Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's Thesis

Grammar-Based Interpreter Fuzz Testing

submitted by

Christian Holler

on June 30, 2011

Supervisor:	Prof. Dr.-Ing. Andreas Zeller
Advisor:	Kim Herzig, M.Sc.
Reviewers:	Prof. Dr.-Ing. Andreas Zeller Prof. Dr. Wolfgang J. Paul

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbruecken, June 29, 2011

_____
Christian Holler

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbruecken, June 29, 2011

_____
Christian Holler

Contents

Abstract

Fuzz-Testing (Robustness Testing) is a popular automated testing method to locate defects within software that has proven to be valuable especially in the area of security testing. Several frameworks for typical applications (e.g. network protocols or media files) have been written so far. However, when it comes to interpreter testing, only a few language specific fuzzers exist. In this thesis we will introduce a new fuzzing tool called "LangFuzz" which provides a language-independent approach for interpreter testing by combining random code generation and code mutation techniques based on syntax. For the languages JavaScript and PHP, we will evaluate the tool and show that it is able to detect real-world defects in the popular browsers Firefox and Chrome. So far, LangFuzz has been awarded with twelve Mozilla Security Bug Bounty Awards and twelve Chromium Security Rewards.

Chapter 1

Introduction

Software security is a topic that has become more and more important with the growing interconnectedness, not only for companies but also for individuals. Security issues can be expensive[1] but even without a financial aspect, it is generally desirable to avoid security issues to protect the privacy of users and companies. The variance of techniques to tackle such security issues is broad. Especially since software systems tend to become more and more complex and thus harder to overlook and secure. One of these techniques is called "fuzz testing" or simply "fuzzing".
Fuzz testing is the process of automatically generating (random) input data for a software that is to be tested and observing its behavior, e.g. to find a crash or assertion. How sophisticated the generated input can be, solely depends on the fuzzer itself. In some cases, even very unsophisticated fuzzers that have only little information about the target and hence lack adaptation, can provoke serious errors. In other cases (e.g. a network protocol) the fuzzer needs additional knowledge about the expected input format. Failing to produce the correct input format will in many cases severely lower the chances to find any errors because such input is usually rejected by the target software at a very early stage. For this purpose, fuzzing frameworks such as the *Peach Framework* [1] include facilities to model the structure of the data that is to be generated.
However, none of these frameworks can perform fuzz testing based on a grammar. Especially in web oriented software, there exist certain facilities processing untrusted input, that can only be described with a grammar. A typical example is the JavaScript engine found in most modern web browsers. As JavaScript is a programming language, its syntax rules are usually described using a context-free grammar (see also Section 2.2.2).
Without following the rules of the underlying grammar, a random input will most likely be rejected by such an engine at the lexer level or during

[1] In 2008, the annual CSI Computer Crime & Security survey has calculated the average loss to be 289,000 USD per incident amongst all survey participators

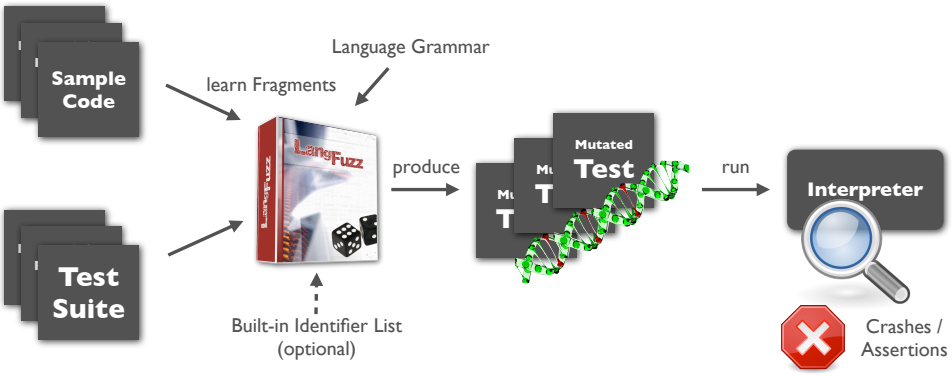

Figure 1.1: Overview on the LangFuzz workflow

further syntax check, thus not reaching high level parts of the engine code (Section 2.2.3 gives a brief overview about interpreter stages).
This thesis aims to develop and evaluate a new approach that allows for black-box fuzz testing (i.e. when we only possess a functional specification of the target program and not the source code) of such an engine based on a context-free grammar. In the following sections, we will first present work related to the topic of grammar-based fuzz testing and to fuzz testing in general. Furthermore, we describe how we derive and evolve our testing techniques. We then evaluate our technique in several experiments and practical tests (Sections 4.1 to 4.4) including a comparison with state of the art. We then cover some treats to our approach (Section 5) and finally conclude on the results and possible future work (Section 6).
As a result, we present our tool called *LangFuzz* which implements a mutation-based approach (3.2) combined with pure code generation (3.1). Figure 1.1 provides an overview of the LangFuzz framework.

Chapter 2

Background

2.1 Related Work

Fuzz testing has a long history. One of the first publications about the topic was by Miller et al. [7] when they tested UNIX utilities using random program inputs in 1990. Since then, the technique has been developed into many different directions, e.g. protocol [3][14] and file format testing [15][16] or mutation of valid input [10][16]. Because it is often easy and cheap to write a fuzzer that actually discovers defects, fuzz testing has become an essential part of many software development life cycles and can often be helpful when other testing method can hardly be applied [10]. Grammar-based fuzz testing is not entirely new either. The following contributions are more closely related to either the subtopic of grammar-based fuzz testing or the approach used in this work.

Introducing jsfunfuzz, written by Rudermann [13] had a large impact when he wrote it in 2007. The jsfunfuzz tool (itself written in JavaScript) is a black-box fuzzing tool for the JavaScript engine which aims at certain categories of errors. It does not only search for crashes but can also detect certain correctness errors by differential testing. Since the tool was released, it has found over 1000 bugs in the Mozilla JavaScript Engine[1] and was quickly spread amongst browser developers [2]. Our work is inspired by this tool, being the first JavaScript fuzzer that was publicly available (it has been withdrawn by now). Our tool however does not specifically aim at a single language, although JavaScript is used for evaluation and experiments. Instead, our approaches aim to be solely based on grammar and general language assumptions.

Grammar-based White-box Fuzzing, researched by Godefroid et al. [4] in 2008 deals with grammar-based fuzzing under the assumption that

[1] according to https://bugzilla.mozilla.org/show_bug.cgi?id=jsfunfuzz
[2] http://news.cnet.com/8301-10784_3-9754201-7.html

the source of the target is available (white-box fuzzing). Their approach combines a grammar with a constraint solver and coverage measurement to produce inputs to a JavaScript Engine (Internet Explorer 7) that reach a certain area of code.

The work is partially related since the authors compare their approach to a black-box approach (which ours is). Their results show that their black-box approach is by far not as efficient as their white-box method. Our work can be seen as an extension of this work to a certain degree: We would like to improve grammar-based black-box fuzzing techniques instead of focusing on a white-box approach. The benefit of the black-box approach is that we are neither bound to a certain language used for implementing the target program nor do we need the source code (which is helpful when testing closed-source software).

However, we believe that simple coverage measurements (line/branch based) are not very expressive in nowadays modern and complicated language engines because they often make use of compilation/JIT techniques and heavily contain on global state.

