Jit-Picking: Differential Fuzzing of JavaScript Engines

Lukas Bernhard
Ruhr-Universität Bochum
Germany
lukas.bernhard@rub.de

Tobias Scharnowski
Ruhr-Universität Bochum
Germany
tobias.scharnowski@rub.de

Moritz Schloegel
Ruhr-Universität Bochum
Germany
moritz.schloegel@rub.de

Tim Blazytko
Ruhr-Universität Bochum
Germany
tim.blazytko@rub.de

Thorsten Holz
CISPA Helmholtz Center for
Information Security
Germany
holz@cispa.de

ABSTRACT

Modern JavaScript engines that power websites and even full applications on the Web are driven by the need for an increasingly fast and snappy user experience. These engines use several complex and potentially error-prone mechanisms to optimize their performance. Unsurprisingly, the inevitable complexity results in a huge attack surface and various types of software vulnerabilities. On the defender's side, fuzz testing has proven to be an invaluable tool for uncovering different kinds of memory safety violations. Although it is difficult to test interpreters and JIT compilers in an automated way, recent proposals for input generation based on grammars or target-specific intermediate representations helped uncovering many software faults. However, subtle logic bugs and miscomputations that arise from optimization passes in JIT engines continue to elude state-of-the-art testing methods. While such flaws might seem unremarkable at first glance, they are often still exploitable in practice. In this paper, we propose a novel technique for effectively uncovering this class of subtle bugs during fuzzing. The key idea is to take advantage of the tight coupling between a JavaScript engine's interpreter and its corresponding JIT compiler as a domain-specific and generic bug oracle, which in turn yields a highly sensitive fault detection mechanism. We have designed and implemented a prototype of the proposed approach in a tool called Jit-Picker. In an empirical evaluation, we show that our method enables us to detect subtle software faults that prior work missed. In total, we uncovered 32 bugs that were not publicly known and received a $10.000 bug bounty from Mozilla as a reward for our contributions to JIT engine security.

CCS CONCEPTS

• **Security and privacy → Browser security**.

KEYWORDS

Fuzzing, Software Security, Differential Testing, JIT Engine, Browser

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560624

ACM Reference Format:
Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. Jit-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560624

1 INTRODUCTION

JavaScript is the predominant programming language on the Web, where the vast majority of all websites use JavaScript code to implement tasks such as content loading, animations, form validation, data logging, or similar functionality. Generally speaking, web browsers include a JavaScript engine that executes the corresponding code on the client side. In recent years, also server-side JavaScript received a lot of attention via frameworks such as *Node.js*, which enable a simpler data sharing between client- and server-side code. In fact, *npm*, the package manager for *Node.js*, is the largest package registry in the world. The JavaScript language itself offers first-class functions, is dynamically typed, single-threaded, and offers prototype-based object-orientation.

In practice, three different JavaScript engines are widely used: First, Google's v8 engine was developed by the Chromium Project for both the *Google Chrome* and *Chromium* web browsers, but it is also used by projects such as *Node.js* and *CouchDB*. Second, the Mozilla Foundation maintains `SpiderMonkey`, which is used in various Mozilla projects, most notably the *Firefox* browser. Finally, `JavaScriptCore` (JSC) is a framework that provides a JavaScript engine for WebKit implementations, which are commonly used in the Apple ecosystem with the *Safari* browser on macOS and iOS. As part of the *browser wars*, the competition for dominance in the browser usage share, these engines played a major role given that a faster and snappier user experience was a main factor driving this competition. One crucial insight is that interpreted code is traditionally slower than native code—JavaScript as an interpreted language especially suffers from this disadvantage. Thus, developers opted to deploy Just-In-Time (JIT) compilers in JavaScript engines to compile frequently exercised code paths and then execute them at higher speeds. While conceptually simple, JIT engines employ sophisticated analysis and optimization passes before emitting native code. As a result of pushing for ever-increasing performance, these engine implementations are heavily optimized for speed and efficiency; many novel techniques [49, 51] have been developed specifically to execute the code in a fast and efficient manner.

Modern JavaScript engines are highly complex pieces of software, e. g., the Google v8 engine consists of about 2.3 million lines of code, and most of it (approx. 1.6 million LoC) is implemented in the memory-unsafe programming language C++ [55]. Given the huge practical importance of these tools and their wide practical deployment, they represent an attractive target for attacks. At the time of writing, vulnerability brokers such as *Zerodium* pay more than 200K USD for remote code execution exploits for common browsers [66], and JavaScript engines play a key role in unwillingly supplying a steady stream of exploitation primitives. In practice, an adversary has fine-grained control over the JavaScript code that is executed in browsers—they can twist every detail and abuse even the smallest inconsistencies arising from subtle bugs. More specifically, an attacker can craft full exploits from one (or multiple) inconsistencies that cause a memory corruption, even if the inconsistency itself is not a memory corruption vulnerability and does *not* lead to a memory safety violation and a resulting crash.

In an attempt to remediate attacks, recent research has focused on techniques for testing JavaScript engines for potential security vulnerabilities, and especially fuzz testing (*fuzzing* for short) played a particularly important role [19, 21, 23, 30, 45, 47, 60]. All the proposed fuzzing approaches have in common that they are highly optimized towards input generation to JavaScript engines, i. e., they attempt to generate diverse JavaScript code to reach as many different code locations as possible. The typical feedback loop based on maximizing code coverage is very successful for other types of software as well [14, 36, 48]. As a result, many different types of vulnerabilities have been detected. However, existing techniques rely on rather general bug detection oracles to detect software faults, e. g., sanitizers [52], or other methods to detect memory safety vulnerabilities. Unfortunately, this is a coarse-grained metric in practice: Many bugs do not crash the program execution or violate memory safety. The complex, optimizing nature of JIT engines often causes only small inconsistencies or slightly wrong results due to logic bugs that are missed by current bug detection oracles. This leads to a somewhat paradox situation: Current fuzzing techniques are already able to exercise logic bugs in the JavaScript engines, but they lack techniques to recognize that they have found something interesting (e. g., an inconsistency in the executed code). More specifically, current methods achieve excellent code coverage, but the actual *symptoms* of the exercised bugs are too subtle for non-domain-specific bug oracles to detect.

To overcome this challenge, recent works proposed to use *differential testing* for JavaScript engines [32, 38, 39]. The basic idea is to execute a given input in multiple programs that implement the same functionality, and check for observable differences in their behavior. If differences are found, a logic bug has likely occurred. However, these recent works compare *different* JavaScript engines against each other and, in particular, they do not target JIT compilation itself. Hence, existing methods cannot spot subtle changes introduced during the JIT optimization process, which is a major source of vulnerabilities in practice. Furthermore, testing different engines against each other requires a significant effort for cross-engine normalization: JavaScript is an evolving language and is not free of implementation-defined behavior (e. g., calling `array.sort()` with a comparator function not inducing a total order on the elements causes the array to be sorted in an engine-specific manner). As a result, many false positives occur, each requiring potentially time-consuming manual analysis. Another drawback is that different engines support different features; hence, only the intersection can be tested (leading to reduced code and feature coverage).

In this paper, we present the design and implementation of Jit-Picker, an approach that effectively finds inconsistencies caused by misguided JIT optimizations using differential fuzzing. We aim at detecting logic bugs manifesting as miscomputations, since such software faults are hard (if not impossible) to find with current state-of-the-art methods. The basic idea is to test the JavaScript engine against *itself* by executing the JavaScript code twice: once with the JIT compiler enabled and once without it, i. e., solely relying on the interpreter. A comparison of the behavior enables a domain-specific and generic bug oracle which can detect even subtle bug symptoms beyond the memory safety violations considered in prior work. While the high-level idea is simple, its correct and efficient implementation is challenging for several reasons: (1) We must ensure deterministic engine behavior to prevent false positives, (2) we need to implement side-effect free observation ("probing") of states to reduce false negatives, and (3) we must test different JIT levels and optimization passes to increase true positives.

