Efficient approach to fuzzing interpreters

Marcin Dominiak
Samsung R&D Institute Poland
m.dominiak@samsung.com

Wojciech Rauner
Samsung R&D Institute Poland
w.rauner@samsung.com

Abstract

Fuzzing has started to gain more recognition over the past years. The basic concept behind it is passing random or otherwise procedurally generated data as input to the tested software. Multiple fuzzers emerged which have an impressive number of bugs discovered. However, their applicability in fuzzing targets requiring highly structured yet arbitrarily nuanced input data, such as interpreters, is questionable. At the same time, an interpreter's input is often untrusted. This, coupled with their widespread usage, makes them a high priority target for security research, especially considering the interpreter's large codebase and high complexity.

In this paper, we introduce the concept of combining coverage guided fuzzing with synthesizing code given arbitrary input data. We will discuss the implementation of this idea in the `Fluff` project and describe its integration with `American Fuzzing Lop` — an open-source fuzzer. We show that by using `Fluff` we can achieve larger code coverage than with current, state of the art solutions.

1 Introduction

1.1 Overview

Fuzzing is a technique used to automatically test software. A basic fuzzer could generate random data and pass it as input to the target. Modern fuzzers often utilize automatic analysis of the program source code or instrumentation in order to gather information about the program and control flow. They use that information as a feedback in order to improve the quality of the test cases. A common metric to measure the quality of an input is, is checking whether the target program behaved in a way which has not yet been observed (for example the program reached a location in the code which has not been visited before, or it crashed). Fuzzers which track executed branches visited across runs and try to maximize this value are called *coverage guided*. Fuzzers can utilize *mutators*, which modify either previously used inputs or a *seed* provided by a user launching the tool, in order to generate new test cases. Often fuzzers utilize multiple approaches, algorithms and heuristics in order to achieve high effectiveness.

Fuzzing, as a technique, has been proven to be highly efficient at discovering bugs and security vulnerabilities. As an example, Zalewski — author of `American Fuzzy Lop` (`AFL`) [1] and Vyukov — author of `syzkaller` [2], display an impressive number of issues discovered by their tools. A wide range of programs is continuously fuzzed by `OSS-fuzz` [3] — an initiative by Google which allows maintainers of open source projects to integrate their codebase with the `ClusterFuzz`, which then automatically synchronizes and tests the software using Google's infrastructure. As of writing this paper, more than 8000 bugs were reported by the `OSS-fuzz` project.

Interpreters of various languages are commonly found in modern software as they allow developers to create a more engaging and interactive user experience. One of the most prominent uses of interpreters are JavaScript execution engines embedded in web browsers. As of March 2019, TIOBE [4], which measures popularity of programming languages, has ranked JavaScript as the 7^{th} most popular technology. Other popular interpreted technologies include Python, PHP and Ruby. All of those technologies are not only widely used, but also highly complex and constantly developed, which makes them perfect security research targets.

Interpreters are a class of software which is difficult to fuzz. The reason for that is the ease with which they reject invalid inputs. On one hand interpreters perform multiple validations on the input in order to reject any erroneous source codes, on the other they accept and execute a very wide range of valid codes. This results in a situation where most test cases generated by a fuzzer are quickly rejected, thus yielding low test coverage.

1.2 Attack scenarios

Exploiting vulnerabilities in an interpreter is especially interesting as it can change the scope of an attack by escaping the restricted execution environment. To visualize the risks connected with security vulnerabilities present in this class of software, we will enumerate several possible attack scenarios:

- Web browsers
  If a malicious entity controls a website (either by legitimately owning it or by compromising an existing service) it can craft JavaScript code which will be executed by default in the browser of each user visiting the webpage. Exploiting the execution engine and browser could lead to the attackers gaining complete control over the visitor's system.

- Online execution services
  Some online services offer a functionality to execute code provided by the user (for example `ideone` [5]). Such services can be used to quickly test, share code or explain certain features. While those services take great measures to make sure that executing arbitrary code does not compromise their safety, exploiting one of the interpreters might allow an attacker to gain control of the system or gather more information about its internals.

- Continuous integration systems
  Those systems are integrated with version control software and allow developers to define tasks which are to be executed periodically. A common task is building the application and running tests. A malicious user could craft code specifically to exploit the execution engine in order to take control or otherwise attack the system itself, or even the machines running it.

- Social engineering
  An attacker could try to post malicious code to online developer forums (like StackOverflow [6]) and ask for help, or otherwise try to trick users to execute his specifically crafted code, hoping to exploit and compromise systems of people thinking that they are helping someone.

1.3 Related work

Grammar based approaches require a user-provided formal grammar and generate a script which will output test cases compliant with that grammar. A sample tool utilizing this approach is `Grammarinator` [7]. Although this approach does not support any sort of feedback loop by design, it can be implemented in a way which does. Vegard has described in his blog post [8] his approach to fuzzing GCC [9] compiler, which is also grammar based but uses `AFL` as the backend and therefore utilizes a feedback loop. This approach is flexible, because one can create an input grammar for many programming languages. However, it is impossible to enforce certain properties, which are sometimes desired, using context free grammars (further elaborated in appendix A.1).

