Representing and Reasoning about Dynamic Code

Jesse Bartels
Department of Computer Science
The University Of Arizona
Tucson, AZ 85721, USA
jessebartels@cs.arizona.edu

Jon Stephens
Department of Computer Science
University Of Texas
Austin, TX 78712, USA
jon@cs.utexas.edu

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

ABSTRACT

Dynamic code, i.e., code that is created or modified at runtime, is ubiquitous in today's world. The behavior of dynamic code can depend on the logic of the dynamic code generator in subtle and non-obvious ways, e.g., JIT compiler bugs can lead to exploitable vulnerabilities in the resulting JIT-compiled code. Existing approaches to program analysis do not provide adequate support for reasoning about such behavioral relationships. This paper takes a first step in addressing this problem by describing a program representation and a new notion of dependency that allows us to reason about dependency and information flow relationships between the dynamic code generator and the generated dynamic code. Experimental results show that analyses based on these concepts are able to capture properties of dynamic code that cannot be identified using traditional program analyses.

KEYWORDS

Program Analysis, Program Representations, Dynamic Code, Self-Modifying Code, Slicing

1 INTRODUCTION

Dynamic code, i.e., code that is created or modified at runtime, is ubiquitous in today's world. Such code arises in many contexts, including JIT-compilation, obfuscation, and dynamic code unpacking in malware. Dynamic code raises a host of new program analysis challenges, arising partly from the fact that the behavior of an application containing dynamic code may depend in part on logic that is not part of the application itself, but rather is in the dynamic code generator. As a concrete example, Rabet describes a JIT compiler bug in Chrome'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) [38]. As another example, Frassetto *et al.* describe how a memory corruption vulnerability can be used to modify the byte code of an interpreted program such that subsequent JIT compilation results in the creation of the malicious payload [14]. To reason about such situations, it would be helpful to be able to start from some appropriate point in the dynamically generated code and trace dependencies back, into and through the JIT compiler's code, to understand the data and control flows that influenced the JIT compiler's actions and caused the generation of the problematic code. E.g., for the CVE-2017-5121 bug mentioned above, we might want to perform automated analyses to identify which analyses/transformations within the JIT-compiler led to removal of the program's initialization code, and which data flows and control-flow logic influenced those transformations. Such analyses, which we refer to as *end-to-end analyses*, can significantly speed up the process of identifying and fixing such problems.

Unfortunately, existing approaches to (static or dynamic) program analysis do not adequately support such reasoning about dynamic code modification. Traditional program representations, such as control flow graphs, cannot handle the effects of runtime changes to the code, which require accommodating the possibility of some memory locations having different instructions at different times during execution. JIT compilers [15, 23] and dynamic binary translators [34] maintain representations of the code being dynamically modified, but not together with that of the code that performs code modification. Whole-system analyses [11, 13, 21, 53, 54] perform dynamic taint propagation, taking into account explicit information flows via data dependencies but not implicit flows via control dependencies. As we discuss later, they also do not take into account dependencies that can arise through the act of dynamic code modification. Thus, existing approaches to automated reasoning about program behaviors suffer from the following shortcomings:

(*a*) They do not provide program representations that let us answer questions such as *"Which code in the dynamic code generator affected the generation of the faulty application code?"* or *"What data flows influenced the behavior of those components of the dynamic code generator, and in what ways?"*.

(*b*) They do not support notions of dependence that can allow us to reason about the computation in ways that can help answer such questions.

This paper shows how this problem can be addressed via a program representation that is able to capture the structure and evolution of code that can change dynamically, together with a notion of dependency that arises from the process of dynamic code generation and which is not captured by conventional notions of data and control dependencies. We also discuss an optimized representation that yields significant improvements in space requirements. Experimental results show that our ideas make it possible to reason about dynamic code in novel ways, e.g., we can construct backward dynamic program slices, starting from incorrect dynamically generated JIT-compiled code, to include the JIT-compiler logic responsible for the problem; and detect situations where a dynamic

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416542

code generator embeds environmental triggers in dynamically generated code. Such end-to-end analyses are not possible using current approaches to program analysis.

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 is a software implementation of a virtual machine (VM). Programs are expressed in the VM's instruction set, with each instruction encoded as a data structure that records relevant information such as the operation, source and destination operands, etc. The computation for each operation x in the VM's instruction set is performed by a piece of code called the handler for x. The interpreter uses a virtual instruction pointer to access the VM instructions encoding the input program and a dispatch routine to transfer control to appropriate handler code.

While interpretation offers a number of benefits such as portability, it incurs a performance overhead due to the cost of instruction decoding and dispatch as well as the limited scope for code optimization resulting from the fact that the user programs executed by the interpreter are not available for analysis when the interpreter is compiled to machine code. Additionally, modern dynamic languages are often implemented using interpreters, and these incur additional overheads due to runtime type checking.