We have implemented a prototype of our approach, Jit-Picker, based on Fuzzilli [19]. We build extensions to the execution model, incorporate an efficient probing mechanism into the generated input, and instrument all three major JavaScript engines to support our approach (e. g., by suppressing non-deterministic behavior of language builtins or out-of-memory exceptions). Our evaluation results show that Jit-Picker can effectively detect new bugs in JavaScript engines that have already been thoroughly tested: In total, we found 32 previously unknown bugs in the tested engines and disclosed our findings to the affected vendors; most of these issues have been patched at the time of writing. An analysis of the detected faults indicates that they do not manifest as segmentation faults or sanitizer violation conditions, hence these vulnerabilities cannot be detected using state-of-the-art methods. Mozilla rewarded our contribution to JIT security with a bug bounty of $10.000 and plans to internally deploy Jit-Picker for testing their engine.

Contributions. In summary, we make the following contributions:

1. We propose a novel approach to efficient differential testing that turns a JavaScript engine against itself: We carefully inject state probes to compare the runtime behavior of the JavaScript interpreter with its own JIT compiler, enabling us to uncover software faults introduced by misguided optimization passes.
2. We show how to use this method as a bug oracle with unprecedented sensitivity: It alerts the fuzzer to even small inconsistencies during execution, allowing us to identify subtle, yet potentially high-impact bugs that state-of-the-art oracles are blind to. We find that the identified faults potentially have high security impact in the victim's browser.
3. We implemented a full prototype of the proposed approach in a tool called Jit-Picker and evaluated it for three major JavaScript engines. In an empirical evaluation, we identified 32 bugs. A closer analysis reveals that most of the bugs we

```
1   function f(a) {
2     return a + a;
3   }
4
5   for (let i = 0; i < 100000; i++) {
6     // "train" engine for numeric addition
7     f(1.0);
8   }
9   // calling f with a different type causes a
10  // bailout due to mismatching types
11  f("abc");
```

Listing 1: Example use case for JavaScript JIT compilation

found had a lifespan of more than a year. We disclosed the identified bugs to the respective vendors in a coordinated way, most bugs have already been fixed.

To foster research on this topic, we release the source code and all research artifacts at https://github.com/RUB-SysSec/JIT-Picker.

2 CHALLENGES SECURING JAVASCRIPT JIT

Securing JavaScript (JS) JIT engines is notoriously difficult [8, 16, 31, 43]. Despite various techniques proposed in the literature [19, 21, 30, 45, 47, 60], the effective identification of software faults in JS engines is still a challenging problem. In the following, we first discuss the challenges of JIT-compiling code. We then present an overview of current techniques employed to find bugs and discuss their limitations in practice.

2.1 JavaScript JIT Compilation

Given that the performance of JS engines is critical for the user experience when surfing the Web, the time when JS code was interpreted statement by statement has long passed. Many types of performance improvements were implemented and today's JS engines determine at runtime which functions are *hot*, i. e., executed particularly often, and compile these functions just-in-time (JIT) to native machine instructions.

As a matter of fact, JS code is dynamically typed, which makes it difficult to optimize correctly. Consider the example code shown in Listing 1. The statement `a + a` within function `f()` (line 2) has different semantics depending on the type of the operand: Numeric addition for numbers, concatenation for strings, and so on. Hence, the interpreter of the JS engine collects runtime type information which in turn are fundamental to optimizing the function for frequently used types. In this example, `a` is observed to hold values of type `Number` for a large number of function calls in the `for` loop in line 7. Consequently, during the JIT compilation phase, the compiler speculates that these types will remain the same going forward, and emits native code to quickly perform the numeric addition. Such speculative optimizations enable efficient code that is specifically optimized per data type.

However, the dynamic typing of JS implies that assumptions about observed types *might* be invalidated in the future. In our example, this is due to the function call of `f()` with a string parameter in line 11. As a consequence, it is mandatory that JIT compilers emit additional type checks which validate the applicability of their optimization assumptions at runtime. Failing validations cause a bailout (i. e., a transition from compiled code back to the interpreter), where the observed type information is updated and taken into account for a potential recompilation.

Any run-time validation not resulting in a bailout consumes precious CPU cycles without actually progressing the program's execution. Therefore, JS engines try to remove unnecessary checks, e. g., those for assumptions which can be proven to hold under all circumstances. As an example, since calls to many built-in `Math` functions return values of type `Number`, code optimization passes might emit native arithmetic instructions without further type checks. Additionally, JIT compilers make use of traditional compiler optimization techniques (e. g., alias analysis, range analysis, dead code elimination, loop-invariant code motion, etc.) to remove even more run-time checks. As a result, JIT compilation yields substantial performance improvements over interpreted code and is used by all JS engines in current browsers.

2.2 Fuzzing JavaScript Engines

Given the complexity of JIT compilers, they result in a multitude of bugs in JS engines [8, 16, 31]. To find these bugs, we could use traditional bug finding techniques such as static or dynamic testing. Particularly *fuzzing* has received a lot of attention in both academia and industry over the past years [2, 4, 19, 23, 47, 65]. In essence, a fuzzing framework generates billions of (random) test cases in an automated way and uses them as input for a program under test, expecting to cause diverse behavior. Ideally, such a randomized test case triggers unforeseen edge cases in the JIT engine's optimization passes, leading to the identification of new bugs.

However, fuzzing JS engines via traditional testing methods proves ineffective in practice: JS code is highly structured, and, for meaningful JS execution to occur, the provided JS code needs high degrees of both syntactic and semantic correctness. As a result, common byte-level fuzzers such as AFL [65] and its many variants (e. g., [3, 6, 14]) are unable to generate valid JS test cases or effectively mutate existing ones. Hence, generic grammar-based fuzzers were developed to help increasing the quality of JS input generation [23, 60]. Following that, specific input generation techniques have been proposed that represent valid JS code in syntax-aware and semantics-aware formats, e. g., in intermediate representations [9, 19] or abstract syntax trees (ASTs) [2, 47]. These representations allow fuzzers to produce high-quality test cases, which achieve a high degree of code coverage. As a consequence, such techniques have proven effective in practice for finding complex software faults.

To detect if a produced input triggers a bug, fuzzers observe if the program under test reaches a faulty state. Generally speaking, current JS engine fuzzers rely on the following *bug oracles*—mechanisms to indicate faulty program behavior—to detect bugs:

- **Segmentation Faults and Signals.** A faulty state is detected if the engine *violates the execution model of the CPU*, such as unmapped memory accesses, division by zero, or similar illegal behavior.
- **Memory-safety Sanitizers.** Sanitizers such as ASAN or MSAN [50, 52, 53] are approaches that use compile-time instrumentation to detect software bugs such as uninitialized memory, use-after-free conditions, and similar memory safety violations. Sanitizers also detect software faults in

```
function miscompute(n) {
  n |= 0;
  if (n < 0) {
    let v = (-n) | 0;
    return Math.abs(n); // miscomputation here
  }
}
```

Listing 2: A simplified trigger of a miscomputation in JSC (CVE-2020-9802). This snippet can be synthesized by automated testing methods such as fuzzers, but does *not* exhibit a segmentation violation. Consequently, state-of-the-art oracles will not consider this input as interesting.

cases where the JS engine does not trigger a CPU signal but merely *violates* memory safety (e. g., an out-of-bounds access or uninitialized read). However, as sanitizers are part of the compiler toolchain, only the C/C++ components are instrumented. Support for sanitizing JIT-compiled code is not yet available for JIT compilers used in browsers. Broadly speaking, sanitizers re-introduce some of the guarantees included in memory-safe languages by design as run-time checks. Still, domain-specific logical flaws generally do not manifest as a sanitizer violation.

- **Assertions.** Assertions are run-time checks which have been manually inserted by a developer to express an invariant which is suspected to be upheld during execution—both in the current version and after later changes to the engine. Assertions within the code allow for the detection of domain-specific logical inconsistencies. However, manually adding and maintaining assertions as an oracle does not scale and only alerts to bugs breaking explicitly encoded invariants.

In summary, JS engines currently rely on traditional bug oracles to detect faulty behavior. Unfortunately, these oracles lack a rigorous and exhaustive approach to identify common bugs such as logic flaws. In particular, current state-of-the-art oracles are not precise enough to identify subtle bugs, unless the violation is severe enough to trigger a sanitizer warning or breaks an invariant explicitly considered by a developer in form of an assertion.

