executed within each JavaScript shell. To determine which individual tests are relevant for failure reproduction we use the *delta debugging algorithm* [24] and the *delta* tool [9] to filter out irrelevant test cases. The very same algorithm also reduces the remaining number of executed source code lines. The result is a suitably small test case.

4.4 Parameters

LangFuzz contains a large amount of adjustable parameters, e.g. probabilities and amounts that drive decisions during the fuzzing process. In Table 3 (see Appendix) we provide the most common/important parameters and their default values. Please note that all default values are chosen empirically. Because the evaluation of a certain parameter set is very time consuming (1–3 days per set and repeating each set hundreds of time times to eliminate the variance introduced by random generation), it was not feasible to compare all possible parameter combinations and how they influence the results. We tried to use reasonable values but cannot guarantee that these values deliver the best performance.

5 Evaluation

To evaluate how well LangFuzz discovers undetected errors in the JavaScript engines, we setup three different experimental setups. The external validation compares LangFuzz to the state of the art in JavaScript fuzzing. The internal validation compares the two fragment replacement strategies used within LangFuzz: random code generation and code mutation. Finally, we conducted a field study to check whether LangFuzz is actually up to the task to detect real defects in current state of the art JavaScript engines.

5.1 LangFuzz vs. jsfunfuzz

The state of the art fuzzer for JavaScript is the jsfunfuzz tool written by Ruderman [17]. The tool is widely used and has proven to be very successful in discovering defect within various JavaScript engines. jsfunfuzz is an active part of Mozilla's and Google's quality assurance and regularly used in their development.

The differences between jsfunfuzz and LangFuzz are significant and allow only unfair comparisons between both tools. jsfunfuzz is highly adapted to test JavaScript engines and contains multiple optimizations. jsfunfuzz is designed to test new and previously untested JavaScript features intensively. This of course required detailed knowledge of the software project under test. Additionally, jsfunfuzz has a certain level of semantic knowledge and should be able to construct valid programs easier. However, for every new language feature, the program has to be adapted to incorporate these changes into the testing process. Also, focusing on certain semantics can exclude certain defects from being revealed at all.

In contrast, LangFuzz bases its testing strategy solely on the grammar, existing programs (e.g. test suites) and a very low amount of additional language-dependent information. In practice, this means that

- changes to the language under test do not require any program maintenance apart from possible grammar updates; and
- through the choice of test cases, LangFuzz can be set up to cover a certain application domain.

The use of existing programs like previous regression tests allows LangFuzz to profit from previously detected defects. However, LangFuzz lacks a semantic background on the language which lowers the chances to obtain sane programs and produce test cases that trigger a high amount of interaction between individual parts of the program.

Although both tools have some differences that make a fair comparison difficult, comparing both tools can unveil two very interesting questions:

Q1. *To what extend do defects detected by LangFuzz and jsfunfuzz overlap?*

By overlap, we refer to the number of defects that both tools are able to detect. A low overlap would indicate that LangFuzz is able to detect new defects that were not found and most likely will not be found by jsfunfuzz. Therefore we define the overlap as the fraction of number of defects found by both tools and the number of defects found in total. This gives us a value between zero and one. A value of one would indicate that both tools detected exactly the same defects. If both tools detected totally different defects, the overlap would be zero.

$$overlap = \frac{\text{number of defects found by both tools}}{\text{number of defects found in total}}$$

The second question to be answered by this comparison is targeted towards the effectiveness of LangFuzz.

Q2. *How does LangFuzz's detection rate compare to jsfunfuzz?*

By effectiveness, we mean how many defects each tool is able to locate in a given period of time. Even though the overlap might be large, it might be the case that either tool might detect certain defects much quicker or slower than the respective other tool. To compare the effectiveness of LangFuzz in comparison against jsfunfuzz, we define the effectiveness as:

$$effectiveness = \frac{\text{number of defects found by LangFuzz}}{\text{number of defects found by jsfunfuzz}}.$$

This sets the number of defects found by LangFuzz into relation to the number of defects found by jsfunfuzz. Since both tools ran on the same time windows, the same amount of time using identical amounts of resources (e.g. CPU and RAM) we do not have to further normalize this value.

