Modeling Code Manipulation in JIT Compilers

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

Xiyu Kang
Department of Computer Science
The University Of Arizona
Tucson, AZ 85721, USA
kangxiyu@email.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 widely used to improve the performance of interpreter-based language implementations by creating optimized code at runtime. However, bugs in the JIT compiler's code manipulation and optimization can result in the generation of incorrect code. Such bugs can be difficult to diagnose and fix, and can result in exploitable vulnerabilities. Unfortunately, existing approaches to automatic bug localization do not carry over well to such bugs. This paper discusses a different approach to analyzing JIT compiler optimization behaviors, based on using dynamic analysis to construct abstract models of the JIT compiler's optimizer and back end. By comparing the models obtained for buggy and non-buggy executions of the JIT compiler, we can pinpoint the components of the JIT compiler's internal representation that have been affected by the bug; this can then be mapped back to identify the buggy code. Our experiments with two real bugs for Google V8 JIT compiler, TurboFan, show the utility and practicality of our approach.

CCS Concepts: • **Theory of computation → Program analysis**.

Keywords: program analysis, jit compiler, optimization, dynamic code generation

ACM Reference Format:
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 '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3520313.3534656

1 Introduction

JIT compilers are ubiquitous in today's world and appear within a wide range of software systems, ranging from widely used applications such as web browsers [14, 16, 18] to specialized performance-critical code in OS kernels [30]. JIT compiler systems are typically large and complex, and often incur rapid code change—a combination that makes them fertile ground for bugs. Particularly challenging, in this context, are dynamic code generation bugs, which cause the JIT compiler to silently emit incorrect code, resulting in incorrect execution behavior in the application being optimized. Such bugs can be difficult to diagnose and correct, for two reasons: first, the program that exhibits incorrect behavior (the application being optimized) is not the program that contains the bug (the JIT compiler); and second, the code that manifests the incorrect behavior is transiently generated at runtime and is not available for static inspection. They also have significant security implications: e.g., an alias analysis bug in Mozilla's IonMonkey JIT compiler results in the erroneous elimination of an array bounds check (CVE-2019-17026) [31], and a bug in the BPF JIT compiler in the Linux kernel results in incorrect branch displacement computations (CVE-2021-29154) [22], in each case resulting in vulnerabilities that can be exploited to achieve arbitrary code execution. The difficulty in diagnosing such bugs, combined with their security implications, makes it important to provide tool support for reasoning about JIT compiler optimization processes.

Unfortunately, existing approaches to automated bug localization [4–6, 8, 20, 21, 23] do not carry over to such bugs. The problem is that these approaches are *application-agnostic*, i.e., do not have any higher-level characterization of what the applications under consideration do. A common approach is to identify differences between the machine-level state sequences observed in "good" and "bad" executions, map these differences to code locations, then use a ranking function to determine their likelihood of being the cause of the bug; code that causes differences earlier in execution are typically ranked higher [3]. This approach, while very general, unfortunately does not carry over well to JIT compilers, whose inputs must first be processed by the interpreter front end (parser, bytecode generator, etc.). The problem is that, given two different input programs—one that triggers a JIT-compiler bug and one that does not—their source-code differences necessarily give rise to execution differences in the interpreter front end that, because they occur early in execution, are ranked higher by the bug localizer. For example, in our experiments with a state-of-the-art bug localization system [3], the root cause of a JIT compiler dynamic code

SOAP '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9274-7/22/06.
https://doi.org/10.1145/3520313.3534656

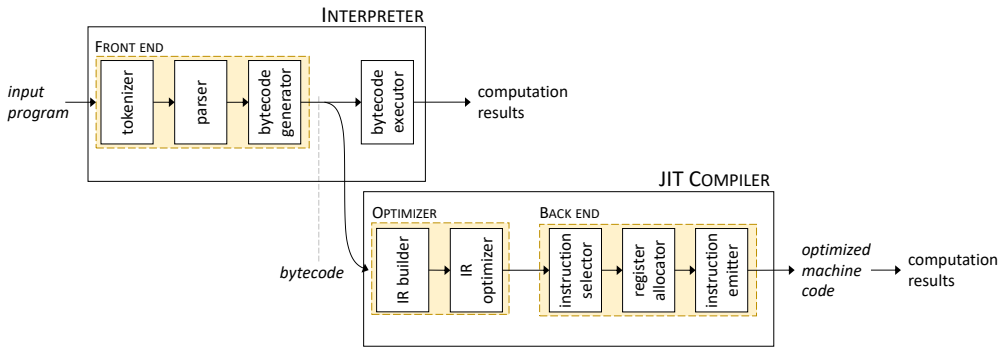

Figure 1. Interpreter/JIT Compiler Systems: Organization and structure

generation bug was erroneously identified as occurring in the parser in the interpreter front end.

