Automatically Localizing Dynamic Code Generation Bugs in JIT Compiler Back-End

HeuiChan Lim
Department of Computer Science
The University Of Arizona
Tucson, AZ 85721, USA
hlim1@arizona.edu

Saumya Debray
Department of Computer Science
The University Of Arizona
Tucson, AZ 85721, USA
debray@cs.arizona.edu

Abstract

Just-in-Time (JIT) compilers are ubiquitous in modern computing systems and are used in a wide variety of software. Dynamic code generation bugs, where the JIT compiler silently emits incorrect code, can result in exploitable vulnerabilities. They, therefore, pose serious security concerns and make quick mitigation essential. However, due to the size and complexity of JIT compilers, quickly locating and fixing bugs is often challenging. In addition, the unique characteristics of JIT compilers make existing bug localization approaches inapplicable. Therefore, this paper proposes a new approach to automatic bug localization, explicitly targeting the JIT compiler back-end. The approach is based on explicitly modeling architecture-independent back-end representation and architecture-specific code-generation. Experiments using a prototype implementation on a widely used JIT compiler (Turbofan) indicate that it can successfully localize dynamic code generation bugs in the back-end with high accuracy.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Software testing and debugging**.

Keywords: JIT Compiler, Back-End, Dynamic Program Analysis, Dynamic Code Generation, Automatic Bug Localization

ACM Reference Format:
HeuiChan Lim and Saumya Debray. 2023. Automatically Localizing Dynamic Code Generation Bugs in JIT Compiler Back-End. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3578360.3580260

1 Introduction

JIT compilers, used to improve the performance of interpreted systems, are used in a wide variety of commonly used software. Bugs in JIT compilers can have a widespread impact; in particular, *dynamic code generation bugs*, where the JIT compiler silently emits incorrect code that results in incorrect execution of the application being optimized, can give rise to exploitable vulnerabilities (e.g., see the numerous JIT compiler exploits described by Google's Project Zero team [1]). The security implications of dynamic code generation bugs make it essential to identify and fix them quickly. Unfortunately, the size and complexity of modern JIT compilers can make manual debugging and fixing such bugs tedious and time-consuming. Therefore, providing automated tool support for reasoning about such bugs is vital.

This paper considers the following bug localization problem: given a single "proof of concept" (PoC) input[1] for triggering a dynamic code generation bug in a JIT compiler back-end, use automated techniques to identify a ranked list of back-end representation objects affected by the bug (i.e., witnesses) and a ranked list of potentially buggy functions (i.e., culprits).

The JIT compilation process can be thought of as a pipeline. The JIT compiler constructs an intermediate representation (IR) of the input program's bytecode, which it receives from the interpreter; performs various optimizations on the IR; then passes the optimized IR to a back-end that maps the optimized IR to native code. As a result, dynamic code generation bugs can arise in the IR optimization process (*optimization bugs*) or when converting the IR to native code (*back-end bugs*). In addition, the data structures and algorithms used for the IR are usually very different from those used for the back-end. For example, the Turbofan JIT compiler in Google's V8 JavaScript engine uses a sea-of-nodes structure for its IR graph [12], while the back-end uses a control-flow graph (CFG) where each block of CFG holds a sequence of back-end objects. Moreover, an IR is the primary data structure used throughout the optimization, i.e., the result of the optimization is a manipulated IR. At the same time, the back-end also affects the code buffer, which is not a part of the representation object. Accordingly, optimization bugs are quite different from back-end bugs. This paper focuses on the latter class of bugs; to the best of our knowledge, it is

CC '23, February 25–26, 2023, Montréal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0088-0/23/02.
https://doi.org/10.1145/3578360.3580260

[1]Vendors require bug reports to specify how the bug can be reproduced (e.g., see [3, 7]). For JIT compiler bugs, this translates to providing a PoC input that triggers the buggy behavior.

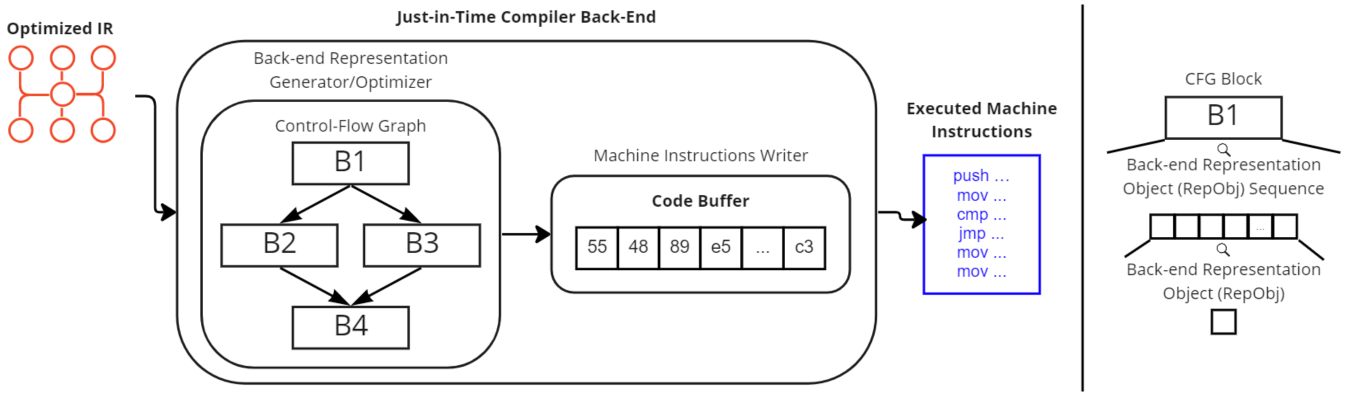

Optimized IR
Just-in-Time Compiler Back-End
Back-end Representation Generator/Optimizer
Control-Flow Graph
B1
B2
B3
B4
Machine Instructions Writer
Code Buffer
55 48 89 e5 ... c3
Executed Machine Instructions
push ...
mov ...
cmp ...
jmp ...
mov ...
mov ...
CFG Block
B1
Back-end Representation Object (RepObj) Sequence
Back-end Representation Object (RepObj)

Figure 1. From Input optimized IR to back-end representation to output code execution (left). Each CFG block holds a sequence of back-end representation objects, i.e., `RepObjs` (right).

the first work on the automatic localization of dynamic code generation bugs in JIT compiler back-ends.