To address this problem, just-in-time (JIT) compilers are widely used alongside interpreters to improve performance by compiling selected portions of the interpreted program into (optimized) code at runtime. The general idea is to take frequently-executed portions of the program (identified via runtime profiling), apply optimizing transformations, and generate optimized machine code. These optimizations are performed at runtime, as the program is being executed, and results in code that is dynamically created or modified. Some JIT compilers support multiple levels of runtime optimization, where the dynamically created code may be subjected to additional rounds of optimization as execution progresses [45].

2.2 Control Flow Graphs

Program analyses are based on representations of the program's structure; for concreteness, we focus on control flow graphs (CFGs). CFG construction for static code via static analysis is well-understood [3]. However, this approach is inadequate for dynamic code because code created at runtime is not available for static inspection; instead, we use dynamic analysis. This has the benefit of being able to handle dynamic code; its drawback is that the constructed CFG may not contain all of the program's code due to incomplete code coverage. We sketch here how CFGs for static code can be constructed from an instruction trace obtained via dynamic analysis. The extension of this approach to dynamic code is discussed in Section 3.4.

Let G denote the CFG under construction. We process instructions in the execution trace as they are encountered. For each instruction I, its properties (e.g., whether or not it is a control transfer) and its status within G (e.g., whether or not it is already in G) determine how it is processed; we refer to this as "*processing I in the context of G.*" If I has not been encountered previously, it is added as a new instruction. If I follows a conditional or unconditional jump, it should begin a basic block: thus, if I is currently in G and is not the first instruction of its block, the block has to be split and control flow edges added appropriately.

Multi-threading introduces additional complexity because adjacent instructions in the execution trace may be from different threads and thus may not represent adjacent instructions in the code. To handle this, we require that each instruction in the trace be flagged with a value indicating the thread that executed it; we refer to this as the *thread-id* of the instruction. The CFG construction process separately maintains a summary of the state of each thread; this summary contains information such as the call stack, previous instruction seen, current function being reconstructed, etc. When constructing the CFG G, each instruction I in the trace is now processed in the context of the state summary for its thread, which is obtained from the thread-id for I. Thus, the last instruction from one thread may be appending an instruction to a basic block whereas a different thread could be splitting a different block.

3 REASONING ABOUT DYNAMIC CODE

This section discusses the concepts underlying our approach to representing and reasoning about dynamic code.

3.1 Design Goals

In devising program representations that support end-to-end analysis of dynamic code, we have the following design goals:

(1) It should be a natural and scalable generalization of existing program representations.
(2) It should provide a basis for extending existing program analyses to handle dynamic code in a natural way.
(3) It should be precise enough to distinguish between conceptually distinct dynamic code changes.

The first two goals aim to avoid reinventing the wheel as much as possible. The third is motivated by the fact that dynamic code changes can be quite complex. For example, JIT compilers typically use shared code buffers that may be repeatedly reused to hold different, and possibly unrelated, pieces of dynamically generated code; different dynamically optimized code fragments may involve different runtime optimizations; pieces of dynamically optimized code may sometimes be "deoptimized" to free up space in the shared code buffer; and such deoptimized code and may later get dynamically optimized again, possibly with a different set of optimizations that involve different parts of the JIT compiler. The third goal aims to obtain program representations that are able to separate out the effects of such complex runtime code changes and allow analyses to reason about them.

3.2 Dynamic Code Modification

Dynamic code modification can give rise to different versions of the program, with different instructions and behaviors, at different points in its execution. A representation suitable for end-to-end analysis of dynamic code should keep track of the different versions of the code resulting from dynamic modification. There are two issues to consider here: (1) what constitutes "dynamic code

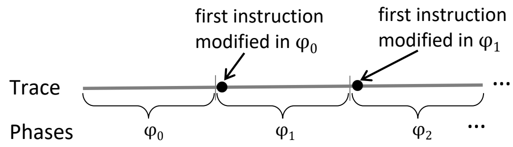


Figure 1: Phases

modification"? and (2) how should such modifications be captured in the program representation? We address these questions as follows. First, we note that in general, heuristic approaches, such as categorizing a memory write as code modification if it targets an executable section of the program's memory, may not be sufficiently precise, e.g., because permissions on memory pages can be changed during execution, making a non-executable memory region executable. We therefore consider a write to a memory location ℓ as "code modification" only if ℓ is part of some instruction that is subsequently executed. Second, even small dynamic code modifications can result in arbitrarily large changes to the program's representation and behavior. In the x86 ISA, for example, the arithmetic instruction "bitwise exclusive or" (opcode: *xor*; encoding: `0x32`) can, by flipping a single bit, be changed to the control transfer instruction "jump short if below" (opcode: *jb*; encoding: `0x72`), with potentially large effect on the control flow graph.

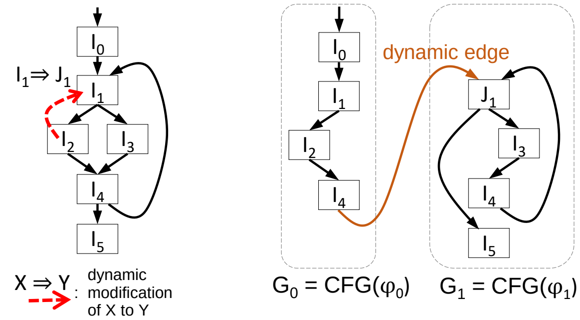
Based on these observations, we build our program's CFG using dynamic analysis, as described in Section 2.2, until we encounter an instruction whose memory locations have been modified. At this point we are confronted with a potentially arbitrary change to the program's behavior and representation. To capture this, we begin construction of a new CFG, which we link to the previously constructed CFG using a special type of edge that we call a "dynamic edge." Each such linked CFG corresponds to a "phase" of the program's execution. We make this notion more precise below.