Instead, we take a very different approach. The problem we are concerned with is: given a "proof of concept" (PoC) input P that triggers a dynamic code generation bug in the JIT compiler (such PoCs are typically provided alongside bug reports), use program analysis techniques to identify potentially incorrectly optimized components of the JIT compiler's representation of the input program. We use directed fuzzing to create a set of variants of P and determine which of them result in buggy JIT compiler executions and which do not.[1] For each of these input programs, we use dynamic analysis to obtain an instruction-level execution trace of the JIT compiler, from which we construct an abstract model of how it manipulates its representation(s) of that program. We then compare these abstract models for buggy and non-buggy JIT compiler executions to determine how they differ, and use these differences to identify structures in the JIT compiler's program representation that have potentially incorrect values. This allows us to pinpoint specific components of the program representation that may be problematic: e.g., a particular intermediate representation node where an edge was incorrectly modified during optimization, or an instance of an instruction data structure in the JIT compiler back end where an operand may have been improperly encoded. This information can then be used to locate the buggy code.

This paper makes the following technical contributions: (1) it describes how the program representations manipulated by JIT compilers can be modeled in a general way; and (2) it discusses how the resulting abstract models can be used for automatic identification of witnesses to dynamic code generation bugs, i.e., specific components of the JIT compiler's representation of the input program that have incorrect values. Preliminary results from a prototype implementation are encouraging and suggest that this approach may be helpful in dealing with such bugs. Our prototype currently targets three JavaScript JIT compilers: TurboFan, (V8), DFG (JavaScriptCore), and IonMonkey (Spidermonkey); due to space constraints we present results for V8.

[1]For our purposes, a JIT compiler execution is non-buggy if the input program's behavior is the same with and without JIT compilation, i.e., if the JIT-compiled code has the same behavior as interpreted code. The JIT compiler's execution is considered buggy if program's behavior with JIT compilation differs from its behavior when it is interpreted.

2 Background

Figure 1 shows the conceptual structure of a typical interpreter/JIT compiler system. The input program is read by the interpreter front end, converted to bytecode, and executed by the interpreter. If a section of bytecode is executed a large number of times, the JIT compiler is invoked and optimizes the bytecode to generate optimized machine code.

JIT compilers typically convert the bytecode generated by the interpreter into a graph-structured intermediate representation (IR) that is used for optimization and translation to machine code. E.g., the TurboFan JIT compiler for Google's V8 JavaScript engine uses a sea-of-nodes representation, where nodes represent operations, control flow, types, and state; and edges represent control flow, data flow, and effect dependencies [19, 28]. The optimized IR is converted it into a different internal representation via *lowering*. E.g., the resulting used in V8's instruction selector, called an `Instruction`, is a data structure that represents a machine code instruction. The `Instruction` data structure has a number of different fields; when modeling the back end we focus on two of these fields: `opcode` and `operands`.

3 An Overviw of Our Approach

We use dynamic analysis using Intel's Pin Tool [25] to collect an execution trace of a JIT compiler, i.e., a sequence of dynamic instances of machine instructions obtained on a particular execution of the input program (including JIT compiler invocations occuring during execution). While analyzing each instruction in a given trace, we extract the following information to model the JIT compiler optimizer and back end `Instruction` Selector:

- We use symbol table information in the JIT compiler executable to map each instruction in the trace to the function it belongs to. Thus, given the names of the JIT compiler's IR node allocation function(s), we can identify entry into and return from these allocation functions, and thereby determine the address and size of the allocated node.
- Given a (direct or indirect) function call instruction in the trace, we collect the return value of the call. The location of the return value can be obtained from the system's application-binary interface (ABI), e.g., on an x86-64 system, an address is returned in the `rax` register.
- For each instruction in the trace, we collect information about (*a*) the registers or memory locations it reads together with the values read; and (*b*) the registers or memory locations it writes together with the values written. We use the read/write information to model the JIT compiler's creation, modification, and manipulation of IR nodes and `Instruction` objects in the course of execution.

