DUMPLING: Fine-grained Differential JavaScript Engine Fuzzing

Liam Wachter*[†][‡], Julian Gremminger[†][‡], Christian Wressnegger[†], Mathias Payer[‡] and Flavio Toffalini[‡][§]
*Asymmetric Research, [†]Karlsruhe Institute of Technology (KIT), [‡]EPFL, [§]Ruhr-Universität Bochum
liam@seine.email, mail@ju256.de, christian.wressnegger@kit.edu, mathias.payer@nebelwelt.net, flavio.toffalini@rub.de

***Abstract*—Web browsers are ubiquitous and execute untrusted JavaScript (JS) code. JS engines optimize frequently executed code through just-in-time (JIT) compilation. Subtly conflicting assumptions between optimizations frequently result in JS engine vulnerabilities. Attackers can take advantage of such diverging assumptions and use the flexibility of JS to craft exploits that produce a miscalculation, remove bounds checks in JIT compiled code, and ultimately gain arbitrary code execution. Classical fuzzing approaches for JS engines only detect bugs if the engine crashes or a runtime assertion fails. Differential fuzzing can compare interpreted code against optimized JIT compiled code to detect differences in execution. Recent approaches probe the execution states of JS programs through ad-hoc JS functions that read the value of variables at runtime. However, these approaches have limited capabilities to detect diverging executions and inhibit optimizations during JIT compilation, thus leaving JS engines under-tested.**

We propose DUMPLING, a differential fuzzer that compares the full state of optimized and unoptimized execution for arbitrary JS programs. Instead of instrumenting the JS input, DUMPLING instruments the JS engine itself, enabling deep and precise introspection. These extracted fine-grained execution states, coined as (frame) *dumps*, are extracted at a high frequency even in the middle of JIT compiled functions. DUMPLING finds eight new bugs in the thoroughly tested V8 engine, where previous differential fuzzing approaches struggled to discover new bugs. We receive $11,000 from Google's Vulnerability Rewards Program for reporting the vulnerabilities found by DUMPLING.

I. INTRODUCTION

Web browsers are the portal to the World Wide Web. Users directly run JavaScript (JS) code on their devices, which is delivered from websites, but also embedded advertisements and third-party front-end frameworks. Due to the ubiquitousness and attack surface, JS engines are the focus of security researchers and malicious attackers, who find bugs that can be used to gain arbitrary code execution. Therefore, it is important to improve vulnerability discovery and broaden the class of automatically detectable vulnerabilities.

Modern web applications heavily rely on the performant execution of JS code. The JS engine has evolved into one of the most crucial components of a web browser. To meet performance expectations, JS engines use optimized *just-in-time (JIT) compilation* [1]. Modern JS engines include multiple execution tiers implementing the same semantics: the interpreter and one or more JIT compilers. Execution tiers represent different trade-offs between execution time and compilation time and are selected based on runtime-profiled usage at function granularity. Different from C/system code, a JS engine may switch between different optimizations at arbitrary points during execution, even during the execution of a function. The interpreter and the compiler's emitted code must have the same semantics even after compiler optimizations. A mismatch in semantics can be as subtle as a JIT compiler wrongly assuming a float value to be 0.0 when it could also be the distinct float value -0.0. This real-world example of a bug in the V8 JS engine allowed arbitrary code execution in the Chrome renderer [2].

The above bug belongs to a logic bug class (Typer bug) unique to JS engines [3], [4]. Starting from a proof-of-concept that merely leads to the JIT compiler making a wrong assumption (e.g., typing a -0.0 value as 0.0), multiple steps are necessary to make the bug observable without introspection (e.g., debugging) into the JS engine. These steps involve, first, making the compiler act on the wrong assumption by producing a wrong value (miscalculation). Second, preserving this wrong value, such that it is propagated during subsequent optimization and execution. Third, manipulating and passing the value to a sink that finally makes the differential observable. The bug may be observable as a crash due to memory corruption, a violation of a manually inserted assertion in the JS engine, or by a wrong output print. Like Typer bugs other phases of JIT compilation regularly feature their own classes exploitable vulnerabilities that share the property of *miscalculation before memory corruption* [5], [6].

Fuzz testing (*fuzzing*) is a powerful technique to detect errors in programs [7]. Fuzzing relies on *bug oracles* for detecting bugs, such as detecting crashes and compiling with address sanitizers [8]. Address sanitizers are limited in their effectiveness for JS engines, as they heavily leverage JIT compiled code and handwritten assembly, which cannot be instrumented. While assertions are an effective bug oracle, they need to be manually inserted by developers and can only check locally available information. To go beyond memory corruption bugs and also detect logic bugs, differential fuzzing is a promising strategy. Differential fuzzing executes a large number of inputs against two implementations of the same

Network and Distributed System Security (NDSS) Symposium 2025
24–28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241411
www.ndss-symposium.org

specification and reports a bug if their outputs differ [9], [10], [11]. If the implementations are not in agreement, the tuple of differing output prints is called a *differential*. Differential fuzzing across JS engines needs to manually account for implementation specific behavior that the ECMAScript specification [12] leaves open [13], [14], [15]. Instead, differential fuzzing can also be done between different execution tiers of the JS engine, such as the interpreter and optimizing JIT compilers [16], [17], [18]. State-of-the-art differential fuzzers for JS engines share the design choice of probing the execution state at JS level with techniques similar to print statements of individual variables. This design choice positions them as generic approaches since they can be easily deployed on different JS engines. However, relying on probes inside the JS code constrains the observation to only a small subset of the execution state, while such bugs can appear in any part of the execution state [19]. Furthermore, injecting JS probes alters the JS code and may inhibit JIT compiler optimizations, and therefore leave the JS engine under-tested.

We propose DUMPLING, a differential fuzzer that extracts and compares the execution state of the JS engine between optimized JIT compiled code and unoptimized code without any modification of the JS code. The key innovation of DUMPLING is to enable a fuzzer to observe the full execution state of the JS engine, instead of only comparing single variables, while not interfering with JIT compiler optimizations. To achieve this goal, we propose a new introspection mechanism for modern JS engines that allows for such state extraction at a high frequency, even in the middle of JIT compiled functions. DUMPLING's representation of individual objects is more in-depth and fine-grained than previous approaches [18], [17], [20]. This new level of sensitivity enables the fuzzer to detect bugs *before* they become observable as wrong output prints or memory corruptions. Another key challenge is to avoid false positives. During our evaluation, we observe that existing differential fuzzing approaches suffer from a high false positive rate [18], [17]. Without a fuzzing approach avoiding false positives, true positives are hidden between many false alarms, which hinders efficient bug fixing in practice. DUMPLING is designed and carefully tuned to avoid false positives, as demonstrated in our evaluation.

DUMPLING's design revolves around the general principles of JS JIT compilers described by Gal et al. [1]. Gal et al.s' work forms the cornerstone of contemporary JS engines. In our work, we choose the V8 engine as the most used implementation of Gal's principles, holding a market share of 75% for web browsers [21]. V8 is built for Google's Chrome and Chromium browsers, but it is also used in Microsoft Edge and Opera. Furthermore, V8 is running in other contexts as well, for example, server-side JS environments, desktop applications, mobile apps, database management systems, and PDF readers. We believe that the ubiquity, criticality, and ongoing discovery of new vulnerabilities warrants research in domain-specific bug oracles for JS engines, at the example of V8. Section VI discusses the porting effort of DUMPLING to other JS engines.

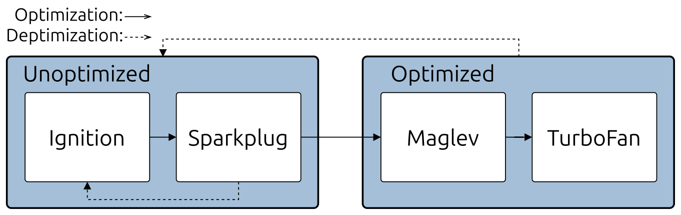


Fig. 1: V8 compiler's tiers, based on Maglev's design [22].

In our evaluation, we compare the overhead, sensitivity, and specificity of DUMPLING to state-of-the-art JS engine differential fuzzing approaches. We show that DUMPLING's bug oracle is significantly more sensitive than the state-of-the-art while maintaining a similar performance in terms of coverage and execution speed. This sensitivity is quantified as the entropy of information given to the bug oracle of which DUMPLING improves on the state-of-the-art by around 30%. Furthermore, DUMPLING's bug finding capabilities are demonstrated by identifying ten bugs in V8, eight of which are new.

The contributions of this paper are as follows:

1) We design a bug oracle that probes the execution state of a JS engine during execution of arbitrary programs. The generated dumps provide fine-grained insight of the execution state, do not modify execution semantics, and have a concise format (allowing for efficient transmission).
2) We propose DUMPLING, a differential fuzzer that employs our bug oracle to find vulnerabilities in the well-tested V8 JS engine. Our evaluation shows that DUMPLING finds more bugs than the state-of-the-art, while we do not observe any false positives from DUMPLING.

We publicly release the prototype of DUMPLING and the necessary materials to replicate our experiments at https://github.com/two-heart/dumpling-artifact-evaluation.

In the following, we introduce core JS engine internals, that are fundamental for understanding the design of DUMPLING. We choose to use V8 terminology throughout the paper to make the implementation easier to follow. Section VI maps this terminology to the equivalent concepts in other JS engines. V8 consists of four execution tiers, the bytecode interpreter Ignition, the non-optimizing baseline compiler Sparkplug, the mid-tier compiler Maglev, and the most optimizing compiler TurboFan. The V8 execution tiers and their interactions are depicted in Figure 1. DUMPLING's design revolves around the concept of execution state and the interaction between the execution tiers.

II. BACKGROUND AND MOTIVATION

In the following, we introduce core JS engine internals, that are fundamental for understanding the design of DUMPLING. We choose to use V8 terminology throughout the paper to make the implementation easier to follow. Section VI maps this terminology to the equivalent concepts in other JS engines.

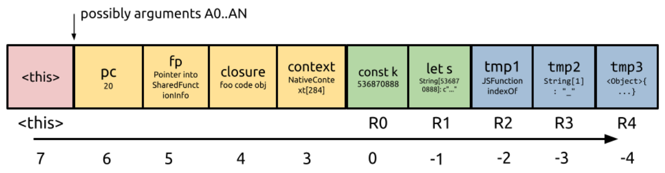


Fig. 2: Example execution state of a JS function: Virtual stack frame used in the bytecode virtual machine. The accumulator is pinned to a hardware register. The notation follows the format used in Ignition [23].

V8 consists of four execution tiers, the bytecode interpreter Ignition, the non-optimizing baseline compiler Sparkplug, the mid-tier compiler Maglev, and the most optimizing compiler TurboFan. The V8 execution tiers and their interactions are depicted in Figure 1. DUMPLING's design revolves around the concept of execution state and the interaction between the execution tiers.