Finding and Understanding Bugs in C Compilers, written by Yang et al. [18] is another very recent example for a language-specific fuzzer. In their work, the authors explain how they constructed a fuzzer called *CSmith* that automatically generates C programs for testing compilers. It is based on earlier work of these authors and on the "Random C Program Generator" written by Turner [17]. One interesting parallel to our work is that CSmith randomly uses productions from its built-in C grammar to create a program. In contrast to our work, their grammar has non-uniform probability annotations which might make the use of a recursive approach (random walk, see Section 3.1.1) easier. 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 with a language-independent approach.

2.1.1 Indirectly Related

The following work is still related to automated testing but not so closely related as the previously discussed work. We provide these references to show other types and applications of fuzz testing but also because some ideas and results can typically be applied to other areas of fuzz testing as well.

A sentence generator for testing parsers published by Purdom [12] in 1972 predates even the first named references to "fuzz testing". It is

however still relevant for us because it is one of the first attempts to automatically test a parser using the grammar it is based on. Especially the idea of the "Shortest Terminal String Algorithm" has been used in this work as well (see Section 3.1.2).

Random testing of C calling convention published by Lindig [5] in 2005 is another example finding compiler problems with random testing. A program called *QUEST* here generates code to specifically stress the C calling convention and check the results later. In this work, the generator also uses recursion on a small grammar combined with a fixed test generation scheme.

Analysis of Mutation and Generation-Based Fuzzing published by Miller and Peterson [8] in 2007 is an evaluation of the two main approaches in fuzzing: generating new input randomly or modifying existing valid inputs instead. In their work, the authors perform fuzz testing on the PNG image format both by mutating existing PNG files and generating new ones from scratch. Their results point out that mutation testing alone can miss a large amount of code due to missing variety in the original inputs. We however believe that this result can only partly be applied to our work. Although we use mutation testing techniques, the mutated inputs we use were already defective at some time in the past (regression test). Furthermore we combine the mutation approach with a generative approach such that a higher variety is more likely.

In **Dynamic test generation to find integer bugs in x86 binary Linux programs** published by Molnar et al. [9] in 2009 the authors present a tool called *SmartFuzz* which uses symbolic execution to triggers integer related problems (overflows, wrong conversion, signedness problems, etc.) in x86 binaries.

Announcing cross_fuzz published by Zalewski [19] is another recent work on browser fuzzing (more precisely DOM fuzzing) that revealed quite a few problems in the most popular browsers. The 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

2.2.1 Defect

We consider only defects that cause abnormal program termination (e.g. a crash due to memory violations or an abort due to an assertion being triggered). Program failures that only manifest in wrong computations/output or any other defective behaviors that do not result in abnormal termination are not within the scope of this work. Such defects can be detected under certain circumstances (see also 7.1). This limitation is reasonable because fuzz-testing for other types of defects is hardly possible without making strong assumptions about the target software. Therefore, it is common to restrict the search to these kind of defects in security-related fuzz testing.

2.2.2 Language Grammar

When talking about language grammars we usually refer to context-free grammars (Type-2 in the Chomsky hierarchy). Such a grammar G is defined by the 4-tuple (N, T, P, S):

- N refers to the set of *non-terminal* symbols.
- T refers to the set of *terminal* symbols.
- P is the set of *productions* which are mappings from N to $(N \cup T)^*$, i.e. they describe the expansion of a single non-terminal symbol to multiple terminal and/or non-terminal symbols.
- S is the start symbol of the grammar, it must be an element of N.

The language described by the grammar is the set of words that can be *produced* from the starting symbol by repeatedly applying rules from P, formally defined by $L(G) = \{w \in T^* \mid \exists\, p_1 \dots p_n \in P : S \to_{p_1} \cdots \to_{p_n} w\}$.

Lexer and Parser

Using a language grammar to process some input usually requires at least the two stages of *lexing* and *parsing*. The lexer is responsible for reading the input and separate into *tokens* which are usually logical groups of characters (which usually refer to low-level symbols of the grammar). The parser then processes the tokens further and groups them to high-level symbols of the grammar.

2.2.3 Interpreter

We consider all programs that receive a program in source code form and then execute it, as interpreters. This also includes so called just-in-time

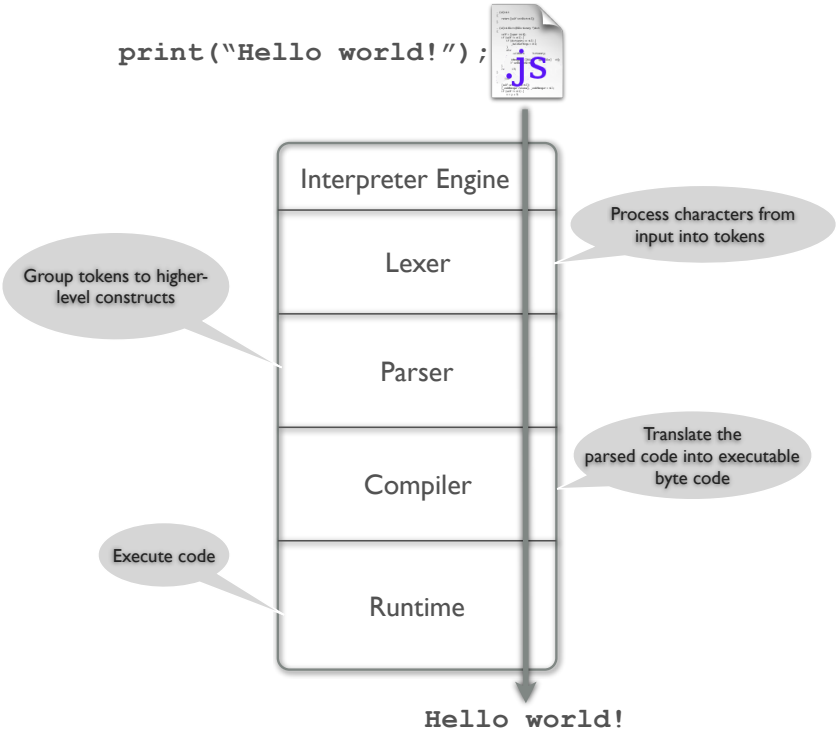

Figure 2.1: Different possible stages of an interpreter

compilers which translate the source code to byte code before or during runtime of the program. The important properties for us are that the input is source code and that the program is executed by our target program. Figure 2.1 gives an exemplary overview of the different interpreter stages. Note that this is just an exemplary model. Especially the compiler and runtime stages are not necessarily separated. For a just-in-time compiler, the compilation is done during runtime whenever it is necessary or advisable for performance reasons. One important fact for fuzz testing here is that lexer and parser stages are able to detect malformed input which causes such input the be dropped at those (early) stages. This is of course not desirable as we would like to find defects in later stages as well, forcing us to produce proper input for lexer and parser.

Chapter 3

Methods

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. Both jsfunfuzz [13] and CSmith [18] use generative approaches. Thus, we will first investigate how we can implement a similar generative but generic approach based on the language grammar (Section 3.1). However, as we would like to stay independent from the language semantics, a purely generative approach is likely to fail due to certain semantic rules not being applied (e.g. a variable must be defined before it is used). Introducing further semantic rules to solve this problem would tie us again to certain language semantics. The logical consequence would be to use existing semantic context, as demonstrated in Section 3.2.