To enhance portability, we push system-specific aspects of trace analysis, in particular the functionality to identify the address, opcode, and size of IR node, into a library that is accessed through a system-independent API, so that the remaining logic of model construction and analysis can be kept system-independent. At this time, we have successfully used this approach for system-independent modeling of IR optimization for three different JIT compilers, namely: TurboFan [13], DFG JIT [12], and IonMonkey [14]. Due to space constraints, this paper provides experimental data for a V8 TurboFan optimization bug. We have not had time to apply this approach to modeling JIT compiler back end structures.

This approach is conceptually simple, does not require a great deal of knowledge about the JIT compiler system beyond recognizing the functions that create the objects manipulated by the JIT compiler, and is portable across different JIT compiler implementations. However, collecting and analyzing such traces can be expensive in both space and time. An alternative would be to instrument the JIT compiler to only emit information about specific events of interest. The latter approach would likely incur less runtime overhead, but has the following disadvantages:

1. It would require deeper knowledge of what to instrument in the JIT compiler as well as more manual effort.
2. It would potentially be more brittle in handling changes to the JIT compiler's code.
3. It would not be portable across different JIT compilers.

4 Model Representations

Although a JIT compiler is a combination of several different components, such as the parser, analyzer, optimizer, machine code generator, etc, we focus on two components that play an important in code manipulation in the JIT compiler. In this section, we describe our abstract models of the code representations used by the JIT compiler for manipulating the input program during optimization and code generation. We focus, in particular, on two components of the JIT compiler: (1) the IR optimizer, which is responsible for converting bytecode into a system-independent intermediate representation (IR) and optimizing it; and (2) system-specific native code generator, which is the back end of the JIT compiler.

4.1 Optimizer - Intermediate Representation

The JIT compiler parses its input, a sequence of bytecode instructions, and builds an initial unoptimized IR. The IR optimizer ("optimizer" for short) then performs a variety of machine-independent optimizations on the resulting IR. Different JIT compilers may differ in the details of their optimization processes, both in terms of the optimizations used and the particulars of how they are implemented.

In order to characterize and reason about the optimization process within the JIT compiler, we build an abstract model of the IR manipulated by the optimizer, as discussed below.

4.1.1 IR Graphs. We extract information about the IR nodes and edges manipulated by the JIT compiler during an execution by examining the instructions in its execution trace. We assume that we know the names of the JIT compiler's IR node allocation function(s).[2] We can therefore identify the instructions that enter and return from these allocation functions, from which we can determine the address and size of the allocated node; by examining values assigned to fields within the node we can determine the node type (i.e., an operation such as *mult* or *div*, or a data type such as *int*). Given an IR node v_i of type t_i and size s_i at address a_i, the corresponding "abstract IR node" is obtained as $\alpha(v_i) = (i, t_i, s_i, a_i)$. The index i refers to the order of node creation during optimization. The size s_i and address a_i allow us to keep track of operations performed on the node during optimization, including addition/deletion of edges.

Given an edge $e \equiv (u, v)$ in the concrete IR graph manipulated by the JIT compiler, the corresponding "abstract edge" is $\alpha(e) = (\alpha(u), \alpha(v))$.

4.1.2 IR Graph Transformations. The optimization process in a JIT compiler on a given input can be thought of as a sequence of IR graph modifications of the form

$$G_0 \xrightarrow{(v_{i_0},\tau_0)} G_1 \xrightarrow{(v_{i_1},\tau_1)} \cdots \xrightarrow{(v_{i_{n-1}},\tau_{n-1})} G_n$$

where each G_i is an IR graph and $G_j \xrightarrow{(v_{i_j},\tau_j)} G_{j+1}$ denotes that a transformation[3] τ_j is applied to $G_j \equiv (V_j, E_j)$ at node $v_{i_j} \in V_j$ to obtain the graph G_{j+1}. In practice the JIT compiler does not construct a sequence of different IR graphs

[2] Identifying and extracting such functions from the source code is typically straightforward: e.g., the functions can be found in *node.h* for TurboFan, *DFGNode.h* for DFG JIT, and *MIR.h* for IonMonkey.

[3] For our purposes, the transformations τ_i are low-level changes to the graph, i.e., addition, deletion, or modification of nodes or edges, rather than high-level optimization transformations such as loop unrolling or inlining.