Unfortunately, existing work on automatic bug localization does not carry over to the bugs we consider. Earlier work by Lim and Debray [18] focuses on optimization bugs; Lim *et al.* [19] mentions modeling both optimization and back-end bugs but do not discuss any algorithms or implementation for the latter. Neither of these works performs bug localization for back-end bugs. Proposals for bug localization for ordinary compilers [4, 5, 27, 29] do not carry over to JIT compilers due to a number of differences between the two kinds of compilers, including, e.g., tight coupling with an interpreter; history-sensitive optimization; fixed-size dynamic code buffers, and the potential for dynamic deoptimization. For example, in a single execution of a program, a function can start out as interpreted byte-code, get JIT-compiled with one set of optimizations, then later get deoptimized to interpreted code, then subsequently get JIT-compiled again, possibly with a different set of optimizations that result in different optimized code than before. Finally, other approaches to automated bug localization [2, 6, 14, 17, 26] do not work because the code that exhibits incorrect execution behavior (the application being optimized) is not the code that contains the bug (the JIT compiler).

This paper proposes a novel approach for automatic localization of dynamic code generation bugs in JIT compiler back-ends. It is based on modeling the JIT compiler's representations of the code being JIT-compiled. We use dynamic analysis to collect low-level execution traces of the JIT compiler's executions. We then construct a model for each such execution, and compare the models for buggy executions of the JIT compiler with models for non-buggy executions to identify similarities and differences between buggy and non-buggy executions of the JIT compiler. We use this to identify specific problematic components of the input program's code representations and use information about the code that manipulated those components to identify potential buggy functions in the JIT compiler's source code.

We evaluated the effectiveness of our approach using a prototype tool that we applied to 40 back-end dynamic code generation bugs (5 real-world; 35 synthetic) in Google V8's JIT compiler, Turbofan. The results demonstrate that our approach can provide high-accuracy source-level localization of such bugs.

This paper makes the following contributions.

1. To the best of our knowledge, this paper is the first to propose an approach to modeling JIT compiler back-end representations.
2. It describes an algorithm to localize the dynamic code generation bugs in the back-end by analyzing the modeled back-end representation.
3. It demonstrates the efficacy of our ideas experimentally using 40 back-end bugs (5 real-world; 35 synthetic) in a widely-used real-world JavaScript JIT compiler.

2 Research: From PoC-to-Localization

This section presents an automated approach for localizing dynamic code generation bugs in JIT compiler back-ends.

Figure 1 shows a conceptual flow of the JIT compiler back-end. The back-end receives an optimized IR from the optimizer constructing/optimizing the back-end representation in a control-flow graph (CFG) structure and emits machine instructions. The CFG is constructed based on the scheduled optimized IR nodes. Each block of CFG holds a sequence of back-end representation objects. In our approach, we consider these sequences as a single sequence. The JIT compiler back-end (e.g., Instruction Selector and Register Allocator in V8) generates and manipulates `RepObjs` to be closer to the architecture (e.g., x64) whose instructions it generates. In the final step, the code generator generates architecture-specific machine code based on the representation object and writes this code to the code buffer, where it is executed.

Different JIT compilers use different data structures for their back-ends, e.g., Google V8's `Instruction` objects [10] are different from JavaScriptCore's `B3 IR` [11] and SpiderMonkey's `LIR` [9] objects. To maintain generality and avoid tethering our ideas and algorithms too closely to the implementation specifics of any one JIT compiler, we use the generic term '`RepObj`' to denote back-end representation objects generated from the IR and converted to machine instructions (Figure 2).

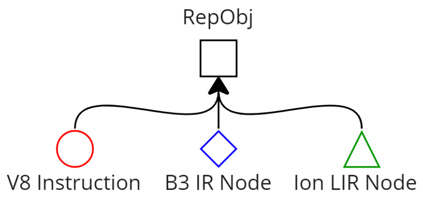


Figure 2. Back-end Representation Object (`RepObj`)

Our approach aims to identify buggy `RepObjs` containing incorrect values and analyze them to locate the culprit functions in the JIT compiler's source code that created those incorrect values. The buggy `RepObjs` and functions identified are ranked in the order of most likely related to the bug to the least. To achieve this goal, we compare the JIT compiler's execution on inputs that trigger the bug with those that do not determine what the buggy executions have in common with non-buggy executions that do not. From each such execution, we construct an abstract model that captures essential characteristics of the JIT compiler's back-end behavior. These abstract models are then compared to extract commonalities and differences.

2.1 Buggy vs. Non-Buggy Executions

To determine whether a JIT compiler's execution on a given input program p is buggy or non-buggy, we run p twice using the interpreter/JIT compiler system: once without JIT compilation and another with JIT compilation. The JIT compiler's execution is considered to be buggy if the result of executing p with JIT compilation enabled is different from the output with JIT compilation disabled.

2.2 Overview of the Analysis Pipeline

Figure 3 demonstrates the architecture of our prototype tool. Our tool has five main parts, from input to output, each responsible for a specific task. The input to the tool is a proof-of-concept (PoC) program (e.g., JavaScript code) that triggers a bug in the JIT compiler back-end.

1. *Input programs generation* (Sec. 2.3). Given a single PoC program, we use directed fuzzing to generate a set of mutated variants of the original program. Some of these variants will continue to trigger the JIT compiler bug while others do not.
2. *Modeling* (Sec. 2.4). Each input program, i.e., the original PoC and its variants, is executed with the Interpreter/JIT compiler system. For each input, we collect an instruction-level execution trace and analyze the executed instructions of the JIT compiler back-end to construct an abstract model of the JIT compiler's manipulation of its back-end representation.
3. *Witness identification and ranking* (Sec. 2.5). Witnesses, i.e., model components suspected to be affected by the JIT compiler bug(s), are identified and ranked.
4. *Culprit instruction selection and ranking.* (Sec. 2.7 and Sec. 2.8). We analyze each ranked witness's access log to identify the executed instructions suspected to be the culprit of the bug.
5. *Rankings analysis and final result generation* (Sec. 2.9). We analyze the two rankings, i.e., witness and instruction rankings, and construct a final result.

The output from our tool is a text file containing (1) a ranked list of buggy witnesses, i.e., abstract `RepObjs` corresponding to the concrete buggy `RepObjs`, information; and (2) a ranked list of suspicious JIT compiler source functions.

2.3 Input Programs Generation

Existing approaches to input mutation for generating input variants for automated bug localization [4, 16, 23] fail to consider the unique characteristics of JIT compilers. For example, while ordinary compilers translate the entire input program to native code, only parts of the input program (the "hot" code) get JIT compiled. This means that mutations made to the original input program using existing approaches might not be JIT compiled, resulting in weak or nonexistent relationships between the mutations and the bug under investigation. The approach introduced by Lim and Debray [18] generates mutated variant programs targeting JIT compilers but making random changes to the abstract syntax tree (AST) of the original program without considering how the change might influence the JIT compiler's buggy behavior.

Given the original program p_0 and a user-provided value n that specifies the number of new programs to generate, our directed fuzzer generates a set of new programs via three different phases: (1) random mutation; (2) target identification; and (3) controlled mutation.

