Automated Bug Localization in JIT Compilers

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

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

Abstract

Many widely-deployed modern programming systems use just-in-time (JIT) compilers to improve performance. The size and complexity of JIT-based systems, combined with the dynamic nature of JIT-compiler optimizations, make it challenging to locate and fix JIT compiler bugs quickly. At the same time, JIT compiler bugs can result in exploitable security vulnerabilities, making rapid bug localization important. Existing work on automated bug localization focuses on static code, i.e., code that is not generated at runtime, and so cannot handle bugs in JIT compilers that generate incorrect code during optimization. This paper describes an approach to automated bug localization in JIT compilers, down to the level of distinct optimization phases, starting with a single initial Proof-of-Concept (PoC) input that demonstrates the bug. Experiments using a prototype implementation of our ideas on Google's V8 JavaScript interpreter and TurboFan JIT compiler demonstrates that it can successfully identify buggy optimization phases.

CCS Concepts: • **Security and privacy → Software security engineering**; *Web application security.*

Keywords: Program Analysis, Debugging, Bug localization, Dynamic Code, Self-Modifying Code

ACM Reference Format:
HeuiChan Lim and Saumya Debray. 2021. Automated Bug Localization in JIT Compilers. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21), April 16, 2021, Virtual, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3453933.3454021

1 Introduction

Many widely-deployed modern programming systems include a language interpreter, which provides portability, together with a just-in-time (JIT) compiler, which provides performance: examples range from JavaScript code executed in web browsers to enterprise software written in Java. Such systems typically consist of multiple sophisticated interacting components (e.g., an interpreter; a runtime system including a profiler, a garbage collector, etc.; a JIT compiler), and as a result tend to be large and complex, and thus may be prone to bugs.

Bugs in the JIT compiler that can be particularly challenging to diagnose and fix are those that result in the generation of incorrectly JIT-optimized application code, causing the application to compute incorrect results or crash. When a buggy JIT compiler emits incorrect code, the problem manifests itself, not in the code that contains the bug (the JIT compiler), but elsewhere, in the application code being optimized. Moreover, the optimized code generated by the JIT compiler is not available for static analysis, as with conventional compilers, but is created dynamically and may be modified multiple times during execution. At the same time, incorrectly optimized code resulting from JIT compiler bugs can result in security problems.

For example, Rabet describes a JIT compiler bug in the Chrome web browser's V8 JavaScript engine that causes some initialization code in the application program to be (incorrectly) optimized away, resulting in an exploitable vulnerability (CVE-2017-5121) [34].

The widespread adoption of systems that use JIT compilers, combined with the potential for security vulnerabilities arising from JIT compiler bugs, makes it important to locate and fix such bugs quickly.

There is a considerable body of research on automated bug localization: Section 7 gives a deeper discussion. To the best of our knowledge, all of this work focuses on static code, i.e., where code is not created or modified during execution.

Crucially, these approaches do not track dependencies arising from the act of runtime code generation, e.g., where code *A* generates code *B* at runtime and a bug in *A* can result in an incorrect instruction sequence generated for *B*. As a result, existing work on automatic bug localization is inapplicable to the situation we consider: namely, where a bug in the JIT compiler is manifested as buggy behavior in the dynamically generated code. To address this situation, we propose an approach that explicitly models the JIT compilation process and uses it to reason about the JIT compiler's behavior.

Experimental results from a prototype implementation of our ideas, evaluated using bug reports for the TurboFan JIT compiler [40] used in Google's V8 JavaScript Engine, indicate that our approach is effective in localizing JIT compiler bugs.

The remainder of this paper is organized as follows. Section 2 briefly summarizes some background on interpreters and JIT compilers. Section 4 discusses our research ideas for bug localization. Section 5 describes experimental results from a prototype implementation of our ideas. Section 6 discusses these results and possible future improvements. Section 7 summarizes related work, and Section 8 concludes.

2 Background

This section briefly discusses some key concepts relevant to our ideas. It may be skipped by readers familiar with this material.

2.1 Interpreters and JIT Compilers

An interpreter implements a virtual machine (VM) in software. Programs are expressed using the VM's instruction set, with each VM instruction represented as a data structure in the interpreter's memory. To mitigate the runtime performance overheads typically incurred by interpreters, they are often coupled with Just-in-Time (JIT) compilers, which dynamically optimize frequently executed code fragments into native code. The overall structure of typical interpreter/JIT-compiler system is therefore as follows: the input program is read in and translated into an intermediate representation (IR), which is then used to quickly generate byte-code or unoptimized native code. Subsequently, as the program is executed, frequently executed code fragments are identified and JIT-compiled to more efficient code. Some JIT compilers support multiple levels of optimization, where dynamically generated code from one set of optimizations may subsequently be subjected to additional rounds of optimization [37].

Commonly used IRs have a tree or graph structure, e.g., abstract syntax tree or control flow graph. For concreteness in this discussion, we will assume that the IR is a graph and thus contains a collection of nodes [18]. Each such node can be thought of as having a set of (system-dependent) *properties*, e.g., its type, the set of its inputs, its register color (if graph coloring is used for register allocation), etc.

Optimizations within a JIT compiler are typically organized as a sequence of *phases*, where each phase refers to a specific optimization to the IR (e.g., constant propagation) together with any supporting program analyses [1, 10]. The effect of performing an optimization is to modify the program's IR. We can use the properties associated with the nodes in the IR to reason about the effect of optimization on a program.