Overall, this comparison answers the question whether LangFuzz is a useful contribution to a quality assurance process, even if a fuzzer such as jsfunfuzz is already used. It is not our intention to show that either tool outperforms the other tool by any means. We believe that such comparisons are non-beneficial since both jsfunfuzz and LangFuzz operate on different assumptions and levels.

5.1.1 Testing windows

We compared jsfunfuzz and LangFuzz using Mozilla's JavaScript engine *TraceMonkey*. There were two main reasons why we decided to choose TraceMonkey as comparison base. First, Mozilla's development process and related artifacts are publicly available—data that required internal permission was kindly provided by the Mozilla development team. The second main reason was that jsfunfuzz is used in Mozilla's daily quality assurance process which ensures that jsfunfuzz is fully functional on TraceMonkey without investing any effort to setup the external tool. But using TraceMonkey as reference JavaScript engine also comes with a downside. Since jsfunfuzz is used daily within Mozilla, jsfunfuzz had already run on every revision of the engine. This fact has two major consequences: First, jsfunfuzz would most likely not find any new defects; and second, the number of potential defects that can be found by LangFuzz is significantly reduced. Consequently, it is not possible to measure effectiveness based on a single revision of TraceMonkey. Instead we make use of the fact that Mozilla maintains a list of defects found by jsfunfuzz. Using this list, we used a test strategy which is based on *time windows* in which no defect fixes were applied that are based on defect reports filed by jsfunfuzz (see Figure 4). Within these periods, both tools will have equal chances to find defects within the TraceMoney engine.

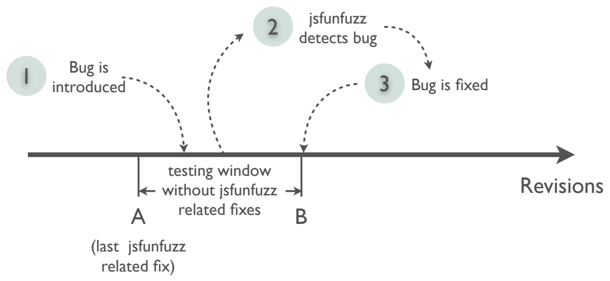
1
Bug is introduced
2
jsfunfuzz detects bug
3
Bug is fixed
testing window without jsfunfuzz related fixes
Revisions
A
(last jsfunfuzz related fix)
B

Figure 4: Example of a testing window with the live cycle of a single defect.

	start revision	end revision
W_1	46569:03f3c7efaa5e	47557:3b1c3f0e98d8
W_2	47557:3b1c3f0e98d8	48065:7ff4f93bddaa
W_3	48065:7ff4f93bddaa	48350:d7c7ba27b84e
W_4	48350:d7c7ba27b84e	49731:aaa87f0f1afe
W_5	49731:aaa87f0f1afe	51607:f3e58c264932

Table 1: The five testing windows used for the experiments. Each window is given by Mercurial revisions of the Mozilla version archive. All windows together cover approximately two months of development activity.

Within the remainder of this paper, we call these periods *testing windows*.

In detail, we applied the following test strategy for both tools:

1. Start at some base revision f_0. Run both tools for a fixed amount of time. Defects detected can solely be used to analyze the overlap, not effectiveness.
2. Set $n = 1$ and repeat several times:
   (a) Find the next revision f_n starting at f_{n-1} that fixes a defect found in the list of jsfunfuzz defects.
   (b) Run both tools on $f_n - 1$ for a fixed amount of time. The defects found by both tools can be used for effectiveness measurement if and only if the defect was introduced between f_{n-1} and $f_n - 1$ (the preceding revision of f_n)[1]. For overlap measurement, all defects can be used.

Figure 4 illustrates how such a testing window could look like. The window starts at revision A. At some point, a bug is introduced and shortly afterwards, the bug gets reported by jsfunfuzz. Finally, the bug is fixed in revision $B + 1$. At this point, our testing window ends and we can use revision B for experiments and count all defects that where introduced between A and B which is the testing window.