A. Unoptimized Execution

Ignition translates JS into bytecode and interprets it instruction by instruction in a register-accumulator virtual machine [23]. For instance, a simple addition of the accumulator with a register `Add r0, (1)`, can result in one of many different implementations depending on the types of the operands and special cases in the ECMAScript specification [12]. In the example instruction, `(1)` is not an operand to the addition in the mathematical sense, but an entry in the *feedback vector* [1]. Early tiers collect type feedback from instructions and store it in such feedback vectors, to later base speculative optimizations on. Technically feedback is already used in Ignition to a limited extent, to more quickly find the right implementations for the types (fast paths), but interpreted code still has an inherent overhead over machine code. Nevertheless, JS always starts execution in Ignition, which can start execution without waiting for compilation to finish. Type feedback must be collected for sensible optimized JIT compilation to occur.

Sparkplug is a JIT compiler designed for quick compilation [24]. Sparkplug walks the bytecode once, lowering every bytecode instruction verbatim to machine code, without speculating on operands. As such the bytecode structure is preserved. Therefore, Sparkplug can be thought of as accelerated interpretation. This allows DUMPLING to set up hooks at any bytecode instruction in code that is JIT compiled by Sparkplug for our introspection.

B. Execution State

The register-accumulator virtual machine of Ignition and Sparkplug directly acts on the execution state. The execution state is tracked per function in a (virtual) stack frame. Such a stack frame along with example values is shown in Figure 2. The accumulator, that is also part of the execution state, is usually pinned to a hardware register for efficiency [23], [25]. A stack of stack frames forms the execution state of a JS program. This execution state contains all information necessary for unoptimized tiers to continue correct execution at any bytecode instruction. Note that the stack does not just contain local JS variables, but also any used global variables.

C. Optimized Execution

V8 contains two optimizing JIT compilers, Maglev and TurboFan. Maglev, a relatively recent addition to V8, serves as a mid-tier compiler [22]. It offers a middle ground between compilation time and code optimization. Only after a function is heavily used it will be JIT compiled with TurboFan, which performs more complex optimizations such as range analysis. Maglev and TurboFan have independent implementations of optimization passes, on their own intermediate representations. However, they are conceptually similar in that they both speculate on collected runtime feedback to perform optimizations. In the previous addition example, one such speculation could be that this instruction will continue to add unsigned integer values less than one machine word. Finally, efficient machine code is emitted that implements the assumed special case.

D. Deoptimization

Speculative JIT compilation requires the guarding of specialized code by runtime checks at so-called *deoptimization points*. If an assumption is violated that further code relies on, the execution needs to continue seamlessly in a non-optimizing tier [26]. Therefore, every deoptimization point points to a bytecode location where execution continues if its guarding check is violated. As defined in Section II-B, unoptimized execution requires the virtual stack frame (and those of inlined functions) to be reconstructed. This part of deoptimization is called *rematerialization* and restores the stack frame to the extent that it is changed by the JIT code at this point. After this, execution will continue in Ignition or if there is Sparkplug code, in Sparkplug. TurboFan and Maglev share an implementation of the deoptimization mechanism starting from rematerialization. For the sake of simplicity, when discussing *deoptimization points*, we implicitly mean *eager deoptimization*, which is distinct from *lazy deoptimization* [27].

Some code patterns will only be generated by optimizations, when a function is compiled once, then deoptimized, called with different feedback, and finally recompiled. For detecting JIT vulnerabilities with a fuzzer it is important to generate inputs that have such behavior with diverse feedback. Previous work wraps all inputs in a template that forces JIT compilation with debug syntax in the single constrained way [17]. Instead, we rely on Fuzzilli, an existing approach to generate code triggering JIT optimization with diverse feedback [3].

III. DESIGN

DUMPLING is a differential fuzzer that deeply introspects the JS engine to detect divergences between optimized and unoptimized code. All modern JS engines root their concepts in Gal et al.'s work [1]. Among them, V8 stands out as the most successful instance with a 75% share for web browsers [21]. We discuss the adoption of DUMPLING in other JS engines in Section VI.

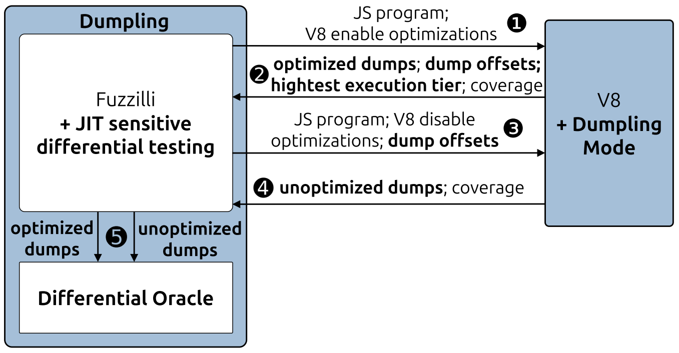

Dumpling
JS program; V8 enable optimizations ❶
❷ optimized dumps; dump offsets; hightest execution tier; coverage
Fuzzilli + JIT sensitive differential testing
V8 + Dumpling Mode
JS program; V8 disable optimizations; dump offsets ❸
❹ unoptimized dumps; coverage
optimized dumps ❺ unoptimized dumps
Differential Oracle

Fig. 3: Design and protocol overview of DUMPLING. Contributions in bold.

DUMPLING's design addresses three key challenges. First, we aim to extract the full execution state at a high frequency and represent it in a detailed and comparable way without changing the execution semantics. To achieve this, we add instrumentation to the JS engine itself (Section III-A). Second, DUMPLING needs to compare the dumps from optimized and unoptimized execution. To this end, we design a differential oracle based on a novel algorithm that efficiently matches the states extracted from different JS tiers and reports any inconsistency observed (Section III-B). Finally, non-deterministic behaviors in the JS engines may lead to false positive alarms. To address this issue, we propose a practical technique that we detail in the implementation (Section IV-C).

DUMPLING's goal is to detect differential bugs in optimized JS executions. Therefore, the architecture is designed to detect the activation of the JIT compilers and enable the differential oracle only when needed. Moreover, we aim to extract and transmit the execution state from the engine to the differential oracle with minimal time and space overhead. Figure 3 shows DUMPLING's components and their interaction at a high level. In ❶, the fuzzer produces a JS program and inputs it to the JS engine configured to use JIT optimizations. In the first run, we enable all tiers of the engine's compiler pipeline, cf. Figure 1. More precisely, V8 is extended with the so-called DUMPLING *Mode*, which dumps the execution state both during optimized JIT code and unoptimized execution in a canonical comparable format. In ❷, the dumps are returned to the fuzzer, along with the highest tier that was used, the positions where the dumps were captured, and the exercised coverage. In cases where optimizing tiers are used, ❸ starts a second execution of the same input program with the optimizing tiers disabled. The second execution produces an unoptimized execution state that serves for comparison with the optimized one. Compared to prior work [18], our technique (*JIT sensitive testing*) steers executions to target JIT bugs. DUMPLING Mode facilitates the dumping of the execution state at the positions supplied in ❸ and returns them in ❹. Finally, the differential oracle receives the dumps from the first and second run in ❺, and compares them to detect divergences.

While we implement our prototype based on Fuzzilli, a widely used state-of-the-art JS engine fuzzer [3], our approach is compatible with any JS input program from any fuzzer or another source. In Section VI, we describe a standalone Python implementation that allowed us to uncover a new JS bug in the latest version of V8 by simply re-executing the official V8 unit test suite.

A. *State Extraction:* DUMPLING *Mode*

DUMPLING Mode allows the fuzzer to obtain a consistent representation of the JS execution that will be then used by our oracle to identify inconsistencies between optimized and unoptimized JS executions (Section III-B). More specifically, DUMPLING Mode instruments V8 to dump the execution state during the execution of JIT compiled functions and at the corresponding positions in the unoptimized tiers. DUMPLING Mode produces three types of dumps: (i) *frame dumps* (cf. `FRAME` type in Listing 3) for representing internal function states, (ii) function entry (`FUNC_ENTER`), and (iii) function exit (`FUNC_EXIT`). The combination of these dumps allows the differential oracle to detect differentials (Section III-B).

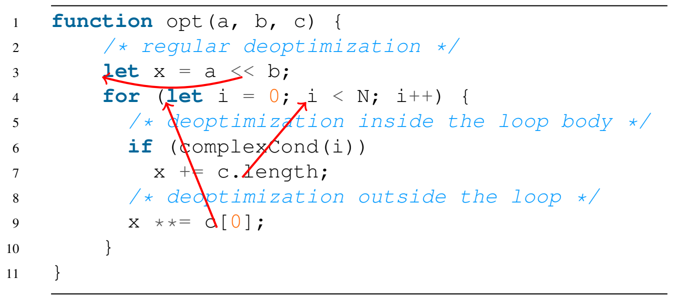
1) State Extraction in Optimized Tiers: Obtaining the execution states from highly optimized JIT compiled code is nontrivial. At a high level, we address three technical challenges in the design of state extraction from optimized tiers. First, obtaining the execution state that is tracked a priori in machine code, such that it is comparable with the state in unoptimized tiers. Second, finding suitable locations at which to extract the execution state, providing high-frequency low overhead introspection into JIT compiled code. Third, we aim to address the previous two challenges without affecting JIT compiler optimizations.

To address the first challenge, we rely on the following observation: Speculative JIT compilers perform a deoptimization back to unoptimized code if the check at a deoptimization point is violated, as described in [1], [27], [28]. Therefore, JS engines have a mechanism to restore the bytecode virtual machine state when transitioning from optimized JIT compiled code to a bytecode location in an unoptimized tier. This mechanism is called rematerialization (Section II-D). The rematerialization restores the JS state for the unoptimized tiers to continue execution on this same state. The same is true at every return from a JS function to unoptimized code. We leverage rematerialization to obtain the frame state at every deoptimization point and such returns. Concretely, we perform the equivalent of the first part of deoptimization, i.e., rematerializing the frame state that the unoptimized tier would have if deoptimized. However, instead of executing the unoptimized code, we continue with the JIT compiled code.