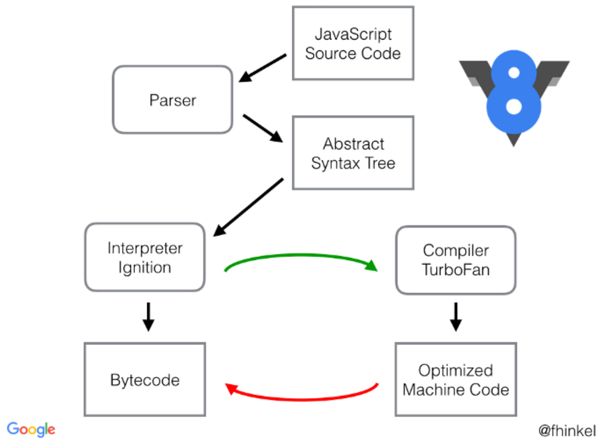

Figure 1. V8 Pipeline[16]

2.2 JavaScript Engine Pipeline

JavaScript engine is an interpreter and JIT compiler system that is implemented specifically for JavaScript language. The JavaScript engines that we can easily found are V8 in Google Chrome[12], Chakra in Microsoft Edge[13], SpiderMonkey in Mozilla Firefox[31], and JavaScriptCore in Apple Safari[11], etc. Internally, each engine has different implementations, but they all follow the general pipeline.

When a JavaScript engine receives input code, it first parses to it to generate the abstract syntax tree for the interpreter. The interpreter generates byte-codes based on the input syntax tree. Then, while running the byte-code, the engine evaluates concurrently to identify which code is being repeatedly executed (a.k.a hot code). If the engine evaluated some code is "hot", then it invokes the compiler to compile and optimize the byte-code to native code [9, 30]. While the optimized is being executed, it performs a check on each piece of a code's assumption that "what needs to be done." If this check fails, it deoptimizes the code and returns to the byte-code stack frame [30, 36]. Figure 1 illustrates Google Chrome's V8 engine pipeline.

3 A Running Example

Figure 2 shows an example of a "Proof-of-Concept" (PoC) for a V8 JIT compiler bug [27] that we will use as a running example. In this program, the high iteration count of the for loop triggers JIT compilation. The JIT compiler has a bug at the simplified lowering phase that gets triggered by the line of code `a = i + -0`, but this is not visible until a garbage collection occurs (in this example, forced via the call `gc()`). In the resulting JIT-optimized code, the value for

```
var a, b;  // should be var
for (var i = 0; i < 100000; i++) {
    b = 1;
    a = i + -0;
    b = a;
}
print(a === b);  // true
gc();
print(a === b);  // false
```

Figure 2. An example of a PoC for a V8 JIT compiler bug

a is incorrectly written to a memory location where it is supposed to be protected by the write barrier.

4 Research

We make the following assumptions about the JIT compiler under analysis:

1. The optimization phases within the JIT compiler, and the function(s) that implement each such phase, are known.
2. We can obtain a machine-instruction-level execution trace of the JIT compiler. There is enough symbol table information available in the JIT compiler executable to map each machine instruction executed to the function it belongs to. This allows us to determine the sequence of optimization phases executed for any given input program.
3. We can identify the input program's IR and determine the values of the properties of IR nodes.

We assume that we have a bug report that contains a Proof-of-Concept (PoC) input program that demonstrates the JIT compiler's buggy behavior. Such PoCs are typically submitted when a bug is found and reported. This section discusses how we use this information to identify the optimization phase that most likely contains a JIT compiler bug.

4.1 Overview

Our approach to automated bug localization, starting from this PoC, consists of the following steps:

1. We begin by automatically modifying this PoC to create a set of new input programs.
2. We run each of these programs P_i and collect an instruction-level trace of their executions.
3. We analyze these execution traces to determine whether or not P_i manifested the bug and to identify P_i's intermediate representation (IR) within the JIT compiler together with the optimization phases executed while optimizing P_i.
4. From the information so gathered, we pick out the candidates of where the bug may reside among the tracked optimization phases.
5. Finally, we rank these candidates to identify the most likely phase for where the bug is located.

The use of an instruction-level trace, rather than a higher-level trace obtained using system-specific options or tools (e.g., using V8's `-trace-turbo-graph`) is motivated by two considerations. First, the use of system-specific features can inhibit portability across systems. Second, these higher-level traces may not provide sufficiently detailed information about how the JIT compiler manipulates IR nodes. However, it does have the downside that the collection of instruction-level traces can be expensive in both time and space.

The remainder of this section describes each of these steps in more detail.

4.2 Modified PoC Generation

Conceptually, we can think of the process of automatic bug localization as taking the code involved in a buggy execution and determining which portions of it might contain the bug and which portions definitely do not. The greater the amount of code that can be excluded as "definitely not buggy" the better the bug localization. To do this, we need a way to distinguish possibly-buggy code from definitely-not-buggy code. For manual debugging, software developers might use their knowledge of the application code and/or programming language to do this, but this does not seem easily automatable. A more easily automated approach, pioneered by Liblit [23–25], is to compare a set of buggy program executions with a set of non-buggy executions to identify execution behaviors that are common to the buggy executions but not the non-buggy ones. This requires multiple program executions, which requires multiple inputs.

In our case, unfortunately, we have only a single PoC input.[1] To deal with this situation, we modify the original PoC input to create a set of additional input programs. This modification process is guided by the following constraints.

1. To ensure that the newly generated programs are syntactically correct, we modify the abstract syntax tree (AST) of the original PoC rather than its source code. We apply tree transformations that ensure that the result is also a valid AST, then map the modified AST back to source code.
2. To ensure semantic similarity between the newly generated programs and the original PoC, all AST node modifications are constrained to preserve the type of the node. Specifically, this means that a literal can only be replaced by another literal; a binary arithmetic operator can only be replaced by another binary arithmetic operator; an integer constant can only be replaced by another integer constant; a string can only be replaced by another string; etc.

[1]It is of course possible that there may be multiple PoC inputs submitted for a particular JIT compiler bug, but this is not something we can count on in general.

3. JIT compiler optimizations are sensitive to the structure of the code being optimized, and large changes to the input program can result in substantially different JIT compiler behavior, making it less useful for automatic bug localization. To this end, we keep the number of edits to the PoC code small. For example, the prototype implementation described in Section 5 uses only a single AST modification to generate each new program.