Grey-box fuzzing is a technique of incorporating information about the target to the fuzzer. In the case of fuzzing interpreters, this can be done by providing keywords or whole code snippets which the fuzzer incorporates into the generated test cases. `AFL` can work in this way if a user provides it with a list of keywords. Another well known tool in this category is `jsfunfuzz` [10], which works by constructing JavaScript code snippets and executing them. Those snippets are generated using predefined templates, magic values, keywords and hard-coded randomness. This project does not use any instrumentation nor symbolic execution. It is also tightly coupled with JavaScript and in particular, with Mozilla's `SpiderMonkey` engine. `Langfuzz` [11] is another fuzzer in this category, which utilizes both generative and mutative approaches in order to create new test cases. It is however a Mozilla internal tool. This approach can be effective, as it is easier for this type of fuzzers to create a valid program and pass parser, thus reaching backend of the interpreter. Unfortunately, fuzzers in this category share the downsides of both grammar based approaches and generic mutative fuzzers, as they are either limited by the hardcoded values and snippets (as is the case with `jsfunfuzz`), or not limited enough and can't reliably pass parser (like `AFL` with dictionary).

csmith [12] is a tool which synthesizes completely valid C programs and passes them for compilation and execution in multiple compilers. The outputs are then executed and the behavior of generated files is monitored in order to spot errors in logic. Any crashes during the compilation process are also stored for analysis. `csmith` is a mature and highly efficient tool, however, it does not cooperate with any fuzzer, nor employ any mutative approach, which could increase its effectiveness. `csmith` focuses on finding bugs in logic, which cause the compiler to generate incorrect machine code. It also supports only C language.

Formal verification could allow to ensure, in a provable way, that a piece of software is secure and correct. It is still difficult to perform formal verification of projects as big as interpreters and compilers due to the amount of required work and computational power, however, there is an ongoing effort to create a fully verified compiler — *compcert* [13]. We are not aware of any ongoing effort aiming to formally verify any existing interpreter.

1.4 Paper outline

Section 2 describes different levels of fuzz testing. Section 3 introduces `Fluff`, which is our contribution to this field. That section contains description of its design, implementation and limitations, as well as extensions and design choices. Section 4 describes the

methodology used to evaluate selected approaches to fuzzing. Section 5 presents results of our research and comparison of `Fluff` with other approaches to fuzzing interpreters. Finally, section 6 summarizes findings of our work, as well as presents several further research ideas.

2 Fuzzing interpreters

In fuzzing we can differentiate between multiple levels of increasing test case quality [14]. Below is the list of levels we are interested in when fuzzing interpreters, especially JavasScript runtimes.

Level 0. Sequence of bytes. This is the most basic level, equivalent of the passing unix `/dev/urandom` as the input to the tested program. This approach is especially useful when practiced on targets, that do not require highly structured input, e.g. audio/video codecs. In the context of fuzzing interpreters this technique is effective in fuzzing components which perform initial processing of the runtime code — parsers and lexers. We have found several bugs in those components, for example ones that occurred during parsing unicode encoded characters in the program source code. Some runtimes allow the bytecode (representation of parsed source code) for runtime virtual machine to be executed, in this case tests at this level can also be used. This feature in most cases is meant for the runtime developers, and is not enabled in e.g. web browsers by default. Examples of fuzzers which operate at this level are `AFL`, `radamsa` [1, 15].

Level 1. Sequence of ASCII characters. This is a simple variation of Level 0 which is not interesting in our case. Usage of only ASCII characters increase the probability of constructing valid keywords and thus passing the parser, but makes it impossible to detect errors mentioned in description of level 0.

Level 2. Sequence of words, separators and white space. Using this approach fuzzer can get past parsing and lexing with more test cases. Unfortunately, most of the test cases still will not be a syntactically correct programs, so the interpreter will throw an exception and stop the execution. This level is an equivalent of fuzzing with prepared dictionary. Success rate of the test cases is strictly correlated with the size and quality of the dictionary. As an example, `AFL` has an option to fuzz with specified dictionary of keywords to incorporate in the test cases [1].

Level 3. Syntactically correct programs. Test cases at this level are correct JavaScript code, utilize various features of the language and therefore can find bugs in the exposed runtime logic. The family of fuzzers which can generate this test cases are grammar fuzzers, e.g. `grammarinator` [7].

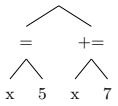

Figure 1: Decomposition of the sample program into AST

Apart from syntactical correctness we want to enforce semantical correctness, in order to make sure, that the generated test cases are not only a valid JavaScript code, but can also be executed by the interpreter.

JavaScript runtimes are complex targets, therefore coverage based fuzzing can be employed to help the fuzzer find more bugs. In our work we evaluate the effectiveness of combining generation of semantically correct test cases with coverage guided fuzzing.

3 Fluff

3.1 Concept

Our goal was to create a fuzzer which could synthesize semantically correct JavaScript source code and pass it for execution using a chosen interpreter. Due to the high number of interpreters and lack of unified API to execute code, easy integration with new target interpreters was also a design goal for us. Throughout this section we will describe how `Fluff` works and the types of generated test cases.

Our tool was designed as a translation layer, working between `AFL` fuzzer and the interpreter. Thanks to this, `Fluff` benefits from `AFL`'s instrumentation, tooling and test harness.

3.2 Synthesising arithmetic expressions

Our approach to synthesizing code from arbitrary data is a bit similar to the process of parsing code written in a programming language. First, the parser performs *lexing* (also called *tokenization*) which breaks down the code into *lexemes* (*tokens*) such as keywords, identifiers etc. Those lexemes are later parsed to create an object abstracting a tree structure of the given source code. Such trees are called *Abstract Syntax Trees* (*AST*). Figure 1 shows an AST of a program demonstrated in listing 1.

```
1 x = 5
2 x += 7
```

Listing 1: Sample program

The tree generated by compilers can be serialized just like any other tree structures, for example using prefix left-to-right traversal.