2.3.1 Phase 1: Random Mutations.

The directed fuzzer first creates n randomly mutated variants of p_0 by making n copies of p_0's AST and making a random change to each copied AST. The change must follow two rules: (1) the change must not violate the syntax rule of the input program's language; (2) the change should be semantically similar to the original. To satisfy these two rules, the fuzzer limits its mutations to existing AST nodes, i.e., the fuzzer does not add or remove nodes but only changes the values.

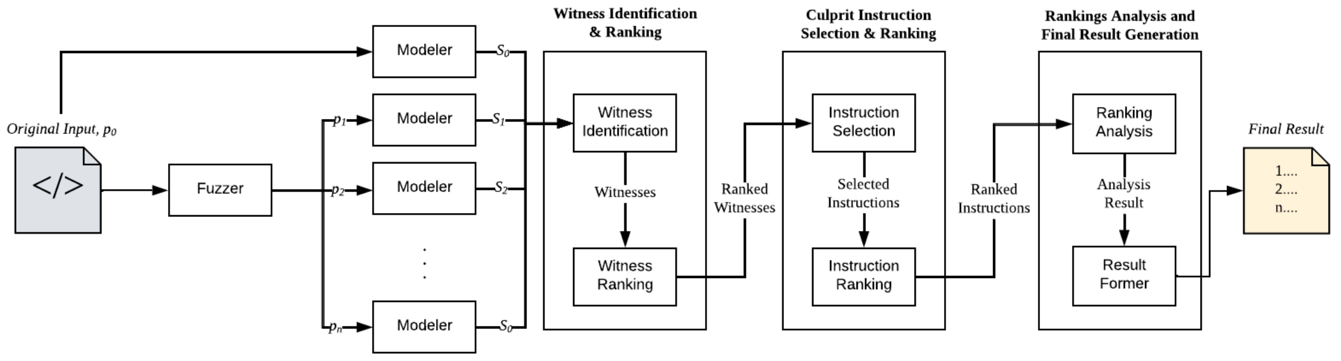


Figure 3. The architecture of our analysis tool

2.3.2 Phase 2: Target Identification. This phase aims to identify the AST nodes related to the bug, i.e., mutating specific AST nodes eliminates the buggy behavior. Such identification lets us know which specific part(s) in the input program is related to the JIT compilation and, more importantly, triggers a bug in the JIT compiler. The output of this phase is a set of node ids of the original AST.

2.3.3 Phase 3: Controlled Mutation. Given a set of node ids of the original AST to target from the second phase, this phase aims to generate a set of n new programs targeted to mutate or avoid specific parts of the original program, p_0. To obtain a program that does not trigger a bug in the JIT compiler, the fuzzer makes a change specifically targeting the nodes of AST with ids in the target id set. On the other hand, we generate the bug-triggering programs by mutating the AST, avoiding the nodes with ids in the target node id set. The output of this phase is a set of new input programs with the original input p_0: $\mathbf{P} = \{p_0, \ldots, p_n\}$.

2.4 Back-end Representation Modeling

As discussed, the JIT compiler back-end receives an architecture-independent graph-structured optimized IR from the optimizer, generates a back-end representation, manipulates them (e.g., register allocation and optimization), and emits architecture- specific machine instructions.

For each p_i in the set of input programs $\mathbf{P}$, we obtain an instruction-level trace of the JIT compiler's execution on input p_i using Intel's Pin dynamic analysis tool [20]. Let $S_i = \langle s_1, \ldots, s_k \rangle$ denotes a *concrete* back-end representation, i.e., a sequence of *concrete* `RepObj`, s_j, $1 \leq j \leq k$, constructed for p_i. For each such s_j we use the execution trace to extract information about the JIT compiler's manipulation of *concrete* `RepObj` to construct an *abstract* `RepObj` $\widehat{s_j}$; we use `aRepObj` to denote an abstract `RepObj`. We construct an *abstract* model $\widehat{S_i} = \langle \widehat{s_1}, \ldots, \widehat{s_k} \rangle$, i.e., a sequence of `aRepObjs`. In constructing the abstract model, we assume that we know the names of the functions that allocate `RepObjs`. These function names can be obtained from the source code of the JIT compiler (e.g., `Emit` function in V8) or provided by the user; the identification of these allocators is orthogonal to the topic of this paper, so we do not pursue it further in this discussion.

In addition to constructing the abstract model $\widehat{S_i}$, we construct an *abstract code buffer*. The final step of JIT compilation is writing architecture-specific machine instructions to a code buffer. We model this code buffer to keep track of values written into it. We analyze the JIT compiler's execution trace to identify where the code buffer allocator function is called and thereby determine the start address and size of the code buffer. This allows us to identify subsequent memory operations in the execution trace that access the code buffer. The abstract code buffer is a mapping (initially empty) from offsets in the code buffer to values; when the JIT compiler writes (the binary encoding of) a machine instruction into the concrete code buffer, we update the abstract code buffer accordingly, as follows. We analyze the JIT compiler's execution trace for memory writes into the address range of the code buffer. If a value w is written into an offset m in the code buffer, we update the abstract code buffer to indicate that the value at offset m is w.

The `aRepObjs` have the following components.

Representation Object Identifier: An identifier is a number uniquely assigned to each `aRepObj` at the generation time. This identifier is used in identifying the order of generation of the `aRepObj`'s corresponding concrete `RepObj`.

Representation Object Address: While analyzing the execution trace, if we encounter a function call to `RepObj` allocator function, we analyze the executed instructions of the callee to extract the value that the function returns to the caller. We can obtain the location of the return value from the system's application-binary interface (ABI). E.g., in x86-64 systems, pointers are returned in the `rax` register.

Representation Object Size: Given the `RepObj` address, i.e., the start address of the allocated memory block, we use the size (in bytes) of `RepObj` to determine the end address of the memory block. Then, we use the start and end addresses to identify the memory read/write accesses to `RepObj`.

Representation Object Opcode: The opcode of RepObj represents the machine operation the object will be translated to. It is assigned based on the target architecture (e.g., a RepObj for 32-bit cmp operation of x86-64 architecture is assigned with an opcode stands for X64Cmp32 mnemonic in Turbofan) but may not be an actual machine-level opcode for that architecture. The code generator determines an architecture-specific opcode based on the opcode and operands of RepObj. The opcode of RepObj can be obtained by analyzing the executed instructions of the allocator function that memory writes to the start address of the allocated object, i.e., the address of the representation object address.

IR Node Opcodes: The back-end uses IR node opcodes to determine the opcode for new RepObjs. We extract the IR node's opcodes and maintain them in our aRepObj.

Assume that we have an optimized IR graph that is an input to the back-end. To identify the opcodes of IR nodes used in generating RepObj, we analyze the executed instructions of the caller function of RepObj allocator. Before the caller function calls the allocator function, it analyzes IR nodes from the IR graph. Our approach is to identify these nodes being analyzed by the JIT compiler to generate specific RepObj.