The second and third constraints produce programs that are generally similar to the original PoC. This makes it likely that the JIT optimizations they experience will resemble the original PoC (though in generali, they will not be identical).

Example 4.1. Consider the PoC code shown in Figure 2. The following are three of the new PoCs generated using the modification process described above:

New program 1:

```
var a, b;
for (var i = 0; i < 100000; i++) {
    b = 1;
    a = i + +0; // Changed from '-' to '+'.
    b = a;
}:
print(a === b);
gc();
print(a === b);
```

New program 2:

```
var a, b;
for (var i = 0; i < 100000; i++) {
    b = 1;
    a = i + -1; // Changed from '0' to '1'.
    b = a;
}
print(a === b);
gc();
print(a === b);
```

New program 3:

```
var a, b;
for (var i = 0; i < 100000; i++) {
    b = 1;
    a = i & -0; // Changed from '+' to '&'.
    b = a;
}
print(a === b);
gc();
print(a === b);
```

□

4.3 Correct and Incorrect Execution of PoCs

After generating new PoCs as described in the previous step, we execute each generated program *P* twice: once with JIT optimization turned off (i.e., using only the interpreter) and one with JIT optimization turned on. Since this work is concerned only with JIT-compiler bugs, the interpreter-only execution is considered to be "correct." Thus, if *P* has the same observable behavior with and without JIT optimization, the JIT compiler's execution on input *P* is deemed to be *non-buggy*; otherwise it is deemed to be *buggy*.

The approach discussed above assumes that the AST modifications will generate a PoC variant that executes correctly. While this cannot be guaranteed with absolute certainty, the likelihood of obtaining a variant that executes without errors can be increased by running the PoC generator a large number of times. This issue is discussed further in Section 6.

4.4 Representing Optimization IR in Graphs

As mentioned earlier, we assume that the function(s) implementing each JIT-compiler optimization phase is known to the bug localization tool. Given an execution trace *T* for a JIT compiler, we define the *scope* of a phase φ in *T* as a subtrace of *T* that begins at the first instruction where a function implementing the phase φ is entered and ends at the instruction where that function call returns. A phase may have multiple distinct scopes in a trace, and a scope for a phase may be nested within (i.e., be a subtrace of) scope for another trace.

We collect a machine-instruction-level trace of the JIT compiler's execution on each PoC code and analyze the trace to determine (*i*) the sequence of optimization phases executed, and (*ii*) how these phases manipulate the input program's IR during JIT optimization. Algorithm 1 shows the algorithm for this. The algorithm proceeds as follows. For each instruction in the trace, we use the symbol table information in the JIT compiler binary to map it to the corresponding function name. We use this to identify the entry into and return from the functions that implement each phase and thereby identify the scope of each phase. This process of phase identification is important as we are grouping the IR nodes and the optimization activities when any modification to the IR happened within the specific phase scope. We scan the execution trace *T* and identify the instructions that generate or modify an IR node for each phase and we update the graph *G* appropriately. We also identify instructions that change any property of an IR node and update *G* to record this information. Our current implementation only considers the property of a node that was removed from the IR and disabled so that it will not be converted to a machine code. At the end of the analysis, this produces an undirected graph that represents the IR that the JIT compiler has generated and optimized. The resulting graph *G* is then passed to next step for analysis.

Example 4.2. Figure 3 illustrates the phase graph *G* for the PoC code shown Figure 2. A point to note is the large number of IR nodes, and the density and complexity of their structure, even for such a small and simple program. This complexity is one of the factors that makes bug localization in JIT compilers challenging. □

Algorithm 1: Optimization tracking on a graph

```
Input: An execution trace $T$
Result: Undirected graph $G$ that represents
        optimized IR for $T$
function current_phase(I):
    $f$ = function that instruction $I$ belongs to
    if $f \neq \perp$ and $f$ implements a phase $p$ then
        if $I$ is the entry to function $f$ then
            push $p$ on PhaseStack
            return $p$;
        else if $I$ is a return from function $f$ then
            $p = pop(PhaseStack)$
            return $p$;
    else if PhaseStack is not empty then
        return $top(PhaseStack)$
    else
        return $\perp$
begin
    $V = \emptyset$; /* the set of vertices of $G$ in the order of
      generation */
    $E = \emptyset$; /* the set of edges of $G$ */
    $G = (V, E)$
    PhaseStack = empty stack
    for each instrucion $I$ in $T$ do
        if $I$ generates a new IR node then
            create a new vertex $v$ corresponding to
              this new node
            $v.properties = \emptyset$
            add $v$ to $V$ in the order of generation
        else if $I$ adds a node $u$ to an existing node $v$
          then
            add an edge $(u, v)$ to $E$
        else if $I$ removes a node $u$ from an existing
          node $v$ then
            remove the edge $(u, v)$ from $E$
        else if $I$ changes a property $q$ of an existing
          node $v$ then
            $p$ = current_phase($I$)
            add $(q, p)$ to $v.properties$
```

4.5 Phase Graph Analysis

The next step of our analysis is to compare the phase graphs constructed in the previous step to identify differences between the phase graphs for buggy and non-buggy executions of the JIT compiler. As noted in Section 4.2, the different PoC codes we consider are obtained by making a set of minimal edits to the original PoC, and so are structurally very similar to the original PoC. Ideally, given two structurally similar input programs where one results in a buggy execution in the JIT compiler while the other gives a non-buggy execution, the difference(s) between these execution behaviors—and, therefore, in the corresponding phase graphs—should arise only from the effects of the bug, thereby allowing us to localize the bug. However, the situation is complicated by the fact that the behavior of a JIT compiler can be highly sensitive to the input program, such that even small differences in the input program can cause significant differences in the behavior of the JIT compiler. In particular, the set of optimizations performed by the JIT compiler can be quite different. Such differences in optimization include: (1) different transformations applied to the generated IR nodes within the same optimization phase; (2) some optimization phases are not triggered; or (3) some additional optimization phases are triggered. The goal of the phase graph analysis phase is to compare the differences between phase graphs and identify the differences between them. We then use the differences so identified to find candidate locations where the bug may be residing.