3.1 Random Generative Approaches

As previously discussed, one possible approach for fuzzing an interpreter based on the grammar would be to randomly generate source code that follows the structure of the grammar. In other words, we attempt to generate random strings that could be produced by a given grammar. For this purpose, different approaches and enhancements are possible, which we will discuss in the next sections.

3.1.1 Random Walk over Grammar

Given the definition for a language grammar in Section 2.2.2, it is natural to perform a *random walk* over the tree of possible expansion series. Such an algorithm can be defined as:

1. Set current expansion e_{cur} to the start symbol S.
2. Loop until e_{cur} contains only terminal symbols:

(a) Pick the first non-terminal symbol n from e_{cur}.

(b) Find the set of productions $P_n \subseteq P$ that can be applied to n.

(c) Pick one production p from P_n randomly and apply it to n, yielding $p(n)$.

(d) Replace that occurrence of n in e_{cur} by $p(n)$.

All possible paths from S to any word $w \in L(G)$ span a graph. The given algorithm can be seen as a random walk over this graph of possible expansions.
But there is a serious problem of the approach: A random walk with uniform probabilities is not guaranteed to terminate at all. Terminating the walk by different criteria without completing all expansions might result in a syntactically invalid word.
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 Yang et al. [18] demonstrate in their work about CSmith.
However, in our case we would like to stay language-independent which greatly limits our possibilities to introduce semantic restrictions. Furthermore, 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 describe a modified algorithm in the next section.

3.1.2 Stepwise Expansion Algorithm

The goal of this approach is to overcome the problems exposed by the random walk approach: We'd like to be able to terminate the algorithm after every iteration step without yielding a syntactically invalid word. To achieve this, we rewrite the algorithm to not perform a depth-first search for every non-terminal but to rather increase in a breadth-first manner:

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 $p(n)$.

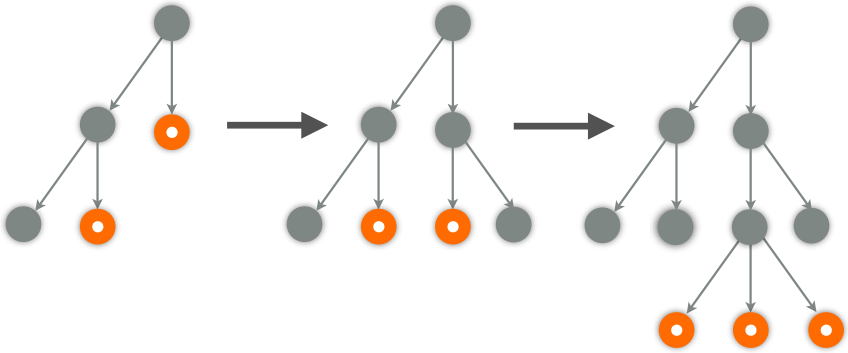

Figure 3.1: Example of a stepwise expansion on the syntax tree: Orange (dotted) nodes are unexpanded non-terminals (can be expanded) while the other nodes have already been expanded before.

In Figure 3.1, you can see how two steps in the algorithm might look like, considering the expansion as a tree. Orange (dotted) nodes are unexpanded non-terminals that can be considered for expansion while the remaining nodes have already been expanded before.
This algorithm alone doesn't yield a valid expansion after *num* iterations. We need to replace the remaining non-terminal symbols by sequences of terminal symbols.

Minimal expansion

One possibility is to determine the set of *minimal* expansions N_{min} for the grammar, i.e. for every $n \in N$ we determine $n_{min} \in T*$ which is the smallest possible expansion of this non-terminal symbol into terminal symbols. Rephrasing the problem, we'd like to find the smallest set of productions P_{min} such that $N \rightarrow_{P_{min}} w$ and $w \in T*$. This can easily be achieved by a breadth-first search (BFS) on all possible expansions of the respective non-terminal symbol. Note that this causes an exponential blow-up, but the search has to be performed once for the grammar only. Given N_{min}, we can replace every remaining non-terminal symbol by its minimal expansion in a single step.
In practice, the minimal expansion is problematic though for two reasons:

1. It is atypical for normal programs and causes bias in the expansion (e.g. shortest *primaryExpression* is always `this` and shortest *unaryExpression* is always `delete`).

2. Certain fragments typically cause infinite loops in their minimal expansion state (e.g. `for (;;)` is the smallest for-loop in many languages).

While the minimal expansion might be interesting in other applications (e.g. syntactic test case minimization), it does not seem suitable to form the majority of our generated code.

Learning Expansions From Code

The only alternative to generating fully expanded code fragments ourselves is using already existing ones. One possibility to obtain such existing code fragments is to learn them from existing code. Using a parser with the given grammar, we can extract possible expansions from given code for every non-terminal symbol. Given a large codebase, we can build up a *fragment pool* consisting of expansions for all kinds of non-terminal symbols. These code fragments might of course be semantically invalid or less useful without the context that surrounds them, but they will at least allow us to complete our code generation yielding syntactically valid code. In Section 3.2 we will see one possibility to enhance our resulting code on a semantic level by tweaking on the identifiers.
Using known expansions from our fragment pool, we can complete the stepwise expansion algorithm as follows:

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 $p(n)$.
3. Loop while e_{cur} contains non-terminal symbols:
   (a) Choose the first non-terminal n in e_{cur}:
   (b) Find a known fragment expansion n_e for n (syntactically equivalent) in the fragment pool.
   (c) Replace n by n_e.

Using this algorithm, we can now generate random code that can either be used to directly test the target program or that can be incorporated into more complex approaches.

3.2 Code Mutations

So far, we've been focusing on generating new programs from scratch. As our primary target is to trigger bugs 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 as well. Especially for interpreted programming languages, there often exist such test cases that are written in the language itself. Using the same parser approach as in the previous section for expansion learning, we can process whole tests and learn the locations and types of all source code fragments within the tests. We can now randomly pick some of these fragments and replace them with other fragments of the same type (syntactically equivalent) that we've generated and/or learned before (3.1.2). There are many adjustments to this method that can be made that are described in the remained of this section.

Adjust 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 (because of language independence), we can only perform corrections based on *generic* semantic assumptions. A perfect 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 has not been declared is considered to be a runtime error).
However, 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. This way, 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 also 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 and can easily be extracted there.

Fragment Type Restrictions We could restrict the method to replace certain fragment types only, e.g. statements or expressions. Such restrictions

make it possible to test even on different levels within the grammar itself. Furthermore, restricting this to a few high-level fragment types will make the whole approach much faster.

Ratio between Generated and Purely Learned Fragments For performance reasons, it would be possible to reduce or completely omit the generation of fragments and instead directly use only known fragments that have been learned from the code base.

3.3 Constructing LangFuzz

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 1.1) in the introduction.
The typical steps performed by our implementation start with a learning phase where the given code base is parsed and fragments are learned (Section 3.3.1). The input here can be either the test suite itself or any other code base in the target language. Once the learning step is complete, LangFuzz starts to process the test suite. All tests are parsed and the result is 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 decide how to interpret each of those fragments if multiple possibilities exist.

3. After test mutation is complete, the mutated test is executed and its result is checked (Section 3.3.4).

3.3.1 Code Parsing

Both when learning code fragments (first step in the workflow) as well as in the mutation step, we need the be able to parse the given source code. For this purpose, LangFuzz contains a language-independent parser subsystem such that concrete parsers for different languages can be added. We decided to use the ANTLR parser generator framework by Parr and Quong [11] 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. When producing a mutated

test, it is used again to find all fragments in the test that could be replaced and to determine which replacements can be made without breaking syntax.