Terminology. In some situations, it may make sense to distinguish between code created at runtime prior to being executed ("dynamic code generation") and code modified at runtime after it has already been executed ("dynamic code modification"). The ideas described here apply to both these situations, and we use the terms "generation" and "modification" of dynamic code interchangeably.

3.3 Concepts and Definitions

3.3.1 Phases. The idea behind phases is to partition an execution of a program into a sequence of fragments $\varphi_0, \varphi_1, \ldots, \varphi_i, \ldots$ such that for each φ_i, none of the locations written by the instructions in φ_i is part of any instruction executed by φ_i. Each φ_i is referred to as a *phase*. Execution begins in phase φ_0 with the program's initial code. When the first dynamic instruction is encountered, we switch to φ_1. Execution continues in φ_1 (including other instructions that may have been created or modified in φ_0) until an instruction is encountered that was modified in φ_1, at which point we switch to φ_2, and so on. This is illustrated in Figure 1. An execution with no dynamic code consists of a single phase.

More formally, given a dynamic instance I of an instruction in a program, let *instr_locs*(I) denote the set of locations occupied by I and *write_locs*(I) the set of locations written by I. These notions extend in a straightforward way to a sequence of instructions S:



(*a*) Static CFG (*b*) Dynamic CFG

Figure 2: DCFG: An example

$$instr\_locs(S) = \bigcup_{I \in S} instr\_locs(I)$$
$$write\_locs(S) = \bigcup_{I \in S} write\_locs(I)$$

Given an execution trace T for a program, let $T[i]$ denote the i^{th} instruction in T, and $T[i:j]$ denote the sequence (subtrace) $T[i], \ldots, T[j]$. We define the phases of T as follows:

Definition 3.1. Given an execution trace T, the phases of T, denoted $\Phi(T)$, is a sequence $\varphi_0, \varphi_1, \ldots, \varphi_i, \ldots$ of subtraces of T such that the following hold:

- $\varphi_0 = T[0:k]$, where $k = \max\{j \mid j \geq 0$ and $write\_locs(T[0:j]) \bigcap instr\_locs(T[0:j]) = \emptyset\}$;
- For $i \geq 0$, let $\varphi_i = T[k:(m-1)]$, then

  $\varphi_{i+1} = T[m:n]$, where $n = \max\{j \mid j \geq m$ and $write\_locs(T[m:j]) \bigcap instr\_locs(T[m:j]) = \emptyset\}$.

3.3.2 Dynamic Control Flow Graphs. We use the notion of phases to construct control flow graphs for dynamic code: we construct a CFG for each phase of the execution, as discussed in Section 2.2, and link them together using special edges, called *dynamic edges*, that represent the control flow from the last instruction of one phase to the first instruction of the next phase. We refer to such a CFG as a *dynamic control flow graph* (DCFG). More formally:

Definition 3.2. Given an execution trace T for a program, let $\Phi(T) = \varphi_0, \ldots, \varphi_n$ denote the phases of T, and let $G_i = (V_i, E_i)$ denote the CFG constructed from the subtrace ϕ_i. Then the dynamic control flow graph for T is given by $G = (V, E)$, where:

- $V = \biguplus_{i=0}^{n} V_i$ is the disjoint union of the sets of vertices V_i of the individual phase CFGs G_i; and
- $E = (\biguplus_{i=0}^{n} E_i) \cup E_{dyn}$ is the disjoint union of the sets of edges E_i together with a set of *dynamic edges* E_{dyn} defined as follows: $E_{dyn} = (last(\varphi_i), first(\varphi_{i+1}))$, where $last(\varphi_i)$ and $first(\varphi_{i+1})$ denote, respectively, the basic blocks corresponding to the last instruction of φ_i and the first instruction of φ_{i+1}.

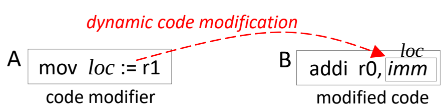
Example 3.3. Figure 2 gives a simple example of a DCFG. The static CFG of the program under consideration is shown in Figure 2(a). When instruction I_2 is executed, it changes instruction I_1 to J_1 (indicated by the dashed red arrow), where J_1 is a conditional branch with possible successors I_3 and I_5. The following is an execution trace for this program along with its phases:

$$\begin{array}{ll} \text{Trace:} & \underbrace{I_0\ I_1\ I_2\ I_4}_{\varphi_0}\ \underbrace{J_1\ I_3\ I_4\ J_1\ I_5}_{\varphi_1} \\ \text{Phases} & \end{array}$$

The first phase, φ_0, consists of the instruction sequence I_0, I_1, I_2, I_4. When control returns to the top of the loop at the end of this sequence, instruction I_1 is found to have been changed to J_1. This ends φ_0 and begins φ_1, which comprises the rest of the trace, J_1, I_3, I_4, J_1, I_5. The CFGs corresponding to phases φ_0 and φ_1 in Figure 2(b) are G_0 and G_1 respectively. Finally, the control transfer from φ_0 to φ_1 is indicated via a dynamic edge from the basic block of the last instruction of φ_0 to the basic block of the first instruction in φ_1, i.e., from the block for I_4 in G_0 to the block for J_1 in G_1.