To this end, let Φ_{buggy} denote the set of phase graphs corresponding to buggy executions of the JIT compiler for the set of PoCs we are analyzing, and $\Phi_{nonbuggy}$ denote the set of phase graphs corresponding to non-buggy executions, determined as discussed in Section 4.3. We consider pairs of

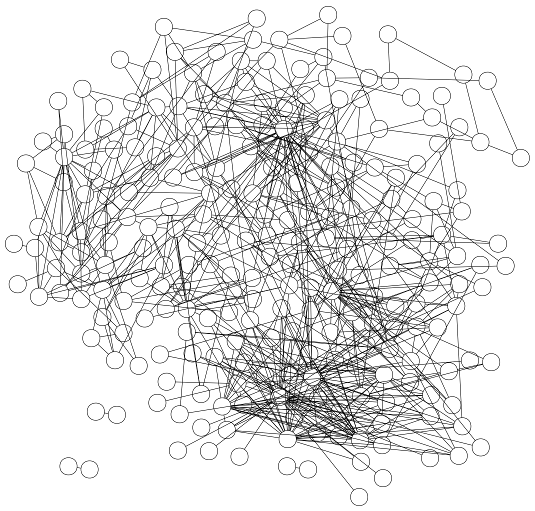

Figure 3. Phase graph for the PoC code of Section 3

phase graphs (g_1, g_2) where $g_1 \in \Phi_{buggy}$ and $g_2 \in \Phi_{nonbuggy}$, and use Algorithm 2 to determine where g_1 and g_2 differ.

Algorithm 2: Graph analysis to find the differences

```
Input: Phase graph $g_1 = (V_1, E_1)$, phase graph
       $g_2 = (V_2, E_2)$
Result: Set of phase candidates $C$
1  function $get\_phase\_diff(g_1, g_2)$:
2      return $(g_1.phases - g_2.phases) \cup (g_2.phases - g_1.phases)$
3  function $get\_node\_diff(g_1, g_2, C, i)$:
4      if $size(v1_i.attached\_nodes) \neq size(v2_i.attached\_nodes)$ then
5          add $v1_i.phase$ to $C$
6      else
7          for $j = 0$ to $size(v1_i.attached\_nodes) - 1$ do
8              if $v1_i.attached\_nodes_j \neq v2_i.attached\_nodes_j$ then
9                  add $v1_i.phase$ to $C$
10         if $v1_i.properties \neq v2_i.properties$ then
11             add $v1_i.phase$ to $C$
12 begin
13     $C = get\_phase\_diff(g_1, g_2)$
14     for $i = 0$ to $size(g_1) - 1$ do
15         $get\_node\_diff(g_1, g_2, C, i)$
```

Suppose that $g_1 \in \Phi_{buggy}$ and $g_2 \in \Phi_{nonbuggy}$. The *get_phase_diff* function identifies the differences in phases between two graphs. In other words, a phase exists in one graph, but not in the other. The *get_node_diff* function compares the nodes in two graphs sequentially for three different things to set the two nodes are different. This is because the nodes are generated sequentially following the order of phases executed, which means that if two exactly same PoCs are run at a different time and obtained graphs from them, these two graphs' node and phase orders are equal.

The criteria we use to compare two nodes for equality are (1) the number of nodes that are attached (line no.4), (2) the nodes that are attached also matches (line no.8), and (3) the properties of nodes (line no.10). If any of them fails to be equal, then the two nodes are considered as not equal and added to the candidates set (line no. 5, 9, and 11).

4.6 Candidate Selection

The graph analysis step described in the previous section allows us to compare the phase graphs for a correct and an incorrect execution and determine the set of optimization phases that are different between these executions. We refer to this as the *difference set* between the two-phase graphs. Given sets of phase graphs Φ_{buggy} and $\Phi_{nonbuggy}$, corresponding to the buggy and non-buggy executions respectively, we use this graph analysis to compute, for each $g_i \in \Phi_{nonbuggy}$ and $g_j \in \Phi_{buggy}$, the size of the difference set between g_i and g_j. To identify the set of possible optimization phases that may contain the bug, we select a pair (g_i, g_j) whose difference set is the smallest among all such pairs. The resulting difference set is taken to be the set of possible buggy phases. We select a pair of phases with the smallest number of differences because we want to find the graphs that are closest to each other in terms of computation and optimization such that one corresponds to the correct execution of the JIT compiler while the other corresponds to an incorrect execution. If there are multiple pairs with minimum values, we choose one of them arbitrarily.

Example 4.3. Suppose that our set of phase graphs is $\{G_1, G_2, \ldots, G_9\}$, where $\Phi_{buggy} = \{G_1, G_2, G_3, G_4, G_5, G_6\}$ and $\Phi_{nonbuggy} = \{G_7, G_8, G_9\}$. In the table T shown below, the value in the cell $T(i, j)$ represents the size of the difference set between the graphs $g_i \in \Phi_{nonbuggy}$ and $g_j \in \Phi_{buggy}$. For example, graphs G_1 and G_7 differ in 11 optimization phases.