For the second challenge, we leverage the following insight: Deoptimization points are scattered in the JIT compiled code and the return captures the state at the end of a function. Moreover, deoptimization points have the property of being inserted after complex operations that the compiler cannot guarantee to be side effect free on the deoptimization condition [27]. As such, we consider deoptimization points as natural probing positions after interesting operations in the

```
-------TurboFan frame dump-------
pc: 7
acc: 13.37
a0: <Object>{
__proto__: <Class C7>{<String[1]: f>[enumerable]<JSArray>[]},
<String[1]: a>[configurable][enumerable]42(enum cache: 2),
<String[1]: f>[configurable][enumerable]13.37(enum cache: 0)
}
r0: -INFINITY
context: <ScriptContext[4]>
receiver: <JSGlobalProxy>
closure: <JSFunction f0>
Function ID: 27
```

Listing 1: A verbose example of a frame dump.



```
1  function opt(a, b, c) {
2      /* regular deoptimization */
3      let x = a << b;
4      for (let i = 0; i < N; i++) {
5        /* deoptimization inside the loop body */
6        if (complexCond(i))
7          x += c.length;
8        /* deoptimization outside the loop */
9        x **= c[0];
10     }
11 }
```

Listing 2: Visualization of deoptimizations from optimized code to earlier points in unoptimized code. Arrows indicate control-flow transfers from optimized to unoptimized if a deoptimization is triggered.

JS engine. Additionally, for completeness, we dump after deoptimization was performed during normal execution.

Finally, we guarantee not to influence the compiler optimizations by positioning our non-invasive hooking at the end of the compiler pipeline, i.e., after all optimizations have completed.

A beneficial byproduct of our approach is that we intensively test the rematerialization code, which during normal fuzzing is less frequently used. Rematerialization featured impactful and hard to spot vulnerabilities in the past [29].

2) State Extraction in Unoptimized Tiers: Optimized states are extracted at every deoptimization point. Deoptimization points point to *target bytecode locations* in non-optimizing execution tiers. The *target bytecode locations* are used in the unoptimized run as *dump locations* (❷, ❸). Consequently, to obtain the dumps from unoptimized execution, DUMPLING hooks the *dump locations*. Once a hook is called, the frame state that unoptimized execution directly operates on is obtained and dumped.

3) State Serialization: Once rematerialized, the dumps need to be transferred to the differential oracle for comparison for matching and comparison of the frame states. Listing 1 shows an example frame dump from TurboFan execution, which contains the main frame information described in Section II-B. The design goals of our serialization are three-fold (i) the dumps must adhere to a canonical format that is invariant across execution tiers; (ii) the dumps must be fine-grained to detect subtle differences in the execution state, such as -0 vs. 0; (iii) the dumps must be concise to minimize the overhead of transmitting the state to the differential oracle.

To address these challenges, we define a custom serialization format for all possible types. The serialization format ensures that the dumps only contain invariant information. In contrast to other solutions, e.g., V8's `%DebugPrint`, we strip all memory addresses and tier specific data structures. We also avoid using `JSON.stringify`, used in prior work [20], as it does not provide the necessary level of detail. To limit the size of an individual frame dump, we control the traversing of JS objects by two parameters: depth and number of properties. By modeling a JS object as a graph structure, the first parameter indicates how deep the serialization moves into the graph, while the second one indicates the number of child nodes to be visited. Setting the ideal dumping parameters is a trade-off between performance and sensitivity: a shallow dump may miss crucial state inconsistencies but improve executions per second. In Section V, we evaluate the impact of this hyperparameter and observe that it has only a minor impact on performance. An effective mechanism for reducing the size of dumps is to only include fields in the dump that changed compared to the previous frame dump. The strings in Listing 1 are expanded for legibility, in the actual prototype we choose shorter representations that carry the same information. A third minor optimization is to explicitly include certain properties only if they differ from the default value, like the prototype of an object.

B. Differential Oracle

The goal of the bug oracle is to detect any divergence in execution states between optimized and unoptimized execution. After the dumps from the unoptimized execution are received (❹), they are passed together with the dumps from the optimized execution to the bug oracle (❺). Matching optimized and unoptimized frame states may appear simple at first glance. However, reviewing the inherent properties of deoptimization points reveals that a simple "one-to-one" mapping does not exists. We first describe this challenge, and then present our matching algorithm that is both robust and sensitive.

To understand the problem of frame matching, consider Listing 2 that visualizes three cases of deoptimization points. For readability, and without loss of generality, we use the corresponding positions in source code instead of bytecode and machine code. More specifically, the code contains three deoptimization points: (a) line 3 shows a regular deoptimization that points to the same line, (b) line 7 points to line 4 inside the loop body, and (c) line 9 points to line 4, the declaration before the loop body. While the compiler tries hard to avoid cases (b) and (c) for efficiency, they can still occur, and are here illustrated in a relatively small code example. Consider the deoptimization (b) at line 7: if `complexCond(i)` is true, we pass by the deoptimization point. Therefore, optimized execution will dump a frame state that is valid for the bytecode offset in unoptimized execution pointed to in line 4 (Section III-A1). Conversely, since line 4 is pointed to by the deoptimization in line 7, the unoptimized execution dumps a frame state every time line 4 is traversed (Section III-A2). During the unoptimized execution, we observe N dumps from this bytecode location, but only $|\{i \mid i \in \text{complexCond}(i) \wedge \{0, \ldots, N-1\}\}|$ dumps in optimized execution. Therefore, we do not know which of the frames produced at line 4 corresponds to the respective optimized frame. Static analysis with the given is theoretically impossible, as in this example `complexCond` can be any Turing-complete calculation. Furthermore, there exists no exact mapping from machine code instruction to bytecode offset due to compiler optimizations, making dynamic tracing impossible. Conversely, deoptimization point (c) shows a case where there is one dump in unoptimized execution, but N dumps from optimized execution. In even more complex cases, a function may get deoptimized and recompiled multiple times in the same invocation. Therefore, we believe that any heuristic attempting an "exact" mapping will miss bugs or be false positive prone.

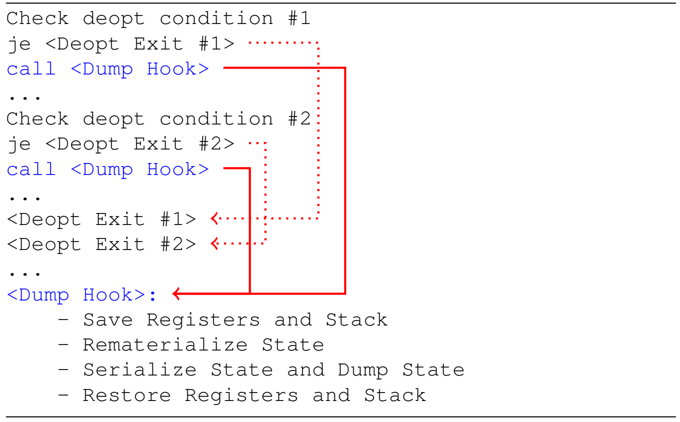
We describe why a generally accurate matching between frames is not necessary for the detection of divergence in execution. In fact a relatively simple algorithm is sufficient to solve the problem. Our algorithm relies on two invariants. First, every frame state in the optimized execution must appear in the unoptimized execution, for execution to be correct. Second, by the pigeonhole principle [30], if a program state, including the program counter, repeats, the program is in an infinite loop. Programs with infinite loops will hit the timeout of the fuzzer, and therefore never reach the bug oracle. Based on these observations, we design an oracle that works in two steps, its pseudocode is shown in Listing 3. First, the `group` function groups frame dumps by function invocation by walking the call stack demarcated by the aforementioned function enter and exit dumps. Inlining is not a problem because V8 maintains the state of the inlined function separately to perform correct deoptimizations. Since groups represent function invocations, every correct JS program execution has to contain the same number of groups in optimized and unoptimized runs. If this invariant is violated, a differential bug is found and reported by our oracle (line 15). Second, the `match` function checks for every invocation of a JIT compiled function if each of its frame dumps appears in the unoptimized frame dumps of the same function invocation. Note, this matching algorithm can be applied to any speculative JIT compiler that deoptimizes to recent bytecode offsets as described in Gal et al.'s work [1].

IV. Implementation

The following illustrates the modification required to implement Dumpling. More precisely, we discuss the V8 modifications to extract the execution state in Section IV-A, while

```
1  def group(dumps):
2      stack = [], chunks = []
3      for dump in dumps:
4          if dump is FUNC_ENTER:
5              stack.push([])
6          elif dump is FUNC_EXIT:
7              chunks.push(stack.pop())
8          elif dump is FRAME:
9              stack[-1].push(dump)
10     return chunks
11
12 def match(opt_dumps, unopt_dumps):
13     opt_groups = group(opt_dumps)
14     unopt_groups = group(unopt_dumps)
15     if len(opt_groups) != len(unopt_groups):
16         return False
17     groups = zip(opt_groups, unopt_groups)
18     for (opt_group, unopt_group) in groups:
19         for opt_frame in opt_group:
20             if opt_frame not in unopt_group:
21                 return False
22     return True
```

Listing 3: Pseudocode depicting the implementation of the matching algorithm.


```
Check deopt condition #1
je <Deopt Exit #1>
call <Dump Hook>
...
Check deopt condition #2
je <Deopt Exit #2>
call <Dump Hook>
...
<Deopt Exit #1>
<Deopt Exit #2>
...
<Dump Hook>:
    - Save Registers and Stack
    - Rematerialize State
    - Serialize State and Dump State
    - Restore Registers and Stack
```

Listing 4: Pseudo-assembly code with JIT hooks that Dumpling inserts at every deoptimization point.

the modifications for Fuzzilli are elaborated in Section IV-B. Finally, we describe our approach to reduce false positives in Section IV-C.

A. V8 Modification

To implement Dumpling Mode, we add 4,580 LoC to V8 based on commit 6e70dc9 (2024-03-21). An inherent design goal of Dumpling is to transparently extract execution states without any modification to JS or interference with the JIT compiler optimizations. Therefore, we insert our hooks in a way that is not observable by any pass in the compilation pipeline. We implement the hooking by emitting a single `call` instruction at every deoptimization point during the lowering phases to machine code of Maglev and TurboFan. Since the respective lowering phases run as the last phase during compilation, it is guaranteed that Dumpling's hooks do not interfere with any compiler optimization. Once a deoptimization point is passed, the inserted hook is called, performing the reconstruction of the corresponding virtual machine stack frame, object rematerialization and serialization of the constructed frame. Calling the hook modifies CPU registers and the hardware stack state that further JIT compiled code relies upon. Consequently, we create a back-up of this information before executing our hook and restore it before returning to JIT compiled code. Listing 4 illustrates our instrumentation with hooks of JIT compiled code produced by TurboFan or Maglev. Deoptimization points in V8 always follow the same structure. First, the compiler inserts a check that has to pass in order for the following code to function correctly. This check determines during runtime, if the JIT compiled code can continue to execute or if it has to be deoptimized. In case the check passes and execution continues in the JIT compiled code, the hook is called and performs the previously mentioned operations to serialize the current state. V8 supports both lazy and eager deoptimizations (Section II-D). Our prototype focuses exclusively on eager deoptimizations for two reasons. First, supporting lazy deoptimization involves only engineering effort. Second, including lazy deoptimizations would result in higher overhead and reduced throughput. Therefore, we consider hooking only eager deoptimizations to be the most practical solution for achieving an efficient prototype with a meaningful security impact.