Our idea is, in a sense, to reverse the work of a parser. Our tool takes as input a serialized AST of a certain program, which we first deserialize and reconstruct the tree, and then emit high level code. Since programs are highly structured, we make our deserializer as permissive as possible, in order to accept and successfully generate code for a wide range of inputs.

We will present a simpler version of `Fluff` — a program which given arbitrary input generates valid, simple arithmetic programs in a language consisting of integer literals, addition, multiplication and variables. Sample programs in that language (which is actually a subset of JavaScript) is proposed in listing 2.

```
1 var x
2 x = 2 + 10
3 x * 40
```

Listing 2: Sample program in a subset of JavaScript

Take into consideration that even such a simple language cannot be defined using a context free grammar (see appendix A.1). Therefore we rely on simple decompositions of languages and provide support for more complex features by design. In this case, a sample grammar/decomposition is presented in figure 2.

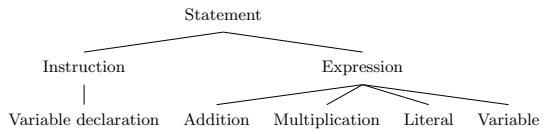

Figure 2: Abstract Syntax Tree for arithmetic

We will use the syntax tree presented in figure 3 to parse input data. The first step is to label each edge with a natural number, such that each vertex has its edges labeled 0, 1, 2, ...

There exist multiple valid labelings of the presented tree, and it does not matter which labeling is chosen. We will use the labeling presented in figure 3.

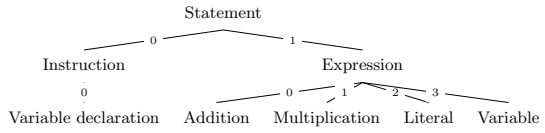

Figure 3: Labeled Abstract Syntax Tree for arithmetic

The labeling is now used to parse input. `Fluff` reads the input byte by byte, reading a number between 0 and 255. This value is used to move in the labeled AST until a leaf is reached. Reading a number x while being in vertex v_1 results in transition to vertex v_2 if there exists an edge from v_1 to v_2 with label $x \mod deg^-(v_1)$.

A sample chain of transitions for the following input: `0x01, 0x07` is presented in figure 4.

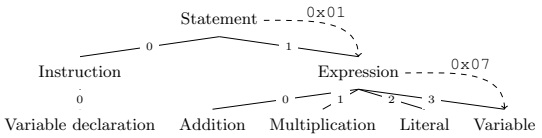

Figure 4: Parser state transition chain when reading input

Once the parser reaches a leaf, it takes actions specific to the current state. We will present sample pseudocode implementations for leafs in the sample tree. Helper functions will be used without presenting implementations. Their role is as follows:

- *ReadByte*() — read and return a single byte from input
- *GetExpression*() — set the parsing state to *Expression* and return the parsed expression
- $AllocateVariableId()$ — returns a unique id for a variable
- $GetVariable(n)$ — return n-th variable identifier

Implementations for the leafs could be defined as shown in listings 3, 4, 5, 6 and 7.

```
1   id = AllocateVariableId()
2   Emit("var " . id)
```

Listing 3: Implementation of Variable declaration leaf

```
1 lhs = GetExpression()
2 rhs = GetExpression()
3 Emit(lhs)
4 Emit("+")
5 Emit(rhs)
```

Listing 4: Implementation of Addition leaf

```
lhs = GetExpression()
rhs = GetExpression()
Emit(lhs)
Emit("*")
Emit(rhs)
```

Listing 5: Implementation of Multiplication leaf

```
value = ReadByte()
Emit(value)
```

Listing 6: Implementation of Literal leaf

```
number = ReadByte()
variable_id = GetVariable(number)
Emit("var " . variable_id)
```

Listing 7: Implementation of Variable leaf

Finally, note that there exist inputs such that the last instruction is unfinished or otherwise malformed — an example of such a situation in our arithmetic language is an input consisting of a single byte `0x01`. The parser would not be able to construct a valid expression without more data. Similarly, the input `[0x01, 0x02, 0x10, 0x01, 0x00, 0x02, 0x17]` would correctly deserialize the first instruction as a single integer literal 16 followed by a malformed addition (due to the lack of the right hand side operand). At first we have decided to ignore this problem and simply reject the malformed instruction. However, after some testing we have discovered that the fuzzer often creates a very deep expression and runs out of data to finish it, resulting in unexpectedly short outputs compared to the number of input bytes. Therefore, we have implemented default values for instructions. As a result, an input consisting of only `[0x01, 0x00]` would be transformed into $(0 + 0)$.