The reader may notice, in Example 3.3, that the basic block containing I_4 occurs in both G_0 and G_1. This illustrates a potential drawback of a naive implementation of DCFGs, namely, that CFG components may be replicated across different phases. It is possible to implement DCFGs to avoid such replication, but in this case it is important to ensure that algorithms that traverse the DCFG (e.g., for slicing) do not follow unrealizable paths. The details for merging phases are discussed in Section 4; Section 6.3.3 briefly sketches the performance improvements we see from implementing sharing of DCFG components across phases.

3.3.3 Codegen Dependencies. Dynamic code modification can induce a dependency between the code performing the modification and the resulting modified code. Consider the following example:



In this example, B is an instruction that adds an immediate value *imm* to the register r0; the bytes of B containing *imm* are at address *loc*. Thus, if *loc* contains the value 5, then B ≡ 'addi r0,5'. Instruction A writes the contents of register r1 to address *loc*, thereby modifying B. When B is executed, the value added to r0 depends on the value written to address *loc* by A. Thus, the execution of A affects the behavior of B through the act of dynamic code modification, independent of any data or control dependencies that may exist in the program. We refer to dependencies arising in this way due to dynamic code modification as *codegen dependencies*. More formally:

Definition 3.4. Given an execution trace T, a dynamic instance of an instruction $I \equiv T[i]$ is codegen-dependent on a dynamic instance of an instruction $J \equiv T[j]$ $(j < i)$ if and only if, for some $loc \in instr\_locs(I)$, the following hold:

(1) $loc \in write\_locs(J)$, i.e., J modifies the location loc; and
(2) $\forall k$ s.t. $j < k < i : loc \notin write\_locs(T[k])$, i.e., J most recently modifies loc before I is executed.

While codegen dependencies resemble data dependencies in some ways, they differ in one fundamental way. If an instruction I is data dependent on an instruction J, then J can change the values used by I, but not the nature of the computation performed by I. By contrast, if I is codegen dependent on J, then J can change the nature of the computation performed by I, e.g., from an *xor* instruction to a *jump-if-below* instruction as discussed earlier.

Algorithm 1: DCFG Construction (Unoptimized)

Input: An execution trace T
Result: A DCFG $\mathbf{G}$ for T

```
1  function instr_starts_new_phase(Instr, WrittenLocs):
2      return (instr_locs(Instr) ∩ WrittenLocs ≠ ∅)
3  begin
4      G = ∅
5      φ ⟵ 0
6      W = ∅
7      G_φ = (∅,∅); add G_φ to G
8      for i = 0 to len(T) − 1 do
9          if instr_starts_new_phase(T[i], W) then
10             φ += 1
11             W = ∅
12             G_φ = (∅,∅); add G_φ to G
13         process T[i] in the context of G_φ (see Sec. 2.2)
14         if instr_starts_new_phase(T[i], W) then
15             add a dynamic edge from last block of G_{φ−1}
                 to first block of G_φ
16         W ⟵ W ∪ write_locs(T[i])
```

3.4 DCFG Construction

Algorithm 1 shows how we construct a DCFG from an execution trace. The algorithm is based directly on Definition 3.2 and constructs an unoptimized DCFG. The DCFG consists of a sequence of CFGs $\{G_\varphi \mid \varphi = 0, 1, \ldots\}$, one per phase, linked together by dynamic edges; we refer to the index φ for these CFGs as their phase index. The algorithm proceeds as follows. We initialize the phase index φ to 0 and the DCFG $\mathbf{G}$ to $\emptyset$. The set W of memory locations written in the current phase is initialized to $\emptyset$. The CFG G_φ is initialized to the empty graph and added to $\mathbf{G}$ (line 7). We then iterate through the trace T processing each instruction $T[i]$ in turn. If $T[i]$ begins a new phase, we increment the phase index (line 10), reset W to $\emptyset$ (since no memory locations have been written in the new phase that has just begun), initialize the CFG V_φ for the new phase to the empty graph, and add this new V_φ to the DCFG $\mathbf{G}$ (lines 10–12). We then process the instruction $T[i]$ in the context of the CFG G_φ, as discussed in Section 2.2 (line 13). At this point, if $T[i]$ is the first instruction of a phase (line 14), it has been added to G_φ, which means G_φ has a basic block for it, so we add a dynamic edge from the basic block of the last instruction of the previous phase to the basic block of the first instruction of the current phase (line 15). Finally, we update the set of written memory locations by adding in the set of locations written by $T[i]$ (line 16). We then continue the process with the next instruction of T.