For all tests, we used the TraceMonkey development repository. Both the tested implementation and the test cases (approximately 3,000 tests) are taken from the development repository. As base revision, we chose revision *46549* (*03f3c7efaa5e*) which is the first revision committed in July 2010, right after Mozilla Firefox 4 Beta 1 was released at June 30, 2010. Table 1 shows the five test windows used for our experiments. The end revision of the last testing window dates to the end of August 2010, implying that we covered almost two months of development activity using these five windows. For each testing window, we ran both tools for 24 hours.[2]

To check whether a defect detected by either tool was introduced in the current testing window, we have to detect the lifetime of the issue. Usually, this can be achieved by using the *bisect* command provided by the Mercurial SCM. This command allows automated testing through the revision history to find the revision that introduced or fixed a certain defect. Additionally, we tried to identify the corresponding issue report to check whether jsfunfuzz found the defect in daily quality assurance routine.

5.1.2 Result of the external comparison

During the experiment, jsfunfuzz identified 23 defects, 15 of which lay within the respective testing windows. In contrast, LangFuzz found a total of 26 defects with only 8 defects in the respective testing windows. The larger proportion of defects outside the testing windows for LangFuzz is not surprising since LangFuzz, unlike jsfunfuzz, was never used on the source base before this experiment. Figure 5 illustrates the number of defects per fuzzer within the testing windows.

To address research question *Q1*, we identified three defects found by both fuzzers. Using the definition from Section 5.1 the overlap between LangFuzz and jsfunfuzz is 15%. While a high overlap value would indicate that both fuzzers could be replaced by each other, an overlap value of 15% is a strong indication that both fuzzers find different defects and hence supplement each other.

LangFuzz and jsfunfuzz detect different defects (overlap of 15%) and thus should be used complementary to each other.

To answer research question *Q2*, we computed the effectiveness of LangFuzz for the defects found within the experiment.

[1] $f_n - 1$ is exactly one revision before f_n and spans the testing window. The testing window starts at f_{n-1} and ends at $f_n - 1$ because f_n is a jsfunfuzz-induced fix.

[2] For both tools we used the very same hardware. Each tool ran on 4 CPUs with the same specification. Since jsfunfuzz does not support threading, multiple instances will be used instead. LangFuzz's parameters were set to default values (see Table 3).

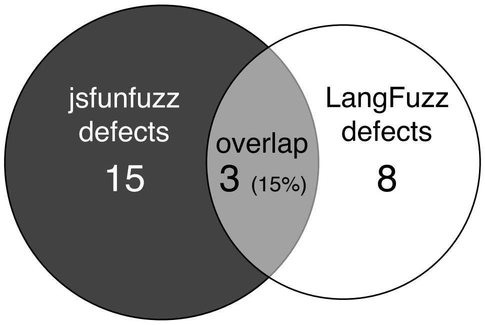

Figure 5: Number of defects found by each fuzzer within the testing windows and their overlap.

Compared to the 15 defects that were exclusively detected by jsfunfuzz LangFuzz with it's eight exclusively detected defect has an effectiveness of $15 : 8 = 53\%$. In other words, LangFuzz is half as effective as jsfunfuzz.

A generic grammar-based fuzzer like LangFuzz can be 53% as effective as a language-specific fuzzer like jsfunfuzz.

For us, it was not surprising that a tried-and-proven language-specific fuzzer is more effective than our more general approach. However, effectiveness does not imply capability. The several severe issues newly discovered by LangFuzz show that the tool is capable of finding bugs not detected by jsfunfuzz.

5.1.3 Example for a defect missed by jsfunfuzz

For several defects (especially garbage collector related) we believe that jsfunfuzz was not able to trigger them due to their high complexity even after minimization. Figure 6 shows an example of code that triggered an assertion jsfunfuzz was not able to trigger. In the original bug report, Jesse Ruderman confirmed that he tweaked jsfunfuzz in response to this report: "After seeing this bug report, I tweaked jsfunfuzz to be able to trigger it." After adaptation, jsfunfuzz eventually produced an even shorter code fragment triggering the assertion (we used the tweaked version in our experiments).