$G_0, \ldots, G_n$, but successively transforms a single data structure, by adding, deleting, or modifying nodes and/or edges, until the final "fully optimized" IR graph G_n is obtained. To avoid confusion, we refer to the IR graphs manipulated by the JIT compiler as *concrete IR graphs*.

In order to model such optimization behaviors of the JIT compiler for bug localization purposes, we summarize the changes occurring in the concrete IR graphs using an *abstract IR graph*. A key consideration here is the handling of node and edge deletions. Suppose that a node (edge) is deleted from a concrete IR graph at some point in the optimization process. We cannot simply delete the corresponding node (edge) in the abstract IR graph, since it would be lost to subsequent reasoning if it were. Instead, we retain the corresponding node (edge) in the abstract IR graph, but flag it as "removed."

Thus, given a concrete IR graph transformation sequence

$$G_0 \xrightarrow{(v_{i_0},\tau_0)} G_1 \xrightarrow{(v_{i_1},\tau_1)} \cdots \xrightarrow{(v_{i_{n-1}},\tau_{n-1})} G_n$$

where $G_i = (V_i, E_i)$ is a concrete IR graph, we model this using an *abstract IR graph* $(\widehat{V}, \widehat{E}, \widehat{H})$ where $\widehat{V}$ is a set of abstract nodes, $\widehat{E}$ is a set of abstract edges, and $\widehat{H}$ is a *transformation history*. The set of nodes $\widehat{V}$ and edges $\widehat{E}$ in the abstract IR graph capture all of the nodes and edges in the concrete IR graphs encountered during optimization:

$$\widehat{V} = \bigcup\{\alpha(v) \mid v \in V_i, 0 \le i \le n\}$$
$$\widehat{E} = \bigcup\{\alpha(e) \mid e \in E_i, 0 \le i \le n\}$$

The transformation history $\widehat{H}$ specifies the location and nature of graph transformations applied during optimization. We consider three kinds of graph modifications, denoted as **Mods** = {*addition*, *removal*, *replacement*}; since we are interested in bug localization, with each transformation we record the JIT compiler source-code function that performs that transformation (this can be deduced by using the symbol table of the JIT compiler binary to map instructions in the execution trace to source-level function names). The transformation history $\widehat{H}$ is a sequence of tuples of the form

$$\widehat{H} = \langle (v_0, m_0, f_0), (v_1, m_1, f_1), \ldots, (v_n, m_n, f_n) \rangle.$$

where $v_i \in \widehat{V}$ is the vertex in the abstract IR graph that is transformed at step i; $m_i \in$ **Mods** specifies the nature of the transformation applied at that step; and f_i is the name of the JIT compiler function that performs the modification.

4.2 Back End Instruction Selector - Instruction

The instruction selector in the JIT compiler back end uses the `Instruction` data structure to represent the optimized machine instructions to be generated. Our analysis examines the JIT compiler's execution trace and extracts the following information for each allocated instance of this data structure:

- Its *address*.
- The *opcode* of the instruction represented by this instance.

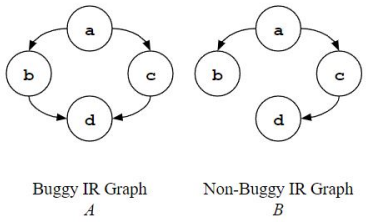

Figure 2. Buggy IR (left) vs. Non-Buggy IR (right)

- The *operands* of that instruction. This field in the source code is a dynamically sized array. Each element in the array is an 8-byte long operand encoding. Each operand encoding describes attributes of the operand. As an example, one of the attributes indicates whether the operand will use a register or a stack slot. Since this field is a dynamically sized array, we do not know its length. We currently consider only the first two elements of this array.
- *State changes* of the opcode and operands fields. Each field has its own sequence of state changes. For each field, let $S = (f, v)$ be the state of the field, which means the value v is written to the field by a function f. The state of the field is changed when a new function writes a new value to the field. When such a change happens, we record the new state of the field. Let S_i be the i-th state of the field. So the state changes of the field is $\{S_1, S_2, ..., S_n\}$.

Let *Instr* to denote the information we extract for each instance. Our model is defined as a sequence $\{Instr_1, ..., Instr_n\}$, where n is the number of allocated instances.