3.3 Extensions and limitations

The basic idea presented in the previous section can be extended to generate a real programming language, in our example JavaScript. It is important to note, that there are many ways to configure the language generation and deciding between those possibilities may have a big impact on the results of research. Another important thing to note is that generating the actual code is highly dependent on the target language. This approach will be applicable for most imperative programming languages, but may not be as meaningful for other paradigms like functional or logic.

```
  number_of_instructions = ReadByte()
  number_of_arguments = ReadByte()
  function_id = AllocateFunctionId()
  Emit("function " . function_id ."(")
  for i in 0..number_of_arguments:
    argument = AllocateVariableId()
    Emit(argument . ", ")
  Emit(") {")
  for i in 0..number_of_instructions:
    instruction = GetInstruction()
    Emit(instruction)
  Emit("}")
```

Listing 8: Implementation of function code generation

In order to produce full-fledged JavaScript code, several features must be considered:

1. Types
   Depending on the target language, types can play an important role. In the case of JavaScript, the type system is permissive, so we did not have to worry about making sure, that for example we are not summing integers with strings or treating floating numbers as boolean values. However, in languages with stronger type systems correctly handling types must be implemented. Our idea to solve this issue is to implement a special register which tracks types of variables, functions and arguments and can be queried for variables or functions returning certain type. Another challenge would be type casting, which allows implicit or explicit casts between different types. A graph-like structure could be used to track this information.

2. Functions
   Functions can be treated similarly to how we approached variables in section 3.2. In JavaScript, functions do not have a fixed number of arguments — you can declare a function with 2 arguments and call it without providing any, or declare a function with no arguments but call it with one, which can later be accessed using the `arguments` variable. Because of JavaScript permissive type system, the program does not have to take into consideration function arguments types or return value type. In our case, the number of arguments and the number of instructions in a function is read by the parser as a single byte. We also do not force return statements, as shown of listing 8.

3. Classes
   JavaScript since ECMAScript 6 contains classes, which in fact are just *syntactic sugar* on top of the function declaration [16]. We treat classes very

similarly to functions — we also simply read the number of methods and fields from input and simply treat the next instructions as part of the class. We also remember which methods have been declared, to allow calling them.

4. Name scopes
   Some identifiers have limited scope — for example function arguments are not visible outside of that functions and variables declared inside one block may not be visible in another. This is more complicated in JavaScript since ECMAScript 6 standard [16], as variables have different visibility depending on how were they defined — whether it was declared with `let`, `const` or `var`. Choosing how to handle this correctly may increase performance or coverage. We have decided to not use variable shadowing, as using it would not increase the number of discovered issues, but only complicate the code of our project.

5. Target interpreter portability
   As mentioned in section 3.1, we wanted to be able to test multiple execution engines. Our solution is using a simple interface for interacting with and abstract JavaScript execution engine and prepared a specific implementation for each of the test targets. This allowed us to abstract away interactions with each specific interpreter, while still allowing to fuzz multiple targets.

From those descriptions we can start to notice the following limitations:

- Integer literals are not greater than 255,
- String literals are not longer than 255,
- Code blocks (functions, loop bodies, etc.) are not longer than 255 instructions,
- No incorrect bracketing is possible,
- Variable and function names follow a set pattern.

Despite those limitations, it is possible to achieve big numbers by using arithmetic, long strings can be achieved by concatenating multiple strings together and function and variable names do not play any role in execution and may only be useful during exploitation. At the same time, incorrect bracketing may sometimes trigger bugs, but it is something that a pure `AFL` fuzzer, without `Fluff` as the middleware, is capable of testing.

3.4 Built-ins

A significant coverage boost can be achieved by allowing `Fluff` to generate code which uses built-in objects and functions, such as the aforementioned `arguments`, or functions like `filter`, `parse` etc. Usually there are also some built-in objects, like `Date` or `JSON`. Those objects, while not being part of the language grammar, should be utilized. We achieve this by feeding them into the sets of variables, objects and functions. Many of the errors discovered by `Fluff` were related to using some of those built-ins.

Approaching the task of composing this list is specific to each interpreter and language. For JavaScript runtimes we were able to write a script that uses `Object.getPrototypeOf` and `Object.getOwnPropertyNames` to iterate through all built-ins available in the runtime. We filled missing ones using documentation for the interpreter.

3.5 Corruptions

Having constructed the syntax tree in memory, we are able to perform arbitrary modifications to it. The approach described up to this point allows us to generate valid programs. However, we wanted to introduce a way to corrupt the generated code (by *sometimes* introducing undefined identifiers, placing a series of strange bytes between keywords or otherwise). Being able to produce corrupted statements in our generated code can help us with fuzz testing some parts in interpreters, specifically the parser and lexer.

One way to implement this feature is to include such corrupted statements in our simplified grammar as new entities. However, this would cause a blow-up of our codebase, as each statement is represented by a separate class and has custom parsing logic implemented for it. We also find this approach inelegant, as we want the language description to be as simple as possible.

Instead, we decided to approach implementing this feature in the following way: after parsing and building any statement we read an extra byte, which we call *corruption byte*, and use it to decide whether this particular statement should be corrupted, and if so — how. We can thus model parsing in the way shown on listing 9.

```
1 statement = ReadStatement();
2 corruption_byte = ReadByte();
3 statement.Corrupt(corruption_byte);
```

Listing 9: Handling of the corruption byte

This approach allows us to not only keep the grammar simple and minimal, but also allows us to control the probability with which the corruptions occur, as well as include multiple types of corruptions for each statement. What the `Corrupt` function actually does is highly dependent on the statement which is being corrupted. For example, a

corrupted integer literal could be a very big number, or it might contain a non-numerical character in it. It could be implemented as shown on listing 10

```
ExprIntLiteral::Corrupt(char corruption) {
  if (corruption < 200) {
    return;
  }
  if (corruption % 5 == 0) {
    this->value = VERY_BIG_VALUE;
  }
  if (corruption % 7 == 0) {
    this->value .= NON_ASCII_CHARACTER;
  }
}
```