3.3.2 Code Generation

The code generation step uses the stepwise expansion (Section 3.1.2) 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 rewritten to remove these operators by introducing additional rules according to the following pattern

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

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 with one or more alternatives and 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 3.1.2. All other remaining non-terminals are replaced by learned code fragments as described earlier. In our implementation, we however introduced a size limitation on these fragments so huge code fragments are not put into small generated code fragments.

3.3.3 Fragment Replacement

The fragment replacement code first modifies the new fragment as described in the first paragraph of Section 3.2. 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”. If the mapping was changed for every identifier occurrence, then we would probably destroy some of the semantics in the fragment. Identifiers that are on the built-in identifier list (“Global Objects”) are not replaced. LangFuzz can also actively map an identifier to a built-in identifier with a certain probability.

3.3.4 Test Running

In order to be able to run a mutated test, LangFuzz must be able to run the test with its proper *test harness* which contain 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 sub directory 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 now is to start the JavaScript engine binary with the appropriate test harness files and the mutated test. While this is relatively easy, it is also very slow because the binary has to be started for every test and the relatively large test harness has to be processed for every test although it remains the same. To solve this problem, LangFuzz uses a *persistent* shell: A small JavaScript program called the *driver* is started together with the test harness. The driver reads filenames line by line from standard input until it receives a special signal that causes it to run all the received files one by one. Once the driver has run all files, it signals completion and is ready to run a new test. 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.
While the original intention of the persistent shell was to increase the test throughput, the results have shown that it also helps to find further defects: Because multiple tests are running in a single shell instance, they can all contribute to a single failure. In our experiments, most of the defects we found did not boil down to a single test but required multiple tests 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.
Of course, running multiple test cases in one shell also means that we have to determine which tests are relevant for failure reproduction. The goal would be to provide a suitably small test case that allows reproducing the original crash. Using the *delta debugging algorithm* by Zeller and Hildebrandt [20], we filter out irrelevant test cases first. Later we apply the same algorithm to reduce the remaining number of executed source code lines. This way, we

can provide a suitably small test case in nearly all cases. In our experiments, we used the *delta* tool [6] which provides an implementation of the original algorithm described by Zeller and Hildebrandt [20].

3.3.5 Parameters

LangFuzz contains a large amount of adjustable parameters, e.g. probabilities and amounts that drive decisions during the fuzzing process. In Table 3.1 we provide the most common/important parameters and their default values.

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 3.1: Common parameters in LangFuzz 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), 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.

Chapter 4

Evaluation

4.1 Comparison with State of the Art

In this section, we compare LangFuzz to state of the art in interpreter fuzz testing. Because of it's wide application and success, we choose Mozilla's jsfunfuzz tool for our comparisons. jsfunfuzz is an active part of Mozilla's and Google's quality assurance and regularly used in their development.

4.1.1 Differences between the Programs

jsfunfuzz is specialized on the JavaScript language. This specialization should allow the program to test even very specific new and/or previously untested JavaScript features intensively. Furthermore, the program has a certain level of semantic knowledge and should be able to construct valid programs easier than any program without such knowledge. However, for every new feature introduced into the language or even only in the implementation, the program has to be modified to incorporate these changes into the testing process. Also, focusing on certain semantics can exclude certain defects from being revealed at all.

LangFuzz is not language specific. It bases its testing strategy solely on the grammar, existing programs (e.g. test suites) and a very low amount of additional language-dependent information. Because of this, the approach is generic and can be easily adapted to different languages. Furthermore, new implementation features are automatically covered if they are tested within the respective test suites. Changes to the language do only require program maintenance if they affect the syntax. The use of existing programs like previous regression tests should allow LangFuzz to profit from previously detected defects. On the other hand, LangFuzz lacks a lot of semantical background on the language which lowers the chances to obtain sane programs and produce test cases that require a high amount of semantical interaction between parts of the program.

4.1.2 Questions and Goals

There are two main questions that we will answer by the comparison:

Overlap How far do defects detected by LangFuzz and jsfunfuzz overlap?

By overlap, we refer to the number of defects that both tools are able to locate. This can tell us how large LangFuzz's contribution is, if a development environment uses already jsfunfuzz. Clearly, it's desirable that LangFuzz's results are not entirely a subset of jsfunfuzz's results. LangFuzz should be able to detect defects that jsfunfuzz missed before.

Effectiveness 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 time. It is also possible that LangFuzz finds defects during the experiment that can only be detected by jsfunfuzz given a larger amount of time. Such situations should also be covered by the effectiveness analysis.

Measuring both quantities requires a special experiment setup as we will see in the next section.
Overall, we'd like to show that LangFuzz is a beneficial contribution to a development process, even if it already uses a fuzzer like jsfunfuzz. We *do not* want to show that LangFuzz is *worse* or *better* than jsfunfuzz or any other specific language fuzzing tool. We believe that such comparisons do not make sense because both programs operate on different levels and have different strengths and weaknesses (as briefly explained in 4.1.1).

4.1.3 Experiment Setup

For our comparison, we choose the TraceMonkey, Mozilla's JavaScript engine, as the evaluation target for several reasons. First of all, Mozilla's development process is largely open and a lot of development information is public. For those parts that are not public (e.g. security bug reports), we had contact with the Mozilla security team to get access to such information if necessary. Also, both tools are already adapted to run on TraceMonkey, so we can especially be sure that jsfunfuzz properly functions on the target during the comparison.
One of the major problems with TraceMonkey is that one of the tools (jsfunfuzz) is already part of its development process. As a result, when looking at revision x of TraceMonkey, almost all defects that jsfunfuzz can identify will be fixed already (except for very recent ones that have not yet been fixed). So, it is not possible to measure effectiveness based on single revisions. But Mozilla maintains a list of all defects that have been identified through jsfunfuzz. Using this information, we can propose a different test strategy:

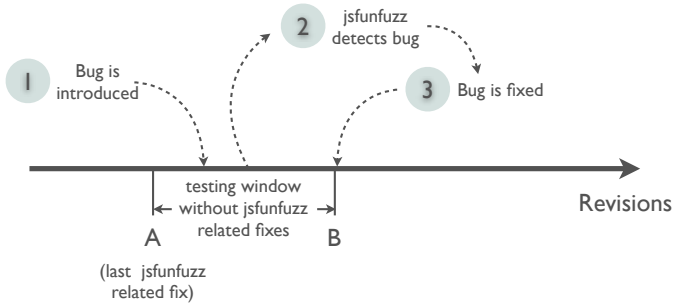

Figure 4.1: Example of a testing window with the live cycle of a single defect

1. Start at some base revision f_0 of TraceMonkey, run both tools for a fixed amount of time. All defects detected here can solely be used to analyze the overlap, not the 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 $f_n - 1$. For overlap measurement, all defects can be used.

   (c) Increase n by 1.

The idea behind this testing strategy is to find periods of time in the revision history where no defects detected by jsfunfuzz were fixed. Within these periods, both tools will have the same chances to find defects in the experiment. From now on, we'll refer to these periods as *testing windows*. Figure 4.1 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 is reported by jsfunfuzz. Finally, the bug is fixed by a developer 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.