		Φ_{buggy}					
T		G_1	G_2	G_3	G_4	G_5	G_6
$\Phi_{nonbuggy}$	G_7	11	12	4	8	17	19
	G_8	13	7	9	12	16	14
	G_9	7	4	6	22	21	11

Each entry in this table is computed using Algorithm 2. We can see that $i = 7$ and $j = 3$ minimizes the value of $T(i, j)$. Thus, the set of possible buggy phases is given by the difference set between G_7 and G_3. □

4.7 Ranking the Candidates

The ranking of candidate phases from most likely to the least likely phase where the bug may reside is decided by two criteria:

1. If a phase $\varphi \in C$, where C is a set of candidate phases, such that $\varphi \in G_x$ and $\varphi \notin G_z$, then φ is ranked higher than other candidates. This is the most straightforward case because this missing phase is the one that makes difference in the output. For example, let's say G_x is a graph from incorrectly executed code and G_z is a graph from the correctly executed graph. Then, it is somewhat clear that the additional phase φ execution caused the incorrect output as not having this phase executed gave correct execution.
2. If there are more than one missing phases or no missing phases that are in C, then we rank them by the order that the phase was executed. This is because there is a possibility that the following phases in the

Table 1. Bug reports considered for our evaluation

Report#	Date	V8 version	Problem summary
5129	June 2016	8.3.1	MachineOperatorReducer changes `x - y < 0` to `x < y` which is not safe when `x - y` can overflow [4]
8056	Aug. 2018	7.0.0	Function reducer assumes that the prototype is an initial one and has no element, but does not implement a check for this assumption [5].
791245	Dec. 2017	6.5.0	Write barrier to heap is sometimes incorrectly eliminated [27]
961237	May 2019	7.6.0	`null` is truncated to `+0` even in contexts such as `-0 == null` because it was not handling the TypeCheck correctly [6]
1072172	April 2020	8.4.0	In the Typer phase `Math.max` and `Math.min` generate the wrong type by mistakenly removing the `Type::MinusZero` property of the input nodes. [7]

candidate are impacted by the first place of the phase that has a bug as a result from one phase flows to the next.

4.8 A Concrete Example

This section discusses a concrete example of the application of the bug localization steps discussed above in the context of our prototype bug localization tool, which is described in Section 5.1. We focus on the bug discussed in Section 3 (bug report 791245 [27]).

4.8.1 PoC Generation.

The first step of the process is to generate a number of input programs that are variants of the original PoC, as described in Section 4.2; for our experiments, we specify a maximum of 20 such variants. Since we want each such variant to be only minimally different from the original PoC, we specify that each such variant should be only one edit different from the original. This ensures that, although the variant PoC codes are all different from each other, the computations of the JIT-compiler are nevertheless close to each other. In practice, the number of variant PoCs generated may be less than the limit of 20 because it is possible for a particular variant to be generated more than once, in which case duplicates are discarded.

4.8.2 Graph Generation and Analysis.

The next part of the prototype is to generate phase graphs from the execution traces for each newly generated input PoC codes, as described in Section 4.4. These graphs represent the IR that the JIT compiler generates and optimizes. In our experiment, we generated a total of 20 graphs, of which 8 correspond to non-buggy executions of the JIT-compiler and 12 correspond to buggy executions. We then analyze pairs of phase graphs, from the buggy and non-buggy executions, to identify the differences between their optimization phases (Section 4.5). For the particular bug report under discussion, the smallest difference sets obtained by this analysis had two candidate buggy phases while the largest difference sets had five candidate buggy phases.

4.8.3 Selecting the Candidates.

The smallest difference set obtained from the previous step contains the following two candidate buggy phases: `SimplifiedLowering` and `GenericLowering`. The largest difference set contains the following five candidate phases: `Inlining`, `LoopPeeling`, `SimplifiedLowering`, `GenericLowering`, and `EffectControlLinearization`. Our algorithm selects the smallest set as the set of possible buggy phases, namely: {`SimplifiedLowering`, `GenericLowering`}.

The other phases that occur in the largest difference set but not in the smallest one, e.g., `Inlining` or `LoopPeeling`, arise due to different optimizations applied by the JIT compiler due to differences in the execution behaviors of some of the PoC variants created by our tool. Our goal is to identify optimization phases that are consistently analyzed as differences across all the PoC variants. For this reason, we choose the phases occurring in the smallest difference set.

4.8.4 Ranking the Candidates.

The final step is to rank the candidate buggy phases identified in the previous step. We use the ranking algorithm discussed in Section 4.7 for this. We first check each optimization phases in the minimum candidate phases to find out whether any of them is missing in one phase graph but captured in the other. In our result, both phases are found to occur in both graphs. We next check the execution order of the phases: a phase that is executed earlier is ranked higher. In this example, `SimplifiedLowering` is found to be executed before `GenericLowering`. The ranking on the candidate phases generated by our tool is therefore

1. `SimplifiedLowering`
2. `GenericLowering`

We checked the bug report and the Github repository commit to find out whether this result matches the phases where the bug was fixed. In this example, it turns out that the actual buggy phase is `SimplifiedLowering`, i.e., the top-ranked candidate output by our tool: the bug reporter has reported that this bug is an optimization bug that has to do `Simplified-lowerer IrOpcode::kStoreField, IrOpcode::kStoreElement` [27].

Table 2. Accuracy of bug localization

Report no.	Possible buggy phases identified by our tool (in descending order of rank)	Actual buggy phase
5129	1. Inlining 2. SimplifiedLowering 3. GenericLowering 4. **EarlyOptimization**	EarlyOptimization
8056	1. **Inlining** 2. TypedLowering 3. LoopPeeling	Inlining
791245	1. **SimplifiedLowering** 2. GenericLowering	SimplifiedLowering
961237	1. **SimplifiedLowering** 2. GenericLowering 3. EffectControlLinearization 4. LoopPeeling	SimplifiedLowering
1072172	1. **Typer** 2. SimplifiedLowering	Typer

5 Evaluation

5.1 A Prototype Implementation

We evaluated our ideas using a prototype implementation and ran our experiments on a machine with 32 cores (@ 3.30 Ghz) and 1 TB of RAM, running Ubuntu 20.04.1 LTS. We used a dynamic analysis tool built on top of Intel's Pin software (version 3.7) [28] for program instrumentation and collecting instruction-level execution traces; and XED (version 8.20.0) [19] for instruction decoding [19]. Additionally, we used esprima-python [14] to generate the syntax-tree for JavaScript code; and escodegen [38] to regenerate the JavaScript code from the syntax-tree.