This differs from the existing approaches, including the model structure that Lim and Debray suggested [18]. The structure of their abstract IR node holds the opcode of the node alone, i.e., they do not map the node to its precedence bytecode. Therefore, a mapping between the IR nodes and bytecodes may increase the accuracy in identifying the IR nodes related to the bug during witness identification (Sec. 2.5.3).

Operands: The operands are synonyms with the definition of instruction operands, i.e., entities operated upon by the instruction [22]. In V8, the register allocator accesses the operands to allocate registers. Despite the fact that the JIT compilers make distinctions among the different kinds of operands, we do not make such distinctions in our implementation. This is because (1) it is non-trivial to distinguish them from the instruction-level execution trace; (2) the number of operands for each RepObj is not fixed, i.e., the numbers are determined at run-time. Thus, we keep a record of operands as values written between the start and end addresses of the concrete RepObj in the corresponding aRepObj.

Offsets of Code Buffer: The code generator writes the constructed machine instructions to the code buffer by analyzing RepObjs, and more than one machine instruction can be constructed from a single object. We maintain the offset of the code buffer in the aRepObj to create a mapping between the aRepObjs and the emitted machine instructions. Figure 4 demonstrates a conceptual level of this mapping. Each s_j is represented with the RepObj opcode, and the binary values in the code buffer are the binary encoding of the x86 instruction opcodes and operands. To create this mapping, we keep a record of RepObj that was last accessed before the machine instruction is written to the code buffer. Suppose we encounter an executed instruction writing machine instruction to a code buffer. In that case, we add the offset of the code buffer to the aRepObj corresponding to the last accessed concrete RepObj.

Maintaining a mapping between aRepObjs and the abstract code buffer makes it possible to reason about a greater variety of bugs that were not possible in earlier work, e.g., bugs resulting from improper manipulation of one data structure (e.g., the code buffer) as a result of incorrect values in a different data structure (e.g., RepObj). Earlier work by Lim and Debray [18] models the JIT compiler's IR manipulation but not any auxiliary data structures, and thus cannot reason about bugs that arise due to misalignment between different data structures in the JIT compiler.

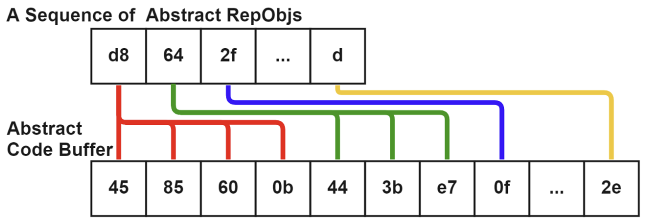


Figure 4. aRepObjs to abstract machine instructions

Access Log: The JIT compiler back-end accesses its RepObjs to read values (e.g., opcodes, operands) or manipulate them (e.g., for register allocation). Our aRepObjs maintain the access log of information about such accesses to the concrete RepObj. While analyzing the JIT compiler's execution trace, when we find an instruction I that accesses a RepObj, we record the following information about I in theaRepObj's access log: (1) the instruction id of I; (2) the source-code function I belongs to; (3) I's opcode and operands; (4) the memory location accessed, i.e., offset from the address of accessed RepObj; (5) value stored (if access overwrites some current value, we maintain the pre-overwrite and post-overwrite values); and (6) access type. The access types referred to are: (a) create a RepObj; (b) write a new value to a RepObj; (c) update an existing value in a RepObj; and (d) write machine instructions to the code buffer.

2.5 Witness Identification

The JIT compiler bug under investigation manifests itself as one or more incorrectly constructed RepObjs that eventually result in the generation of incorrect optimized code; we refer to the corresponding aRepObjs as *witnesses.* Given a set of abstract models, $\widehat{\mathbf{S}} = \{\widehat{S}_1, \ldots, \widehat{S}_n\}$, we take the following steps to identify witnesses:

1. Partition $\widehat{\mathbf{S}}$ into two sets: $\widehat{\mathbf{S}}_B$ and $\widehat{\mathbf{S}}_{NB}$, denoting the abstract models for the *buggy* and *non-buggy* JIT compiler executions respectively.
2. Find the set of aRepObjs that the models in $\widehat{\mathbf{S}}_B$ have in common; and similarly for $\widehat{\mathbf{S}}_{NB}$.
3. Find the differences between the two sets in the earlier step.

2.5.1 Model Partitioning. We use the approach discussed in Section 2.1 to determine whether the execution of the JIT compiler is buggy or non- buggy. If the execution is buggy, we add the model $\widehat{S_i}$ constructed for that execution to $\widehat{\mathbf{S}}_B$; otherwise we add $\widehat{S_i}$ to $\widehat{\mathbf{S}}_{NB}$.

2.5.2 Finding Commonalities between Models. To find the aRepObjs that a set of models have in common, we first designate a *base model* for each of $\widehat{\mathbf{S}}_B$ and $\widehat{\mathbf{S}}_{NB}$, denoted by $base(\widehat{\mathbf{S}}_B)$ and $base(\widehat{\mathbf{S}}_{NB})$ respectively: $base(\widehat{\mathbf{S}}_B)$ is the model obtained from the original PoC p_0, while $base(\widehat{\mathbf{S}}_{NB})$ is the non-buggy model that is "most similar" to $base(\widehat{\mathbf{S}}_B)$. We determine the similarity of models using sequence alignment between the opcode sequences in the aRepObjs in the models. The idea is that similar models have similar opcode sequences between their RepObjs, and therefore, the better the alignment between the opcodes of the two aRepObj sequences in the models, the greater their similarity. Our current implementation uses the Needleman-Wunsch sequence alignment algorithm for this [21]. The output of this phase is the two different sets of aRepObj positions, i.e., a set of aRepObj positions common to the buggy group and a set of aRepObj positions common to the non-buggy group.

2.5.3 Finding Differences between Sets of Models. Our next task is to determine how the commonalities between all the buggy models differ from the commonalities of all the non-buggy models. This is given by the symmetric difference between the aRepObjs that appear in buggy commons and non-buggy commons and captures the witnesses culpable for the buggy behavior of the JIT compiler. Algorithm 1 demonstrates the steps of our approach to finding the witnesses. Given the two base models and two sets of common aRepObj positions, the algorithm to identify witnesses proceeds as follows:

1. Find the alignment of positions from the buggy base to the non-buggy base (line 18).
2. Select witnesses from $base(\widehat{\mathbf{S}}_B)$ (lines 19-20).
3. Find the alignment of positions from the non-buggy base to the buggy base (line 21).
4. Select witnesses from $base(\widehat{\mathbf{S}}_{NB})$ (lines 22-23).
5. Merge two sets found in steps 2 and 4 (line 24).
6. Return the merged set, *WITNESSES* (line 25).