Repository, Base Revision and Testing Windows For all tests, we will use the TraceMonkey development repository. Both the tested implementation and the test cases (approximately 3000 tests) are taken from that repository. As base revision, we chose revision 46549 (03f3c7efaa5e) which is the first revision committed in July 2010, right after Mozilla Firefox Beta 1 was released at June 30, 2010. We used the following five test windows for our experiments:

Start Revision	End Revision
46569:03f3c7efaa5e	47557:3b1c3f0e98d8
47557:3b1c3f0e98d8	48065:7ff4f93bddaa
48065:7ff4f93bddaa	48350:d7c7ba27b84e
48350:d7c7ba27b84e	49731:aaa87f0f1afe
49731:aaa87f0f1afe	51607:f3e58c264932

The end revision of the last testing window dates to the end of August 2010, so we covered roughly 2 months of development time using these five windows.

Time Frame, Resources and Settings For each testing window, both tools will be granted 24 hours of testing. Both tools will be run on 4 CPU cores with the same specification. As jsfunfuzz does not support threading, multiple instances will be used instead. LangFuzz's parameters are set to their defaults as specified in Section 3.3.5.

Defect Processing Once a defect has been found, we still need to find the appropriate bug report and the lifetime of the bug (at least if it was introduced in the current testing window). Usually, this can be achieved by using the *bisect* command, that the Mercurial SCM provides. This command allows automated testing through the revision history to find the revision that introduced or fixed a certain defect.

4.1.4 Experiment Results

During the experiment, jsfunfuzz identified a total of 23 defects, of which 15 where within the respective testing window. LangFuzz identified a total of 26 bugs, of which 8 where in the testing window. The larger number of defects outside the testing window for LangFuzz was expected because LangFuzz, unlike jsfunfuzz, was previously never used on the source base. Figure 4.2 illustrates the number of defects per fuzzer, that where within the respective testing window.

Overlap

As Figure 4.2 shows, only 3 defects where found by both fuzzers during the experiment. Expressing the overlap as a fraction of all defects found can be achieved by calculating

$$\text{Overlap} = \frac{\text{Number of defects found by both tools}}{\text{Number of defects found in total}} \tag{4.1}$$

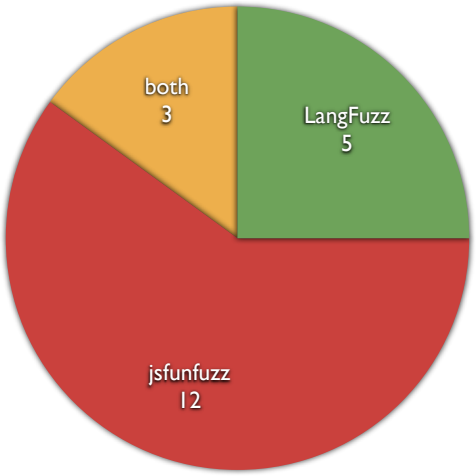

Figure 4.2: Number of defects found by each fuzzer within the testing window

which gives us an overlap of only 15%. This clearly shows that LangFuzz and jsfunfuzz find different defects and hence supplement each other.

Note: During the experiment, LangFuzz found 3 defects that jsfunfuzz did not find in the same experiment, but that were reported as jsfunfuzz defects outside the experiment. As jsfunfuzz is part of the daily Q&A process at Mozilla (i.e. runs 24/7 on multiple servers), it has much more time and resources in reality than given in our experiment. To get a fair comparison, we only count defects found during the experiment. This also applies for LangFuzz: At least one bug was found by LangFuzz only outside the experiments and within the experiment only by jsfunfuzz. Again we only counted the findings within the experiment.

Effectiveness

In order to get an expressive effectiveness, we need to relate LangFuzz's absolute effectiveness to a reference effectiveness (in this case jsfunfuzz). This can be done by taking the fraction of defects found by each tool:

$$\text{Effectiveness} = \frac{\text{Number of defects found by LangFuzz}}{\text{Number of defects found by reference (jsfunfuzz)}} \tag{4.2}$$

With LangFuzz identifying 8 defects and jsfunfuzz 15, we see that LangFuzz's effectiveness is around 53% of that of jsfunfuzz. In other words, jsfunfuzz is twice as effective as LangFuzz. We can conclude that the language specific fuzzer is usually more effective (which is also intuitive) but we also

see that LangFuzz's approach is nonetheless practical as our effectiveness is still within the same order of magnitude compared to jsfunfuzz.

4.2 Generative vs. Mutative Approach

In this experiment, we analyze the impact of each of the different approaches used in LangFuzz. Given the results from the last experiment, it is evident that LangFuzz can locate defects that jsfunfuzz misses. It is however unclear, what importance the generative algorithm has and what role mutations play in these results.

4.2.1 Questions and Goals

In this experiment, we will answer the following questions:

- How important is it that LangFuzz generates new code?
- How important is the role that mutation plays in LangFuzz?

As a result, we will especially investigate if mainly 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.

4.2.2 Experiment Setup

To identify the influence of the different approaches, we perform several LangFuzz runs on selected versions that we've been testing before in 4.1. All runs are limited to the same amount of time and resources, however the settings differ from run to run:

Mutation without code generation Run with code generation completely disabled

Mutation with only code generation Mutate tests but generate every code fragment used

Intuitively, we would like to add another run which does not perform mutation at all. The results would however not be comparable to the other runs for several reasons:

Code Size When generating new code, one usually does not aim to create code as large as a whole regression test (because the larger the code, the higher is the chance to introduce some error and most of the code will most likely remain meaningless). Even jsfunfuzz aims to generate code that is much smaller than most of the regression tests.

Environment Adjustment When mutating code, we can adjust the newly introduced fragment to the environment (see also 3.2). With purely generated code, this is not possible in the same way as there exists no consistent environment around the location where a fragment is inserted (in the syntax tree at the end of generation). It would be possible to track the use of identifiers during generation but the result would most likely not be comparable to what happens during code mutation.

Therefore, we will try to determine the impact of code generation from the difference between the mutation runs with and without code generation enabled.

Versions, Time and Parameters

Because we are only comparing different settings of LangFuzz, we do not need the testing windows that we've been using in the comparison with jsfunfuzz. Hence we will not repeat the experiment on all versions used during that comparison but restrict it to those two versions that showed most defects in the previous experiment. This restriction allows us to increase the time that is granted to all runs, minimizing the randomization impact on the experiment even further. In order to keep the experiment feasible, we grant each run 3 days on each version (yielding an experiment time of 12 days). The two selected revisions for this experiment are `03f3c7efaa5e` and `f3e58c264932`. Both revisions showed a high defect rate in the previous experiment and additionally, over 5000 revisions lie between them, so we should be able to find different defects in both. All runs will be done with default parameters (see Section 3.3.5), except for the `synth.prob` parameter being set to 0.0 for the run without code generation (respectively 1.0 for code generation only).

4.2.3 Experiment Results

Figure 4.3 shows the results of our experiment. We can see that there is no clear advantage of any of the settings over the other: On the first revision, code generation only outperforms the approach running without any code generation. In the second revision though, code generation only is slightly worse.
As a conclusion we can say that the ideal approach should be a mixed setting where both code generation and direct fragment replacement is done, both with a certain probability.