2.3 From Buggy Snippets to Crashing Inputs

Despite the recent progress in JS fuzzing, critical security vulnerabilities are still uncovered by humans on a regular basis that cannot be found with state-of-the-art fuzzing approaches. Examples for this observation include complex logical errors and hard-to-reach code locations that depend on a specific engine state to trigger a crash. As an example, consider the JS snippet shown in Listing 2 triggering a miscomputation in JSC. Due to a logical flaw in its common subexpression elimination optimization, the result of `Math.abs(n)` may return a *negative* number at runtime. This miscomputation cannot be detected by traditional bug oracles, as the miscomputation itself does *not* result in a crash or memory safety violation.

Groß [20] showed that this simple miscomputation is sufficient to be turned into a memory corruption vulnerability (CVE-2020-9802). Let us analyze the example shown in Listing 3 which turns this miscomputation from Listing 2 into an actual memory corruption condition. First, the code stores the miscomputed value in variable `idx` (line 6) and later uses it to access array `arr` (line 10). Despite not knowing the concrete value of `idx`, the optimizer may assume the following properties:

```
1   function oobBug(arr, n) {
2     n |= 0;
3     if (n < 0) {
4        let v = (-n)|0;
5        // miscomputation here may return negative number
6        let idx = Math.abs(n);
7        // following code triggers a segfault
8        // due to OOB access
9        if (idx < arr.length) {
10        arr[idx]; /* optimizer assumes idx is a positive number
11        and smaller than the length or the arr, therefore the
12        bounds-check can be eliminated */
13       }
14     }
15   }
```

Listing 3: Example code [20] for turning the triggered miscomputation in JSC (CVE-2020-9802 Listing 2) into an actual memory safety violation that can be exploited.

- as a result of `Math.abs()`, `idx` has a positive value
- when accessing the array, `idx` is smaller than the length of `arr` due to the guarding `if` clause in line 9

Combining these two properties, the access to `arr` is ruled to be always within the bounds of the array. Runtime bounds checking of the array access is thus deemed unnecessary and eliminated for performance reasons. As a consequence, the array access at a *negative* index is not guarded against at runtime. Therefore, the memory access might be outside of the intended bounds. Adversaries can use this memory safety violation as a stepping stone to compromise the entire process.

As we can see, JS bugs triggering a miscomputation have the potential to be a precursor for a memory corruption. However, the corresponding steps are highly engine-specific and entirely depend on the particular miscomputation. Automatically detecting such bugs via traditional bug oracles would mandate fuzzers to generate JS code which leverages intricate properties of engine-specific optimization passes. Unfortunately, this is entirely out of reach for the current generation of fuzzing methods.

As a key insight, we identify a significant gap between the ability to generate inputs triggering bugs in JIT engines, and the capability to detect them as such. To identify subtle bugs during JS JIT execution, we need to detect miscomputations given no carefully crafted exploit (which traditional oracles require to detect the bug), but the type of incidental trigger that fuzzers are able to generate. On a conceptual level, what we require for JS JIT fuzzing is comparable to ASAN [50] being able to detect memory safety violation without a segmentation fault crashing an ordinary C program.

3 DETECTING SUBTLE BUGS IN JS ENGINES

To overcome the shortcomings of these imprecise bug oracles, we propose to use *differential fuzzing* to receive more fine-grained feedback on miscomputations. Generally speaking, differential fuzzing compares the behavior of multiple implementations of a given functionality for the same input [10, 24, 28, 38, 42]. Given the highly dynamic nature of JS, it is not feasible to use this method to test the behavior of *different* JS engines to uncover bugs. The JS standard leaves too much room for interpretation, hence different implementations exhibit several differences in their behavior.

In contrast, we propose to compare a *single* JS engine against *itself.* Broadly speaking, a JS engine contains two implementations to execute JS code with identical semantics: a JS interpreter evaluating statement by statement and a JIT compiler producing aggressively optimized, native machine code. Depending on the JS engine, one or multiple intermediate JIT tiers [17] sit between the interpreter, and the aggressively optimizing compiler. These tiers offer a middle ground between the low startup overhead of the interpreter and the efficient code execution of a fully optimizing JIT engine.

To uncover software faults in JS engines, we can compare the execution of the interpreted code to the execution of code generated by the JIT compiler. We can selectively allow all JIT tiers to run or (optionally) individually disable them. If both executions perform the same computation, we can ignore this test case. If, however, the two executions *differ* in their computation, we can use this as a fine-grained oracle to identify the test case as triggering a bug. A major technical challenge is to implement such a comparison mechanism in an *effective* way given the highly optimized nature of today's JIT engines. A naive implementation of such a comparison mechanism would simply make all computations externally visible, e. g., by printing the result to `stdout`. Unfortunately, this rather crude way of observation significantly interferes with the machinery of the JIT engine. The mere fact that the result of some computation becomes externally visible (e. g., by printing) suppresses dead-code elimination. Alongside dead-code elimination, optimizing JIT compilers implement a whole range of classic optimizations (e. g., instruction reordering, scalar replacement, and constant folding [1]) and JS-specific passes (e. g., garbage-collector modeling). The applicability of individual optimizations depends on the result of analysis passes (e. g., alias analysis or type analysis [25]) and profitability assessments. While all observation techniques incur *some* interference w.r.t. optimization and analysis passes, a well-designed mechanism minimizes its impact. Failing to reduce interferences stifles optimizations passes and consequently suppresses miscomputations.

With the optimizing nature of JIT engines in mind, let us revisit the code snippet reproducing CVE-2020-9802 in Listing 3. Our goal is to simplify the code such that diverging behavior becomes apparent. Such a minimized snippet is actually shown in Listing 2, the starting point of our discussion. It triggers the same miscomputation, and the triggering code snippet is much easier for existing fuzzing methods to generate. Unfortunately, current bug oracles based on segmentation faults, sanitizers, or assertions are blind to the mere fact that a miscomputation occurred. As a consequence, these existing methods fail to correctly detect the bug. This motivates our work for developing a more sensitive and effective bug oracle, which allows us to detect subtle triggers of bugs in JS code.

4 DESIGN

Figure 1 shows a high-level overview of Jit-Picker, our differential fuzzing approach to detect even subtle software faults in JS JIT engines. With this approach, we aim to efficiently find logic bugs and miscomputations that have previously been hard to identify. At the core of our approach, we perform two executions of fuzzing inputs, once with the JIT compiler enabled and once solely using the engine's interpreter. Side-by-side executions in the same engine allow us to compare states with high fidelity and detect miscomputations even if no actual memory corruption has occurred.

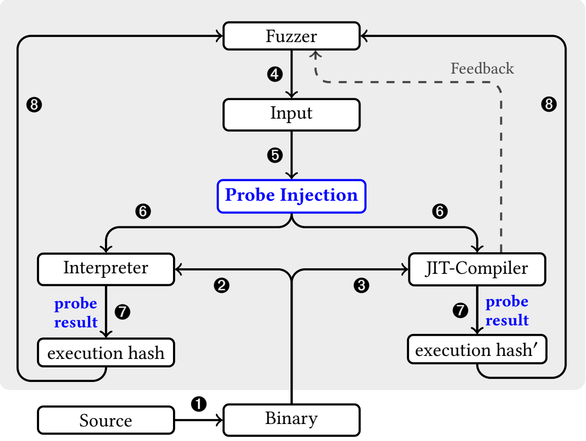

Figure 1: Overview of our design. Fuzzing inputs are passed to both the interpreter and optimizing JIT-compiler; comparing their internal state during execution (using probes injected into the input) reveals miscomputations.

To prepare fuzzing, the JS engine is compiled ❶ with code instrumentation. The instrumentation emits runtime code coverage information which assists the fuzzer in identifying interesting inputs. Then, the JS engine is instantiated twice. One instance ❷ is configured via command-line arguments such that JIT compilation is prohibited. Hence, the engine has to fall back to interpreting the JS code statement by statement. Note that this execution mode is slower, albeit significantly less likely to contain bugs given that the interpreters are mature and have been extensively tested. The second JS engine instance ❸ is configured to eagerly JIT-compile during execution. As an additional optimization, we selectively enable/disable the various JIT-tiers, optimizations, and code generation parameters. Next, the fuzzer generates JS inputs ❹. Jit-Picker now post-processes the generated inputs by injecting a probing mechanism. This centerpiece ❺ of our work extracts miscomputed results with unprecedented sensitivity. After probe injection, each input is executed ❻ by both instances of the JS engine. During both executions in the respective JS engine instances, the injected probes extract ❼ a subset of computational results (e. g., value of a local variable) and sends ❽ them to the fuzzer. The fuzzer now compares whether the extracted computations are identical. Deviations indicate that one of the two instances suffers from a miscomputation. Inputs provoking such diverging behaviors are extracted and stored for later analysis.