To additionally capture the less likely case of a deoptimization condition being met, DUMPLING also dumps the state before execution continues in an unoptimized tier. Dumping the *continuation state* in unoptimized execution after the engine performs a deoptimization allows DUMPLING to specifically catch errors during the deoptimization process. Additionally, once a compiled function finishes and returns to unoptimized execution, the state is dumped as well. Incorporating those frames ensures that every function invocation of compiled code has at least one dump associated with it, even if no deoptimization point was generated for the corresponding function.

To allow state extraction for unoptimized code, Ignition is instrumented by inserting a hook at the beginning of every bytecode handler. This approach inserts a hook into every bytecode handler thus automatically tracing every bytecode instruction. We perform a lightweight check to determine if the current bytecode location is a dump location. With the Ignition frame already being in place, no rematerialization has to be performed and only the serialization has to be done in the hook.

This same approach can be applied to Sparkplug as well. During the single pass that Sparkplug does over the bytecode, it directly maps each bytecode instruction to a piece of machine code. DUMPLING inserts hooks at the beginning of each resulting machine code piece. Effectively, this provides the same capabilities, with regards to dumping, as the hooks in the bytecode handlers, allowing state extraction in Sparkplug.

B. Fuzzilli Modification

To leverage the extracted states as the information for our differential oracle during fuzzing, we extend Fuzzilli. DUMPLING is based on Fuzzilli commit 5696921 (2023-11-19). Fuzzilli is written in Swift. We add or modify 1,773 LoC in Fuzzilli, implementing the JIT sensitive differential testing and differential oracle components of DUMPLING. Concretely, we extend Fuzzilli with the capability of differential executions (Section III). JIT-Picker [18] already provides a good foundation to implement differential executions on top of Fuzzilli from which we take inspiration. Furthermore, we implement deserialization of dumps, and the differential oracle described in Section III-B. To aid our evaluation, we extend Fuzzilli to measure additional runtime statistics.

C. Avoiding False Positives

For differential fuzzing, assuming a correct execution, we require that repeating the execution of the same JS code traverses the same states. A violation of this invariant would lead to false positives being reported by a differential oracle, without further mechanisms for false positive avoidance. However, JS permits valid operations that introduce non-deterministic behavior. Examples of this include `Math.random()` and `Date.now()` but also performance and memory measurement functionality that is exposed to JS. State-of-the-art differential fuzzers [18], [17] proposed different techniques to avoid false positives. Unfortunately, we observe the available artifacts still suffer from this problem (Section V-A). In practice, it is easy to provide a JS program to any of the state-of-the-art differential fuzzers that will trigger a false positive. Therefore, we systematically study the implementation of V8 and design DUMPLING to use V8 deterministically. During the implementation of DUMPLING, we also asked multiple experts in JS programming and JS engine exploitation to provide false positives, but none were found. Furthermore, we encounter no false positives in our evaluation runs (Section V). We therefore believe that our study provides practical insights on how to avoid false positives also for other researchers.

We divide our approach to avoid false positives into multiple techniques. First, we perform a best effort approach to find and mock non-deterministic behavior inside the JS engine by patching known non-deterministic functionality inside V8 to return deterministic values. While V8 provides a `predictable` runtime flag that is used by some prior work [18], we observe this technique is insufficient to avoid all non-deterministic behaviors. In addition, false positives can arise from native syntax and stack-depth measurement. These can be partially mitigated by the `-allow-natives-for-differential-fuzzing` flag and the `correctness-fuzzer-suppressions` flag. For those behaviors that are in scope for the options but not currently covered, we developed and submitted our patch to the V8 team. Similar to JIT-Picker [18], we re-execute both runs to verify that they produce the same output dumps [16]. Additionally to re-executing, we prepend JS code with wraps

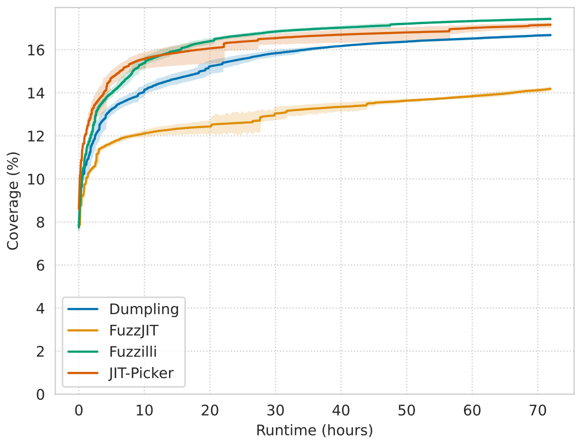


Fig. 4: Stable edge coverage [3] graph showing the 95% confidence interval in a ten fold cross evaluation.

to some JS functionality that otherwise has intended non-deterministic behavior. This includes a rather small set of functions, namely `Realm`, `FinalizationRegistry`, `Worker`, `Atomics.waitAsync`, typed array maximum size, NaN patterns inside `DataView`, stack traces, and `Math.pow` with an argument with exponent -0.5. We base this functionality on the preexisting mocking of jsfuzzer [16]. Installing the mocks only in case a differential is provisionally detected the introduction of false positives while imposing only minimal overhead. Only if a differential is detected again on re-execution, the sample is saved as a differential. Otherwise, the sample is discarded and removed from the corpus to avoid the generation of further samples that exhibit non-deterministic behaviors.

V. Evaluation

We evaluate the effectiveness of Dumpling and compare it against the state-of-the-art by answering the following research questions:

1) Can Dumpling discover yet unknown bugs in the newest version of the thoroughly tested V8 engine? (Section V-A)
2) How sensitive is the differential oracle of Dumpling? (Section V-B)
3) What overhead does our bug oracle introduce? (Section V-C)
4) How do hyperparameters influence Dumpling? (Section V-D)
5) What is the cost to maintain Dumpling? (Section V-E)

Baselines and Parameters: We compare Dumpling against Fuzzilli as the state-of-the-art in non-differential JS engine fuzzing [31]. Furthermore, Dumpling is compared against JIT-Picker [18] and FuzzJIT [17] as the state-of-the-art in differential fuzzing of JS engines. Both are based on Fuzzilli and are therefore comparable with respect to the effectiveness of the proposed differential oracle. We run FuzzJIT (a3d3f6d) and Fuzzilli (722e036) as-is on their newest commit at the time of writing. FuzzJIT is based on an older version of Fuzzilli. Their custom code generation makes it infeasible for us to rebase it to the newest version of Fuzzilli, and preserve the properties of their grammar modifications. For JIT-Picker, we use the version of their Pull Request (#378, 7a56b45) to Fuzzilli, which contains bug fixes compared to the research artifact. We safely rebase JIT-Picker to the current version of Fuzzilli, which Dumpling and the Fuzzilli baseline both use. JIT-Picker includes a feature for disabling random compiler optimizations [18]. However, we observe that this feature changes the settings for each fuzzing campaign. Therefore, we disable it to always obtain the same JIT-Picker configuration. Dumpling and FuzzJIT are run in their default configuration, i.e., no flags are set except for exporting statistics. For JIT-Picker, we had to set two parameters that indicate the frequency at which the probes are injected. Since these settings are not provided in the paper or the artifact, we contacted the authors who recommend 0.01 for the “inline hash rate” and a value in the range $[0.01, 0.1]$ for the “out-of-line hash rate”. We choose 0.075 for the latter.

Experimental Setup: The evaluation is conducted on an AMD EPYC 7302P 16-Core Processor. We allocate 15 cores to the respective fuzzer. The machine has 64 GB RAM and is running Ubuntu 22.04. If not mentioned otherwise, we conduct a ten-fold cross-evaluation with three-day runs. We do not use a seed corpus, but let the fuzzer generate the initial corpus. We adopt this configuration by following JIT-Picker recommendations, which are more extensive than FuzzJIT (i.e., 24-hour $\times$ ten-folds). Our setup shows a 95% confidence interval, thus giving sufficiently stable results to draw robust conclusions.

A. Bug Finding Capability

We investigate the bug-finding capabilities of Dumpling under multiple aspects. First, we analyze the bugs detected from the aforementioned ten-fold evaluation campaign and sporadically during development. Then, we study the false positive reports observed by JIT-Picker and FuzzJIT and investigate their causes. Finally, we illustrate two interesting bugs as case studies of the errors discovered by Dumpling, one of which is illustrated in Section A-A. From the reports found in our evaluation, Google’s Vulnerability Rewards Program [32] recognizes a bounty of $11,000 due to their security relevance.

Dumpling finds ten bugs in the well-tested V8 engine. The continuous testing of V8 currently includes large-scale fuzzing with multiple specialized JS engine fuzzers, unit tests, regression tests, telemetry from users, and incentives in the form of bug bounties for outside parties to report bugs [33], [32]. We report all discovered bugs to Google in a coordinated way. Of the ten bugs found in total, four are found during the evaluation runs, while the remaining six are found during development. We count bugs initially found during development, that are rediscovered during the evaluation runs, as found during the evaluation runs. Table I shows the bugs found by Dumpling and differentiates them by their triggered

TABLE I: Bugs discovered by DUMPLING. All issues CR are accessible at `crbug.com/<number>`. The column "By" indicates the fuzzer finding the bug: D - DUMPLING, J - JIT-Picker. The status "available" indicates a bug that is acknowledge by V8's developers but has not yet fixed.

Bug Id	**Kind**	**Status**	**Changes**	**When**	**By**	**Description**
CR41488094	Diff	fixed	+28/-23	eval	D, J	Enumerating properties eagerly, has incorrect side effect
CR335310000	Diff	fixed	+15/-0	eval	D	Property not marked as enumerable by Maglev and TurboFan
CR332745405	Diff	fixed	+5/-0	eval	D	DefineOwnProperty called the setter of an existing accessor property
CR329330868	assert	dup	N/A	eval	D, J	array.shift does not update pointers in spill slots
CR41484971	Diff	fixed	+52/-40	dev	D	Store inline cache handlers are incorrectly used for defining properties
V8:14605	Diff	fixed	+1/-1	dev	D	The JumpLoop bytecode does not clobber the accumulator in all cases
V8:14556	Diff	available	N/A	dev	D	The arguments array is handled differently in optimizing compilers
CR345960102	Diff	fixed	+6/-4	dev	D	TurboFan incorrectly optimizes 64 bit BigInt shifts
CR346086168	Diff	fixed	+109/-107	dev	D	Overflow in BigInt64 shift optimization
CR40945996	assert	dup	N/A	dev	D	The profiler in Maglev interferes with deoptimization

bug oracle. We found CR345960102 and CR346086168 while triaging one differential that pointed to two distinct bugs in the TurboFan optimization of `BigInt`. More interesting, the bug CR346086168 represents a subtle semantic error derived by an incorrect implementation of the shift operation for `BigInt` types. This error can affect cryptographic algorithms thus showcasing the capability of DUMPLING to explore attack surfaces beyond classic memory corruptions. We leave a detailed report of this bug in Section A-A.