Listing 10: Corruption implementation in the Integer class

It might be beneficial to have some test cases generated without corruptions at all. Using this approach it is highly unlikely that not a single statement will get corrupted, which may decrease the overall effectiveness of fuzzing if the corruptions cause the generated test cases to be rejected without execution or other deeper processing. We have decided to remedy this by treating the first byte of input (which we call *sanitization byte*) and treating it specially — based on its value we can choose to not include any corruptions. Therefore the final version of parser could work in the way described on listing 11.

```
global sanitization_byte = ReadByte();

...

statement = ReadStatement();
if (sanitization_byte % 3 != 0) {
  corruption_byte = ReadByte();
  statement.Corrupt(corruption_byte);
}
```

Listing 11: Sanitization byte implementation

3.6 Integrating Fluff with American Fuzzy Lop

In order to use `Fluff` in a real life scenario a testing harness and a data generator must be provided. We recommend to use `AFL` with `Fluff`, because it can provide not only the aforementioned components, but also its own instrumentation which can be useful in finding new, interesting test cases.

Example setup used in evaluating our project is shown on figure 5. In this setup `AFL` is responsible for repeatedly invoking `Fluff` binary, detecting crashes and hangs and generating and mutating input data. Data from `AFL` are passed to `Fluff`, and using input

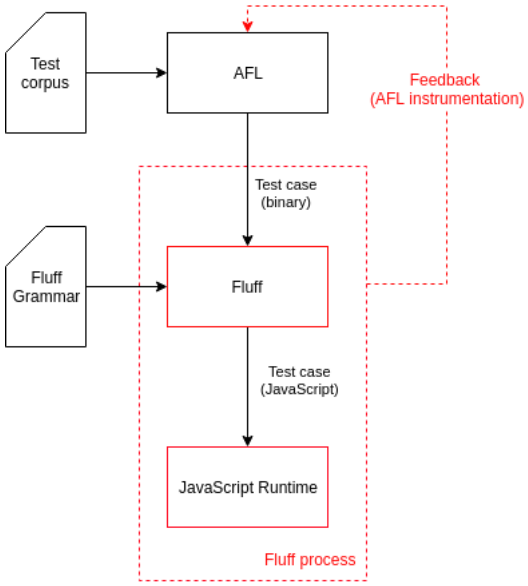

Figure 5: `AFL` and `Fluff` integration

grammar file transformed into JavaScript code, which is being executed in JavaScript runtime. Another benefit of using `AFL` is its built-in timer, which automatically detects and terminates test cases on timeouts.

Because both `Fluff` and JavaScript runtime are built with `AFL` wrappers over C/C++ compilers, they have `AFL` own instrumentation. During execution `AFL` collects feedback which consists of paths taken by the program during the execution. This kind of instrumentation is especially useful with `Fluff`, because it will help `AFL` generate payloads triggering every possible path in `Fluff` parser. This results in producing JavaScript test cases that use various features of the target language.

Because we designed `Fluff` with modularity in mind, it is easy to change JavaScript runtime. We provide examples of `Fluff` interfaces to many JavaScript runtimes (including `ChakraCore`, `jerryscript` and others). It is also possible to invoke the target runtime using system functions, such as `execve`.

4 Evaluation

4.1 Target selection

From the available JavaScript runtimes we selected `ChakraCore` to benchmark Fluff performance. It is a full-fledged JavaScript runtime with a handful of features, made by Microsoft corporation.

Open sourced. `ChakraCore` is an open source project, therefore we can measure code coverage, which

is a useful metric to compare Fluff with other tools. Furthermore, it allows us to perform quicker assessment of found crashes, in order to determine their exploitability.

ES2015 support. ChakraCore supports most of the ECMAScript 6 features, that makes it relevant and enlarges the possibility of bug occurrence [17]

Large codebase. As measured by llvm-gcov, ChakraCore on Linux x64 (git commit SHA1 22f0a427031bcdab990b59ceead8bfa329166a49) has 225960 lines of code. Compared to other engines like JerryScript (21717 lines of code), it is a larger attack surface. One drawback from large number of source code lines is slower execution of build with coverage enabled.

Relevant. ChakraCore is used in various products, including Microsoft Edge, Universal Windows applications, Azure Document DB and others [18].

Embedable. Developers can easily embed ChakraCore as JavaScript runtime in their applications, because of that it is easy to integrate Fluff with ChakraCore.

Easy to build. We had no issues with building ChakraCore in reasonable time for our tests.

4.2 Performance metrics

There are many performance metrics that can be used to benchmark fuzzers. The ideal parameter used for comparison is crashes identified in predefined amount of time [19]. From our trial runs on various JavaScript engines we noticed that achieving substantial amount of crashes took too much time, and therefore we abandoned this approach.

Another metric that can be used to benchmark performance of fuzzers is the generated code coverage of the tested target. This metric differentiates Fluff from non-grammar fuzzers like AFL. Using this metric we can check how many features of the language test cases generated by Fluff are utilizing. Another argument in favor of using code coverage as a metric is that fuzzing will not find a bug that exists in non-executed line of code. Therefore we assume that the higher percentage of code coverage we get from testing, the better chance of finding a crashing input.

Because of the nature of the target and limited time we decided to use this metric for comparison. Each fuzzer will be measured by running for 25 hours on selected target.

4.3 Fuzzers selection

As the first fuzzer to measure performance of Fluff against we decided to use AFL, which has been described before in section 2.

In order to show other features of AFL, we will also compare Fluff performance to AFL using JavaScript dictionary, provided by the author. This feature can help AFL generate valid JavaScript test cases, that do not fail on parsing, but instead can be executed. Sample JavaScript dictionary is provided with AFL source code.