5.2 Generation vs. Mutation

The last experiment compares LangFuzz with the state of the art JavaScript engine fuzzer jsfunfuzz. The aim of this experiment is to compare the internal fragment replacement approaches of LangFuzz: code generation against code mutation. The first option learned code fragments to replace code fragments while the second option uses code generation (see Section 4.2) for replacement instead.

```
options('tracejit');
for (var j = 0; uneval({'-1':true}); ++j) {
        (−0).toString();
}
```

Figure 6: Test case generated by LangFuzz causing the TraceMonkey JavaScript interpreter to violate an internal assertion when executed. Reported as Mozilla bug 626345 [2].

This experiment should clarify whether only one of the approaches accounts for most of the results (and the other only slightly improves it or is even dispensable) or if both approaches must be combined to achieve good results.

Q3. *How important is it that LangFuzz generates new code?*

Q4. *How important is it that LangFuzz uses learned code when replacing code fragments?*

To measure the influence of either approach, we require two independent runs of LangFuzz with different configurations but using equal limitation on resources and runtime. The first configuration forced LangFuzz to use only learned code snippets for fragment replacement (mutation configuration). The second configuration allowed code fragmentation by code generation only (generation configuration).

Intuitively, the second configuration should perform random code generation without any code mutation at all—also not using parsed test cases as fragmentation replacement basis. Such a setup would mean to fall back to a purely generative approach eliminating the basic idea behind LangFuzz. It would also lead to incomparable results. The length of purely generated programs is usually small. The larger the generated code the higher the chance to introduce errors leaving most of the generated code meaningless. Also, when mutating code, we can adjust the newly introduced fragment to the environment (see Section 3.3). Using purely generated code instead, this is not possible since there exists no consistent environment around the location where a fragment is inserted (in the syntax tree at the end of generation). Although it would be possible to track the use of identifiers during generation the result would most likely not be comparable to results derived using code mutation.

Since we compared different LangFuzz configurations only, there is no need to use the testing windows from the previous experiment described in Section 5.1. Instead, we used the two testing windows that showed most defect detection potential when comparing LangFuzz with jsfunfuzz (see Section 5.1): W_1 and W_5. Both windows

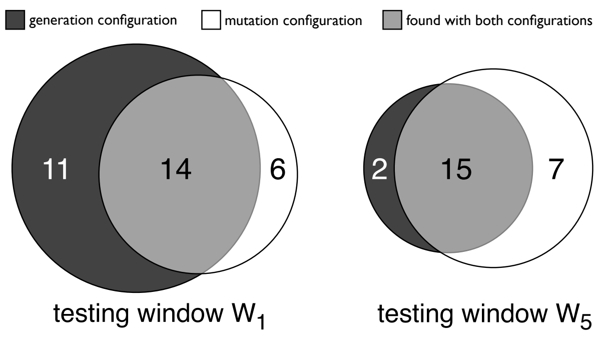

Figure 7: Defects found with/without code generation.

showed a high defect rate in the experimental setup described in Section 5.1 and spanned over 5,000 revisions giving each configuration enough defect detection potential to get comparable results.

Limiting the number of testing windows to compare the two different configurations of LangFuzz, we were able to increase the runtime for each configuration, thus minimizing the randomization impact on the experiment at the same time. Both configurations ran on both testing windows for 72 hours (complete experiment time was 12 days). For all runs we used the default parameters (see Table 3), except for the `synth.prob` parameter. This parameter can be used to force LangFuzz to use code generation only (set to 1.0) and to ignore code generation completely (set to 0.0).

Since the internal process of LangFuzz is driven by randomization, we have to keep in mind that both runs are independent and thus produce results that are hard to compare. Thus, we should use these results as indications only.

5.2.1 Result of the internal comparison

Figure 7 shows the results of the internal comparison experiment as overlapping circles. The left overlapping circle pair shows the result of the comparison using test window W_1, the right pair the results using test window W_5. The dark circles represent the runs using the generation configuration while the white circles represent runs using the mutation configuration. The overlap of the dark and white circles contains those defects that can be detected using both fragment replacement strategies. The numbers left and right of the overlap show the number of defects found exclusively by the corresponding configuration.