While we use our differential oracle, existing bug oracles remain active. This allows DUMPLING to detect two debug assertion failures, both are marked as duplicates. Duplicate entries cannot be avoided, because unfixed potential security-relevant bugs are only published 14 weeks after the fix in the Chromium bug tracker. Therefore, it is common that bugs found by a preexisting bug oracle in Fuzzilli are discovered by Google internal fuzzing first. Nevertheless, finding traditional bugs with DUMPLING demonstrates that our approach broadens the classes of discoverable bugs compared to Fuzzilli.

Eight of our discovered bugs are differentials. The reward received from Google's Vulnerability Reward Program demonstrates DUMPLING's ability to find security vulnerabilities.

With a total of eight reported bugs that are not marked as duplicates, we conclude that DUMPLING uncovers yet unknown bugs in the newest version of V8 and that the attack surface explored by DUMPLING falls beyond memory corruption errors.

To evaluate the false positive rate of DUMPLING compared to the state-of-the-art, we manually analyze every differential found by the fuzzers. FuzzJIT reports 54 differentials during the ten three-day runs. After analyzing the reports, we conclude that all differentials found by FuzzJIT are false positives. The cause of the false positives is attributed to FuzzJIT not doing two isolated runs of the engine, but calling a function twice, i.e., optimized and unoptimized. This strategy might leave artifacts behind, for example in global state. As such, the differential is observed because of an artifact in the first function invocation. Manual triage of the differentials indicates that 41 differentials come from a combination of first modifying the prototype of an object and then accessing it again with `Reflect.apply`. This state is global, i.e., persistent across function invocations. Six differentials are caused by `Math.random`. Another four are caused by setting and later using a key on a global object such as `JSON` or `Math`. The remaining three measure stack depth.

JIT-Picker reports 6,823 differentials during the evaluation runs. JIT-Picker reports findings either as flaky or deterministic, depending on if the same differential appears again when re-executing the same JS input. 626 of the differentials are reported as deterministic. However, we observe that JIT-Picker's mechanism is insufficient to prevent false positives. 6,683 (522 of the ones reported as deterministic) of the differentials are caused by JS `testRunner` related functionality. Another large class 132 (99) are due to measuring performance properties inside the JS input. Another three (zero) are due to the intended non-deterministic behavior of JS `Realms`. The ten true positive differentials found by JIT-Picker are all duplicates of the same deterministic report, and have CR41488094 as a root cause.

DUMPLING reports 16 differentials during the evaluation runs. 12 of them are caused by CR41484971, two by CR332745405, and one by CR335310000. For one differential, we are unsure if it is caused by CR332745405 or CR335310000 or a combination of both.

Given the large number of false positives observed in the related works compared to DUMPLING, we conclude that our technique to reduce false positives produces tangible benefits in practical scenarios.

a) Case Study: CR335310000: Listing 5 shows an interesting V8 issue that was found by DUMPLING. The given proof-of-concept is minimized for readability. In line 10, after a few loop iterations, the object construction of `B` is JIT compiled. It is expected for this object construction to execute the base constructor `A` first and only then define the class field `x`, thus overwriting the property definition in `A`. Since the ECMAScript specification states that public class fields need to contain the enumerable attribute [12], this should lead to an object creation with one enumerable property `x`. Ignition handles this case properly and produces the correct object. However, in the case of JIT compilation, V8 produces

```
function A() {
    Object.defineProperty(this, "x", {
↪   writable: true, configurable: true, value:
↪   undefined });
}

class B extends A {
    x = {};
}

for (let i = 0; i < 100; i++) {
    new B();
}
```

Listing 5: Minimized PoC for the bug CR335310000.

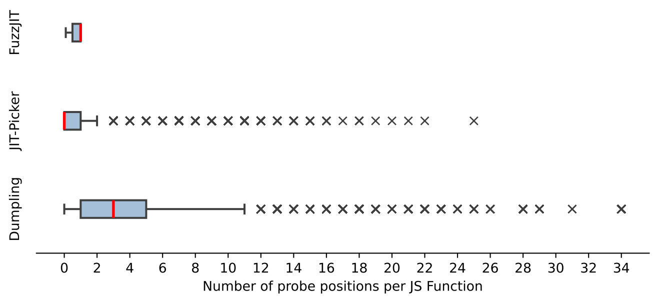


Fig. 5: Number of probe positions per JS Function. Observed during the first 10,000 differential oracle invocations of fuzz cases.

an incorrect object failing to install the enumerable attribute on the property. Our case study shares important similarities with existing exploitable bugs [34]. More specifically, if the missing attribute had been configurable instead of enumerable, our bug would have been exploitable with known exploitation techniques. This showcases that DUMPLING observes the same type of semantic errors common to critical security issues.

DUMPLING detects this differential even though the invalidly constructed object is not assigned to any JS variable and only lives in the full execution state. This bug showcases the advantage of inspecting objects in depth during differential fuzzing. The only way for JIT-Picker or FuzzJIT to detect this bug is to generate additional JS code that makes the differential visible directly in JS, e.g., by assigning the created object to a variable and using `isPropertyEnumerable` with the correct parameters. On the contrary, DUMPLING's design allows for earlier detection without any additional JS code support.

B. Bug Oracle Sensitivity

In the absence of ground truth for buggy behavior in the V8 engine, we conduct three proxy experiments to measure the sensitivity of the differential oracles. First, we evaluate the overlap of bugs that are found by the considered state-of-the-art fuzzers compared to DUMPLING during the evaluation runs. Second, we assume that a differential oracle that adds more probing positions that are more often hit is more sensitive.

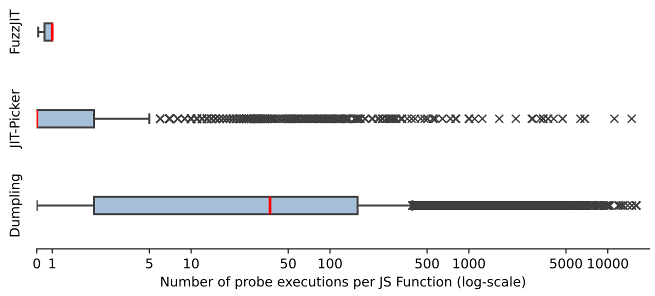


Fig. 6: Number of probe executions per JS Function. Observed during the first 10,000 differential oracle invocations of fuzz cases.

Therefore, we statically and dynamically measure the amount of probing positions in DUMPLING and the state-of-the-art fuzzers. Third, we quantify the amount of information that reaches the bug oracle for each fuzzer, assuming that more detailed observations of the program state contribute to higher sensitivity.

This experiment considers bugs found during the evaluation runs, listed in Table I. DUMPLING finds a superset of bugs, i.e., every bug found by other fuzzers in the evaluation was also found by DUMPLING. One of the differentials is also found by JIT-Picker. FuzzJIT and vanilla Fuzzilli did not uncover new bugs during our evaluation runs. Finding more bugs, while reaching slightly less coverage than Fuzzilli and JIT-Picker, cf. Figure 4, suggests our differential oracle is more sensitive.

From a design perspective, FuzzJIT, JIT-Picker, and DUMPLING probe the JS execution state during a script's execution [17], [18]. However, they differ in the following properties with respect to the bug oracle's input.

a) The frequency at which probes are inserted in JIT compiled code:

- **DUMPLING:** At every dump position: Deoptimization points and returns to unoptimized code.
- **JIT-Picker:** A variable is queried with a probability of 0.01 during normal usage and additionally with a probability of 0.075 in the visible scope.
- **FuzzJIT:** One probe or zero probes per JS program.

b) The subset of the JS execution state that is queried per probe:

- **DUMPLING:** The execution state of the bytecode VM.
- **JIT-Picker:** A single JS variable per probe.
- **FuzzJIT:** A single return value.

c) The detail at which a probed state is compared:

- **DUMPLING:** Fine grained representation of the state as described in Section III-A.
- **JIT-Picker:** Primitive values are compared, but any object is equal to any other object/array.
- **FuzzJIT:** The returned values are compared in detail.

By inspecting the source code of FuzzJIT, we observe an inconsistency between the claims in publication [17] and the current prototype. More specifically, the FuzzJIT prototype does not return *all visible variables* from the current function for comparison in the bug oracle. Instead, it behaves like vanilla Fuzzilli that it returns a single value and sometimes no value at all, limiting the sensitivity of the FuzzJIT bug oracle.

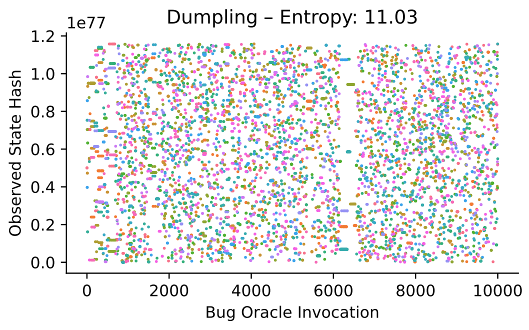
We empirically investigate the frequency of inserted probes. Sensitive differential oracles should extract state from multiple points in the execution. Figure 5 shows the number of probe positions per JS function. DUMPLING has a higher number of probe positions than JIT-Picker and FuzzJIT, with an average and median of three. Figure 6 shows the number of probe executions per JS function. We observe that DUMPLING has a higher number of probe executions than JIT-Picker and FuzzJIT, with a median of 37 dumps and average of 50 dumps per function. This demonstrates a high dumping frequency, indicating a tight-knit probing of the execution state. Furthermore, the experiment provides insight into the efficiency and relatively low overhead of our bug oracle. DUMPLING extracts, transmit and process executions with over 10,000 frame dumps.

As outlined, each of the fuzzers might miss differentials, because the state is only partially queried at varying intervals. Therefore, our third experiment quantifies the amount of information given to the bug oracle, measured as entropy. Higher entropy suggests a higher sensitivity of the bug oracle. We modify FuzzJIT, JIT-Picker, and DUMPLING to hash and save the inputs to the differential oracle by using sha256 as a hash function. For FuzzJIT, we transform their `deepEquals` function to a `deepToString` function, that contains the same information as the original function, but is hashable. JIT-Picker, instead, accumulates an internal hash over one JS execution that is reported back to the fuzzer. We hash it again with sha256, such that values are mapped collision-free to same output space. For DUMPLING, we use the dump file from the first run, as it reflects what will be compared by our matching algorithm. The first 10,000 usages of the FuzzJIT bug oracle have an entropy of 6.08, JIT-Picker reaches 8.42, and DUMPLING 11.03. Figure 7 visualizes this distribution over time. It can be observed from the horizontal lines that FuzzJIT and JIT-Picker keep comparing the same values in the differential oracle, while DUMPLING observes more uniformly distributed values. We are unsure of the exact reason for Dumpling having brief phases of observing a smaller number of states. We deem it is caused by Fuzzilli finding new coverage while mutating a function that is not JIT compiled.