Our prototype targets Google's JavaScript engine V8, focusing in particular on TurboFan, V8's JIT compiler. The objective of our experiments is to determine the accuracy of our algorithm in automatically localize bugs to phase level in such a large and complex JIT compiler system. We used a number of bug reports from the V8 bug report site, `bugs.chromium.org`, to check whether the candidate buggy phases identified by our prototype match the place where the bug has been fixed as mentioned in the reports.

The bug reports we selected for our evaluation focused on incorrect optimized code generated by the JIT compiler; in our experience, such bugs are especially challenging to localize and can potentially benefit the most from automatic localization. They were based on the following criteria.

1. The bug resides in the JIT compiler source code. In other words, the misbehavior (i.e. system crash or incorrect output) is not caused by the bug in other parts of the system, such as interpreter or parser, etc.
2. An extension criteria from the first one, we selected a bug report that produces different output behavior in the optimized code from the interpreted code. For example, the interpreted execution returns a boolean value *true* while the optimized code returns a value *false*. This confirms that the problematic behavior is in fact due to incorrect optimization by the JIT compiler.
3. If a bug causes a crash during the optimization process, we exclude the bug report; whereas we select the bug report that indicated that the crash happens during the execution of optimized code. This is because if the system crashes during the optimization process, it usually gives a stack trace from the crashed function, which is, not easy, but somewhat straightforward to localize the bug. However, if the system crashes during the execution of optimized code, it is not so trivial to localize the bug in the optimizer as the information about who optimized the crashed code is not sufficient.
4. Finally, the misbehavior in the optimized code is due to incorrect optimization on the IR nodes. This is because we are comparing the differences in the IR nodes. If the bug resides outside of the IR node generation or modification, our algorithm is not suitable. Therefore, we studied the fixes that the V8 team has made, and if the fix was made on the code that has to do with the IR node then we select the bug report.

The bugs we considered in our evaluation are described in Table 1. For each such bug report, we proceeded as follows.

1. We compiled the appropriate version of V8 and confirmed that we could manually replicate the buggy behavior described in the PoC given as part of the bug report.
2. We used our tool to generate a ranked list of possible buggy phases.
3. We obtain the ground truth of the actual location of the bug using the next released version of V8 where the

bug has been fixed. We use two different approaches for this:

a. We compare the fixed source code in the new version with the buggy code in the old version to identify code changes, and thereby determine the location of the buggy phase. This is done with code where it is straightforward to identify the phase, e.g., if the fixed code is under simplified-lowering or typer file, etc.
b. The second approach is to locate the source code where the bug was fixed and add a marker that can be used to identify that code fragment in an instruction-level execution trace of the progam trace. A marker is a value that is (a) very unlikely to naturally generated during the program execution (e.g., `unsigned long long xyz = 0x3f4f5f6f` where `xyz` is a new variable); and (b) that does not change the program's execution behavior. We then collect an execution trace, use the marker to find the code fragment that was marked, and use the *current_phase* function shown in Algorithm 1 to determine its phase. This approach is used to confirm that first phase identification was correct and for cases where the same functions are used multiple times across different phases.

4. We compare the possible bug locations obtained from our tool with the ground-truth location of the fixed bug.

5.2 Accuracy of Bug Localization

Table 2 compares the ranked list of candidate buggy phases obtained from our tool against the actual buggy phases identified from examining the code of the fixed versions. The first column is the report number identifying the bug. The second column of the table shows the output from our tool giving the possible buggy phases in descending order of rank. The third table gives the ground truth location of the bug at the phase level. In this table, the number of candidate bug locations (column 2) is different for different bugs considered. This is because (a) different inputs result in the invocation of different optimizations, e.g., loop-peeling is not triggered for code without loops; (b) the number of differences found between the graphs for buggy and non-buggy executions of the JIT compiler are different for different inputs; and (c) we only select the smallest difference set.

It can be seen from Table 2 that for each of the five bugs considered, the actual buggy phase is in fact identified as one of the possible locations by our tool. For four out of the five bugs, the phase containing the actual bug is in fact ranked at the top of the list of possible bug locations obtained using our tool. For the fifth bug (report no. 5129 [4]), the actual buggy phase is ranked fourth in the ranked list of candidates given by our tool. The reason for this is that our ranking algorithm prioritizes candidate buggy phases in terms of their relative execution order. For this bug, the `EarlyOptimization` phase was executed after other phases, resulting in its lower ranking. An additional possible source of imprecision in our tool in this example is that the buggy function, `MachineOperatorReducer::Reduce()`, was also invoked by the other higher-ranked phases to modify the input program's IR. We are currently working on improving and refining our ranking algorithm.

There are roughly 30 optimization phases in TurboFan (the exact number differs between different V8 versions). Out of these 30 or so phases, our tool is able to isolate just a small number as being the potentially buggy ones; and in four out of the five bug reports we considered, our tool accurately lists the actual buggy phase at the top of its list of candidates.

As noted earlier, TurboFan is a large, complex, and mature software system. To provide some context for the accuracy numbers from the previous section, it is useful to consider the size of the code under consideration. The complexity of the code base makes it nontrivial to give static line counts for the source code. Instead, Table 3 gives dynamic instruction counts from the execution traces we collected. The columns in this table are as follows:

- **Sum**: the total number of instructions executed over all optimization phases.
- **Max**: the maximum number of instructions executed by any single phase across all optimization phases.
- **Med**: the median number of instructions executed by any single phase
- **Min**: the minimum number of instructions executed by any single phase.
- **Phases**: the total number of optimization phases executed during the JIT compilation.