For W_1 a total of 31 defect were found. The majority of 14 defects is detected by both configurations. But 11 defects were only found when using code generation whereas six defects could only be detected using the mutation replacement strategy. Interestingly, this proportion is reversed in test window W_5. Here a total of 24 defects and again the majority of 15 defects were found using both configurations. But in W_5 the number of defects found by mutation configuration exceeds the number of defects found by code generation. Combining the number of defects found by either configurations exclusively, code generation detected 13 defects that were not detected by the mutation configuration. Vice versa, code mutation detected 9 defects that were not detected during the code generation configuration run. Although the majority of 29 defects found by both configurations, these numbers and proportions show that both of LangFuzz internal fragmentation replacement approaches are crucial for LangFuzz success and should be combined. Thus, an ideal approach should be a mixed setting where both code generation and direct fragment replacement is done, both with a certain probability.

The combination of code mutation and code generation detects defects not detected by either internal approach alone. Combining both approaches makes LangFuzz successful.

```
( 'false'? length(input + ''): delete(null?0:{}),0 ).
    watch('x', function f() { });
```

Figure 8: Test case generated by LangFuzz using code generation. The code cause the TraceMonkey JavaScript interpreter to write to a null pointer. Reported as Mozilla bug 626436 [3].

5.2.2 Example of defect detected by code generation

The code shown in Figure 8 triggered an error in the parser subsystem of Mozilla TraceMonkey. This test was partly produced by code generation. The complex and unusual syntactic nesting here is unlikely to happen by only mutating regular code.

5.2.3 Example for detected incomplete fix

The bug example shown in Figure 9 caused an assertion violation in the V8 project and is a good example for both an incomplete fix detected by LangFuzz and the benefits of mutating existing regression tests: Initially, the bug had been reported and fixed as usual. Fixes had been merged into other branches and of course a new regression test based on the LangFuzz test has been added to the repository. Shortly after, LangFuzz triggered exactly the same assertion again using the newly added regression test in a mutated form. V8 developers confirmed that the initial fix was incomplete and issued another fix.

```
var loop_count = 5
function innerArrayLiteral(n) {
  var a = new Array(n);
  for (var i = 0; i < n; i++) {
    a[i] = void ! delete 'object'%
˜ delete 4
  }
}
function testConstructOfSizeSize(n) {
  var str = innerArrayLiteral(n);
}
for (var i = 0; i < loop_count; i++) {
  for (var j = 1000; j < 12000; j += 1000) {
    testConstructOfSizeSize(j);
  }
}
```

Figure 9: Test case generated by LangFuzz discovering an incomplete fix triggering an assertion failure in the Google V8 JavaScript engine. Reported as Google V8 bug 1167 [4].

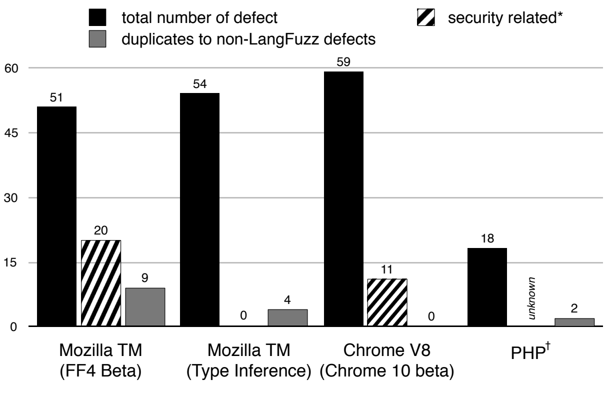

Figure 10: Real defects found on Mozilla, Google V8, and PHP. These defects were reported as customer defects and their numbers are not comparable to the defect numbers in earlier figures.*The security lock might hide the issue report from public because it might be exploitable. †Defects reported for PHP were not classified security relevant.

5.3 Field tests

The first two experiments are targeting the external and internal comparison of LangFuzz. But so far, we did not check whether LangFuzz is actually capable of finding real world defects within software projects used within industry products. To address this issue, we conducted an experiment applying LangFuzz to three different language interpreter engines: Mozilla TraceMonkey (JavaScript), Google V8 (JavaScript), and the PHP engine. In all experiments, we used the default configuration of LangFuzz including code generation. An issue was marked as security relevant if the corresponding development team marked the issue accordingly as security issue. Thus the security relevance classification was external and done by experts.