Function *buggies* (lines 1 - 9) demonstrates the steps of selecting witnesses from the buggy base. We prepare an empty *Witnesses* set (line 2). For every position i in the buggy base aligned to the non-buggy base, check if the position i is a common position in the buggy models (lines 3 - 4). If *true*, retrieve the actual aRepObj from the buggy and non-buggy bases (lines 5 - 6). One thing to mention is that the algorithm simplified the process of retrieving the aRepObjs from the non-buggy base by using the same position i as a buggy base. In the actual implementation, we compute the position of non-buggy aRepObj aligned with the buggy aRepObj at position i separately. Given the buggy and non-buggy aRepObjs, we compare the properties of the two to verify that they are the same objects. The properties of the aRepObjs we compare are (1) opcodes, (2) sizes, (3) IR node opcodes, (4) operands, and (5) mapped machine instructions. Suppose any one of the comparisons fails, i.e., $\widehat{s} \neq \widehat{t}$. In that case, we determine they are different and add the aRepObj from buggy base to *Witnesses* set (lines 7-8). Then, return *Witnesses* (line 9).

Function *nonbuggies* (lines 10-16) demonstrates the steps of selecting witnesses from the non-buggy base. First, we prepare an empty *Witnesses* set (line 11). Then, for every position i that is common in the non-buggy models, check if the aRepObj at the position i aligns with any of the aRepObjs in the buggy base (lines 12-13). If there is no aligned aRepObj in the buggy base, extract the actual aRepObj from the non-buggy base (line 14). Contrary to the approach of selecting the witnesses from the buggy base, we do not compare the properties of the non-buggy aRepObj to the buggy. This is because there is no aligned aRepObj in the buggy base that we can compare to, so we add the extracted non-buggy aRepObj to *Witnesses* (line 15). Then, we return *Witnesses* (line 16).

2.6 Ranking the Witnesses

Given the set of selected witness back-end representation objects, *WITNESSES*, we rank the aRepObjs through two phases: (1) sort by occurrence; (2) sort by order of generation. Before we sort the aRepObjs in *WITNESSES*, we filter out nop aRepObj, e.g., X64ArchNop RepObj in V8. This is based on our experiment that the corresponding concrete RepObjs are not translated to machine instructions. Figure 5 shows an example of witness ranking steps. The numbers in each cell represent the model identifiers.

2.6.1 Sort by Occurrence. This phase groups the selected witnesses by their occurrence. We identify and group the aRepObjs that occur either on the buggy or the non-buggy side, then group the rest, i.e., aRepObjs that occur on both sides separately. Then, we prioritize aRepObjs that only occur in the buggy sequences. This is based on the observation that aRepObjs that occur only on one side are more likely to be directly related to the JIT compiler bug: an aRepObj that occurs in the buggy executions but not in the non-buggy ones are indicative of the buggy executions doing something that the non-buggy ones do not;. In contrast, an aRepObj that occurs in the non-buggy executions but not in the buggy

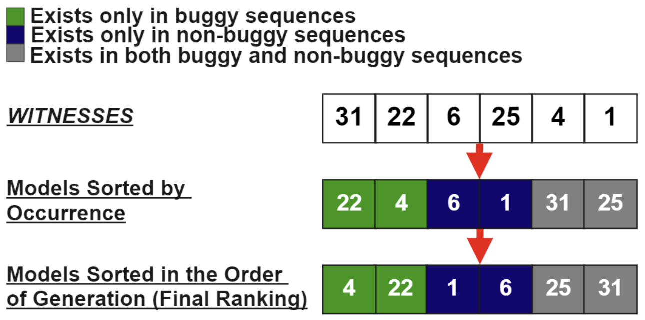
Algorithm 1: Selecting the Witness Models

```
Input: $CMM(\widehat{\mathbf{S}}_B)$: Common buggy object positions.
Input: $CMM(\widehat{\mathbf{S}}_{NB})$: Common non-buggy object
        positions.
Input: $base(\widehat{\mathbf{S}}_B)$: Buggy base.
Input: $base(\widehat{\mathbf{S}}_{NB})$: Non-buggy base.
Result: $WITNESSES$: Identified Witnesses.
function $buggies(Comm, AlignedPos, Base1, Base2)$:
    $Witnesses \leftarrow \emptyset$;
    for $i \in AlignedPos$ do
        if $i \notin Comm$ then
            $\widehat{s} \leftarrow Base1_i$
            $\widehat{t} \leftarrow Base2_i$
            if $\widehat{s} \neq \widehat{t}$ then
                $Witnesses \longleftarrow Witnesses \cup \{\widehat{s}\}$
    return $Witnesses$;
function $nonbuggies(Comm, AlignedPos, Base)$:
    $Witnesses \leftarrow \emptyset$;
    for $i \in Comm$ do
        if $i \notin AlignedPos$ then
            $\widehat{s} \leftarrow Base_i$
            $Witnesses \longleftarrow Witnesses \cup \{\widehat{s}\}$
    return $Witnesses$;
begin
    $BugToNonBug \leftarrow align\_pos_{base(\widehat{\mathbf{S}}_B)}(base(\widehat{\mathbf{S}}_{NB}))$;
    $FROMBUG \leftarrow buggies(CMM(\widehat{\mathbf{S}}_B),$
        $BugToNonBug, base(\widehat{\mathbf{S}}_B), base(\widehat{\mathbf{S}}_{NB}))$;
    $NonBugToBug \leftarrow align\_pos_{base(\widehat{\mathbf{S}}_{NB})}(base(\widehat{\mathbf{S}}_B))$;
    $FROMNONBUG \leftarrow nonbuggies(CMM(\widehat{\mathbf{S}}_{NB}),$
        $NonBugToBug, base(\widehat{\mathbf{S}}_{NB}))$;
    $WITNESSES \leftarrow FROMBUG \cup FROMNONBUG$;
    return $WITNESSES$;
```

ones is indicative of the non-buggy executions doing something that the buggy executions fail to do.

2.6.2 Sort by Order of Generation. For each group, we sort the witnesses by the order of generation. As discussed in Section 2.4, the identifiers of `aRepObjs` represent the order of generation. Thus, we sort the witnesses by identifiers in ascending order.

2.6.3 Witness Ranking: An Example. An example of witness sorting by occurrence and order of generation is shown in Figure 5. The second sequence shows an example of how *WITNESSES* is sorted by occurrence. The cells in green (cells 1 and 2) are `aRepObjs` that occur only in the buggy models, while the cells in blue (cells 3 and 4) are `aRepObjs`



🟩 Exists only in buggy sequences
■ Exists only in non-buggy sequences
▨ Exists in both buggy and non-buggy sequences
WITNESSES
31 22 6 25 4 1
Models Sorted by Occurrence
22 4 6 1 31 25
Models Sorted in the Order of Generation (Final Ranking)
4 22 1 6 25 31

Figure 5. Ranking the witnesses

that occur only in the non-buggy models, and cells in grey (cells 5 and 6) are `aRepObjs` occur in both buggy and non-buggy models.