4 SPACE OPTIMIZATION OF DCFGS

DCFGs constructed using the straightforward approach described in Algorithm 1 may contain redundancies. This is illustrated in Figure 3, which shows the execution of a program where a function f is JIT-compiled and the resulting code is executed, after which a different function g is JIT-compiled and executed. Suppose that the program's execution begins in phase φ_0. The memory writes that

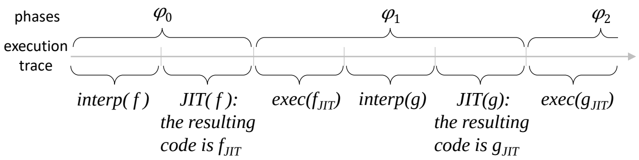


Figure 3: Potential redundancies in DCFGs

create the JIT-compiled code for f are thus in φ_0. The execution of the JIT-compiled code for f therefore causes a transition to a new phase φ_1. Subsequently executed instructions, including the JIT-compiled code for f and the JIT-compilation of g, are then a part of φ_1. When the JIT-compiled code for g is executed, there is a transition to a new phase φ_2. Thus, the JIT-compiler code executed when compiling f is part of φ_0; while the JIT-compiler code executed when compiling g is part of φ_1. The control flow graphs constructed from these two invocations of the JIT-compiler are therefore replicated, once in φ_0 and once in φ_1, means that there is potential for a significant amount of redundancy in a naively constructed DCFG. In general, the situation described arises if the same code is invoked multiple times from different phases.

A natural approach to addressing the redundancy problem would be to merge the repeated components of the DCFG. For example, if the JIT compiler is invoked multiple times in the course of execution, as in Figure 3, we can coalesce the various replicated control flow graphs for the JIT compiler into a single copy and redirect all control flow edges accordingly. However, a naive approach to such coalescing can lead to a loss in precision of analysis by propagating information along unrealizable paths, similar to the issue of context-sensitivity in interprocedural program analysis [32, 40, 43, 51].

An important difference between the general problem of context-senstive interprocedural analysis (i.e., k-CFA) and the issue of merging replicated code in DCFGs is that of the nature and complexity of the context relationships that arise. Programs can have arbitrarily complex call graphs, and increasing the amount of context information maintained during interprocedural analyses can therefore increase the precision of analysis, albeit at increased cost [22]. Phases in a DCFG, on the other hand, have a predictable linear progression, with phase n transitioning to phase $n+1$ on encountering dynamic code. This predictable structure of inter-phase relationships means that, given the phase number of a function or basic block in a DCFG, identifying the phase number of the previous or next phase is straightforward. This allows us to implement this optimization efficiently at all levels of granularity—namely, instructions, basic blocks, edges, and functions—without incurring the complexity and cost of general k-CFA.

Our implementation of merged DCFGs associates a set of phase identifiers with each DCFG component (instruction, basic block, and edge). In the simple case, there are N identical blocks, each containing the same sequence of instructions, that appear in N phases $a_1, \ldots, a_N$. We merge these into a single block, which is then associated with a set of phase identifiers $\{a_1, \ldots, a_N\}$. The resulting merged block must also account for merging the edges into/out of it. An edge that occurs in a single phase gets the phase identifier for that phase. Shared edges, on the other hand, are edges that connect the same blocks in multiple phases. These are merged

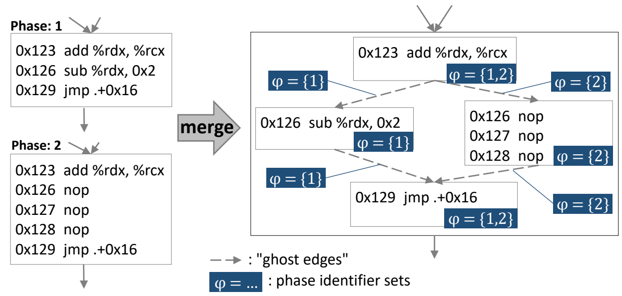


Figure 4: Merging sub-parts of a basic block. The dashed edges internal to the block are "ghost edges".

into a single edge whose set of phase identifiers is the union of the phase identifiers for the phases in which that edge appears.

Merging basic blocks becomes more complex when sharing similar but non-identical blocks. We take advantage of the similar portions of the blocks using a notion of "splitting a block across a phase." To split a block across a phase we introduce a new type of edge which we call a *ghost edge*. Conceptually, a ghost edge e is an intra-block connector and indicates that, for the given phase identifiers associated with e, the two sub-blocks connected by e should be treated as a single block. Using ghost edges we can split a block, merging the shared components across multiple phases while still keeping unique portions of the block that could not be shared. Figure 4 shows an example of merging sub-parts of a block.

When traversing a merged DCFG, a traversal along the edges and basic blocks of one phase should not take an edge leading out of a shared basic block associated with a different phase if the outgoing edge is not shared between the two phases. We use the sets of phase identifiers associated with bsic blocks and edges to enforce this requirement and only allow traversals across components with matching phase identifiers.