Mozilla TraceMonkey We tested LangFuzz on the trunk branch versions of Mozilla's TraceMonkey JavaScript engine that were part of the Firefox 4 release. At the time we ran this experiment, Firefox 4 and its corresponding TraceMonkey version were pre-release (beta version). Changes to the TraceMonkey trunk branch were regularly merged back into the main repository.
Additionally, we ran LangFuzz on Mozilla's type inference branch of TraceMonkey. At that time, this branch had alpha status and has not been part of Firefox 4 (but was eventually included in Firefox 5). Since this branch was no product branch, no security assessment was done for issues reported against it.

Google V8 Similar to the Mozilla field test, we tested LangFuzz on the Google V8 JavaScript engine contained within the development trunk branch. At the time of testing, Chrome 10—including the new V8 optimization technique "Crankshaft"—was in beta stage and fixes for this branch were regularly merged back into the Chrome 10 beta branch.

PHP To verify LangFuzz's language independence, we performed a proof-of-concept adaptation to PHP; see Section 6 for details. The experiment was conducted on the PHP trunk branch (SVN revision 309115). The experiment lasted 14 days.

5.3.1 Can LangFuzz detect real undetected defects?

For all three JavaScript engines, LangFuzz found between 51 and 59 defects (see Figure 10). For the Mozilla TraceMonkey (FF4 Beta) branch, most left group of bars in Figure 10, 39% of the found security issues where classified as security related by the Mozilla development team. Only nine defects were classified as duplicates of bug reports not being related to our experiments. The relatively low number of duplicates within all defect sets shows that LangFuzz detects defects that slipped through the quality gate of the individual projects, showing the usefulness of LangFuzz. Although the fraction of security related defects for the Google V8 branch is lower (19%), it is still a significant number of new security related defects being found. The number of security issues within the Mozilla TraceMonkey (Type Inference) branch is reported as zero, simply because this branch was not part of any product at the time of the experi-

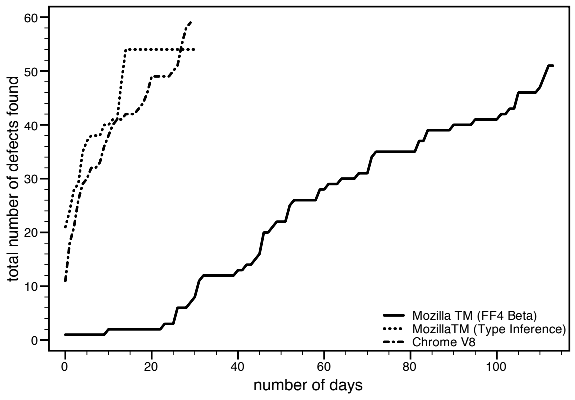

Figure 11: Cumulative sum of the total number of defects found depending on the length of the experiment. Approximate 30 days for Mozilla TM (FF4 Beta) and Google Chrome V8. We did no such analysis for PHP.

ment. This is why the Mozilla development team made no security assessment for these issue reports.

For PHP, the number of detected defects is much lower (see Figure 10). We considered the PHP experiment as a proof-of-concept adaptation and invested considerably less time into this experiment. Still, the total number of 18 defects was detected just within 14 days (1.3 defects per day). The majority of these defects concerned memory safety violations in the PHP engine. We consider these to be potential security vulnerabilities if an engine is supposed to run on untrusted input. However, the PHP development team did not classify reported issues as security relevant.

Figure 11 shows the cumulative sum of the total number of defects found depending on the length of the experiment. The length of the different plotted lines corresponds to the number of days each experiment was conducted. The result for the Mozilla TraceMonkey (FF4 Beta) branch differs in length and shape. This stems from the fact that during this experiment (which was our first experiment) we fixed multiple issues within LangFuzz which did not affect the two later applied experiments. For both Mozilla projects, LangFuzz detected constantly new defects. The curves of the two shorter experiments show a very steep gradient right after starting LangFuzz. The longer the experiment, the lower the number of defects found per time period. Although this is no surprising finding, it manifests that LangFuzz quickly find defects produced by test cases that differ greatly from other test cases used before.