The last sequence in Figure 5 shows an example of a sorted result, i.e., final ranking. Let *RANKING* denote a list of ranked witnesses from left to right.

2.7 Culprit Instruction Selection

We analyze the access logs of the ranked witnesses to identify the culprit executed instructions, i.e., executed instructions that improperly manipulated the concrete `RepObj` or failed to make proper manipulation(s). As discussed in Section 2.4, the access information is constructed from analyzing the executed instructions meaning that we can easily reconstruct the executed instructions from the information.

Given the ranked list of witnesses, the set of buggy models, and the set of non-buggy models, we take the following steps to identify the culprit instructions:

1. *Find the common information* (Sec. 2.7.1). Find the information that is in the ranked witness's access log and all its correspondings within the same group.
2. *Find the unaligned information* (Sec. 2.7.2). Find the information that is in the ranked witness's access log, but does not appear in one or more access logs of corresponding `aRepObjs` on the other side.
3. *Find the intersecting information from the above two steps* (Sec. 2.7.3). Find the ranked witness's information that is commonly found in all corresponding `aRepObjs` on the same group while do not appear in one or more times on the other side.

2.7.1 Find the Common Access Information. Given the two model access logs A and B, we align the access information positions in A to the positions in B. The two access information aligns if they have the same (1) function symbol, (2) executed instruction's opcode and operands, (3) accessed location, (4) accessed value (including the written value if the access was for a value write), and (5) access type. There are five different access types: new object creation, new value write, value overwrites, and machine instruction emit. Given an access log Q of some ranked witness $\widehat{s}_j$ and access log sequences $\mathbf{L} = \{L_1, L_1, ..., L_n\}$, here L_i is an access log of corresponding aRepObjs of $\widehat{s_i}$ in the same group, we find the common access information, i.e., access information that is found in all access logs Q and $\mathbf{L}$.

2.7.2 Find the Unaligned Access Information. Given an access log Q of some ranked witness $\widehat{s_j}$, $1 \leq j \leq n$, we select all access information that does not align to one or more access information of the corresponding aRepObjs on the other side. The idea is that if the same instruction was executed on both the buggy and non-buggy concrete RepObjs, the instruction does not influence the decision to make the concrete RepObjs either buggy or non-buggy. Thus, the instruction can be removed from the candidacy for culprit instruction. On the other hand, if an instruction was executed on one side, but not on the other side, implicates that an instruction may have influenced the decision. Thus, such instruction is worth considering for further analysis.

2.7.3 Find the Intersecting Access Information. The final step in selecting the culprit instruction(s) is finding the executed instructions that are common to one side (buggy or non-buggy) but not on the other side.

2.8 Culprit Instruction Ranking

Given the set of selected culprit executed instruction information, we rank the information in ascending execution order, i.e., instructions that are executed earlier during execution are ranked higher than those that are executed later. The intuition is that those culprit instructions that are executed earlier are more likely to have played a causal role in creating incorrect RepObjs compared to those executed later. Since the instruction ids represent the order of instruction execution, we can rank culprit instructions by comparing their instruction ids.

2.9 Rankings Analysis and Final Result Generation

The final step of our tool is ranking analysis and producing the final result file. Our result file holds (1) a list of ranked witness information and (2) lists of ranking functions, which each list is nested under each ranked witness information.

```
1  Ranking 1. Opcode: 0xabc, IR Opcode(s): 0xcbd
2  |--Ranking 1. Function ABC.
3  |--Ranking 2. Function KLM.
```

2.9.1 Ranked Witness Information. We provide three main pieces of information about the ranked witness, which are the ranking number, RepObj opcode, and the list of IR node opcodes. For example, in the example, the 1^{st} ranked witness corresponds to the concrete RepObj with opcode 0xabc, which is from an IR node with opcode 0xcbd.

2.9.2 Lists of Ranked Functions. Lines 2-3 show examples of function names listed under each ranked witness. These functions are identified by analyzing the ranked culprit executed instruction information. While traversing the ranking, we map the information to the function level while maintaining the ranked order.

3 Evaluation

We developed a tool to evaluate the efficacy of our ideas. We use esprima-python [8] to generate an abstract syntax tree (AST) for JavaScript programs, and escodegen [24] to generate JavaScript programs from the ASTs. Modeler is built on top of Intel Pin Tool [20] using C/C++. The analyzers for selecting, ranking, and producing the final result are built using Python3.8. The evaluation was done on a machine with 32 cores (@3.30 GHz) and 1 TB of RAM, running Ubuntu 20.04.1 TLS.

3.1 Methodology

Our automatic bug localization tool targets dynamic code generation bugs in Google V8's JIT compiler back-end. We experimented on 40 different bugs obtained in two different methods. We obtained five bugs from Google's bug report Community (https://bugs.chromium.org/), and we introduced 35 synthetic bugs in the JIT compiler source code.

We classify the bugs into three different classes depending on where the bug is located. The classes are (1) Instruction selector bugs; (2) register allocation bugs; and (3) code generation bugs. For example, the real bugs no.1196683, 1336869, 5129, and 9113 are instruction selector bugs, i.e., incorrect type has been assigned to the back-end representation object. The real bug no.913296 is a code generation bug, i.e., incorrect machine instructions are generated. Then, we further segment the bugs into three different types: (1) a bug in the JIT compile source code that is not expected to execute, but executed; (2) a bug is in the code that is expected to execute, but failed to execute, and (3) a bug in the code that is accessing the value of representation, i.e., reading or writing a bad value to the representation or code buffer.

To thoroughly assess the efficacy of our tool, we selected our synthetic bugs to ensure that they covered all relevant parts of the JIT compiler back-end and included each of the three types of bugs discussed above. To measure the accuracy of our ranking result, we compare our result to the ground truth. The ground truth is obtained using the following steps:

1. If a bug is from the report, we identify the ground truth from the developers' discussion and the committed fix. If a bug is introduced by ourselves, we keep a record of which function we are introducing the bug to.
2. We add print statement(s) in the buggy function to print the address and opcode of ground truth RepObjs that the function manipulates.
3. We run the PoC program with the buggy version of V8 executable, d8, to get the representation object's address and opcode.