5 APPLICATIONS

This section discusses a few applications of DCFGs and codegen dependencies to reasoning about dynamic code.

5.1 Program Slicing for Bug Localization and Exploit Analysis in JIT Compilers

Program slicing refers to identifying instructions that (may) affect, or be affected by, the value computed by an instruction in a program [2, 30, 48]. Slicing can be static or dynamic; and, orthogonally, forward or backward. By eliminating instructions that are provably irrelevant to the computation of interest, slicing reduces the amount of code that has to be examined in order to reason about it. In the context of dynamic code modification, DCFGs play a crucial role in providing control flow information needed to construct backward slices. Analyses that reason about dynamic code solely through data dependencies, e.g. using taint propagation [11, 13, 21, 54] are unable to capture the effects of control dependencies and therefore are unsound with respect to slicing.

We implemented backward dynamic slicing as an application for evaluating the efficacy of DCFGs and codegen dependencies, with the goal of bug localization and exploit analysis in JIT compilers.

Backward dynamic slicing aims to identify the set of instructions that may have affected the value of a variable or location at some particular point in a particular execution of the program. Our implementation is based on Korel's algorithm for dynamic slicing of unstructured programs [30]; however, any slicing algorithm for unstructured programs would have been adequate.

In Korel's slicing algorithm [30], an instruction I at position p in a trace T (i.e., $I \equiv T[p]$) *depends on* an instruction $J \equiv T[q]$ (written $I \rightsquigarrow_{(Korel)} J$) if and only if, for some source operand a of I, J is the *last definition* of a at position p. More formally:

$$
\begin{aligned}
I \rightsquigarrow_{(Korel)} J \text{ iff } (\exists \text{ a source operand } a \text{ of } I)\text{:}\\
&[a \in \mathit{write\_locs}(J); \text{ and}\\
&(\forall n : q < n < p : a \notin \mathit{write\_locs}(T[n])]
\end{aligned}
$$

When processing an instruction I, Korel's algorithm (lines 5 and 16 of Fig. 11 [30]) marks all instructions J such that $I \rightsquigarrow_{(Korel)} J$. To work with dynamic code, we modify this notion to also take codegen dependencies into account, writing the resulting notion of dependency as $I \rightsquigarrow J$:

$$I \rightsquigarrow J \quad \text{iff} \quad I \rightsquigarrow_{(Korel)} J \text{ or } I \text{ is codegen-dependent on } J.$$

Our slicing algorithm is identical to Korel's except for two generalizations:

(1) Codegen dependencies are taken into account in propagating dependencies. In the marking step of the algorithm (lines 5 and 16 of Fig. 11 [30]) we use the $\rightsquigarrow$ relation rather than the $\rightsquigarrow_{(Korel)}$ relation used by Korel [30].
(2) The structure of the DCFG is taken into account by treating dynamic edges similarly to jumps (in the terminology used by Korel [30], this corresponds to the notions of *j-entry* and *j-exit*).

5.2 Detecting Environmental Triggers in Malware

Malware sometimes use environmental triggers to evade detection by performing malicious actions only if the right environmental conditions are met, e.g., if the date has some specific value. Current work on detecting such behaviors is geared towards static code, e.g., identifying conditional branches with input-tainted operands [6]. The idea is to use dynamic taint analysis to identify conditional branches of the form '**if** *expr* **then** *behavior*$_1$ **else** *behavior*$_2$' where *expr* is tainted from (i.e., influenced by) some input values. Once such conditionals have been identified, other techniques, e.g., using SMT solvers to generate alternate inputs, can be used to further explore the program's behavior.

Dynamic code opens up other ways to implement environmental triggers, e.g., by using the environmental input to directly affect what instruction bytes are generated. This idea can be illustrated by adapting an example of evasive behavior, described by Brumley *et al.* [6], to use dynamic code instead of a conditional. The code, shown in Figure 5, uses bit-manipulation instead of conditionals to evaluate the trigger expression, thereby rendering inapplicable techniques that rely on tainted conditionals. The variable `day_bits` is set to 1 or 0 depending on whether or not the most significant bit of the value of the expression `day-9` is 0, i.e., whether or not the predicate `day ≥ 9` is true. Similarly, `mth_bits` is 1 or 0 depending on whether or not `month ≥ 7` is true. Thus, the variable `trigger` is 1 or 0 depending on whether the environmental trigger—in this example, the predicate `day ≥ 9 && month ≥ 7`—is true or not. The assignment to `*(addInstrPtr+11)` writes this value into the source byte of an assignment to a variable that is used in a conditional to determine whether the malicious behavior is manifested.[1] Note that the conditional that controls the execution of the `payload()` function is neither data-dependent nor control-dependent on the input; instead there is a codegen dependency between this conditional and the patching instructions, which are data dependent on the input.

Our current implementation generalizes the approach of Brumley *et al.* [6] to incorporate codegen dependencies: we taint the values obtained from any environmental inputs of interest, then propagate taint in a forward direction. We determine that an environnmental trigger is present if either of the following hold:

(1) A conditional jump instruction with one or more tainted operands is executed; or
(2) There is a codegen dependency where the value written is tainted (equivalently: one or more memory locations containing an executed instruction are tainted).