From the available grammar fuzzers we have selected a previously described (in section 1.3) grammarinator [7] to compare its performance to proposed solution. It is an open source fuzzer utilizing ANTLR v4 grammars, many of which are available as open source projects — including those for JavaScript. Output from grammarinator should be correctly parsed and executed in JavaScript runtime. On the other hand, grammarinator does not utilize the same feedback mechanism as AFL, so this fuzzer cannot adjust its input data to cover new paths in the target binary.

We compare the code coverage produced by Fluff and the selected fuzzers (AFL, AFL with dictionary and grammarinator) in order to evaluate our approach.

4.4 Fuzzing code with coverage

In order to measure the code coverage, target must be built with the coverage support. This is handled by adding specific compilation and linking flags to the building process. This produces an executable that is instrumented in such a way, that during execution detailed information about source code lines the program executed is stored in a separate file, unique for every source code file. To get a full result we can use a tool such as gcovr to get summary coverage [20].

This method combined with fuzzing imposes a significant problem. Because fuzzed binary is accessing large amount of files with coverage information, it slows down execution rate significantly. For the chosen target we observed approximately 100 times slower execution.

To perform benchmarking using code coverage as a metric with much higher execution speeds we decided to use different approach. The testing procedure was split into two steps:

1. Fuzzing — 25 hours of fuzzing target binary without coverage, with AFL instrumentation,
2. Code coverage — calculating code coverage of target binary.

We performed fuzzing with AFL on the target binary without code coverage, and measured code coverage by executing target binary with code coverage enabled for every test case generated by the fuzzer, saved in the queue folder. According to AFL documentation, queue folder consists of files from input directory (test corpus) and mutated inputs that discovered new paths in fuzzed binary [21].

4.5 Fuzzing with grammarinator

A challenge while fuzzing using `grammarinator` is the fact that it requires a custom test harness, as out of the box this fuzzer only generates test cases. This fuzzer also does not accept seed from the user. To give this fuzzer a fair comparison with others, we calculated approximate number of the test cases other fuzzers were generating (based on the observed execution speed) within 25 hours. This value approximated at 3 456 000. Because of the slower execution of target binary with code coverage enabled, we decided to use `AFL` tool `afl-cmin` on this test corpus and target binary without code coverage, in order to reduce test corpus size. Then we performed code coverage calculation only on the test cases, that discovered new paths, and therefore increased code coverage.

4.6 Experimental setup

Because of the random nature of fuzzing, each fuzzer was tested on 24 independent runs for 25 hours. Using Mann-Whitney U test we calculated the p-value to make sure that results are statistically significant, as described in section 4.2.

All tests and compilations were run on `Ubuntu` version 16.04 (x64). `AFL` (version 2.52b) was built using `clang` version `7.0.0 (trunk 337895)`. Both target `ChakraCore` and `Fluff` were built using `afl-clang-fast++`.

Fuzzing was running in parallel on a virtualized machine with 24 cores (CPU Intel Xeon Gold 6130) and 48 GB of RAM. The resources were shared between 24 fuzzer instances.

5 Results

Figure 6 shows the measured code coverage. On average, after fuzzing the target for 25 hours on 24 independent instances, `Fluff` had the largest code coverage.

	$\bar{x}$	$\bar{x}$ [%]	σ
`Fluff`	60282	27	7874
`AFL 2.52b` with dictionary	54744	24	8563
`grammarinator 18.10`	42741	18	463
`AFL 2.52b`	41094	18	1174

Table 1: Results after 25 hours of fuzzing, x — number of lines covered, σ — standard deviation

Comparing `Fluff` with its closest competitor — `AFL` with dictionary, gave the result of $p = 0.001$, which means that obtained results are significant.

The standard deviation values in the case of `Fluff` and `AFL` with dictionary are larger than `AFL` and `grammarinator`. The high deviation values are expected, as they reflect the process of discovering interesting test cases and mutating them further to discover more paths. We observed high increases of coverage on some of the instances, meaning that they managed to construct an interesting test case and used it to discover multiple paths. Such test cases were semantically correct programs or usages of a new feature.

The slow increase of coverage in the case of AFL visualises the difficulty of constructing a syntactically correct program without utilizing prior knowledge. Low deviation in the results from `grammarinator` reflect the lack of feedback loop, which does not allow the fuzzer to focus on interesting test cases.

During the test there were no crashes reported by any of the fuzzer. During the development phase of `Fluff` we discovered multiple bugs in various JavaScript runtimes, as shown in appendix A.2.

6 Summary

6.1 Conclusions

Interpreted programming languages are very common in modern software. Due to their widespread usage, discovering and exploiting vulnerabilities of their interpreters poses a high security risk to their users. At the same time, interpreters are often very complex projects, including not only parser and runtime environment, but often also garbage collectors, libraries providing additional functionalities or even JIT compilers. Because of that, fuzzing operating at level 0 or 1, can be ineffective.

In this paper, we presented a novel approach to fuzzing interpreters, which uses both generation of syntactically valid code and coverage guidance. This approach can be used on a wide range of software, which has been difficult otherwise.

Apart from being well suited for testing software that requires highly structured input, `Fluff` provides the developer with complete control over the generated test cases. Moreover, the presented methodology that `Fluff` is based upon is generic and can be applied not only to interpreters, but also compilers or any other type of software, which has a strictly defined input form. Our implementation of this approach in the `Fluff` project proved the effectiveness of this approach by discovering numerous issues in several JavaScript interpreters A.2.