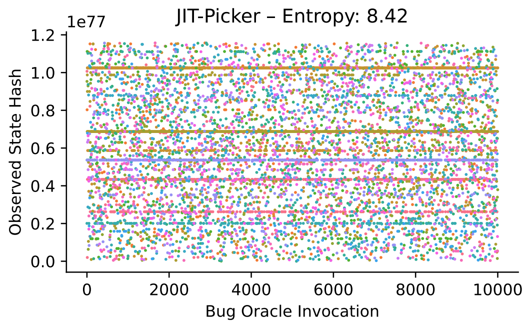
With DUMPLING finding a superset of bugs and extracting more detailed state at a higher frequency, we conclude that DUMPLING has a more sensitive bug oracle than the state-of-the-art.

C. Bug Oracle Overhead

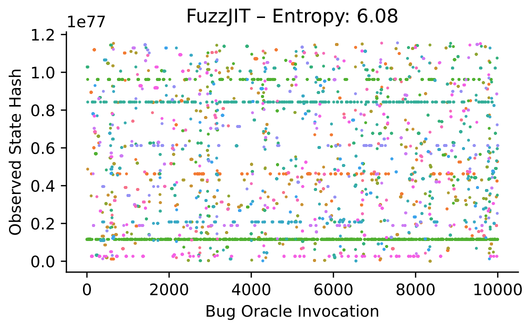
Naturally, a bug oracle that interrupts JIT execution, inspects objects in depth, and serializes them, transmits them to the fuzzer, where they are deserialized and matched, will have a measurable overhead. Therefore, we compare the tested code as measured by edge coverage, as reported by clang. Figure 4 shows the coverage obtained during the ten-fold cross validation of all compared fuzzers with a 95% confidence interval. As expected, Fuzzilli, which does not suffer from the overhead of differential oracles, explores the most code. However, it is closely followed by JIT-Picker and then DUMPLING, in the order of complexity of the bug oracle. After the ten three-day runs DUMPLING has on average 0.75 percentage points less coverage than Fuzzilli and 0.48 percentage points less



(a) States observed for DUMPLING.



(b) States observed for JIT-Picker.



(c) States observed for FuzzJIT.

Fig. 7: The states observed from the first 10,000 differential oracle inputs generated during fuzzing for DUMPLING, JIT-Picker, and FuzzJIT. The states are hashed with sha256 to better compare their values and apply colors in modulo 10 to distinguish neighbouring values.

coverage than JIT-Picker. FuzzJIT reaches significantly less coverage than the competitors, with 2.49 percentage points less than DUMPLING.

We investigate the causes of the coverage reduction in DUMPLING. By construction, DUMPLING does not modify the original Fuzzilli mutators or the V8 code optimizations. Therefore, we hypothesize that the reduction in coverage over time is due to the oracle overhead. To prove our hypothesis, we measure the total execution after 72 hours of fuzzing, where we also count the reference run as an execution. In particular, Fuzzilli reaches 64M executions, JIT-Picker 99M, FuzzJIT 61M, and DUMPLING 52M. JIT-Picker outperforms Fuzzilli because it does not need to generate a new input for the reference execution, and unoptimized executions are faster on average for Fuzzilli-generated inputs, where the threshold for JIT compilation is lowered to benefit testing. Since we observe a reduction in total executions, while Fuzzilli's mutators and the V8 code remains unchanged, we conclude that the drop in coverage is caused by DUMPLING's overhead. It is noteworthy that, in industrial settings, the JS engines undergo month-long campaigns that easily reach a coverage plateau. Therefore, given the improved sensitivity of DUMPLING — measured in new bugs found (Table I) and states observed (Figure 7) — we consider the observed overhead to be a justified trade-off.

In order to reduce the overhead of DUMPLING, we employ a particular performance optimization that only serializes state in a frame that is distinct from the previous frame. This leads to a significant reduction of the sizes of dumps that have to be transferred from V8 to the fuzzer and, as a result, to a significant performance increase. During a three hour five-fold cross validation run with this optimization being enabled and without, we measure a 49% increase in executions per second and a 83% and 77% reduction in dump size respectively for optimized and unoptimized executions.

With DUMPLING being competitive in terms of coverage and being in the same order of magnitude of execution speed compared to the state-of-the-art, we conclude that the overhead of the bug oracle is acceptable.

D. Hyperparameters

DUMPLING has two hyperparameters. The first parameter is the *depth* that is used while traversing an object for serialization. The second parameter is the *number of properties* or elements that are serialized per object. All other experiments are conducted with a depth of 3 and 5 properties, which results from the observation of existing differential bug reports. We evaluate the impact that those hyperparameters have on the effectiveness of DUMPLING by conducting an experiment with combinations of [3, 5, 10] and [5, 10, 50] for the depth and the property count, respectively. For the first 100,000 executions that reach the bug oracle, sensitivity, executions per second and the sizes of dumps in optimized (opt) and unoptimized (unopt) executions is measured. Table II shows the results of this experiment where executions per second and dump sizes are given as mean with standard deviation over five-fold cross validation. Except for insignificant variation that we attribute to statistical noise, the impact of the hyperparameter choice is neutral in regards to bug oracle sensitivity and overhead. By analyzing the JS input generated, we explain these measurements with the fact that Fuzzilli does not generate JS with such deep nested object structures that would make it beneficial to choose a depth larger than three. Similarly, Fuzzilli does not usually generate JS with very large arrays or objects with many properties.

TABLE II: Influence of object traversal depth and number of properties considered during serialization on Entropy (H), execution speed, and size of dumps per program in the first (Opt) and second (Unopt) run.

Depth	**Props**	**H**	**Execs/s**	**Dump Size (B)**	
				Unopt	**Opt**
3	5	11.322	10.3 ± 9.5	36K ± 15K	17K ± 7K
3	10	11.320	10.6 ± 10.0	37K ± 17K	16K ± 6K
3	50	11.322	10.4 ± 10.5	48K ± 20K	22K ± 9K
5	5	11.318	10.9 ± 10.5	32K ± 13K	17K ± 8K
5	10	11.321	11.0 ± 11.0	41K ± 17K	19K ± 10K
5	50	11.321	10.2 ± 10.3	47K ± 23K	20K ± 8K
10	5	11.322	10.8 ± 10.4	34K ± 14K	17K ± 10K
10	10	11.318	12.4 ± 11.8	33K ± 14K	15K ± 6K
10	50	11.320	11.4 ± 10.5	40K ± 13K	18K ± 6K

TABLE III: Time effort for one person to rebase DUMPLING to new versions of V8.

Δ Time (days)	83	21	31	21
Δ V8 LoC	75K	43K	43K	14K
Δ #V8 Commits	1,190	372	521	229
Approx. Rebase Time (min)	30	10	< 1	<1

Depth and property count have only minor impact the sensitivity of the bug oracle (approximated as entropy) and the overhead.

E. *Maintaining* DUMPLING

During the course of our development, we rebased DUMPLING multiple times to the newest available V8 version. The design of DUMPLING specifically aims to decouple the introduced changes from the existing fast moving V8 codebase. Table III shows statistics on the four rebases we performed. Specifically, the table shows the code motion of V8, measured in LoC and V8 commits added, and the estimated time it took to port DUMPLING to the newest version at that time. By having employed a loose coupling of components, we managed to trivially rebase our prototype multiple times even when the code motion of V8 is more massive. Section VI discusses future plans to upstream DUMPLING to the V8 codebase, making it a part of the continuous fuzzing infrastructure.

Consequently, we conclude that the cost to maintain DUMPLING is low.

VI. Discussion

We now discuss the limitations of DUMPLING and future directions for testing JS engines.

Porting DUMPLING *to other JS Engines:* With DUMPLING, we demonstrate the need for an in-depth introspection to detect more subtle bugs in JS engines. DUMPLING follows key design principles that are common to all modern JS engines [1]. Future work may implement DUMPLING for more JS engines, namely *SpiderMonkey* and *JavaScriptCore*. Both engines follow a multi-tiered execution model, with a register-based bytecode VM and speculative JIT compilers [35], [36], [37]. Similar to V8 where execution state is stored in Ignition virtual stack frames, JavaScriptCore features so-called `CallFrames` that are restored during deoptimization [35]. Likewise, in SpiderMonkey, these execution states are called `Activations` [38]. The deoptimization and rematerialization mechanism in V8, SpiderMonkey, and JavaScriptCore are conceptually similar with regards to all aspects relevant to implementing DUMPLING [27], [28], [35]. Adopting DUMPLING for other JIT compilers outside of JS engines is also an interesting direction for future work to explore.

Hardware Architecture: Hooking into the last phase of the optimizing JIT compilers requires writing native code. We implement DUMPLING for the x86-64 architecture. However, most JS engine bug classes are not architecture-dependent, with a few exceptions of instruction selection and hardware register allocation bugs [39], [40]. We deem that opting for implementing DUMPLING for popular architectures is a reasonable trade-off to detect the majority of bugs in JS engines.

Upstreaming DUMPLING *Mode:* While our data on the cost of maintaining DUMPLING is promising, a more sustainable approach would be to upstream DUMPLING Mode to the V8 codebase. We envision DUMPLING Mode as an extension of the existing Fuzzilli Mode. Besides enabling continuous fuzzing with DUMPLING, our contributions specifically in the areas of comparable fine-grained serialization of JS objects, exposing JIT tier usage, and avoiding false positives could be beneficial to existing and future fuzzers.

Using DUMPLING *with other JS Fuzzers:* DUMPLING is designed to be fuzzer agnostic. While we evaluate DUMPLING with Fuzzilli, we find one bug during development by just replaying V8's own test suite in a python script and a stand-alone variant of the matching algorithm. The so-found bug V8:14605 is not an impactful bug, but it demonstrated that DUMPLING found a forgotten edge case in the interpreter without fuzzing techniques. As such DUMPLING can be used as a stand-alone bug oracle that can be successfully used early in the development cycle. Furthermore, DUMPLING can be used in conjunction with any JS fuzzer. In the future, DUMPLING could be integrated with other fuzzers as a general bug oracle to augment their bug finding capabilities.