The first condition is that originally used by Brumley *et al.* [6], while the second condition incorporates the effects of dynamic code modification. Analysis of the code shown in Figure 5 proceeds as follows. The values obtained from the call `localtime()` are tainted. This causes the variables `day_bits` and `mth_bits`, and thence the variable `trigger`, to become tainted; this tainted value is then written to memory via the assignment

```
*(addInstrPtr+11) = trigger
```

When the function `hide()` is subsequently executed, the location written by the above assignment is found to be a code location, thereby indicating a codegen dependency where the value written is tainted. This indicates the presence of an environmental trigger.

6 EVALUATION

6.1 Overview

We built a prototype implementation to evaluate the efficacy of our ideas and ran our experiments on a machine with 32 cores (@ 3.30 Ghz) and 1 TB of RAM, running Ubuntu 16.04. We used Intel's Pin software (version 3.7) [31] for program instrumentation and collecting instruction-level execution traces; and XED (version 8.20.0) [24] for instruction decoding. We iterate over the instruction trace to construct a DCFG for the execution. We identify dynamic code and determine codegen dependencies using taint analysis: we taint writes to memory, with each memory write getting a distinct taint label. For each instruction in the trace we check whether any of its instruction bytes is tainted, in which case the instruction is flagged as dynamic.

Our evaluations focused on the following questions:

[1] This code relies on the appropriate byte of the modified instruction being at a specific offset—in this case, 11 bytes—from the beginning of that function's code, and therefore is oviously highly compiler- and system-dependent. This is not atypical of malware, which are usually launched as system-specific binary executables.

```
void hide() {
  volatile int environmental_trigger = 0;
  if (environmental_trigger) {
    payload(...);    // perform malicious action
  }
}

void patch() {
  int pg_sz = sysconf(_SC_PAGE_SIZE);
  mprotect((void*) ((((long) &hide) / pg_sz) * pg_sz),
           pg_sz * 2, PROT_READ | PROT_WRITE | PROT_EXEC);
  time_t rawtime;
  struct tm * systime;
  time(&rawtime);
  systime = localtime(&rawtime);

  int day = systime->tm_mday;
  int day_test = ~(day - 9);
  int day_bits = day_test >> 31;  // day_bits == 1 iff day >= 9

  int month = systime->tm_mon+1;
  int mth_test = ~(month - 7);
  int mth_bits = mth_test >> 31;  // mth_bits == 1 iff month >= 7

  // trigger == 1 iff (day >= 9 && month >= 7)
  int trigger = day_bits & mth_bits;

  unsigned char* addInstrPtr = ((unsigned char*) &hide);
  *(addInstrPtr+11) = trigger;
}

int main() {
  hide();
  patch();
  hide();
  return 0;
}
```

Figure 5: Environmental trigger based on dynamic code

(1) *How capable are existing state-of-the-art dynamic analysis tools at end-to-end reasoning of dynamic code?*
To answer this question we used two small synthetic benchmarks to evaluate three widely-used modern dynamic analysis tools: PinPlay [36], angr [44, 47], and Triton [42].

(2) *How effective are our ideas in reasoning about dynamic code in scenarios involving problems in real-world software?*
To evaluate this question, we consider two kinds of experiments: (1) dynamic slicing for bug reports and exploits for the JIT compiler in V8, the JavaScript engine in Google's Chrome browser; and (2) two benchmarks that use dynamic code for environmental triggers in malware.

(3) *What is the performance impact of the merging optimizations discussed in Section 4?*
The bug/exploit proof-of-concept code used in the slicing experiments mentioned are deliberately constructed to crash the software quickly, and thus do not reflect typical application behavior. We use the Jetstream benchmarks (Sec. 6.4) to more accurately evaluate the impact of our memory optimizations on typical application code.

The code for our prototype implementation is available at `https://github.com/skdebray/ASE-2020/` and `https://www2.cs.arizona.edu/projects/lynx-project/Samples/ASE-2020`. Our data samples are available at `https://www2.cs.arizona.edu/projects/lynx-project/Samples/ASE-2020/DATA`.

6.2 Assessing the Capabilities of Existing Tools

We evaluated the capabilities of existing state-of-the-art tools using three widely-used modern dynamic analysis tools that implement backward dynamic slicing, namely: PinPlay [36] (revision 1.29), angr [44, 47] (commit bd3c6d8 on github), and Triton [42] (build no. 1397).

		Our approach	PinPlay	angr	Triton
Synthetic	Benchmark 1	Y	N	N	N
	Benchmark 2	Y	N	N	N
Exploit analysis	V8 OOB to JIT code pages	Y	X	X	X
	V8 Escape analysis bug	Y	X	X	X
	LuaJIT exploit	Y	N	N	N
Bug localization	OOB Read	Y	X	X	X
	JIT type confusion	Y	X	X	X
	Scoping issue	Y	X	X	X

Key:
Y: Picks up dynamic code generator from backwards slice of dynamic code.
N: Does not pick up dynamic code generator from backwards slice of dynamic code.
X: Crashes or fails to load.