Trials using `Fluff` and other fuzzers have shown, that after 25 hours of fuzzing this approach results in larger code coverage.

The idea that we presented will allow developers and other researches to thoroughly test and verify the security of interpreters, compilers and other similar software. We also think that this work may encourage re-

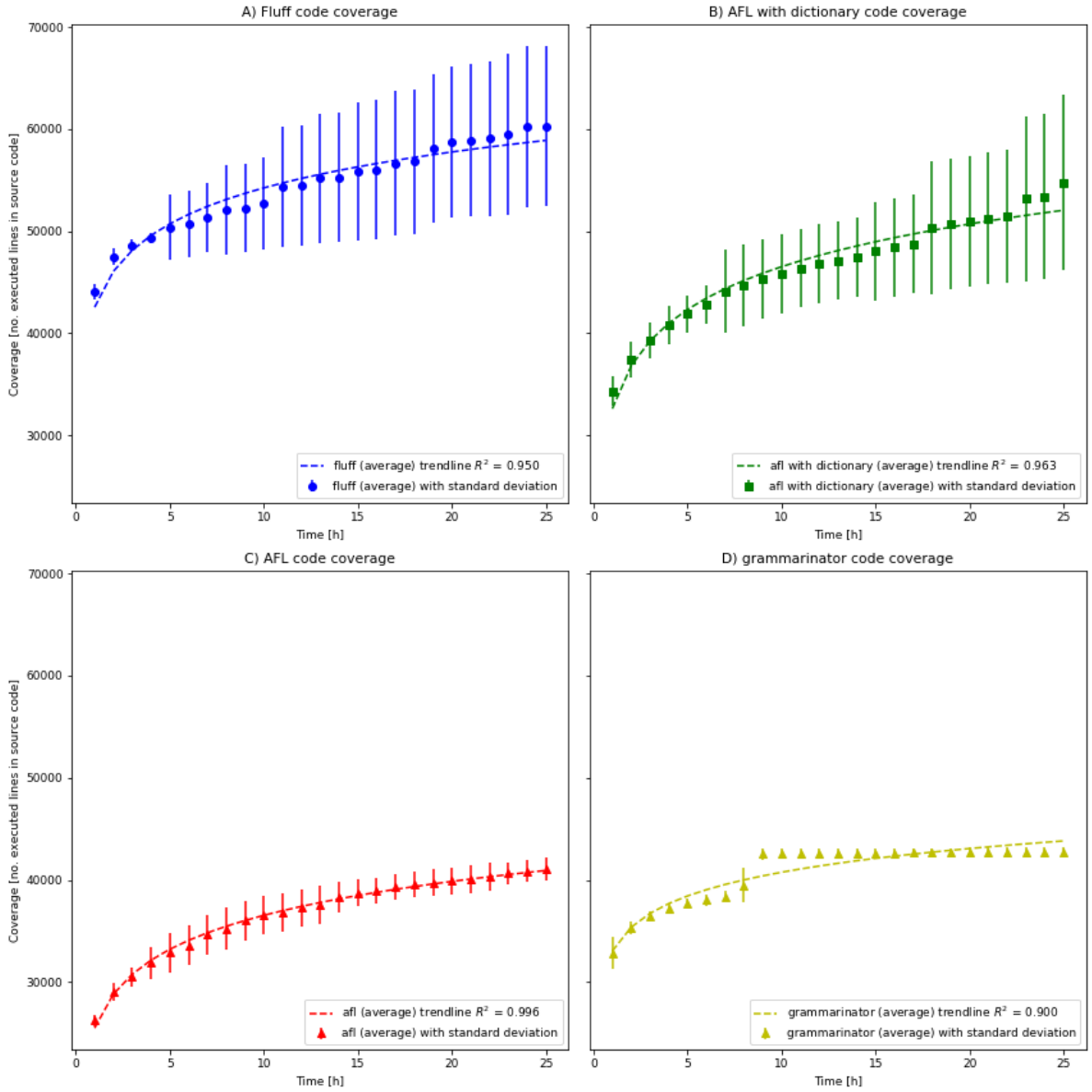

Figure 6: Code coverage measured for 25 hours of fuzzing with A) `Fluff`, B) `AFL` with dictionary, C) `AFL`, D) `grammarinator`. All trendlines were approximated to $y(x) = a + b * log_{10}(x)$, where A) $a = 42556$, $b = 5073$, B) $a = 32626$, $b = 6035$, C) $a = 25512$, $b = 4790$, D) $a = 33054$, $b = 3347$

searchers and security engineers to use fuzzers in creative ways, and not rely only on the tool smartness to discover vulnerabilities.

6.2 Future work

In this section, we will present our ideas which could be researched, tested and implemented.

1. Testing different languages
   Our implementation of `Fluff` is tightly coupled with JavaScript. The design however (both high- and low-level) is generic and applicable to any programming language. Implementing this project for different technologies would allow to discover bugs and vulnerabilities in other popular interpreters.

2. Language agnostic implementation
   Implementing this approach in a language agnostic way is a natural generalization. It is more difficult than the previous proposal, but if done correctly, it would allow to quickly test multiple technologies, including interpreters and compilers of various languages, but also parsers of structured documents (XML, YAML, etc).

3. Extend functionality to compilers
   Our aim was to test runtimes of interpreters. However, our approach can be used to test compilers. In such case, the generated code could be compiled and the fuzzer would check if it crashed, without running the generated code.