5 Applications of Models

One of the useful applications of the abstract models discussed above is in identifying the bugs of the JIT compiler which can happen during the code manipulation, i.e., optimization and instruction selection. Our models hold enough information about the result of the operation, e.g., the structure of an IR node after edge replacement graph modification or the operand of `Instruction` instance after encoding, etc., which can be used in the analysis to identify the buggy IR nodes and buggy `Instruction` instances. The buggy structures so identified can then be mapped back to the code that manipulated them to identify the actual buggy functions in the JIT compiler source code. For the purposes of this paper, we focus on applying our abstract models to identify the buggy IR nodes and buggy `Instruction` instances.

To identify buggy IR nodes and `Instruction` instances, we compare the buggy and non-buggy models to find the differences between them. For example, in the case of optimizer IR graph, we compare the nodes from two IR graphs to determine the nodes in the buggy IR graph that shows

the difference in any one of *metadata*, *structure*, and the *optimization*. The nodes found to be different are the candidates to be selected as the buggy IR nodes.

For example, in Figure 2, let A be the modeled buggy IR and B be the modeled non-buggy IR. While nodes a, c, and d are the same, the node b in the buggy IR graph has an extra edge to the node d whereas the non-buggy IR graph does not. This indicates that (1) this difference makes one IR graph buggy while the other non-buggy; (2) the bug is likely to be in the code that created the extra edge from b to d.

Note that, in the reality, the models and comparisons are much more complicated than the given example. We provide specific examples of both optimizer IR and back end `Instruction` instance model applications in Section 6.

6 Evaluation

To evaluate our ideas, we built two prototype tools, one that models the IR optimizer and a (partially implemented) tool that models the back end. Our experiments were run on a machine with 32 cores (@ 3.30 GHz) and 1TB of RAM, running Ubuntu 20.04.1 LTS. A dynamic analysis tool built using Intel's Pin software (version 3.7) [25] is used for program instrumentation and instruction-level execution trace collection; XED version 8.20.0 [7] is used to decode instructions.

Our prototypes target the JIT compiler, TurboFan, used in JavaScript engines Google Chrome V8 [16] and Node.js [15] to present the results on the use cases discussed in Section 5. Specifically, (1) identify and rank the IR nodes that are suspicious to be buggy, and (2) identify potentially buggy instances of Instruction data structure. In our experiments, we took the following procedures:

1. Search and retrieve the separate bug reports for optimizer and back end that are marked as fixed from Google Chrome's bug report community [17].
2. Get the proof-of-concept (PoC) codes from the reports. Use our fuzzer to generate a set of variants of the PoC.
3. Run our prototype tools on these PoC variants to identify a candidate set of buggy instances of JIT compiler data structures.
4. Check that the results from the prototypes are correct. We do this by checking that the result returned by our tool matches that targeted by the fix in the source code.

6.1 Optimizer Buggy IR Identification

Bug issue 5129 [9] was reported in June 2016 explaining that the V8 JIT compiler version 8.3.1 incorrectly optimizes the nodes for *subtract* and *less-than* operations: e.g., given the expression `x - y < 0`, the optimizer transforms the IR graph to generate `x < y` expression instead. Mathematically, the two expressions are equivalent, which makes the conversion seems to be reasonable. However, according to the developers, this can cause an overflow resulting to wrong evaluation, e.g., *true* for `x - y < 0`, but *false* once converted to `x < y`. The fix was made in the `MachineOperatorReducer::Reduce` optimizer function optimizing the operator nodes for *less-than* and *subtract*. Thus, our goal is to model the IR graph and analyze the model to confirm that we can identify the buggy *subtract* and *less-than* operator nodes.

For this bug, our tool modeled a total of 300 nodes with 260 different opcodes in a single IR graph. We confirmed that our model is correct by manually adding the print statements in the optimizer source code to print the generated nodes with opcode. We used the print-statements in the optimizer source code and V8's default tracing options, i.e., `–trace-opt`, to confirm that our IR modeler tool has properly modeled the optimization and constructed the model.

Table 1. Ordered List of Potentially Buggy Nodes

Order no.	Node ID	Opcode	Mnemonic
1	242	007c	NumberSubtract
2	243	006f	NumberLessThan
3	285	014c	Word32Equal

Our IR modeler analyzed both buggy and non-buggy modeled IRs to compute their differences. We used a custom fuzzer, which we built, to generate 13 additional JavaScript program inputs for this PoC, for a total of 14 input programs; of these, 4 programs trigger the JIT compiler bug and 10 do not. The results are shown in Table 1. The analyzer selected and returned only 3 nodes out of 300 nodes, namely, `NumberSubtract`, `NumberLessThan`, and `Word32Equal`, and ranked them in the order shown. Of these, `NumberSubtract` and `NumberLessThan` are the actual buggy nodes.