4.1 Key Components

After this overview, we now present some key components in more detail while deferring the actual probing mechanism to Section 4.2. First, we describe a way to instantiate the JIT engine ❸ such that miscomputations become more likely. We proceed with input generation ❹, the process of generating individual JS files. Afterwards, we focus on the requirements posed by efficient differential fuzzing when actually executing individual inputs ❻. In particular, matching the side by side execution in the interpreter and the JIT compiler requires a level of determinism not normally available in JS engines, i. e., we must identify and remediate potential non-deterministic behavior to avoid false positives.

JIT Instantiation ❸. When instantiating the JS engine with JIT compilation enabled, we probabilistically select configuration options that influence code generation, e. g., inlining heuristics or register allocation. Furthermore, for JS engines including multiple JIT tiers, we randomly disable a subset. Past research [18] demonstrated that this randomization process increases the diversity of code paths exercised, and in turn increases the likelihood of uncovering miscomputations.

Input Generation ❹. Emitting JavaScript for detection of miscomputations during JIT compilation poses a major challenge for input generation. Not only must inputs pass syntactic and semantic validation, the generated code must also trigger JIT compilation. This happens only for code deemed *hot*, i. e., executed often enough to warrant further optimization. Therefore, language-specific input generation based on intermediate representation (IR) of JavaScript [9, 19] seems most suitable. As the focus of our work lays on increased bug detection abilities, we will not discuss generation and mutation strategies in more detail. We show in Section 6 that relying on off-the-shelf JS generation techniques is sufficient for the purpose of identifying miscomputations.

Deterministic Execution ❻. The next step after generating individual inputs is executing them in two different instances of the engine, once in interpreter mode and once with JIT compilation enabled. The purpose of these executions is to detect deviating results between the two, in search of miscomputations. However, there is a multitude of reasons why such differences can be entirely benign, e. g., JS offers language-builtin functions which either purposely return non-deterministic (e. g., `Math.random`) or timing-dependent (e. g., `Date.now`) values. Generated and mutated JS code might call any of these functions, either directly or via a complex traversal of prototype chains.

While direct calls to non-deterministic functions could be prevented by adapting code generation, indirect calls can not. For example, randomly generated code might traverse just the right chain of object prototypes to reach a non-deterministic function. After invoking any non-deterministic function, subsequent execution behavior might differ. In a differential fuzzing setup such as ours, this would cause a significant number of false positives. Resolving this issue requires engine modifications that retrofit deterministic behavior into any builtin non-deterministic function. Details of the required engine-specific changes follow in Section 5.

4.2 Probing

During execution, we want to observe the computations and strategically place a certain number of observation points. At these points, we would like to inspect a subset of local variables and check for any inconsistencies between the interpreter and the JIT-compiled execution. While probing each and every computation is feasible, the resulting overhead may degrade execution throughput. As explained in Section 2.3, seemingly irrelevant minutiae such as computing negative 0 instead of positive 0 can compromise a JS engine if optimization passes rely on incorrect assumptions during code generation. Consequently, even the most minuscule miscomputations are of interest to us.

```
function main() {
  let result1; // declare in function scope for probing
  let result2 = 1 + 3.0;
  for (let idx = 0; idx < 1000; idx++) {
    let ret = some_compute(idx);
    result2 = ret + "aa"; // cannot declare result2 here
  }
  probe_state(result1); // updates execution hash
  probe_state(result2);
}
main();
output_state(); // sends execution hash to fuzzer
```

Listing 4: We emit calls to a probing function at the end of main. Before finishing the script execution, we output the accumulated execution hash. This value is expected to be independent of any applied code-optimizations during JIT-compilation, changes suggest a miscomputation.

A mechanism for probing the state of variables should therefore be sensitive to subtle changes in types and values. To achieve this, our design extends JS engines with an additional builtin function: `probe_state(value)`. Jit-Picker injects ❻ calls to this function as a post-processing step running after input generation. Upon each invocation of `probe_state`, the input parameter is serialized and accumulated into a hash value. After executing a JS input, this hash summarizes the observed computations and is hence dubbed *execution hash*. The execution hash is sent to the fuzzer (❽), which uses this value to compare multiple executions of the same input. An example of a JavaScript snippet generated by Jit-Picker is shown in Listing 4. In this example, variable `result1` and `result2` are probed at the end of `main`.

Transparent Probing Probing computations in the straightforward manner just introduced is already capable of identifying miscomputations, but it suffers from a set of architectural limitations: First, as all probing runs are at the very end of the execution, we can potentially miss miscomputations that happen within loops. As an example, consider the loop body in Listing 4: Assuming the loop miscomputes `result2` only for certain loop iterations, Jit-Picker misses the miscomputed value—unless it somehow propagates out of the loop. As a real-world example, `SpiderMonkey` Bug 1761947 in Table 1 uncovered by Jit-Picker only manifests in very specific loop iterations. Second, for them to be available at the end of `main()`, all variables we wish to probe must be declared in the outermost scope. However, this extended variable lifetime inhibits some compiler optimizations. As aggressive code transformations are the most likely culprits for miscomputations, we want to avoid reducing their applicability by increasing the lifetime of variables.

It may appear only logical to emit calls to the probing function not only at the end of code, but throughout the entire execution to address both these limitations. This would allow us to place calls within loops, uncovering miscomputations within loops, and avoid increasing variable lifetimes. See Listing 5 for an example of such a

```
1   function main() {
2     let obj = {a: 1};
3     for (let idx = 0; idx < 1000; idx++) {
4       probe_state(obj);
5       // engine must conservatively assume the native function
6       // call to probe_state updates obj. Hence obj.a cannot
7       // be identified as constant, in turn rendering
8       // constant folding optimizations inapplicable
9       let v = 1 + obj.a;
10      probe_state(v);
11    }
12  }
13  main();
14  output_state();
```

Listing 5: Emitting calls to a state-probing function inhibits code optimization, as the JavaScript engine cannot reason about which analyses remain valid during probing.

probing. We probe variable v within the loop (line 10) and hence observe all computational results therein. However, such probing implicitly assumes that the probe_state function is transparent to the JIT compiler optimization passes, which it is not.

As the probing function itself is still opaque to the JS engines, code optimization passes must fall back to conservative assumptions as a consequence. For example, the engines invalidate alias analysis results whenever the probing function is called, which in turn prevents transformations such as code motion. This hinders the detection of miscomputations, and, therefore, is undesirable. To overcome this limitation, and to enable us to place probing calls throughout the execution, we propose a so-called *transparent probing* mechanism. This advanced mode tightly integrates probing into the engine. Comparable to other JS builtins such as Math.log2, the transparent probing function is no longer treated as an opaque function call. Instead, it gets translated into the internal representation of JIT engines. This translation facilitates the declaration of the probing's properties, e. g., its (absent) impact on alias analysis and how it may be reordered relative to other instructions. As this declaration of properties happens before any compiler passes run, they unlock aggressive code optimizations despite our probing. While improving benefits w.r.t. to observing internal state, transparent probing requires significant (but one-time) effort to implement and is JS engine-specific. In summary, an analyst can choose between regular probing, which effortlessly applies to all engines, or decide to undergo the efforts to integrate transparent probing into specific JIT engines and focus on maximizing observations.

5 IMPLEMENTATION

We implemented the proposed method in a tool called Jit-Picker. The prototype is based on Fuzzilli, forking from commit bc057d1. As a result, the two fuzzers share parts of the overall architecture. However, differential fuzzing requires extensions to the execution model, injection of a probing mechanism, as well as changes to the targeted JS engines. In the following, we explain the implementation challenges and our solutions in more detail. Jit-Picker and all changes to the respective JS engines are available at https://github.com/RUB-SysSec/JIT-Picker under an open-source license.

Instrumentation. Jit-Picker consumes coverage feedback information generated by instrumenting the JS engine. Instrumentation is added by compiling the target with clang's compilation flag -fsanitize=coverage. At runtime, the instrumentation updates a bitmap stored in a shared memory segment which is mapped into Jit-Picker as well as the target engine. This feedback mechanism allows steering input mutation towards unexplored code, successively increasing the amount of code covered.