It can be seen that while the minimum instruction counts are small, the median instruction count for the optimization phases ranges from 35K to 57K instructions. Overall, the optimization phases in the JIT compiler incur between 3.2M and 8.6M instructions. Additionally, although not all optimization phases apply optimizations to the IR nodes, all the phases get executed to evaluate the node to decide whether the evaluated node requires optimization or not. And, these executed, but did not perform optimization, phases still take large portions of the instructions. Moreover, it is not clear which optimization phases were executed to actually optimize the IR nodes until we analyze the IR, which our implementation identifies all the executed phases (approx. 30-ish) and identifies only those that actually performed optimizations to the IR nodes (approx. 9 to 12-ish) and narrow down to the potentially buggy phases (approx. 1 to 5-ish).

6 Discussion

As the evaluation results from the previous section indicate, our algorithm is effective in localizing JIT compiler bugs

Table 3. Size of Phases per V8 Bug Report PoC(dynamic instruction count)

	Dynamic instruction counts				
Report#	**Sum**	**Max**	**Med**	**Min**	**Phases**
5129	5,628,850	269,335	35,393	950	29
8056	5,456,457	326,382	30,944	516	30
791245	3,237,583	406,720	57,374	696	33
961237	5,591,942	271,468	35,687	401	33
1072172	8,650,393	383,706	53,329	1380	34

down to the level of individual optimization phases. It would be desirable, however, to be able to further narrow the possible bug location, e.g., down to the function level. This is a limitation of our algorithm that we are currently working on improving.

6.1 Ambiguity in Function Calls

There are two main reasons our algorithm is currently unable to localize bugs to the level of individual functions. The first is that, while we are able to identify clearly the entry and exit points for optimization phases, call-return relationships between functions can sometimes be tricky to resolve. For example, GCC's *sibling call optimization* (which is enabled by default at optimization levels -O2 and higher) can replace some function calls with jumps, where control does not come back to the originating function. Additionally, assigning function-level blame for IR node modifications can be tricky. For example, suppose we have the following function call chain that results in the buggy modification of an IR node within the function $h()$:

$$f() \rightarrow g() \rightarrow h()$$

In this case, the bug may be that $g()$ incorrectly calls $h()$, but it is also possible that the problem really is in the function $f()$.

Moreover, further narrowing down the buggy location can also be done by learning more about the node properties. So far, we are only identifying the properties of a node by the function names. V8 has some specific functions that access node to add/remove/modify the properties. We seek these functions and identify which node that it's accessing to add/remove/modify the node properties. However, some node properties can be modified directly without calling the modifier functions, which we are facing difficulties to capture the pattern in the trace instructions. Thus, we are continuing our research to come up with a solution that can capture the node properties in general to solve this problem.

6.2 Scope of the Current Approach

Additionally, our approach has a scope where the bug resides in the optimization phase functions that optimize the IR nodes. As mentioned in the evaluation section, where it discusses how the bug reports were selected, our approach has limitations in localizing the bug in the JIT compiler where it does not generate or modify the IR node. More explicitly, for example, if the bug is in the JIT compiler code where it generates some faulty object that is not related to the IR node, but will be used in the optimized code, then our tool won't be able to recognize such bug. Therefore, we are currently investigating improving our approach to recognizing all declared objects not only the IR nodes, which are in fact just special kind of objects, and analyze them. This can, possibly, be done by recognizing the patterns of memory allocation and manipulations for objects in the low-level instructions.

6.3 Assumption in the Correct Execution

Finally, our approach currently assumes that at least one modified PoC will execute correctly. While this was true for the experiments described, it cannot be guaranteed in general. Possible solutions to this include allowing more than one edit operation to the AST of the original PoC (our tool currently limits itself to a single edit). This is a problem we are currently investigating.

7 Related Work

There is a considerable body of work on automated bug localization, which we summarize below. To the best of our knowledge, none of this work considers code that is dynamically generated, as in the case of JIT compilers, and so is inapplicable to the problem we address in this paper. The issue with JIT compiler bugs is that they result in the generation of incorrect code that causes the application being optimized to crash or compute incorrect results. It seems to us that, in order to effectively localize bugs in such systems, the bug localization system needs to model the causal dependencies between the data manipulated by the JIT compiler (e.g., the program IR being optimized) and the execution behavior of the resulting dynamically generated application code. Existing approaches to automated bug localization do not do this.

Automated bug localization approaches can be broadly classified as either static or dynamic. Static bug localization approaches typically use information retrieval techniques [29, 35, 43]. We are not aware of applications of static bug localization to software systems such as JIT compilers that generate code during execution. Dynamic bug localization techniques, by contrast, use the dynamic analysis to monitor the execution behavior of the program on buggy and non-buggy inputs [8, 15, 20, 21, 23–26]. The work described in this paper falls in the latter category. More recently, there has been a great deal of interest in the application of machine learning techniques to automatic bug localization [22, 32, 33, 41, 42]. As noted above, these works do not consider systems that generate code during program execution.

Research on debugging optimized code has been carried out by a number of researchers [2, 3, 17], but to the best of our knowledge all of this works are to debug an optimized code at source-level. The approaches include mapping the binary to source-level, modifying the compiler to produce more information about the optimizer, or deoptimizing the optimized code to retrieve the source-level code, etc. However, these approaches are not very suitable when they have to debug optimized code that was generated by JIT compilers, which compiles byte-code, as they won't be able to retrieve the source-level code from the optimized code. Instead, they will have to figure out the mappings between the optimized code to byte-code, then again mapping the byte-code to source-level, which such an approach is not implemented in any of the papers.

Tice and Graham proposed another method of debugging the optimized codes. Instead of directly mapping the optimized code, which is in binary, to the original source code, it generates a new source code that represents optimized code[39]. Nonetheless, this approach, again, shows the limitation as (1) JIT compilers does not generate code from the source code, but from byte-codes, so it won't be able to regenerate the source code that represents the optimized code and (2) the complexity of optimization has increased hugely since the paper was written.