3.1.1 Accuracy of Rankings. Table 1 shows the accuracy of ranking results from our tool using the Top-n metric (we

Table 1. Top-*n* Ranking Results ($n = 1, 5, 10, 20$)

Ranking	Top-1	Top-5	Top-10	Top-20
Model	16 (40.0%)	31 (77.5%)	35 (87.5%)	40 (100%)
Function	26 (65.0%)	39 (97.5%)	39 (97.5%)	40 (100%)

use $n = 1, 5, 10, 20$), which counts the number of test inputs where the ground truth occurs in the top *n* rankings produced by our tool, e.g., 'Top-1' gives the number of test inputs where the ground truth is ranked first by our analysis and 'Top-5' counts the number where the ground truth is within the top 5 items in our ranking.

The table header (first row) shows the labels of each column. The second row of the table shows the ranking result of witnesses that correspond to the ground truth `RepObjs` in Top-1/5/10/20. Finally, the last row of the table shows the ranking result of the ground truth function in Top-1/5/10/20. Our tool successfully localized 16, 31, 35, and 40 bugs within Top-1, Top-5, Top-10, and Top-20, respectively. In percentage, the witnesses are ranked 40%/77.5%/87.5%/100% bugs within Top-1/5/10/20 while the culprit functions are ranked 65%/97.5%/100%/100% bugs within Top-1/5/10/20.

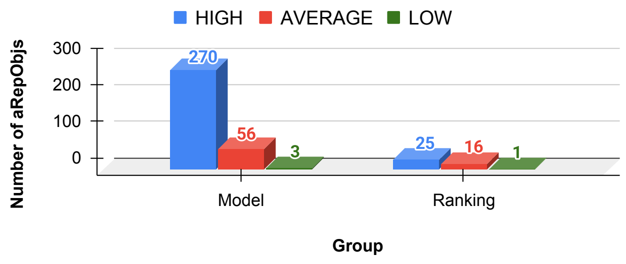


Figure 6. Number of abstract RepObjs by group

Figure 6 shows the number of `aRepObjs` by the group in high, average, and low. 'Model' (left) shows the number of `aRepObj` in models we constructed for 40 bugs. High means the highest number of `aRepObjs` constructed for a single model among the models for all input programs, which is 270 objects in our experiments. Average means the average number of `aRepObjs` constructed for 40 bugs, which is 56. Low means the lowest number of `aRepObjs` constructed for a single model among the models for all input programs, which is 3. 'Ranking' (right) shows the number of `aRepObj` in the ranking. Among the 40 ranking results, the highest number of `aRepObjs` ranked in a single result is 25. On average, 16 `aRepObjs` are ranked. The lowest number of ranked `aRepObjs` is 1. Comparing the 'Ranking' to 'Model,' the smaller the numbers is better. Our approach can reduce the number of `aRepObj` in the rankings by 90.74%, 71.43%, and 66.67% for high, average, and low, respectively, from the models. The higher the percentage is better.

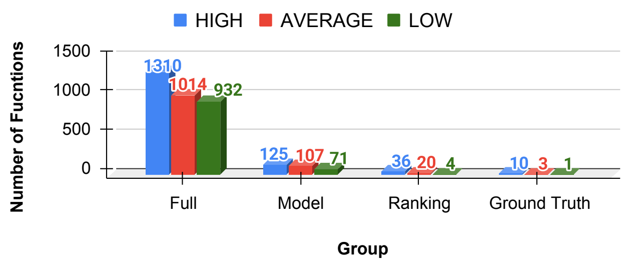


Figure 7. Number of functions by group

Figure 7 shows the number of JIT compiler functions by the group in high, average, and low. 'Full' shows the highest, average, and lowest number of functions executed when JIT compiling 40 bug PoC programs. The highest number of JIT compiler functions executed when JIT compiling one of 40 bug PoC programs are 1310, 1014 is the number of functions executed on average, and 932 is the lowest number of functions executed. 'Model' shows the number of functions executed on the back-end representations and code buffers. 'Ranking' shows the number of functions listed in the ranking. 'Ground Truth' shows the number of functions ranked under the ground truth `RepObj` in the ranking. The result shows that our approach was able to reduce the number of functions from the complete list of executed functions to functions ranked underground truth `RepObj` by 99.24%/99.71%/99.89% for high, average, and low, respectively.

The study [15] shows that developers' acceptability level of automatic bug localization results is if the ground truth is ranked within the Top-10, and the developers' most preferred method (function) level granularity. This suggests that for the automatic bug localization approach to be used in practice,it is essential to remove irrelevant elements in debugging and ranking the appropriate targets. Our results suggest that we can significantly remove the irrelevant elements and, in most cases, rank the ground truths (back-end objects and functions) within the Top-10.

3.2 Example: Bug Report No.1196683

```
Candidate Rank #1. Opcode: 0xb3, IR Opcode(s): []
   - Function Rank# 1. VisitRO
Candidate Rank #2. Opcode: 0x60, IR Opcode(s): 0x1d1, 0x131
   - Function Rank# 1. EmitWithContinuation
...
Candidate Rank #7. Opcode: 0xd6, IR Opcode(s): 0x1cc
   - Function Rank# 1. VisitChangeInt32ToInt64
...
Candidate Rank #24. Opcode: 0xdd, IR Opcode(s): 0xea
   - Function Rank# 1. AddInstruction
```

Figure 8. Example of the output generated for bug no.1196683. The ground truth witness is ranked no. 7 and the culprit function is ranked no. 1.

Figure 8 shows an example of the output generated by our tool for bug report number 1196683 (CVE-2021-21220) [25]. The bug is in the function `VisitChangeInt32ToInt64()`. It causes a `RepObj` with an incorrect opcode `0xd6` (`kX64Movl`) to be generated for `0x1cc` (`ChangeIntToInt64`) IR node; the correct opcode should be `0xd7` (`kX64Movsxlq`). This is a type mismatch bug that can be abused in the exploitation by creating a corrupted array to achieve the out-of-bound (OOB) primitive. Our tool successfully analyzed the JIT compiler's back-end generating `0xd6 RepObj` from the `0x1cc` IR node leads to buggy behavior, i.e., mismatch of execution output between the bytecode-only and JIT-compiled code. As a result, the witness corresponds to the ground truth is ranked at 7 (line 6). Additionally, our tool identified `VisitChangeInt32ToInt64` function as the culprit and only listed in the function ranking (line 7).

4 Discussion

4.1 Target Identification for Mutation

The quality of input programs can affect the quality of rankings. This is because a difference in the input programs can cause the JIT compiler to execute different optimizations, which can lead to a different back-end behavior. In some cases, fuzzer's target identification fails to identify the code parts that are directly related to the bug. For example, if the bug triggering code is within the if-else statement, altering the control-flow by changing the statement's condition can remove the buggy behavior. Our fuzzer identifies the condition as target to mutate. Although, mutating the condition can remove the buggy behavior, the accuracy of rankings may be lower than the rankings obtained from the input programs with the mutation of code that is directly related to the bug in the JIT compiler. We are working on improving the target identification to improve the quality of rankings.