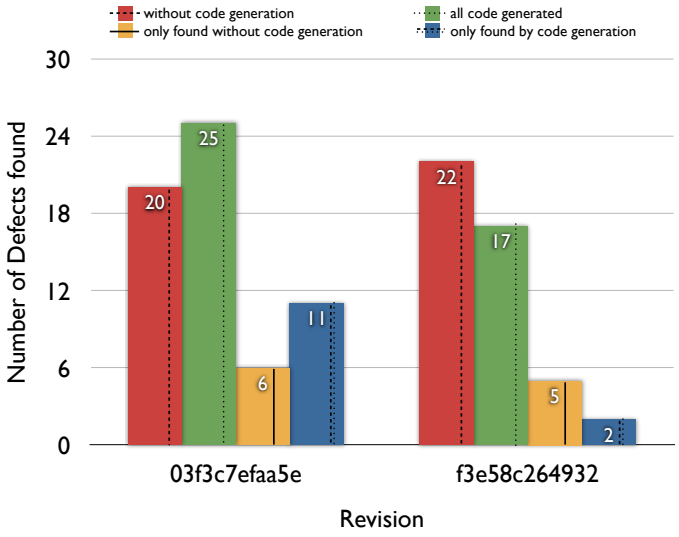

Figure 4.3: Results of comparison with/without code generation

4.3 Field Tests with Mozilla and Google

To demonstrate that LangFuzz is able to find defects in large and popular software projects, we decided to apply it both to Mozilla and Google software during their development cycles. Table 4.1 shows the results of our field tests while the exact target branches are described more detailed afterwards.

Mozilla TraceMonkey in Firefox 4 Development The Mozilla TraceMonkey trunk versions used during our tests where part of the Firefox 4 development (at that time in beta stage). Changes to TraceMonkey trunk were regularly merged back to the main repository for direct inclusion into Firefox 4 betas.

Mozilla Type Inference Branch The Mozilla TraceMonkey type inference branch is a more experimental branch where a new type inference technique is being developed. This branch has alpha quality and is not part of Firefox 4 yet but will most likely be included in Firefox 5 or its successor. Because this branch is not part of any product yet, no security assessment is done for these bug reports.

Google V8 in Chrome 10 Development We also tested LangFuzz on the Google V8 engine development trunk. At the time of testing, Chrome 10 (including the new V8 optimization technique "Crankshaft") was in beta stage and fixes for this branch where regularly merged back into the Chrome

Table 4.1: Results on Mozilla and Google software

Tested Software	Time Span	Number of Defects
Mozilla TraceMonkey (Firefox 4 Beta)	4 months	**51 defects** 9 duplicates to others 20 security locked[1]
Mozilla TraceMonkey (Type Inference)	1 month	**54 defects** 4 duplicates to others
Google V8 (Chrome 10 Beta)	1 month	**59 defects** 0 duplicates to others 11 confirmed security defects

[1]The security lock hides the bug report from public because it might be exploitable

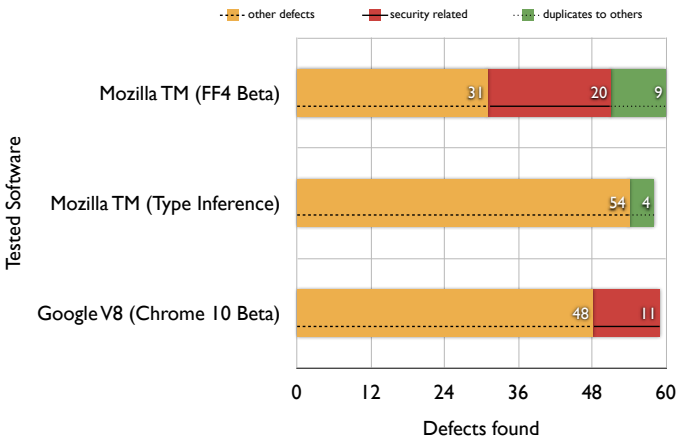

Figure 4.4: Visualized results of Table 4.1

10 beta branch. For our helpful commitment on Chrome 10, Google included included a note of thanks on our work in their official announcement [2].

4.3.1 Example for a bug missed by jsfunfuzz

There are several bugs (especially garbage collector related) where we assume that jsfunfuzz is not able to trigger them due to their high complexity even after minimization. We can however give at least one example, where jsfunfuzz was not able to trigger the bug: In bug 626345[3], 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."

4.3.2 Example for detected incomplete fix

Bug 1167 in the V8 project 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 pushed to other branches and of course a new regression test based on the LangFuzz test was added to the repository. Shortly after that happened, LangFuzz triggered exactly the same assertion again, this time the test mutated though was the *new* regression test, recently added. V8 developers confirmed that the fix was indeed incomplete and issued another fix.

4.3.3 Example for defect detected through code generation only

The following code triggered an error in the parser subsystem of Mozilla TraceMonkey and was reported as bug 626436[4]:

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

This test was partly produced by the code generation and is highly unlikely to be uncovered by mutation, only. The complex and unusual syntactic nesting here is unlikely to happen by only mutating regular code.

4.3.4 Further code examples

The appendix of this work contains lists of all bugs that we reported, including URLs where these reports can be inspected. Each of these bug reports usually contains the test case either attached or inlined. If you are interested in further code examples, inspecting these bug reports is the best choice.

[2] http://googlechromereleases.blogspot.com/2011/03/chrome-stable-release.html
[3] https://bugzilla.mozilla.org/show_bug.cgi?id=626345
[4] https://bugzilla.mozilla.org/show_bug.cgi?id=626436

4.4 Proof-of-Concept Adaptation to PHP

In this section we show results derived from running LangFuzz on PHP as a proof of concept adaptation to demonstrate that LangFuzz is not limited to a single language. For this purpose, we will first explain the necessary steps to run LangFuzz on PHP. We will then run LangFuzz with the modifications to investigate if we can find defects also in the PHP interpreter.
The PHP language was chosen for this experiment for several reasons:

Dynamic/Weak Typing PHP is a dynamically typed language with weak typing, similar to JavaScript.

Grammar available An ANTLR grammar is readily available from the PHPParser project [2]

Tests available The PHP project has a large set of unit- and regression tests available to public.

We assume that especially the typing properties of the target language should affect the success rate of LangFuzz. The more static restrictions the language imposes, the harder it should be for tools like LangFuzz to randomly generate a runnable program when only working syntax-based.

4.4.1 Steps required to run on PHP

Although LangFuzz's design is language-independent, the implementation requires changes to adapt to a new language. These changes are related to reading/running the respective project test suite (most projects use their own test suite mechanisms), integrating the generated parser/lexer classes and supplying additional language-dependent information (list of built-in identifiers, name of the identifier non-terminal).

Integration of Parser/Lexer Classes

Using the grammar obtained from the PHPParser project [2], we first have to generate the Parser/Lexer Java classes using ANTLR (automatic step).
LangFuzz uses so called *high-level parser/lexer classes* that override all methods called when parsing non-terminals. These classes simply extract the non-terminals during parsing (similar to a visitor pattern). These classes can be automatically generated from the classes provided by ANTLR.
All classes obtained in this step are included into the project and combined in a class representing the actual language.

Integration of Tests

LangFuzz provides a test suite class that must be derived and adjusted depending 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

The PHP 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.