Engine Modifications: Determinism. As explained in Section 4.1, JS engines have to be modified to suppress non-deterministic behavior of language builtins. Furthermore, some JS inputs generate out-of-memory exceptions in interpreter mode, but not under JIT compilation (or vice versa). Even though both execution modes have the same amount of memory available, runtime allocation slightly differs. As a consequence, an input might successfully execute in interpreter mode, but throws an exception when JIT-compiling. As such out-of-memory conditions are most likely a false positive, they are signaled to the fuzzer, which in turn discards the input file.

We patched the JS engines SpiderMonkey and JSC such that various builtin functions returning non-deterministic values either return a constant or are not available during fuzzing at all. Runtime errors such as out-of-memory are changed to print errors to stderr, allowing the fuzzer to suppress false positives. The v8 engine already has support for suppressing non-deterministic behavior of builtins, hence no modifications to builtin-functions are necessary here. Due to intricacies of the floating point format, NaN can be represented by many different bit-representations, which in turn can result in spurious differentials. As a consequence, JS engines require a normalization of floating point values. We implemented this support by replacing NaN values with a canonical bit-pattern when encountered during probing.

Engine Modifications: Transparent Probing. All of the generated JavaScript, and importantly, our probing function, should be amenable to aggressive compiler optimizations. Hence we proposed in Section 4.2 to make the properties of the probing function known to the JS engines. As each engine differs, this requires an engine-specific engineering effort. Even though this is a one-time effort, we focus on implementing this technique for only a single JS engine, SpiderMonkey, to demonstrate the practical feasibility of this technique while reducing the engineering burden on our side. The probing mechanism alone requires adding more than 600 LoC into SpiderMonkey. We stress that the same concept can also be applied to the other two engines, v8 and JSC.

A rough sketch of the transparent probing implemented for SpiderMonkey is as follows:

- Declare the probing function as an inlinable native, similar to builtins like Math.log2.
- Map the inlinable native to 2 new instructions introduced at the medium-level intermediate representation (MIR) [35] of SpiderMonkey. Declaring the MIR instructions' properties allows aggressive optimizations to run, as the conservative defaults no longer apply.
- Introducing additional instructions to the low-level intermediate representation (LIR) [34]. One of the instructions specializes for typed values, whereas the other handles untyped values.

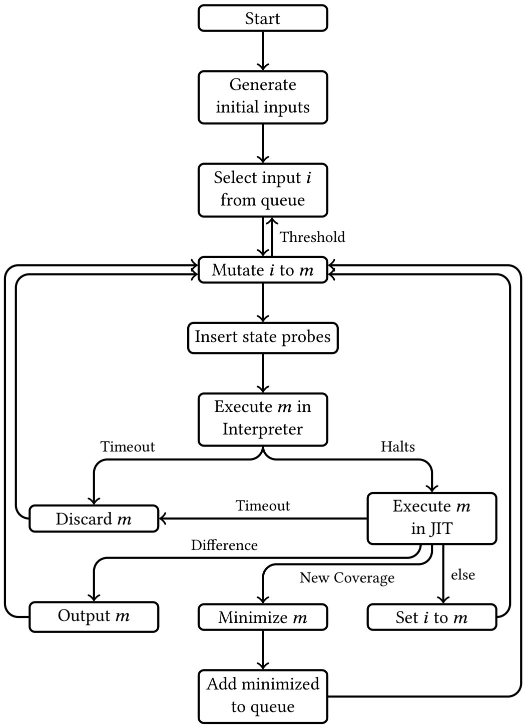

Figure 2: Conceptual overview of the fuzzing workflow implemented in Jit-Picker with a focus on passing a single input to baseline and JIT execution.

- Implementing code generation functions for the introduced LIR instruction, which translate the probing mechanism to the CPU architecture machine code it is generated for.

In Section 6, we evaluate the mechanism w.r.t. the number of values captured during execution and show its superiority to regular probing just at the end of the `main()` function. As we have not adapted `v8` and `JSC`, we use regular probing instead of transparent probing for these two JS engines.

Fuzzing Workflow. Figure 2 shows the workflow of Jit-Picker. We initialize the input queue by generating an initial sample containing a minimal amount of JavaScript. The next step is selecting a random sample i from the queue. This sample is scheduled for multiple fuzzing rounds, until we reach a predefined threshold and select a different sample from the queue. Within each round, the sample i is modified via a randomly chosen mutation, deriving a new input candidate m. Next, we place our state probes into the sample. If transparent probing is supported, we scatter the probes throughout the entire sample, as described in Section 4.2. Otherwise, we inject the probes at the end of the sample.

Executing m in a JS interpreter leaves us with two potential outcomes: (i) The execution can time out and we attempt a different mutation, or (ii) the execution of m can halt. In case of the latter, we extract a hash value (*execution hash*) derived from the state probe inputs. Proceeding with the second execution of m in a JIT-compiler enabled version of the JS engine leaves us with a set of four different outcomes:

- **Timeout:** If m does not terminate in the JIT-compiling engine, we cannot reasonably compare the derived execution hash. Instead, we have to discard the input and test a different mutation.
- **Differential:** If the execution hash derived from the JIT-compiled input differs, we detected a differential execution. These samples indicate a bug and are stored for manual analysis.
- **New Coverage:** If new coverage is detected, m is minimized and added to the queue. Later, the minimized sample will be selected for further fuzzing rounds.
- **Default:** If none of the previous cases materializes, we set i to m and resume mutation.

Generally speaking, this fuzzing workflow successively explores code paths in the target JS engine, while keeping samples that provoke a miscomputation.

6 EVALUATION

To evaluate our approach, we tested Jit-Picker on the three most widely-used JS engines with support for JIT compilation. These are `SpiderMonkey` (Firefox), `v8` (Chrome/Chromium) and `JSC` (WebKit/Safari). The engines already received significant attention [19, 21, 30, 45, 47, 60] by security researchers and their respective vendors, and hence can be considered well-tested, challenging targets. While other JS engines exist [12, 15, 57], they generally do not support JIT compilation and hence are unamenable for our current implementation of differential fuzzing.

In our evaluation, we investigate three research questions:

RQ 1: Can Jit-Picker identify new bugs in already well-tested JS engines?

RQ 2: Can the software faults identified by Jit-Picker also be detected by current state-of-the-art fuzzing methods relying on traditional bug oracles?

RQ 3: Does differential fuzzing incur a significant penalty w.r.t. the number of inputs exercised and code coverage reached?

Identified Software Faults. For the identification of unknown bugs, Jit-Picker ran over the course of 10 months while continuously rebasing our changes to the latest versions of the JS engines. Table 1 shows a summary of the identified bugs. In total, we found 32 previously unknown bugs. While not all of our findings were rated as security issue (e. g., incorrect rounding optimizations or incorrect value recovery with no security impact), each miscomputation flagged as *JIT-Bug* posed a dormant threat. As aggressive code optimizations are continuously added, more and more miscomputations turn into security issues. Hence it is of significant importance to establish a sensitive bug oracle today. The majority of identified bugs had a lifespan of more than a year and a significant proportion of `JSC` bugs was even older. While JIT engine security received a fair amount of attention recently [22, 40, 60], these contributions failed to uncover the issues presented in Table 1. Due to their lack of a differential oracle, miscomputations generally elude their approaches. In particular, the *Age* column shows that many bugs predate these works, in turn emphasizing the contribution of Jit-Picker. We disclosed the bugs to the respective vendor in a coordinated way and by the time of writing, the majority of the identified bugs have already been fixed.

Table 1: The table shows differential execution bugs found by Jit-Picker over the course of multiple months. All identified issues where reported to the respective vendor and we worked together with them to get the vulnerabilities fixed. The *JIT-Bug* column indicates whether the bug is present in the JIT component of the engine, as judged by analyzing the patches. The *Age* column indicates the minimum number of months between introduction of the miscomputation and the commit fixing the issue. Introduction of the root cause might be even earlier. The *Changes* column indicates the number of lines changes by the fix, not counting any supplemental changes such as test cases.