4. Verifying standard compliance
   It could be possible to include logic into `Fluff` which would analyze the generated code and assess what the interpreter (or different target) should do with it, i.e. what errors should be raised or how should the output look like. Verifying whether those expectations are met would allow to detect behaviors which are not compliant with standard or specification.

5. Cross validation
   This is a similar idea to the previous one, but instead of implementing the logic of predicting behavior of a given code snippet, one could use the same test case on multiple targets and check if they behave in a similar way. Any differences between executions could be indications of bugs or lack of standard compliance.

6. Continuous fuzzing
   As mentioned in section 1.1, `OSS-fuzz` is an effort to continuously fuzz open source software. Because `Fluff` does not depend on the backend fuzzer, and our targets are open source projects, it could be possible to use `Fluff` as a middleware on the infrastructure used by this project, thus increasing the effectiveness of `OSS-fuzz`.

Acknowledgments

We would like to thank those, who have helped conduct this research, as well prepare this paper and the presentation:

1. Botwicz Jakub, Ph.D
2. Agria Andrzej
3. Grzelewski Bartłomiej
4. Kowalski Paweł
5. Opasiak Krzysztof
6. Pszeniczny Krzysztof

References

[1] Michał Zalewski. *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl/.

[2] Dmitry Vyukov. *syzkaller*. URL: https://github.com/google/syzkaller.

[3] Mike Aizatsky et al. *OSS-fuzz*. URL: https://github.com/google/oss-fuzz.

[4] *TIOBE Index*. URL: https://www.tiobe.com/tiobe-index/.

[5] Sphere Research Labs. URL: https://ideone.com/.

[6] URL: https://stackoverflow.com/.

[7] Renata Hodovan, Ákos Kiss, and Tibor Gyimóthy. *Grammarinator: a grammar-based open source fuzzer*. Nov. 2018. DOI: 10.1145/3278186.3278193.

[8] *Compiler fuzzing, part 1*. URL: http://www.vegardno.net/2018/06/compiler-fuzzing.html.

[9] *GNU Compiler Collection*. URL: https://gcc.gnu.org/.

[10] Jesse Ruderman et al. *jsfunfuzz*. URL: https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz.

[11] Christian Holler, Kim Herzig, and Andreas Zeller. *Fuzzing with Code Fragments*.

[12] Xuejun Yang et al. *Finding and Understanding Bugs in C Compilers*.

[13] *CompCert*. URL: http://compcert.inria.fr.

[14] William M. McKeeman. "Differential Testing for Software". In: *Digital Technical Journal* 10.1 (1998).

[15] Aki Helin. *Radamsa - a general purpose fuzzer*. URL: https://gitlab.com/akihe/radamsa.

[16] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 6th ed. 2015. URL: https://www.ecma-international.org/ecma-262/6.0/.

[17] *ECMAScript compatibility table*. URL: https://kangax.github.io/compat-table/es6/.

[18] Nawaz Dhandala. *Open Sourcing Chakra Core by Microsoft and what it means to the Javascript community*. 2016. URL: https://blog.cloudboost.io/open-sourcing-chakra-core-by-microsoft-and-what-it-means-to-the-javascript-community-92217245924.

[19] George Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18* (2018). DOI: 10.1145/3243734.3243804. URL: http://dx.doi.org/10.1145/3243734.3243804.

[20] *gcovr*. URL: https://gcovr.com/.

[21] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.

A Appendix

A.1 Programs are not context free

In this section we present a proof that the language of programs which do not contain errors such as usage of undefined identifiers is not context free.
For this proof, we will work with the C language, but analogous proofs can be made to match syntaxes of other programming languages.

Proof. Assume that the language of C programs which compile (denote L) is in fact context free.
Now, recall that an intersection of a context free language and a regular language is a context free language.
We will use the following regular expression:

$$void\ f()\ \{int\ aa^*;\ aa^* = aa^*;\}$$

to intersect it with L. The resulting language is

$$\{void\ f()\ \{int\ a^k;\ a^k = a^k;\} \mid k \in \mathbb{N}, k > 0\}$$

which is isomorphic to

$$\{a^k b a^k b a^k \mid k \in \mathbb{N}\}$$

which is known to not be context free.
This contradiction proves that L is not context free. □

A.2 Discovered issues

Product	Underlying issue (linking to the bug tracker)
jerryscript	Uncontrolled recursion
jerryscript	**Null pointer dereference**
jerryscript	Heap buffer overflow
Espruino	Memory leak
Espruino	Uncontrolled resource consumption
Espruino	Buffer overread
Espruino	Buffer overread
Jsish	**Buffer overread**
Jsish	**Null pointer dereference**
Jsish	Assertion reachable
Jsish	**Buffer overwrite**
Jsish	Buffer overread
iv/lv5	Uncontrolled recursion
iv/lv5	Uncontrolled resource consumption
Jsish	**Buffer overread**
Jsish	Null pointer dereference
Jsish	Null pointer dereference
Jsish	Buffer overread
Jsish	Null pointer dereference
Jsish	Uncontrolled recursion
Jsish	Assertion reachable
ChakraCore	Heap buffer overflow (no public listing)
Jsish	Buffer overread

The bugs with issue types in bold have been assigned the following CVE numbers (in order): CVE-2018-1000636, CVE-2018-1000655, CVE-2018-1000661, CVE-2018-1000663, CVE-2018-1000668.