4.4.2 Experiment/Results on PHP Interpreter

With the additions made in the last section, we performed several runs on the PHP trunk (SVN revision 309115). After a runtime of 14 days we ended up with the 18 issues as shown in Table 4.2. Some of the bugs affected also the stable branch of PHP, as shown by the version column in the table.

Bug #	PHP Version	Title	Classification	Notes
54280	5.3.5	Crash with substr_replace and array	Use after free (assigned CVE-2011-1148)	Duplicate
54281	5.3.5	Crash in spl_recursive_it_rewind_ex	Memory corruption	
54282	Trunk	Crash in zend_mm_check_ptr	Controlled crash, detected memory corruption	
54283	5.3.5	Crash in zend_object_store_get_object	Null-pointer crash	
54284	5.3.5	Crash in zend_object_store_get_object	Memory corruption	Duplicate
54285	Trunk	Crash in _zval_ptr_dtor	Use after free	
54291	5.3.5	Crash in spl_filesystem_object_get_path	Null-pointer crash	
54292	Trunk	Wrong parameter causes crash in SplFileObject::__construct()	Memory corruption	
54296	5.3.5	Crash in SQLite3Stmt internal object destructor	Use after free	
54304	5.3.5	Crash in php_pcre_replace_impl	Arbitrary read-/memory corruption	
54305	Trunk	Crash in gc_remove_zval_from_buffer	Memory corruption	
54322	Trunk	Crash (null pointer) in zif_get_html_translation_table	Null-pointer crash	
54323	Trunk	Accessing unset()'ed ArrayObject's property causes crash	Use after free	
54324	5.3.6	Crash in date_object_compare_date	Duplicate	
54332	5.3.6	Crash in zend_mm_check_ptr Heap corruption	Memory corruption	
54348	5.3.6	Crash (Call stack overflow) in ExchangeArray	Call stack overflow	
54349	Trunk	Crash in zend_std_write_property	Use after free	
54350	5.3.6	Memory corruption with user_filter		

Table 4.2: Results on the PHP Interpreter

Chapter 5

Threats to Validity

5.1 Generalization

5.1.1 Language

In our field experiment (Section 4.3), we have evaluated our approach on two major JavaScript implementations to demonstrate that the success of our technique is not specific to a single implementation. Furthermore, we have shown that it is not specific for a single language by evaluating it with a proof-of-concept on PHP (Section 4.4). Nevertheless, we cannot generalize from these results that the approach will find issues in interpreters for different languages or what requirements/properties the language must satisfy for LangFuzz to be effective.

5.1.2 Tested Software

Our direct comparison with jsfunfuzz (Section 4.1) is not only limited to a single implementation but due to time constraints also limited to certain versions. It is therefore not clear how both tools would perform on different targets and which tool would be more effective. It is certainly possible that the LangFuzz approach could catch up with jsfunfuzz when used on a different implementation, because it is less implementation-dependent and uses tests which were made for this specific implementation.

5.1.3 Test Suite Quality

The Mozilla test suite we used contains over 3000 test cases that try to cover the entire JavaScript specification. We assume that the size and quality of the test suite used with LangFuzz has a major impact on its performance. Projects with less test cases or biased tests could severely decrease the performance and make the tool less helpful in such cases.

5.1.4 Runtime and Randomness

Both jsfunfuzz and LangFuzz make extensive use of randomness to drive their actions. While some defects show up very quickly and frequently in all runs, others are harder to detect. Their discovery might heavily depend both 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 still practical for us. It is however impossible to tell if one of the tools would have performed better in our experiments, given a larger time window.

5.2 Bug Duplicates

In both experiments (Section 4.1 and Section 4.2) as well as in the field evaluations (Section 4.3 and Section 4.4) we give the number of bugs found as part of our results. Some of these bugs have the same cause and are therefore duplicates. During the field tests, developers looked at our bug reports and flagged duplicates accordingly, which is a difficult task. However, in our closed experiments we only have duplicate information available for known bugs. Bugs that have been discovered by our tool on previous versions and that have no corresponding bug report could be duplicates without us being able to recognize them as such. Of course we compared assertion messages and crash traces to filter duplicates but for some complicated bugs, this method does not work. However, we believe that this subset of bugs (no bug report, no longer present in most recent version) does not contain a high number of duplicates, if any. The reason is that most non-trivial duplicates (different traces/assertion message) are actually memory corruptions that manifest in various different crashes, according to our experiences. Such problems are usually recognized much faster because they have a larger impact on the correctness of the program. It is therefore highly unlikely that our set of discovered bugs contains such a bug without any corresponding bug report being present.

Besides the problem of counting one defect as two or more (duplicate problem), the other direction is also possible. Memory corruptions are often caught by the same assertions, so it is also possible that two distinct defects are only seen and counted as one bug during the experiments. While the duplicate problem could make our results look better than they are, this problem would make the results worse.

Chapter 6

Conclusion

This work provides several contributions to state of the art fuzz testing of language interpreters. First of all, we introduce a syntax-based testing approach that combines pure code generation with test mutation. We outline several benefits such as genericness and low maintenance and provide an implementation that compares well to state of the art: Although our approach is generic, we reach over 50% of jsfunfuzz's effectiveness[1] while the majority of LangFuzz-detected bugs have not been detected by jsfunfuzz.
Furthermore, we tested our tool on the two major browsers Mozilla Firefox and Google Chrome and found over 50 defects in every product/branch we tested. Some of these defects where highly security critical, yielding twelve Mozilla Security Bug Bounty Awards and twelve Chromium Security Rewards. This part of our evaluation clearly shows that our technique is practical and can be an important utility in quality assurance. Both Mozilla and Google have shown interest in adapting LangFuzz (or its technique) into their own toolchains.
Finally, we demonstrate that LangFuzz works on a second language to ensure that our initial success is not solely caused by our initial choice of language. With our proof-of-concept implementation for PHP support, we quickly found several problems in the current PHP engine of which some are clearly security-related with a high impact.

[1] Number of defects found per time unit

Chapter 7

Further Work

While LangFuzz has proven to be helpful in the development cycles of at least two major products (Mozilla Firefox and Google Chrome), we believe that there is still a lot more that can be achieved by extending and improving this technique and we invite all interested researchers to work on this. The following is a list of possible topics that we think could be useful in the future.

7.1 Differential Testing

Differential Testing is another commonly used technique in fuzzing that can be applied whenever several implementations of the same standard (e.g. ES3 JavaScript) are available. The usual problem with fuzzing beyond assertions and crashes is to recognize faulty behavior (i.e. missing test oracle). By using multiple implementations on the same test, different behavior can be detected which should indicate a defect if all engines claim to implement the same standard. Both Rudermann [13] and Yang et al. [18] make use of this technique and LangFuzz should be able to do the same either with multiple engines or different options with the same engine.

7.2 Further Languages and Island Grammars

LangFuzz currently only supports ES3 JavaScript and PHP. By extending the language support in LangFuzz, we can not only investigate further which languages are most suited for our testing techniques but also provide a helpful tool for the further development of that language. Another important aspect are mixed languages, such as HTML and ES3 or ES3 and XML (E4X). For these combinations, the main grammar must be modified to include the second grammar ("island grammar"). We believe that this could be especially helpful in browser fuzzing.

7.3 Generic Semantics Support

During a discussion with Graydon Hoare, a language expert at Mozilla, we came to the conclusion that it should be possible to create fuzzers also on levels between the purely syntactic one like LangFuzz, and the language-specific one like jsfunfuzz: Many languages share common characteristics and semantics (control flow constructs, types, etc.). It could be worth investigating in how far these generic semantic rules that apply for many languages can be used in a semi-generic language fuzzer.