Our tool takes about 2 minutes to model a single IR graph for the PoC for this bug, i.e., a total of 28 minutes to model 14 IR graphs. Generating additional input programs via fuzzing, analyzing the modeled IR graphs to identify the buggy IR nodes, and ranking them like in the Table 1 takes less than a minute. Thus, starting from a single input PoC program, it takes approximately 30 minutes to get the result.

6.2 Back End Buggy Instance Identification

We evaluated the back end model on issue 9980 [10]. This bug was reported in November 2019 explaining that the V8 JIT compiler version 8.0.0. causes a crash in the generated dynamic code due to the following description.

The `InstructionSelector::VisitS8x16Shuffle` function allocates instances of `Instruction` data structure that have the opcode `pshufd`. This opcode reorganizes bytes in a specified order. Among the `Instruction` instances, there is an instance whose operand encoding encodes that the operand will be stored to a stack slot. If `pshufd` has a stack operand, the operand is required to have 16-byte memory alignment. However, V8 does not check for this memory alignment, which results in a crash. We

call such an instance a "buggy instance". We can see that the bug roots in the buggy instance which has a buggy operand encoding that eventually causes the crash. In the next V8 version, we see that the fix [1] was made in the `InstructionSelector::VisitS8x16Shuffle` function.

In our experiment, our task is to identify the buggy instance. The proof-of-concept we use is here [11]. Given a PoC retrieved from the bug report, we generated an additional 20 buggy and 17 non-buggy input programs. In summary, 21 buggy input programs and 17 non-buggy input programs.

Table 2 shows the summary of our result. Initially, in all the buggy models, there are 2143 potentially buggy instances. We found that each buggy model has 102 potentially buggy instances on average for this specific bug. We compute the intersections of buggy instances to select only the instances that commonly appear among the buggy instances. The result shows that there are only 35 potentially buggy instances out of 2143 instances. Then, we compute the difference between the intersection result and non-buggy instances to select the instances that only appear on the buggy side. We call this process *subtraction*. The result shows that the number of potentially buggy instances was reduced to 15 instances. The whole process took approximately 60 minutes.

Table 2. Number of Potentially Buggy Instances

Initial Total	Average	Intersection	Difference
2143	102	35	15

Initially, there were 102 potentially buggy instances on average. Our model narrowed this down to 15, and one of them is the actual buggy instance. Furthermore, we can tell which operand and which state of the operand is possibly buggy. If we compare the potentially buggy instances with the non-buggy instances, we can tell which operands and states are different. Those operands and states are possibly buggy in the potentially buggy instances.

We have not fully implemented modeling the back end at this time. However, our initial results are encouraging, and we plan to continue work on modeling the back end.

7 Future Work

As our evaluation shows, our approach is capable of identifying the buggy IR nodes and `Instruction` instances via using dynamic analysis on execution traces of Google V8's JIT compiler optimizer and back end. Nevertheless, our idea was experimented only with JIT compilers for JavaScript language. Therefore, we plan to experiment with other language JIT compilers, e.g., HHVM JIT and/or PHP 8 JIT, etc., to show that the idea is generalizable. Additionally, we aim to model other internal components of the JIT compiler, e.g., register allocator, etc, and improve the approach to handle more types of bugs, e.g., performance bugs that do not show clear distinct behavior between the normal and abnormal execution.

8 Related Work

Lim and Debray [24] discuss bug localization in the TurboFan JIT compiler for Google's V8 JavaScript engine. The work is considerably less precise than ours, in that bug localization is done at the granularity of optimization phases, and only for a limited class of JIT compiler bugs. It also requires detailed symbol information for the code implementing each optimization phase, which makes generalizing to other JIT compilers difficult.

The work on formal verification of JIT compilers [2, 26, 27] has the laudable advantage of proving the correctness of the verified JIT compiler. However, the size and complexity of real-world JIT compilers, combined with the fact that they are typically not written with verification in mind, make its application to real-world JIT compilers challenging.

Static approaches used in automated bug localization [29, 32] typically uses the information retrieval technique. These approaches require source code information in their analysis. There are two main problems using the approach in JIT compilers: (1) JIT compilers generate code at run-time and execution behavior tend to change per execution; (2) JIT compilers, which generally is a part of larger system, tend to share many functions with other components, e.g., interpreter. Thus, source code information used in the analysis might not necessarily belong to the JIT compiler execution but, for example, the interpreter.