Incorporating Compiler Speculations: Currently, DUMPLING mainly includes the execution state of the bytecode VM (more precisely, possible representation of which if the JIT compiler would deoptimize). As such we catch bugs before they become visible to existing bug oracles. To catch bugs even before they result in miscalculations in the execution state, we include optimization information that must stay consistent between execution tiers. An example of this are the enum-cache sizes that we include in our dumps (Listing 1) [41]. To further improve the sensitivity of DUMPLING as a JS JIT bug oracle, the DUMPLING instrumentation could be extended to verify compiler speculations. A check may be added that the assumed types and range analysis hold during execution. The key challenge here is to perform a relation between the intermediate representation of the JIT compiler and the execution states observed by DUMPLING. We leave this challenge for future work, as it is not specific to differential fuzzing.

VII. Related Work

A. JavaScript Engine Fuzzing

Fuzzing is a proven technique for finding bugs and vulnerabilities in software. The key insight behind successful fuzzers like AFL and AFL++ is combining coverage-guided mutation on the bytes of the input, with high throughput, to automatically test programs [42], [7]. While applicable to a wide range of programs, JS is a complex and highly structured kind of input. Therefore, byte-level mutations lead to syntactically invalid JS programs in an overwhelming majority of cases. Syntactically incorrect JS code is rejected by the parser, leading to low coverage of the JS engine's code.

An improvement to byte-level mutation is making AFL aware of tokens such as keywords, operators, and literals. This approach has been successful in finding variants of previously reported bugs [43]. A fuzzer can be made more structure-aware by mutating on the abstract syntax tree, often combined with a tree-based grammar of the JS language [44], [45], [46], [47], [48], [15], [49]. Alternatively, existing JS programs can be mutated by designing JS-specific mutations on the source code level [4], [16]. To reduce the complexity of defining JS-specific mutations and to aid program analysis, mutation on a custom intermediate language has proved successful [3]. Other works use deep learning for the generation and mutation of interesting JS [50], [51].

Besides syntactic correctness, the semantic correctness of JS engines improves fuzzing. If syntactically correct JS throws a runtime error early in its execution, the rest of the program is not executed. Therefore, there is research that specifically focuses on semantic correctness, which is improved by tracking types statically [52], [31] or with dynamic analysis [53], [31], [54]. We build on the extensive research in the field of input generation, specifically on Fuzzilli, that itself adopts ideas from the before-mentioned papers during its continuing development. Our focus is not improving input generation, or fuzzer performance optimization, but proposing a new bug oracle. By extension we do not expect to improve code coverage.

B. JavaScript Engine Differential Fuzzing

To find issues in an engine's implementation, differential fuzzing can be done across JS engines [15]. Additionally, COMFORT and JEST parse the ECMAScript specification to improve their grammar and additionally test for specification violations [13], [14].

A promising application of differential fuzzing for JS engines is finding differentials between execution tiers. These differences inside the same engine are more indicative of vulnerabilities, as they can be directly used as unexpected input for other executions tiers [20]. Google [16] and Mozilla [49] employ differential fuzzing with their regular fuzzers, validating that the stdout is the same with and without optimizations enabled. Our approach is geared to probe the state space more thoroughly efficiently. Furthermore, we specifically target JIT code execution.

Diffuzzilli [20] purposefully inserts `JSON.stringify` statements in Fuzzilli generated code to probe a random subset of variables in scope. Unfortunately, their approach is not open source, such that we cannot compare with it in our evaluation. Similarly, JIT-Picker inserts custom `fhash` probes in the JS input [18]. They apply a relatively small patch to the JS engine to accumulate an internal hash over the `fhash` results that they report back to their modified Fuzzilli. DUMPLING differs from both approaches, in that it captures more state at a higher frequency. Our state probing (dumping) is completely transparent in that it does not inhibit JIT optimizations, as it does not require adding sinks to the JS code. Furthermore, DUMPLING has more fine-grained dumping, that is able to detect even subtle differences. For example, DUMPLING can discern variances in float values, such as $-\infty$, which `JSON.stringify` represents as `NaN`, and it can deeply inspect objects, unlike the `fhash` implementation that treats all JS objects the same by just adding 64 to the hash.

FuzzJIT performs this differential testing inside one JS run [17]. Concretely, FuzzJIT wraps generated code in an input template. This input template always follows the same fixed structure, the generated code is wrapped into a predefined function. This function is executed in the interpreter and its return value is saved. Then this function is force-compiled with TurboFan, executed, and the return value is compared with the previously returned value using a specially crafted deep-equals operation inside JS. In contrast, we do not opt for a rigid force-compilation to collect diverse feedback. Furthermore, comparing interpreter and JIT code in one JS execution requires the function executions to be independent of each other. This limits the input space they can explore. Moreover, we observe a high false positive rate from their prototype due to modifying global state. We therefore do two runs, allowing us to provide arbitrary JS as input, and perform more frequent probing in it.

VIII. Conclusion

In this work, we presented DUMPLING, a novel approach for differential fuzzing that inspects the execution state instead of only observing output prints. We evaluated DUMPLING on the V8 JS engine and found ten bugs. Compared with existing state-of-the-art JS fuzzers, our evaluation shows that DUMPLING is more sensitive, exhibits a lower false positive rate, and has comparable performance. While DUMPLING deeply introspects the V8 engine, we decoupled our instrumentation, making it easy to maintain. The results of our research demonstrate the need for specialized bug oracles to detect more subtle bugs. We envision the bug oracle of DUMPLING to be used in conjunction with other fuzzers to augment their bug finding capabilities at an acceptable overhead. To foster further research in this area, we release DUMPLING as open source at https://github.com/two-heart/dumpling-artifact-evaluation. We invite researchers and practitioners to use and evolve DUMPLING to improve the security of JS engines.

Acknowledgment

We thank the anonymous paper and artifact reviewers for their feedback. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, a gift from Intel corporation, by the Helmholtz Association (HGF) within topic "46.23 Engineering Secure Systems", and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff *et al.*, "Trace-based just-in-time type specialization for dynamic languages," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[2] S. Röttger. (2018) Chrome: V8: incorrect type information on Math.expm1. https://crbug.com/project-zero/1710.

[3] S. Groß, "Fuzzil: Coverage guided fuzzing for JavaScript engines," 2018.

[4] R. Freingruber, "Variation analysis of exploitable browser vulnerabilities," 2020.

[5] S. Glazunov. (2023) Security: Out-of-bounds access in ReduceJSLoadPropertyWithEnumeratedKey. https://issues.chromium.org/issues/40068915.

[6] M. Y. Mo. (2023) Getting RCE in chrome with incomplete object initialization in the maglev compiler. GitHub Security Lab Blog. [Online]. Available: https://github.blog/2023-10-17-getting-rce-in-chrome-with-incomplete-object-initialization-in-the-maglev-compiler/

[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. of the USENIX Workshop on Offensive Technologies*, 2020.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. of the USENIX Annual Technical Conference*, 2012.

[9] W. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, 1998.

[10] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: a system for finding application leaks with black box differential testing," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*, 2008.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[12] (2024) Ecmascript 2024 language specification. https://tc39.es/ecma262.

[13] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated conformance testing for JavaScript engines via deep compiler fuzzing," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021.

[14] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "Jest: N+1-version differential testing of both JavaScript engines and specification," in *Proc. of the IEEE/ACM International Conference on Software Engineering*, 2021.

[15] J. Patra and M. Pradel, "Learning to fuzz: application-independent fuzz testing with probabilistic, generative models of input data," TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664, 2016.

[16] O. Chang. (2020) Js-fuzzer. Google Source Repository. [Online]. Available: https://chromium.googlesource.com/v8/v8/+/master/tools/clusterfuzz/js_fuzzer

[17] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "FuzzJIT: oracle-enhanced fuzzing for JavaScript engine jit compiler," in *Proc. of the USENIX Security Symposium*, 2023.

[18] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential fuzzing of JavaScript engines," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[19] S. Groß. (2021) Attacking JavaScript engines: A case study of JavaScriptCore and CVE-2016-4622. Phrack Magazine. [Online]. Available: http://www.phrack.org/issues/70/9.html

[20] J. Jimenez and V. Rao, "Safari, hold still for nan minutes!" in *Objective by the Sea Conference*, 2023.

[21] statscounter GlobalStats. (2023) Browser market share worldwide. https://gs.statcounter.com/browser-market-share/all/worldwide/2023.

[22] J. Gruber, L. Swirski, and T. Verwaest. (2022) Maglev. V8 Design Document. [Online]. Available: https://docs.google.com/document/d/13CwgSL4yawxuYg3iNlM-4ZPCB8RgJya6b8H_E2F-Aek/

[23] O. Flückiger. (2016) Ignition: V8 interpreter. V8 Design Document. [Online]. Available: https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44

[24] L. Swirski and T. Verwaest. (2021) Sparkplug. V8 Design Document. [Online]. Available: https://docs.google.com/document/d/13c-xXmFOMcpUQNqo66XWQt3u46TsBjXrHrh4c045l-A

[25] A. Ertl, "Stack caching for interpreters," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[26] U. Hölzle, C. Chambers, and D. Ungar, "Debugging optimized code with dynamic deoptimization," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[27] J. Franco. (2017) Lazy deoptimization without code patching. V8 Design Document. [Online]. Available: https://docs.google.com/document/d/1ELgd71B6iBaU6UmZ_lvwxf_OrYYnv0e4nuzZpK05-pg

[28] S.-Y. Guo. (2014) Debugging in the time of jits. https://rfrn.org/~shu/2014/05/14/debugging-in-the-time-of-jits.html.

[29] J. Fetiveau. (2020) Modern attacks on the chrome browser: optimizations and deoptimizations. https://doar-e.github.io/blog/2020/11/17/modern-attacks-on-the-chrome-browser-optimizations-and-deoptimizations/.

[30] W. Trybulec, "Pigeon hole principle," *Journal of Formalized Mathematics*, vol. 2, no. 199, 1990.

[31] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities," in *Proc. of the Network and Distributed System Security Symposium*, 2023.

[32] Google. (2024) Chrome vulnerability reward program rules. https://bughunters.google.com/about/rules/5745167867576320/chrome-vulnerability-reward-program-rules.

[33] M. Bynens, M. Hablich, S. Sauleau, and S.-y. Guo. (2024) Implementing and shipping JavaScript/WebAssembly language features. V8 Documentation. [Online]. Available: https://v8.dev/docs/feature-launch-process

[34] G. Google Security Research. (2023) Chrome read-only property overwrite. [Online]. Available: https://packetstormsecurity.com/files/174669/Chrome-Read-Only-Property-Overwrite.html

[35] F. Pizlo. (2020) Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in-javascriptcore.

[36] T. Zagallo. (2019) A new bytecode format for JavaScriptCore. https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore.

[37] D. Minor. (2019) Spidermonkey byte-sized architectures. https://spidermonkey.dev/assets/pdf/SpiderMonkey%20Byte-sized%20Architectures.pdf.