The issue reports corresponding to the defects reported during field experiments can be found online using the links shown in Table 2. Each link references a search query within the corresponding issue tracker showing exactly those issue reports filed during our field experiments. Due to the fact that some of the bug reports could be used to exploit the corresponding browser version, some issue reports are security locked requiring special permission to open the bug report. At the time of writing this paper, this affects all Google V8 issue reports.

LangFuzz detected 164 real world defects in popular JavaScript engines within four months, including 31 security related defects. On PHP, LangFuzz detected 20 defects within 14 days.

5.3.2 Economic value

The number of defects detected by LangFuzz must be interpreted with regard to the actual *value* of these defects. Many of the defects were rewarded by bug bounty awards. Within nine month of experimenting with LangFuzz, defects found by the tool obtained 18 Chromium Security Rewards and 12 Mozilla Security Bug Bounty Awards. We can only speculate on the potential damage these findings prevented; in real money, however, the above awards translated into 50,000 US$ of bug bounties. Indeed, during this period, LangFuzz became one of the top bounty collectors for Mozilla TraceMonkey and Google V8.

6 Adaptation to PHP

Although adapting LangFuzz to a new language is kept as simple as possible, some adaptations are required. Changes related to reading/running the respective project test suite, integrating the generated parser/lexer classes, and supplying additional language-dependent information (optional) are necessary. In most cases, the required effort for these changes adaptation changes is considerably lower than the effort required to write a new language fuzzer from scratch. The following is a short description of the changes required for PHP and in general:

Integration of Parser/Lexer Classes. Given a grammar for the language, we first have to generate the Parser/Lexer Java classes using ANTLR (automatic step). For PHP, we choose the grammar supplied by the PHPParser project [5].
LangFuzz uses so called *high-level parser/lexer classes* that override all methods called when parsing non-terminals. These classes extract the non-terminals during parsing and can be automatically generated from the classes provided by ANTLR. All these classes are part of LangFuzz and get integrated into the internal language abstraction layer.

Integration of Tests. LangFuzz provides a test suite class that must be derived and adjusted depending

Experiment branch	Link
Mozilla TM (FF4 Beta)	http://tinyurl.com/lfgraph-search4
Mozilla TM (Type Inference)	http://tinyurl.com/lfgraph-search2
Google V8	http://tinyurl.com/lfgraph-search3

Table 2: Links to bug reports files during field tests. Due to security locks it might be that certain issue reports require extended permission rights and may not be listed or cannot be opened.

on the target test suite. In the case of PHP, the original test suite is quite complex because each test is made up of different sections (not a single source code file). For our proof-of-concept experiment, we only extracted the code portions from these tests, ignoring setup/teardown procedures and other surrounding instructions. The resulting code files are compatible with the standard test runner, so our runner class does not need any new implementation.

Adding Language-dependent Information (optional) In this step, information about identifiers in the grammar and global built-in objects can be provided (e.g. taken from a public specification). In the case of PHP, the grammar in use provides a single non-terminal in the lexer for all identifiers used in the source code which we can add to our language class. Furthermore, the PHP online documentation provides a list of all built-in functions which we can add to LangFuzz through an external file.

Adapting LangFuzz to test different languages is easy: provide language grammar and integrate tests. Adding language dependent information is not required but highly recommended.

7 Threats to Validity

Our field experiments covered different JavaScript engines and a proof-of-concept adaptation to a second weak typed language (PHP). Nevertheless, we cannot generalize that LangFuzz will be able to detect defects in other interpreters for different languages. It might also be the case that there exist specific requirements or properties that must be met in order to make LangFuzz be effective.

Our direct comparison with jsfunfuzz is limited to a single implementation and limited to certain versions of this implementation. We cannot generalize the results from these experiments. Running LangFuzz and jsfunfuzz on different targets or testing windows might change comparison results.

The size and quality of test suites used by LangFuzz during learning and mutating have a major impact on it's performance. Setups with less test cases or biased test suites might decrease LangFuzz's performance.

Both jsfunfuzz and LangFuzz make extensive use of randomness. While some defects show up very quickly and frequently in all runs, others are harder to detect. Their discovery heavily depend on the time spent and the randomness involved. In our experiments, we tried to find a time limit that is large enough to minimize such effects but remains practical. Choosing different time limits might impact the experimental results.