8 Conclusion

Many widely-deployed modern programming systems use just-in-time compilers to improve performance. However, we are not aware of any existing automated systems to automatically locate JIT compiler bugs in a large and complex JIT-based system. This paper discusses how this problem can be addressed by automatically capturing the patterns of the JIT compiler's optimization phases and the intermediate representations that it generates and optimizes as well as analyzing them by comparing the captured IRs and rank them in the order of most likely location to least likely locations for the bug.

Although there are plenty of spaces for improving our algorithm and the implementation to more precisely localizing the bug, our experiments with a prototype implementation on a number of real-world examples show that re-generating JIT compiler's IRs and analyzing them to rank the optimization phases led to localizing the bug to a smaller part of the system.

Acknowledgments

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

References

[1] Mohammed Aboullaite. 2017. *Understanding JIT compiler (just-in-time compiler).* https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/

[2] Ali-Reza Adl-Tabatabai and Thomas Gross. 1996. Source-Level Debugging of Scalar Optimized Code. In *SIGPLAN Notices.* 33–43.

[3] Gary Brooks, Glibert J. Hansen, and Steve Simmons. 1992. A new approach to debugging optimized code. *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (1992), 1–11.

[4] bugs.chromium.org. 2016. Issue 5129: Turbofan changes $x - y < 0$ to $x < y$ which is not equivalent when $(x - y)$ overflows. https://bugs.chromium.org/p/v8/issues/detail?id=5129

[5] bugs.chromium.org. 2018. Issue 8056: [turbofan] Optimized Array#indexOf and Array#includes ignore a prototype that is not initial. https://bugs.chromium.org/p/v8/issues/detail?id=8056

[6] bugs.chromium.org. 2019. Issue 961237: Security: jit difference on comparison in d8. https://bugs.chromium.org/p/chromium/issues/detail?id=961237

[7] bugs.chromium.org. 2020. Issue 1072171: Security: missing the -0 case when intersecting and computing the Type::Range in NumberMax. https://bugs.chromium.org/p/chromium/issues/detail?id=1072171

[8] 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.

[9] Adam C. Conrad. 2018. *How JavaScript Engines Work.* https://adamconrad.dev/blog/how-javascript-engines-work

[10] Max Copperman. 1992. Debugging Optimized Code Without Being Misled.

[11] Apple developers. [n.d.]. *JavaScriptCore.* https://developer.apple.com/documentation/javascriptcore

[12] Google V8 developers. [n.d.]. *V8.* https://v8.dev/

[13] Microsoft Chakra developers. [n.d.]. *ChakraCore.* https://github.com/chakra-core/ChakraCore

[14] JS Foundation. [n.d.]. esprima-python. https://github.com/Kronuz/esprima-python

[15] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* 263–272.

[16] Franziska Hinkelmann. 2017. *Understanding V8's Bytecode.* https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775

[17] Urs Holzle. 1992. Debugging optimized code with dynamic deoptimization. *ACM Sigplan Notices* 27 (1992). Issue 7.

[18] Fedor Indutny. 2015. *Sea of Nodes.* https://darksi.de/d.sea-of-nodes/

[19] Intel Corp. [n.d.]. Intel XED. `https://intelxed.github.io`.

[20] 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.

[21] 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.

[22] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC).* IEEE, 218–229.

[23] Benjamin Liblit. 2004. *Cooperative Bug Isolation*. Ph.D. Dissertation. University of California, Berkeley.
[24] 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.
[25] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *Acm Sigplan Notices* 40, 6 (2005), 15–26.
[26] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 286–295.
[27] lokihardt. 2017. *Issue 791245: Security: V8: JIT: Simplified-lowererer IrOpcode::kStoreField, IrOpcode::kStoreElement optimization bug*. https://bugs.chromium.org/p/chromium/issues/detail?id=791245
[28] 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.
[29] Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2010. Bug localization using latent Dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972 – 990.
[30] Benedikt Meurer. 2017. *An Introduction to Speculative Optimization in V8*. https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8
[31] Mozilla. [n.d.]. *SpiderMonkey: The Mozilla JavaScript runtime*. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
[32] Zhendong Peng, Xi Xiao, Guangwu Hu, Arun Kumar Sangaiah, Mohammed Atiquzzaman, and Shutao Xia. 2020. ABFL: An autoencoder based practical approach for software fault localization. *Information Sciences* 510 (2020), 108–121.
[33] Sravya Polisetty, Andriy Miranskyy, and Ayşe Başar. 2019. On Usefulness of the Deep-Learning-Based Bug Localization Models to Practitioners. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. 16–25.
[34] Jordan Rabet. 2017. Browser security beyond sandboxing. Microsoft Windows Defender Research. `https://cloudblogs.microsoft.com/microsoftsecure/2017/10/18/browser-security-beyond-sandboxing`.
[35] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 345–355.
[36] Jaroslav Sevcik. 2016. *Deoptimization in V8*. https://docs.google.com/presentation/d/1Z6oCocRASCfTqGq1GCo1jbULDGS-w-nzxkbVF7Up0u0/htmlpresent
[37] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes*. Elsevier.
[38] Yusuke Suzuki. [n.d.]. Edcodegen. https://github.com/estools/escodegen
[39] Caroline Tice and Susan L. Graham. 1998. OPTVIEW: A New Approach for Examining Optimized Code. In *Proceedings of ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*. ACM.
[40] Ben L. Titzer. 2015. Digging into the TurboFan JIT. `https://v8.dev/blog/turbofan-jit`.
[41] Geunseok Yang, Kyeongsic Min, and Byungjeong Lee. 2020. Applying deep learning algorithm to automatic bug localization and repair. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20)*. 1634–1641.
[42] Sai Zhang and Congle Zhang. 2014. Software Bug Localization with Markov Logic. In *Companion Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, 424–427. https://doi.org/10.1145/2591062.2591099
[43] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.