Engine	Bug ID	JIT-Bug	Age [months]	Status	Changes	Description
v8	v8 12215	✓	> 6	collision[1]		Negative array indices may return incorrect values
JSC	WebKit 229869	✓	>30	fixed	+37/-13	Incorrect casting between int52 and int32 values
JSC	WebKit 230802	✓	>30	fixed	+6/-2	Strength reduction analyzes `RegEx.exec` incorrectly
JSC	WebKit 230804	✓	19	fixed	+6/-4	Engine did not properly handle callable `Proxy`'s
JSC	WebKit 229939	✓	>30	fixed	+13/-4	`in` statements incorrect covert non-objects
JSC	WebKit 230823	✓	>30	fixed	+19/-9	Incorrect backwards propagation during OSR
JSC	WebKit 229951	✓	>30	fixed	+66/-7	Incorrect `this` used for getter access
JSC	WebKit 231321	✓	2	fixed	+43/-21	Missing property recovery
JSC	WebKit 231322	✘	10	fixed	+2/-3	Incorrect handing of `ArrayBuffer` watchpoints
JSC	WebKit 232679	✓	>30	fixed	+3/-3	Incorrect conversion from `Date` to `Number`
JSC	WebKit 232753	✓	>30	fixed	+10/-2	Constant folding to wrong type
JSC	WebKit 232754	✓	7	fixed	+57/-15	Incorrect conversion from `Symbol` to `String`
JSC	WebKit 232966	✓	>20	fixed	+8/-5	Spread incorrectly handles negative indices
JSC	WebKit 233408	✓	18	fixed	+30/-14	Incorrect property check for in-bounds computation
JSC	WebKit 233682		>16	reported		
SM	1716231	✓	1	fixed	+9/-8	Incorrect rounding optimization on float32 values
SM	1716931	✓	2	fixed	+38/-5	Incorrect interaction between `yield*` and OSR
SM	CVE-2021-29982	✓	14	fixed	+16/-11	Register clobbering leaks information to JavaScript
SM	1720093	✓	13	fixed	+7/-0	Incorrect value recovery during OSR
SM	1720032	✓	16	fixed	+21/-9	Exception tracking causes DCE of live code
SM	1738676	✓	13	fixed	+1/-1	Incorrect dropping of negative zero sign
SM	1745949	✓	>24	fixed	+76/-71	Range analysis and truncation interact incorrectly
SM	1749460	✘	1	fixed	+2/-2	Baseline insufficiently checks argument spreading
SM	1750496	✓	8	fixed	+0/-1	Incorrect instruction folding
SM	1751660	✓	>24	fixed	+8/-0	Incorrect recovery of `Function.arguments`
SM	1757634	✓	>18	fixed	+78/-87	Mishandling of frozen prototypes
SM	1759029	✓	11	fixed	+5/-2	Incorrect value recovery in `catch` blocks
SM	1761947	✓	4	fixed	+26/-32	Incorrect value recovery from scalar-replaced objects
SM	1762343	✓	11	fixed	+80/-75	Incorrect optimization on float32
SM	1763012	✓	>22	fixed	+7/-0	Incorrect range analysis results in DCE of live code
SM	1767196	✓	> 6	fixed	+7/-2	Incorrect `phi` specialization
SM	1785200	✘	> 8	fixed	+116/-5	Sparse elements are accessed erroneously

[1] Public discovery of a bug hidden by the vendor

Answer to RQ 1: Jit-Picker uncovers bugs not publicly known in all tested JS engines, despite the fact that browser vendors and security researchers already invest significant resources in testing their implementations.

False Positives A JS input yielding different results during multiple executions might be caused by a miscomputation. However, there are cases where differing results are entirely benign. As an example, `Math.random` returns a different value for each invocation. Such benign differences do not constitute a bug and are considered a false-positive w.r.t. Jit-Picker. Our efforts to reduce such benign differences are described in Section 5. During our fuzzing campaigns spanning 10 months, we reported all findings directly to the respective vendor. These campaigns were deployed on 1-4 servers, depending on the available resources. None of the bug reports were deemed invalid or irreproducible. We hence conclude that the false-positive rate of Jit-Picker is negligible.

Case Study: Firefox Infoleak CVE-2021-29982. During our evaluation, we identified a differential execution in Firefox. The code generation function `isCallableOrConstructor()` takes two parameters `obj` and `output`, each encoding a register. Due to a logical flaw in the register allocation, `obj` and `output` were assigned to the same register. The situation is comparable to a C/C++ function taking two pointer parameters, while expecting them to be non-overlapping. This leads to multiple operations of a generated code sequence unexpectedly operating on an identical register.

```
1  function main() {
2    let result;
3
4    async function f() {
5      const val = false;
6      const neg = -val; // neg is negative 0
7      result = neg;
8      for (let idx = 0; idx != 100000; idx++) {
9        // trigger JIT
10     }
11   }
12
13   [1,1,1].filter(f);
14   // result is -0 in interpreter mode and 0 in JIT mode.
15   // probe_state is sensitive to the difference, in turn
16   // allowing identification of the miscomputation.
17   probe_state(result);
18 }
19 main();
```

Listing 6: JavaScript snippet identified by Jit-Picker (minimized and simplified) triggering a miscomputation in JSC.

In this particular case, an internal pointer value got partially exposed to JavaScript. As a consequence, the result of a JS operation returned different results depending on internal pointer values. Note that bugs like this which leak (initialized) values to JavaScript do *not* trigger segmentation faults or sanitizer violations. Hence, they remain invisible to existing bug oracles. In contrast, information leaks are highly valuable to adversaries, as they allow bypassing security mitigations based on information hiding (e. g., ASLR and stack canaries [41, 59]).

Case Study: WebKit Bug 230823. During fuzzing of WebKit, Jit-Picker identified a miscomputation in JSC. A minimized and simplified version of this miscomputation is shown in Listing 6.

The miscomputation's root cause is a logical flaw in the interaction between graph pruning and a backpropagation algorithm. Our state probing function in line 17 detects the change in `result`, while a simple call to `print` would be unable to detect any change. We modified such findings before submission to the relevant bug tracker of each tested JS engine, e. g., by replacing the call to `probe_state(result)` with `print(Object.is(-0.0, result))`. This information allowed the developers to reproduce our finding and the underlying issues were promptly fixed.

Case Study: WebKit Bug 230802. One of the differential executions identified by Jit-Picker is caused by incorrect behavior of `RegEx.exec()` in WebKit. During JIT compilation, `RegEx.exec()` erroneously returned a value of type `Number`. However, only `Null` and `Object` are valid return types for this function call. The root cause for this bug was a logical flaw during a strength reduction optimization [1]. Because WebKit optimizations currently do not take the expected return type of `RegEx.exec()` into account, the bug was rated as a non-security issue.

Optimization passes of other JS engines, e. g., v8's typer phase, already exploit the type information of builtin functions. In particular, past vulnerabilities in v8 [46] did in fact exploit mismatches between typer phase optimizations and actual runtime types. In essence, any non-security miscomputation today is "just a compiler optimization away" from becoming a remote-code execution vulnerability in the future and thus these bugs need to be considered carefully.

Answer to RQ 2: As shown in our case studies, differential execution bugs generally do not manifest as segmentation fault or sanitizer violation. Hence, existing bug oracles are blind to these bugs, despite input generation being able to produce faulty samples.

Experimental Setup for Comparative Experiments. For the subsequent comparative evaluation, we tested Jit-Picker on the three most-widely used JS engines, for which we used the latest available versions at the time of the experiments:

- For `v8`, we based our changes on version `9.6.99` which corresponds to Git commit `29abb5d847`.
- For `SpiderMonkey`, our changes are applied to Git commit `3fa5cc437a49`.
- For WebKit, we modified Git commit `29c8d02c3b11` with our changes.

The tests were distributed on three identical machines, each with two Intel Xeon Gold 5320 CPUs clocked at 2.20GHz, 256GB RAM, and Ubuntu 21.04 as operating system. Each test ran for three days and was repeated five times. A dedicated leader instance synchronized the individual fuzzing queues of the 100 workers per test via a virtualized network. Coverage was measured by re-running the Jit-Picker queues on a separate JS engine build compiled to generated Clang code coverage feedback [56]. Note that for code coverage measurements we disabled the randomization of JIT options and tiers described in Section 4.1 as this would result in an unfair advantage of Jit-Picker over Fuzzilli. The `SpiderMonkey` implementation had transparent probing enabled, while `v8` and `JSC` utilized baseline probing.