The dynamic analysis approach used in the statistical debugging [5, 23] targets to instrument specific types of predicates, which are then analyzed and ranked. The approach assumes that the program will always take the same execution path that identifying the predicates appear the most within the buggy programs' execution paths gives the idea of where the bug is located in the source code. Nonetheless, the JIT compiler's execution paths are inconsistent.

9 Conclusion

In this paper, we present a new approach in analyzing the execution behaviors of JIT compilers by modeling the intermediate representation, which the optimizer builds and optimizes, and the instruction selector, which the back end builds from the optimized IR to generate native code. Our experiments on the real bugs show that our models are useful in bug localization. Nevertheless, our approach was tested on JIT compilers for JavaScript language only. Thus, we plan to test on other language JIT compilers, e.g., PHP.

Acknowledgments

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

References

[1] Ng Zhi An. 2019. Issue 9980 Fix. https://chromium.googlesource.com/v8/v8.git/+/d9feec111268d796d46b3e48511ba11738006dc8%5E%21/#F0.

[2] Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2020. Towards Formally Verified Just-in-Time Compilation. In *Proc. Sixth International Workshop on Coq for Programming Languages (CoqPL'20)*.

[3] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *Proceedings of the 29th USENIX Security Symposium*. 235–252.

[4] 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.

[5] 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.

[6] Max Copperman. 1994. Debugging Optimized Code without Being Misled. 16, 3 (1994). https://doi.org/10.1145/177492.177517

[7] Intel Corp. 2019. Intel XED. https://intelxed.github.io. Accessed 2020-08-23.

[8] D. S. Coutant, S. Meloy, and M. Ruscetta. 1988. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. 125–134. https://doi.org/10.1145/53990.54003

[9] V8 Developer. 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

[10] V8 Developer. 2019. Issue 9980. https://bugs.chromium.org/p/v8/issues/detail?id=9980&q=9980&can=2.

[11] V8 Developer. 2019. Proof-of-Concept of Issue 9980. https://bugs.chromium.org/p/v8/issues/attachmentText?aid=422581.

[12] WebKit Developers. 2014. DFG JIT. https://trac.webkit.org/browser/trunk/Source/JavaScriptCore/dfg. Accessed 2022-04-23.

[13] Jeremy Fetiveau. 2019. Introduction to TurboFan. https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/. Accessed 2022-01-22.

[14] Mozilla Foundation. 2016. IonMonkey/MIR. https://wiki.mozilla.org/IonMonkey/MIR. Accessed 2022-01-22.

[15] OpenJS Foundation. 2009. node.js. https://github.com/nodejs/node. Accessed 2022-02-18.

[16] Google. 2008. v8 JavaScript Engine. https://v8.dev/. Accessed 2022-01-18.

[17] Google. n.d. Google Chromium Bug Report Community. https://bugs.chromium.org/p/v8/issues/list. Accessed 2022-01-15.

[18] Apple Inc. 2014. JavaScriptCore DFG Source Code. https://trac.webkit.org/browser/trunk/Source/JavaScriptCore/dfg. Accessed 2022-01-22.

[19] Fedor Indutny. 2015. *Sea of Nodes*. Accessed 2022-02-22.

[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] Piotr Krysiuk. 2021. [CVE-2021-29154] Linux kernel incorrect computation of branch displacements in BPF JIT compiler can be abused to execute arbitrary code in Kernel mode. https://www.openwall.com/lists/oss-security/2021/04/08/1.

[23] 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.

[24] 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

[25] 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.

[26] Magnus O Myreen. 2010. Verified just-in-time compiler on x86. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 107–118.

[27] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 41–61.

[28] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, Saul Wold (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/jvm01/paleczny.html

[29] 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.

[30] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. 2018. Performance Implications of Packet Filtering with Linux eBPF. In *2018 30th International Teletraffic Congress (ITC 30)*, Vol. 01. 209–217. https://doi.org/10.1109/ITC30.2018.00039

[31] Max Van Amerongen. 2020. Exploiting CVE-2019-17026 - A Firefox JIT Bug. https://labs.f-secure.com/blog/exploiting-cve-2019-17026-a-firefox-jit-bug/.

[32] 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.