For most experiments, we report the number of defects found. Some of the reported bugs might be duplicates. Duplicates should be eliminated to prevent bias. Although we invested a lot of efforts to identify such duplicates, we cannot ensure that we detected all of these duplicates. This might impact the number of distinct defects discovered through the experiments.

8 Conclusion

Fuzz testing is easy to apply, but needs language- and project-specific knowledge to be most effective. LangFuzz is an approach to fuzz testing that can easily be adapted to new languages (by feeding it with an appropriate grammar) and to new projects (by feeding it with an appropriate set of test cases to mutate and extend). In our evaluation, this made LangFuzz an effective tool in finding security violations, complementing project-specific tools which had been tuned towards their test subject for several years. The economic value of the bugs uncovered by LangFuzz is best illustrated by the worth of its bugs, as illustrated by the awards and bug bounties it raised. We recommend our approach for simple and effective automated testing of processors of complex input, including compilers and interpreters—especially those dealing with user-defined input.

Acknowledgments. We thank the Mozilla, Google and PHP development teams for their support. Guillaume Destuynder, Florian Gross, Clemens Hammacher, Christian Hammer, Matteo Maffei, and Eva May provided helpful feedback on earlier revisions of this paper.

References

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=610223.

[2] https://bugzilla.mozilla.org/show_bug.cgi?id=626345.

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=626436.

[4] https://code.google.com/p/v8/issues/detail?id=1167.

[5] The phpparser project. Project website. http://code.google.com/p/phpparser/.

[6] AITEL, D. The advantages of block-based protocol analysis for security testing. Tech. rep., 2002.

[7] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. *SIGPLAN Not. 43*, 6 (2008), 206–215.

[8] LINDIG, C. Random testing of c calling conventions. *Proc. AADEBUG.* (2005), 3–12.

[9] MCPEAK, S., AND WILKERSON, D. S. The delta tool. Project website. http://delta.tigris.org/.

[10] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Commun. ACM 33* (December 1990), 32–44.

[11] MILLER, C., AND PETERSON, Z. N. J. Analysis of Mutation and Generation-Based Fuzzing. Tech. rep., Independent Security Evaluators, Mar. 2007.

[12] MOLNAR, D., LI, X. C., AND WAGNER, D. A. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 67–82.

[13] NEUHAUS, S., ZIMMERMANN, T., HOLLER, C., AND ZELLER, A. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (October 2007).

[14] OEHLERT, P. Violating assumptions with fuzzing. *IEEE Security and Privacy 3* (March 2005), 58–62.

[15] PARR, T., AND QUONG, R. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810.

[16] PURDOM, P. A sentence generator for testing parsers. *BIT Numerical Mathematics 12* (1972), 366–375. 10.1007/BF01932308.

[17] RUDERMAN, J. Introducing jsfunfuzz. Blog Entry. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[18] SHU, G., HSU, Y., AND LEE, D. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems* (Berlin, Heidelberg, 2008), FORTE '08, Springer-Verlag, pp. 299–304.

[19] SUTTON, M., AND GREENE, A. The art of file format fuzzing. In *Blackhat USA Conference* (2005).

[20] SUTTON, M., GREENE, A., AND AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[21] TURNER, B. Random c program generator. Project website. http://sites.google.com/site/brturn2/randomcprogramgenerator, 2007.

[22] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2011), ACM SIGPLAN, ACM.

[23] ZALEWSKI, M. Announcing cross_fuzz. Blog Entry. http://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html, 2011.

[24] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* (2002), 183–200.

Appendix

Parameter	Default Value
`synth.prob` – Probability to generate a required fragment instead of using a known one.	0.5
`synth.maxsteps` – The maximal number of steps to make during the stepwise expansion. The actual amount is 3 + a randomly chosen number between 1 and this value.	5
`fragment.max.replace` – The maximal number of fragments that are replaced during test mutation. The actual amount is a randomly chosen number between 1 and this value.	2
`identifier.whitelist.active.prob` – The probability to actively introduce a built-in identifier during fragment rewriting (i.e. a normal identifier in the fragment is replaced by a built-in identifier).	0.1

Table 3: Common parameters in LangFuzz and their default values. See Section 4.4 on how these default values were chosen.