Inputs Exercised. To answer the first aspect of RQ 3, the impact of differential testing on fuzzing throughput, we measure the number in tested samples over a fuzzing session of three days. Intuitively, the number of samples tested should be reduced by more than 50%. This slowdown is presumed as samples are not only processed by the faster JIT compiler, but in addition by the slower JS interpreter.

Figure 3 compares the number of samples exercised by Jit-Picker to a baseline run of Fuzzilli over the course of three days testing `SpiderMonkey`. Taking the throughput measurements on `SpiderMonkey` as an example, Jit-Picker tests 235 million samples within three days ($Samples_{\text{Jit-Picker}}$). Of these samples, 190 million are executed a second time in order to detect differential executions ($DiffTests_{\text{Jit-Picker}}$). The difference between the number arises due to samples not terminating, i. e., producing a timeout. Remember, as shown in Figure 2, samples triggering a timeout in the interpreter execution are not scheduled for differential testing. Comparing these numbers to our baseline Fuzzilli, which tests 271 million samples, we notice a decrease in throughput of 13%. This decrease is significantly less severe than intuitively expected, but still indicates that differential fuzzing leads to a certain overhead.

Code Coverage. Answering the second aspect of RQ 3, the impact of differential fuzzing on code coverage, is again evaluated on a fuzzing run of three days. While we could have included code

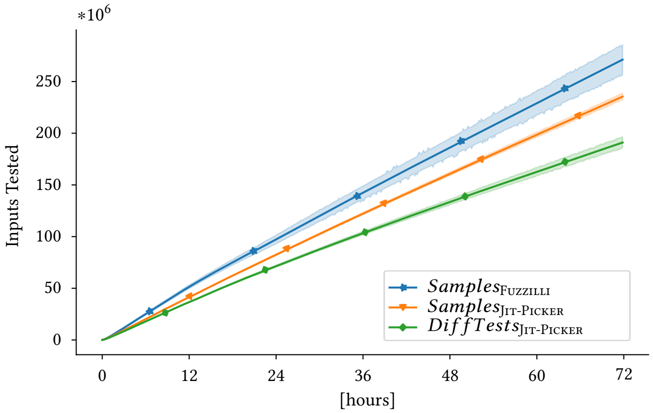

Figure 3: Number of inputs tested and corresponding 60% confidence intervals. Testing an input file two times in a differential fuzzing setup does not degrade the number of inputs tested by 50%. Instead, we observe a decrease in throughput by 13%.

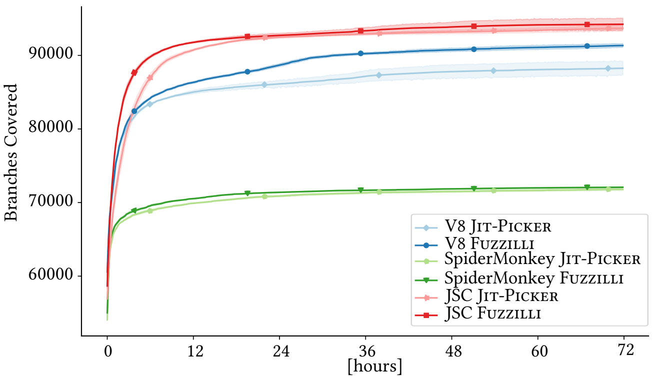

Figure 4: Branch coverage and 60% confidence intervals of Jit-Picker and Fuzzilli while fuzzing three JS engines over 3 days. For all engines, additional differential executions has little effect on the progress of achieving high code coverage.

coverage measurements of various other JS fuzzers, the focus of our work is *not* on improving input generation and increasing said coverage. Instead, our stronger oracle should be shown to increase bug detection capabilities with the *same* amount of code coverage. Indeed, as we can see in Figure 4, there is no significant change for SpiderMonkey and WebKit compared to our baseline after completing the measurement. Only a small noticeable delta in code coverage shows on v8. We explain the difference with decreased fuzzing throughput of Jit-Picker, as shown in Figure 3.

Answer to RQ 3: While the number of inputs tested by Jit-Picker decreases compared to our baseline, the same code coverage is reached over the course of a fuzzing session.

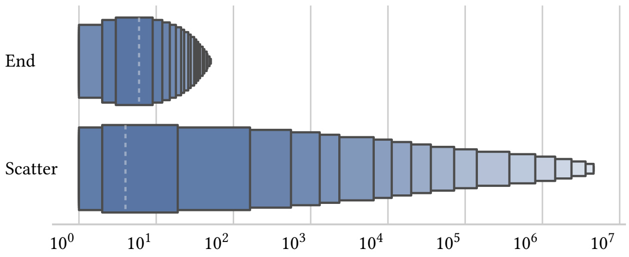

Figure 5: Boxen-plot showing the impact of the state probing function's placement on the number of runtime observations. Scattering the function throughout the program strongly increases the number of observations.

Number of State Probes. In Section 4.2, we introduced the idea of placing state probes not only at the end of the generated JS inputs, but instead scattering them throughout the entire input (*transparent probing*). This was motivated by the fact that variables holding a miscomputed value might no longer be available at the end of the computation. In particular, variables stored within loops are continuously redefined, leaving only the last redefinition available at the end of the execution. Hence, any miscomputation happening only during certain loop iterations would likely remain invisible.

In this part of the evaluation, we compare the number of observations our execution hash depends on, i. e., the number of calls to probe_state. Our baseline is placing the state probes at the end of the execution only. In comparison, we scatter the same number of probes throughout the entire input sample. The boxen plot in Figure 5 shows a visualization of our measurement results. The interval bound by the highest boxes accounts for 50% of all executions, with the dashed line indicating the median. The second-largest boxes each account for 12.5% of all executions, the third-largest boxes for 6.25%, successively cutting in halve the remaining measurements. When placing our probes at the end only, we rarely encounter executions with more than 20 individual observations. This stands in stark contrast to placing our probing throughout the entire sample. In > 12.5% of executions we observe at least 10^2 values and in 3% of executions at least 10^3. This is congruent with our hypothesis that random placement will emit the probing functions within loops, in turn increasing the likelihood of detecting a miscomputation only occurring during a subset of loop iterations.

7 DISCUSSION

In the following, we discuss potential shortcomings of Jit-Picker.

Other Targets. While we focused on JS engines of web browsers in this work, differential fuzzing is per se applicable to any language or program featuring both an interpreter as well as a JIT engine. In the context of web browsers, an example of this is regular expression engines, which rely on JIT compilation [11] to speed up matching. Furthermore, compiling untrusted WebAssembly [44] modules to native code poses an opportunity to deploy differential fuzzing.

Besides browsers, untrusted PHP code running in a sandbox environment [54] of a hosting provider or the *Extended Berkeley Packet Filter* (eBPF) in the Linux kernel are amenable to differential fuzzing.

Widening the scope to non-security applications, the correctness of JIT compilers for Python, Ruby, and other programming languages can be tested. Depending on the target, this may require adaptation of the probes and their injection into the fuzzing inputs. In this work, we focus on JS engines due to their practical importance, high value to attackers, and exposure to malicious code.

Integration into Other Fuzzers. Our approach is designed in a generic and fuzzer-agnostic way. As such, it can be built on top of any fuzzer that allows adapting the fuzzer's input generation process to insert the state probing function. For our prototype implementation of Jit-Picker, we have chosen Fuzzilli, the industrial state of the art, as base due to its high efficiency and excellent track record w.r.t. finding bugs.

Bug Triaging and Impact. Fuzzers are notorious for generating a high number of inputs even for a small number of underlying bugs, requiring additional tools to reliably identify the true number of bugs and their root cause [5, 27]. As Jit-Picker produces inputs that do not necessarily cause a crash, existing tools cannot be used to minimize the findings or triage the root cause of identified bugs. We have manually triaged the root cause for the bugs that Jit-Picker uncovered. More principled analysis methods are needed to improve the assessment of identified bugs.