[38] L. Smyth, A. Bargull, G. Squelart, J. Walden, and T. Campbell. (2022) Spidermonkey activations. https://hg.mozilla.org/mozilla-central/file/1fb01ce743b3c8bac67a091af0bac9a121661a43/js/src/vm/Activation.h.

[39] N. Baumstark, B. Keith, and H. Lotfi. (2021) Understanding the root cause of CVE-2021-21220. https://www.zerodayinitiative.com/blog/2021/12/8/understanding-the-root-cause-of-cve-2021-21220-a-chrome-bug-from- pwn2own-2021.

[40] N. Baumstark, S. Groß, and B. Keith. (2021) Writeup for firefox rce. https://bug1493900.bmoattachments.org/attachment.cgi?id=9011702.

[41] C. Bruni. (2017) Fast for-in in v8. V8 Developer Blog.

[42] M. Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[43] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-level fuzzing," in *Proc. of the USENIX Security Symposium*, 2021.

[44] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. of the USENIX Security Symposium*, 2012.

[45] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proc. of the IEEE/ACM International Conference on Software Engineering*, 2019.

[46] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proc. of the European Symposium on Research in Computer Security*, 2016.

[47] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," in *Proc. of the Network and Distributed System Security Symposium*, 2019.

[48] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: data-driven seed generation for fuzzing," in *Proc. of the IEEE Symposium on Security and Privacy*, 2017.

[49] J. Ruderman. (2007) Introducing jsfunfuzz. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz.

[50] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proc. of the IEEE/ACM International Conference on Software Engineering*, 2020.

[51] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided JavaScript engine fuzzer," in *Proc. of the USENIX Security Symposium*, 2020.

[52] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines," in *Proc. of the Network and Distributed System Security Symposium*, 2019.

[53] X. He, X. Xie, Y. Li, J. Sun, F. Li, W. Zou, Y. Liu, L. Yu, J. Zhou, W. Shi *et al.*, "SoFi: reflection-augmented fuzzing for JavaScript engines," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[54] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript engines with aspect-preserving mutation," in *Proc. of the IEEE Symposium on Security and Privacy*, 2020.

Appendix A
Manuscript Appendix

A. Case Study: 345960102

```
1  y = BigInt("0xffffffffffffffff");
2  function b() {
3      let x = BigInt.asIntN(64, -1n);
4      let broken = x >> (y);
5      return BigInt.asIntN(64, broken);
6  }
7
8  b();
9  b();
10 %OptimizeFunctionOnNextCall(b);
11 b();
```

Listing 6: Minimized proof-of-concept triggering CR345960102 found by Dumpling.

Listing 6 shows the minimized PoC of an issue in V8's optimizing compiler TurboFan that was found by Dumpling. This issue manifests as a differential between optimized and unoptimized execution in the state of the function `b` at line 4. More specifically, the bug root cause is the incorrectly implemented the `BigInt` shift operation. TurboFan converts the variable `y` into `0n` instead of `-1n`. This bug may lead to security flaws in client-side cryptographic operations in the browser.

In line 4, once the function is JIT compiled, TurboFan optimizes the right shift between `x` and `y` by converting the right shift to a left shift and negating the shift amount. Additionally, during the conversion, the `BigInt` value `x` is truncated to an `int64_t`. The ECMAScript specification states that negative `BigInt` values have to be treated as a two's complement binary string with bits set infinitely to the left [12]. However, TurboFan's shift optimization does not respect the specification during the conversion to an `int64_t`, thus leading `x` to be treated as a positive number. As a result, TurboFan incorrectly optimizes the shift operation to `0n` instead of `-1n`.

We study if this bug can be detected by FuzzJIT, JIT-Picker, or Fuzzilli. FuzzJIT suffers from a limited observable JS state given by its design (Section V-B). JIT-Picker may detect this violation in principle. However, we consider this scenario unlikely since JIT-Picker's probes may break this optimization. Moreover, JIT-Picker places its probes probabilistically, thus not guaranteeing detection. Fuzzilli, instead, can observe this error only if a runtime assertion fails. After manually assessing the V8 code, we conclude that the closest assertions are difficult to trigger. Specifically, this bug has been in V8 since November 2022,[1] and no one reported or patched it even after extensive fuzzing campaigns from Google. Therefore, we conclude that the detection of this bug via traditional fuzzing techniques is difficult. Conversely, DUMPLING observes this subtle state inconsistency after a relatively short campaign. This case study demonstrates the need for fine-grained probing techniques. Our report of this issue has been awarded $11,000 by Google's Vulnerability Rewards Program [32].

APPENDIX B
ARTIFACT APPENDIX

In this Appendix, we provide the requirements, instructions, and further details necessary to reproduce the experiments from our paper (DOI: 10.14722/ndss.2025.241411).

A. Description and requirements

The artifact contains the material to reproduce the results in Section V.A, V.B, V.C, and V.D. This material is released under the Apache2 License. The project is built on V8 (`6e70d`) and Fuzzilli (`56969`).

1) *Accessing the artifact*: We release the artifact on a public GitHub repository. Additionally, the code is available on Zenodo with the DOI:10.5281/zenodo.14249678.
2) *Hardware dependencies*: The artifact requires an Intel or AMD machine with at least 64GB of RAM and a 100GB disk. We tested on a 16 physical core machine.
3) *Software dependencies*: The artifact was tested on Ubuntu 22.04 and require the ability to run Docker containers and Docker compose. An active internet connection is also necessary for downloading third party packets, e.g., V8 and Fuzzilli. Additionally, `curl`, `git`, `docker`, and `pip` should also be installed. As some experiments run for extending period of time, we recommend running inside a `tmux` or `screen` session.

[1] https://chromium.googlesource.com/v8/v8.git/+/2690e2e3a39f6c1325dae0743682c714cbbc98db

B. Artifact installation

The initial step is to clone the repository and install Docker compose.

```
# Clone the repository
sudo apt install -qq -y python3-pip git
↪  curl screen
git clone \
  https://github.com/two-heart/\
  dumpling-artifact-evaluation.git
cd dumpling-artifact-evaluation

# Install docker compose
if docker compose ; then
  echo "docker compose already installed"
else
  curl -sSL https://get.docker.com/ |
↪  sudo sh
  sudo groupadd docker || true
  sudo usermod -aG docker $USER
  sudo systemctl start docker
  newgrp docker
  docker run hello-world
fi
```

Each experiment is encapsulated in one or multiple Docker container, which are automatically compiled for each reproduced experiment. The `docker-compose.yml` is available at the root of the artifact repository. We do not provide support for running the experiments locally.

C. Experiment workflow

Our artifact aims at reproducing the results from three experiments presented in the paper. The first aims at evaluating the sensitivity of DUMPLING compared to the state-of-the-art. The second experiment evaluates the runtime overhead of DUMPLING bug oracle over a 24 hours fuzzing campaign. We choose to scale down the fuzzing campaign to simplify the experiment, we provide additional information to set up custom campaigns. Lastly, the third experiment evaluate the two hyperparameters configurable in DUMPLING, *depth* and *number of properties*, and their impact on the fuzzer in terms of execution per second and size of program dumps.

We propose to run these experiments sequentially as they are presented in the paper. The artifact provides scripts to run the experiments and collect the results. The scripts will also generate a table and figures similar to the ones presented in the paper.

More complete and detailed instructions, as well as a minimal example, are available in the README file of the repository.

D. Major claims

For all three experiments, the numbers can slightly differ from the ones in the paper due to the different hardware configurations and statistical noise due to a reduced number of repetitions, but we expect the global trend across the benchmark to remain consistent.

- (C1) *Sensitivity*: DUMPLING has more sensitivity compared to the state-of-the-art. In particular, DUMPLING exhibits a more uniform distribution of states across the fuzzing campaign, as highlighted by Figure 7 in the paper. These results and figures can be reproduced through our experiment E1.
- (C2) *Bug oracle overhead*: DUMPLING's bug oracle incurs a reasonable performance overhead. Our experiment E2 highlights this trend after a *three* hours run repeated 10 times. The results might vary from the one presented in the paper. In the paper, the overhead numbers, in terms of total number of executions and reduced coverage, are available in Section V.C. Additionally, this experiment reproduces Figure 4 showing the coverage trends.
- (C3) *Hyperparameters*: To better assess the effect of the two hyperparameters on DUMPLING, we conduct an exploration of the parameter space. The results are presented in Table II and can be reproduced through the experiment E3.

Additionally, the repository contains the bugs found through our campaign in `./poc` folder as evidence for the experiment described in Section V.A.

E. Evaluation

In this section, we provide the detailed steps to run the experiments and process the results to get the table and figures presented in the paper. Overall, this process requires around two to three days of computation time on a powerful 16-core server. These instructions are also available in the README file of the artifact repository.

Experiment 1 (E1) - Claim (C1): Sensitivity:
[2 humans minutes + 2 compute-hours] The experiment evaluates the sensitivity of DUMPLING. The experiment consists running DUMPLING and measuring the number of states observed in the 10,000 first differential oracle invocations. Additionally, the experiment plots the distribution of the hashes of the observed states.

[Preparation] No specific preparation is required.

[Execution] Run the commands:

```
cd dumpling-artifact-evaluation
docker compose up -d
  ↪  dumpling-eval-sensitivity
ls ./sensitivity/data
```

[Results] Upon completion, the script will generate three figures similar to Figure 7 in the paper. The file `./sensitivity/data/sensitivity.pdf` contains the results.

Experiment 2 (E2) - Claim (C2): Bug oracle overhead: [2 humans minutes + 24 compute-hours] The experiment evaluates the overhead of DUMPLING and the reached coverage. The experiment consists running DUMPLING and measuring the reached coverage and and the number of execution performed.

[Preparation] No specific preparation is required. In case of limited resources or to reproduce more closely the results from the paper, the variables `EXP_RUN_HOURS` and `EXP_NUM_RUNS` can be modified in the `docker-compose.yml` file to reduce the duration of a run and the number of replication, respectively. Additionally, `NUM_JOBS` can limit the number of logical cores allocated.

[Execution] Run the commands:

```
cd dumpling-artifact-evaluation
docker compose up -d
  ↪  dumpling-eval-bug-finding-and-overhead
ls ./bug_finding_and_overhead/data/
```

[Results] Upon completion, the script will print numbers similar to the one available in Section V.C of the paper and generate a figure similar to Figure 4. The file `coverage.pdf` contains the image.

Experiment 3 (E3) - Claim (C3): Hyperparameters: [2 humans minutes + 6 compute-hours] This experiment explores the hyperparameters space of DUMPLING.

[Preparation] No specific preparation is required.

[Execution] In a shell, run the following command:

```
cd dumpling-artifact-evaluation
docker compose up -d
  ↪  dumpling-eval-hyperparameters
ls ./hyperparameters/data
```

[Results] Upon completion, the script will generate a table similar to Table II, which are stored in `./data/hyperparam_eval.tex`.