4.2 Efficiency of Modeler

JIT compilers perform multiple passes of optimizations and code generation for different input programs. Based on the complexity of JIT compilation, the size and time for obtaining the execution traces are different. Since our modeler builds models by analyzing the execution traces, the performance of the modeler is inconsistent. For example, our modeler takes approximately 3 minutes to build a model for the PoC program of bug report no.1196683, while the modeling can take less or more time for other programs. We seek to minimize the modeling time to improve the overall performance.

4.3 Threats to Validity

To reduce the threats to validate the effectiveness of our evaluation, we performed experiments on 40 bugs, which include real-world bugs and the bugs introduced referencing the real-world bugs. Moreover, we partitioned the bugs based on the components the bug is located in the back-end and in 3 different types. Such partitioning is to validate our approach is able to cover different component bugs and the types. In best of our knowledge, there are no existing approaches consider to segment the bugs carefully as ours.

5 Related Work

There are only a limited number of literature on automated bug localization in JIT compilers. JIT compilers have a number of characteristics that make them very different from ordinary compilers; as a result, existing work for ordinary compilers [4, 5, 27, 29] do not carry over to JIT compilers.

Statistics-based techniques [4–6, 13, 14, 17, 27, 29], are widely used in automatic bug localization. The approaches require many inputs (a few hundred to thousands). These inputs are collected either from user reports (sampling) or fuzzers, e.g., AFL [28]. Unfortunately, the approach to wait for many user reports is not very practical when it comes to the systems that change rapidly and the problem can be security sensitive. The random generation of inputs without the fuzzer aware of JIT compiler characteristics and a bug is hard to guarantee the high accuracy in identifying the bug.

While the work done by Lim and Debray [18] and Lim *et al.* [19] have similar concept to ours, i.e., bug localization through an explicit modeling of JIT compiler behavior, their approach is not applicable to JIT compiler back-end bugs.

6 Conclusion

Just-in-Time (JIT) compilers are used in a wide variety of software. Thus, a bug that can rise to an exploitable vulnerabilities can have a high impact. This paper proposes an approach to automatically localize dynamic code generation bugs in the JIT compiler back-end. Empirical studies on Google Turbofan show that our approach can successfully model the behavior of back-end and use the models in localizing the bugs with high accuracy.

Acknowledgements

This research was supported in part by the National Science Foundation under grant no. 1908313.

References

[1] n.d. Project Zero: News and updates from the Project Zero team at Google. https://googleprojectzero.blogspot.com/

[2] Mickey R. Boyd and David B. Whalley. 1993. Isolation and Analysis of Optimization Errors. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 26–35. https://doi.org/10.1145/155090.155093

[3] bugzilla.mozilla.org. n.d.. Bug Writing Guidelines. https://bugzilla.mozilla.org/page.cgi?id=bug-writing.html

[4] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August*

26-30, 2019, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 223–234. https://doi.org/10.1145/3338906.3338957

[5] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 78–89. https://doi.org/10.1145/3324884.3416570

[6] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 34–44.

[7] chromium.org. n.d.. Reporting Security Bugs. https://www.chromium.org/Home/chromium-security/reporting-security-bugs/

[8] JS Foundation. 2012. esprima-python. https://github.com/Kronuz/esprima-python

[9] Mozilla Foundation. n.d.. IonMonkey/LIR. https://wiki.mozilla.org/IonMonkey/LIR

[10] Google. n.d. V8 Instruction Class. https://github.com/v8/v8/blob/main/src/compiler/backend/instruction.h#L859

[11] Apple Inc. n.d.. Bare Bones Backend/B3 Intermediate Representation. https://webkit.org/docs/b3/intermediate-representation.html

[12] Fedor Indutny. 2015. *Sea of Nodes*. Accessed 2021-02-22.

[13] Lingxiao Jiang and Zhendong Su. 2007. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 184–193.

[14] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 241–255.

[15] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization, Andreas Zeller and Abhik Roychoudhury (Eds.). 165–176. https://doi.org/10.1145/2931037.2931051

[16] Xia Li and Durga Nagarjuna Tadikonda. 2022. Improving Mutation-Based Fault Localization via Mutant Categorization. In *The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022*, Rong Peng, Carlos Eduardo Pantoja, and Pankaj Kamthan (Eds.). KSI Research Inc., 166–171. https://doi.org/10.18293/SEKE2022-157

[17] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. 2003. Bug isolation via remote program sampling. *ACM Sigplan Notices* 38, 5 (2003), 141–154.

[18] HeuiChan Lim and Saumya Debray. 2021. Automated bug localization in JIT compilers. In *VEE '21: 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual USA, April 16, 2021*, Ben L. Titzer, Harry Xu, and Irene Zhang (Eds.). ACM, 153–164. https://doi.org/10.1145/3453933.3454021

[19] HeuiChan Lim, Xiyu Kang, and Saumya Debray. 2022. Modeling Code Manipulation in JIT Compilers. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2022)*. 9–15.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL, 190–200.

[21] Vijay Naidu and Ajit Narayanan. 2016. Needleman-Wunsch and Smith-Waterman Algorithms for Identifying Viral Polymorphic Malware Variants. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2016, Auckland, New Zealand, August 8-12, 2016*. IEEE Computer Society, 326–333. https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2016.73

[22] Oracle. 2010. Instructions, Operands, and Addressing.

[23] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verification Reliab.* 25, 5-7 (2015), 605–628. https://doi.org/10.1002/stvr.1509

[24] Yusuke Suzuki. 2012. Edcodegen. https://github.com/estools/escodegen

[25] Adrian Taylor. 2021. Issue 1196683: Security: 2021 pwn2own entry. https://bugs.chromium.org/p/chromium/issues/detail?id=1196683

[26] David B. Whalley. 1994. Automatic Isolation of Compiler Errors. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1648–1659. https://doi.org/10.1145/186025.186103

[27] Jing Yang, Yibiao Yang, Maolin Sun, Ming Wen, Yuming Zhou, and Hai Jin. 2022. Isolating Compiler Optimization Faults via Differentiating Finer-grained Options. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 481–491. https://doi.org/10.1109/SANER53432.2022.00065

[28] Michal Zalewski. 2013. american fuzzy loop. https://lcamtuf.coredump.cx/afl/

[29] Zhide Zhou, He Jiang, Zhilei Ren, Yuting Chen, and Lei Qiao. 2022. LocSeq: Automated Localization for Compiler Optimization Sequence Bugs of LLVM. *IEEE Trans. Reliab.* 71, 2 (2022), 896–910. https://doi.org/10.1109/TR.2022.3165378

Received 2022-11-10; accepted 2022-12-19