8 RELATED WORK

As an established tool for excavating bugs in software, fuzz testing has received increased attention from security researchers over the years. AFL [65] introduced the concept of feedback-driven fuzzing, which gradually builds and mutates a corpus of interesting inputs by observing code coverage within the program under test. By performing bit-oriented and byte-oriented mutations on an input, flexible inputs are generated, and a wide range of programs were tested with great bug finding results. Researchers subsequently followed suit by proposing generic improvements of several core aspects of fuzzing, such as selecting mutations [33], selecting inputs to mutate [6, 64], and decreasing the overall overhead of fuzzing [36, 62]. Such generic progress extended to programs which expect highly structured inputs, for which researchers proposed approaches based on structure-awareness and grammar awareness [2, 4, 9, 23, 58, 60].

As browser security became increasingly relevant to protect the privacy of users, researchers tailored fuzzing systems more and more towards JS engines. These approaches focused on generating increasingly diverse and high-quality JS test cases to thoroughly exercise JS interpreters and JIT compilers [19, 21, 30, 40, 45, 47].

In addition to optimizing the input generation mechanisms that drive the fuzzing process, bug oracles play an important role in identifying bugs in the system under test. During fuzzing, sanitizers may be used to detect faulty conditions even in the absence of crashes, such as accesses to uninitialized or previously freed memory. Sanitizers have received some academic attention in the form of optimizing their performance [26], making them available for closed-source targets [13], or using their injected checks as prioritized targets during fuzzing [37]. However, the corpus of work dedicated to input generation mechanisms far outweighs similar works on bug oracles. To this day, state-of-the-art JS engine fuzzers still largely rely on traditional crash signals and generic sanitizers [50, 53] to detect buggy behavior.

One notable line of work that aims to detect bugs without requiring crashes is differential testing, where the output of different, but related target components is compared. Previously, differential testing was used to uncover bugs in CPUs [24], x509 certificate validation [7, 42], C compilers [29, 63] and the JVM [10]. With regards to browser testing, intuitively, one might want to locate browser implementation bugs by comparing a specific execution's result to the result produced by competing browsers. However, the leeway introduced by implementation-specific behavior in the JS standard `ECMAScript` [61] makes it impossible to compare the outputs, and fully normalizing JS output across JS engines is infeasible: Global structures are laid out differently between JS engines, and functions like sorting algorithms are allowed to produce inconsistencies in the face of unexpected inputs. To circumvent this pitfall, Park et al. [38, 40] rely on the language specification to derive test cases for which the outcome is well-defined and known a-priori, leading to comparable outputs. In contrast, a fuzzer will (and should) explore all JS features, including those that are defined as implementation-specific in `ECMAScript`. These implementation-specific details lead to benign differences between observed output behavior of JS engines, which makes it challenging to judge whether a difference is benign or introduced by faulty behavior. This results in a high number of false positives, rendering the approach of comparing the output behavior of different JS engines unattractive for fuzzing JIT optimization passes. In contrast, we propose to evaluate a JS engine against itself.

9 CONCLUSION AND FUTURE WORK

We presented the design and implementation of Jit-Picker, a new method for efficiently testing JavaScript JIT compilers by combining differential testing with fuzzing. We leveraged the strict consistency requirement demanding that both interpreted and JIT-compiled code perform computations in a substitutable manner. This criteria allowed sidestepping the issue of implementation-dependent behavior, previously preventing differential testing on all but well-defined inputs. Instead, we unlocked differential testing on fuzzer-generated JavaScript code by testing individual engines against themselves. In our evaluation, Jit-Picker identified 32 bugs not publicly known in the three most-widely used browser JS engines.

While differential fuzzing already proved its usefulness, we see multiple optimization opportunities for testing JIT compilers. First, the transparent probing mechanism could be implemented for the remaining two JS engines, but this is merely an engineering challenge. Second, the probing mechanism could be extended to incorporate more engine-specific details (such as reconstructed type information for object fields). This would increase the sensitivity to miscomputations even further, in turn strengthening Jit-Picker's bug detection abilities. Finally, we see opportunities to automate the process of deduplicating the produced bugs, reducing the manual effort required to triage the reported findings.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the German Federal Ministry of Education and Research (BMBF, project KMU-Fuzz – 16KIS1523).

REFERENCES

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2003.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[5] Tim Blazytko, Moritz Schloegel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. Aurora: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security Symposium*, 2020.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[7] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[8] Mathias Bynens. Temporarily Disabling Escape Analysis. https://v8.dev/blog/disabling-escape-analysis, 2007.

[9] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed Differential Testing of JVM Implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[11] Erik Corry, Christian Plesner Hansen, and Lasse Reichstein Holst Nielsen. Irregexp, Google Chrome's New Regexp Implementation. https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html, 2009.

[12] Facebook. Hermes. https://hermesengine.dev.

[13] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *IEEE Secure Development (SecDev)*, 2020.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[15] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra Lightweight JavaScript Engine for Internet of Things. In *ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2015.

[16] Robert Gawlik and Thorsten Holz. Sok: Make JIT-spray Great Again. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[17] Google. Sparkplug - a Non-optimizing JavaScript Compiler. https://v8.dev/blog/sparkplug.

[18] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm Testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[19] Samuel Groß. Fuzzilli: (Guided)-Fuzzing for JavaScript Engines. *Offensive Con*, 2019.

[20] Samuel Groß. JITSploitation I: A JIT Bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html, 2020.

[21] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[22] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[23] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *USENIX Security Symposium*, 2012.

[24] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[25] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *International Static Analysis Symposium (SAS)*, 2009.

[26] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *USENIX Annual Technical Conference (ATC)*, 2020.

[27] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. Igor: Crash Deduplication Through Root-Cause Clustering. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[28] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[29] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[30] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A Neural Network Language Model-guided JavaScript Engine Fuzzer. In *USENIX Security Symposium*, 2020.

[31] Zhenhuan Li and Shenrong Liu. Using the JIT Vulnerability to Pwning Microsoft Edge. *Black Hat Asia*, 2019.

[32] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo d'Amorim. Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing. *Software Quality Journal*, 29:129–158, 2021.

[33] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, 2019.

[34] Mozilla. IonMonkey/LIR. https://wiki.mozilla.org/IonMonkey/LIR.

[35] Mozilla. IonMonkey/MIR. https://wiki.mozilla.org/IonMonkey/MIR.

[36] Stefan Nagy and Matthew Hicks. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[37] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium*, 2020.

[38] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification. In *International Conference on Software Engineering (ICSE)*, 2021.

[39] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. JISET: JavaScript IR-Based Semantics Extraction Toolchain. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020.

[40] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[41] PaX Team. Address space layout randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt, 2003.

[42] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient Domain-independent Differential Testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[43] Chris Rohlf and Yan Ivnitskiy. Attacking Clientside JIT Compilers. *Black Hat USA*, 2011.

[44] Andreas Rossberg, Ben L Titzer, Andreas Haas, Derek L Schuff, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, and Michael Holman. Bringing the Web Up to Speed with WebAssembly. *Communications of the ACM (CACM)*, 61:107–115, 2018.

[45] Jesse Ruderman. Introducing jsfunfuzz. *https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/*, 2007.

[46] Stephen Röttger. Incorrect type information on Math.expm1. https://bugs.chromium.org/p/chromium/issues/detail?id=880207.

[47] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level Fuzzing. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2021.

[48] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[49] Marija Selakovic and Michael Pradel. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *International Conference on Software Engineering (ICSE)*, 2016.

[50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[51] Manuel Serrano and Marc Feeley. Property Caches Revisited. In *International Conference on Compiler Construction*, 2019.

[52] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[53] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[54] Dmitry Stogov and Zeev Suraski. PHP RFC: JIT. https://wiki.php.net/rfc/jit, 2019.

[55] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[56] The Clang Team. Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[57] Sami Vaarala. Duktape. https://github.com/svaarala/duktape.

[58] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing A Test Corpus with Bonsai Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2021.

[59] Perry Wagle, Crispin Cowan, et al. Stackguard: Simple Stack Smash Protection for GCC. In *Proceedings of the GCC Developers Summit*, 2003.

[60] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-Aware Greybox Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2019.

[61] Allen Wirfs-Brock. *ECMAScript 2021 Language Specification.* Ecma International, 12 edition, 2021.

[62] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[63] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[64] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security Symposium*, 2020.

[65] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[66] Zerodium. ZERODIUM: Payouts for Mobiles. https://zerodium.com/program.html, 2021.