Acknowledgements

Many people directly or indirectly helped me to get this work done and I'm thankful for all the support I received.
First, I would like to thank the many people at Mozilla who provided me with helpful comments, suggestions, information and tools required to get my experiments done and to overall improve LangFuzz. Especially I would like to thank Jesse Ruderman, Gary Kwong and Lucas Adamski for granting me access to jsfunfuzz and providing me with a lot of helpful information on the state-of-the-art at Mozilla. I would also like to thank the whole JS development team at Mozilla, especially Jason Orendorff, Brian Hackett, Jan de Mooij, Andreas Gal, Paul Biggar and all the others who were working on the bugs I reported and that encouraged me to continue the project.
I would also like to thank Google for working together with me on V8/Chrome testing, especially Chris Evans from Google Security for providing me helpful information and feedback as well as Mads Ager and the whole V8 team for working on my bug reports.
Furthermore, I'd like to thank my advisor Kim Herzig and my supervisor Prof. Zeller for granting me to write the thesis on this topic and providing me with lots of helpful feedback on it. I'd also like to thank the staff and alumni at the chair for numerous helpful suggestions, especially Stephan Neuhaus, Gordon Fraser, Kevin Streit and Clemens Hammacher.
Special thanks go to Sascha Just, Maximilian Grothmusmann and Sebastian Hafner for technical assistance and their overall support and friendship. Last but not least I'd like to thank my girlfriend Mado Wohlgemuth for her never-ending support and motivation.
Due to the large number of helpful comments, this list might be incomplete. I apology if I inadvertently omitted any person who would have deserved to be mentioned here.

Bibliography

[1] The peach fuzzing platform. Project website. http://peachfuzzer.com/.

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

[3] Dave Aitel. The advantages of block-based protocol analysis for security testing. Technical report, 2002.

[4] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375607.

[5] Christian Lindig. Random testing of c calling conventions. *Proc. AADEBUG.*, pages 3–12, 2005.

[6] Scott McPeak and Daniel S. Wilkerson. The delta tool. Project website. http://delta.tigris.org/.

[7] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33:32–44, December 1990. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/96267.96279. URL `http://doi.acm.org/10.1145/96267.96279`.

[8] Charlie Miller and Zachary N. J. Peterson. Analysis of Mutation and Generation-Based Fuzzing. Technical report, Independent Security Evaluators, March 2007. URL `http://securityevaluators.com/files/papers/analysisfuzzing.pdf`.

[9] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1855768.1855773`.

[10] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3:58–62, March 2005. ISSN 1540-7993. doi: 10.1109/MSP.2005.55. URL `http://portal.acm.org/citation.cfm?id=1058224.1058339`.

[11] T.J. Parr and R.W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[12] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12:366–375, 1972. ISSN 0006-3835. URL `http://dx.doi.org/10.1007/BF01932308`. 10.1007/BF01932308.

[13] Jesse Rudermann. Introducing jsfunfuzz. Blog Entry. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[14] Guoqiang Shu, Yating Hsu, and David Lee. 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*, FORTE '08, pages 299–304, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68854-9. doi: http://dx.doi.org/10.1007/978-3-540-68855-6_19. URL `http://dx.doi.org/10.1007/978-3-540-68855-6_19`.

[15] Michael Sutton and Adam Greene. The art of file format fuzzing. In *Blackhat USA Conference*, 2005.

[16] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, 2007. ISBN 0321446119.

[17] Brian Turner. Random c program generator. Project website. http://sites.google.com/site/brturn2/randomcprogramgenerator, 2007.

[18] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM SIGPLAN, ACM, June 2011. URL `http://www.cs.utah.edu/~{}regehr/papers/pldi11-preprint.pdf`.

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

[20] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, pages 183–200, 2002.

Appendix

The following pages contain the bug numbers (IDs) of all bugs related to LangFuzz. The bug report usually contains the (minimized) test case, either as an attachment or inlined. Note that some bug reports are not accessible at the time of writing because they were classified as security-critical bugs. These will be released to public after a certain period of time. Furthermore, keep in mind that the given lists also contain all duplicates, both with other tools and our own. A star indicates a bug classified as security-related (we only have this mapping available for Mozilla TraceMonkey).

Bug IDs https://bugzilla.mozilla.org/show_bug.cgi?id=<*BugID*>					
610223	*	612837	*	615657	*
615859		616454		616465	
616491		617139		617405	
617617		617745		617770	
617935	*	619064		619970	
620232		620348		620637	
621068		621121		621123	
621137	*	621202	*	621374	
621432		621988		621991	*
622011		622167	*	623297	*
623301	*	623863	*	624439	*
624455		625685		626345	
626436		626464	*	626521	
626592		627106		629650	*
630048		630064	*	631788	
633122		634593		635195	
635594		635598		635599	*
636879		636889		637010	
637011		637202		637205	
638212		638735		642146	
642151		642154		642157	
642159		642161		642164	
642165		642172		642177	
642772		648438		648746	
649017		649259		649761	
650621		651129		651244	
652415		652438		652439	
653396		655499		661586	
666701		667108		667293	

Table 7.1: Bug IDs for bugs found in Mozilla TraceMonkey

Bug IDs https://bugzilla.mozilla.org/show_bug.cgi?id=<*BugID*>			
642198	642206	642209	642222
642247	642248	642254	642285
642307	642319	642326	642405
642422	642592	642758	642760
642979	642985	642988	643113
643266	643272	643277	643279
643281	643284	643285	643299
643376	643543	643552	643693
644970	645044	645293	645301
645493	645991	646001	646004
646006	646012	646026	646060
646215	646393	646411	646429
646498	646587	646594	647167
647183	647199	647424	647428
647537	647547	647559	648747
648757	648839	648843	648849
648852	648999	649005	649011
649013	649152	649261	649263
649272	649273	649278	649775
649824	649936	649937	650148
650658	650662	650663	650673
651147	651155	651199	651209
651218	651232	652422	653243
653249	653262	653395	653397
653399	653400	653467	654001
654392	654393	654665	654668
654710	655504	655507	655769
655954	655963	655990	655991
656132	656259	656753	656914
657225	657245	657247	657287
657288	657304	657587	657624
657633	657881	658016	658211
658212	658215	658217	658287
658290	658293	658294	658561
658777	659448	659450	659452
659456	659639	659766	659779
659965	660202	660203	660204
660597	661859	662044	662047
662338	663628	663910	664422

Table 7.2: Bug IDs for bugs found in Mozilla TypeInference Branch

Bug IDs https://code.google.com/p/v8/issues/detail?id=<*BugID*>		
1103	1104	1105
1106	1107	1108
1109	1110	1111
1112	1113	1118
1119	1122	1123
1124	1125	1126
1128	1129	1130
1131	1132	1134
1135	1136	1137
1138	1145	1146
1147	1148	1149
1151	1152	1160
1165	1166	1167
1170	1172	1173
1174	1175	1176
1177	1182	1184
1200	1206	1207
1208	1209	1210
1213	1227	1229
1230	1231	1232
1234	1235	1236
1237	1238	1337
1351	1362	1363
1404	1500	1501

Table 7.3: Bug IDs for bugs found in the Google V8 engine