Table 1: Assessing Existing Dynamic Analysis Tools

We invoked these tools to incorporate support for self-modifying code as follows: we set the flags `smc_support` and `smc_strict` flags to true for PinPlay, and loaded our project with `auto_load_libs` and `support_selfmodifying_code` set to true for angr.

To avoid potentially confounding factors such as code size or complexity, we considered two small synthetic benchmarks of 15 and 55 x86 instructions respectively. Both programs are simple in structure: one adds a constant to the target operand of a jump instruction; the other adds a constant to the immediate operand of an add instruction. The fixed and unconditional nature of these code modifications means that there is nothing tricky, e.g., no data or control dependencies, between the instructions being dynamically modified and the instructions performing dynamic modification. This allows us to focus entirely on questions of representation and analysis of dynamic code: any problems in analyzing such simple programs relate directly to shortcomings in the underlying program representations and analysis algorithms when applied to dynamic code.

We used the three tools mentioned above, along with our prototype implementation of slicing (Section 5.1) to carry out backward dynamic slicing on our synthetic benchmarks. In each case, we computed a backward dynamic slice with the slice criterion being the value computed by the function whose code was dynamically modified. The results of these experiments are summarized in Table 1. It can be seen that while all three tools successfully included all of the relevant non-codegen-dependent instructions in the slices they computed, none of them are able to pick up the code that performs dynamic modification. Given that soundness for slicing algorithms is defined as not excluding any statement that can influence the slicing criterion, this indicates that the resulting slices were unsound. On further investigation, we found that the reason for this is a fundamental limitation of the underlying CFGs constructed by these tools, which do not represent the different versions of code resulting from dynamic code modification. By contrast, we found that our implementation, using DCFGs and codegen dependencies, computed slices that correctly contained the instructions that performed dynamic code modification.

		Tracing		DCFG Construction					Slicing		
	Test program	N_{trace}	T_{read}	N_{instrs}	N_{blocks}	N_{edges}	N_{phases}	T_{DCFG}	N_{slice}	T_{slice}	Δ_{slice}
Exploit analysis	V8 OOB to JIT Code Pages	11,134,237	10.68	191,613	41,302	117,158	4	146.88	81,986	433.25	57 %
	V8 Escape analysis bug	135,295,168	130.76	245,935	52,929	153,922	3	1,793.23	120,885	10,193.08	50 %
	LuaJIT Exploit	464,743	0.60	18,248	4584	12,606	2	7.47	5,139	7.76	71 %
Bug localization	OOB Read	14,720,437	14.25	150,115	31,469	92,254	2	196.29	61,511	579.78	59 %
	JIT Type Confusion	9,663,365	9.49	158,849	32,536	93,132	9	130.26	67,765	146.47	57 %
	Scoping issue	7,882,295	7.56	99,378	22,394	62,204	4	102.31	47,023	970.95	52 %

Key:

N_{trace}	:	No. of instructions in execution trace
T_{read}	:	Time to read trace (seconds)
N_{instrs}	:	No. of instructions in DCFG
N_{blocks}	:	No. of basic blocks in DCFG
N_{edges}	:	No. of basic blocks in DCFG
N_{phases}	:	No. of phases
T_{DCFG}	:	DCFG construction time (seconds)
N_{slice}	:	No. of instructions in slice
T_{slice}	:	Slice construction time (seconds)
Δ_{slice}	:	Fraction of DCFG removed from slice $= (N_{instrs} - N_{slice})/N_{instrs}$.

Table 2: Slicing: Performance

Additionally, to assess the applicability of these tools to real-world software that makes use of dynamic code, we evaluated them on six bug and exploit reports for the V8 JIT compiler. As shown in Table 1, none of them were able to successfully analyze these examples: they all crashed with internal errors when loading V8. All three tools were able to process the LuaJIT example without crashing, but none of the slices they computed contained the JIT-compiler or exploit code that created the dynamic code.

6.3 Analysis Efficacy on Real-World Examples

To evaluate our approach on real world software that uses dynamic code, we consider three applications: (1) analysis of exploits involving JIT code; (2) bug localization in JIT compilers; and (3) detection of trigger-based evasive behaviors that use dynamic code. Our goal was to perform end-to-end analyses on these examples, i.e., start from the problematic dynamic code and compute a backward dynamic slice that includes the culprit portions of the dynamic code generator where the bug/security exploit originates. The results are shown in Table 1.

6.3.1 Exploit Analysis. We consider three examples of exploits, two of them involving dynamic code in Google's V8 JavaScript engine:

1. malicious shellcode originating from an out-of-bounds (OOB) write to the JIT code pages in V8 [9];
2. escape analysis bug in V8's JIT compiler (CVE-2017-5121) [38]; and
3. malicious bytecode used to escape a LuaJIT sandbox [8].

For each of these exploits, we used the proof-of-concept code to compute a DCFG/backward dynamic slice starting from the dynamically generated exploit code. Separately, we used the write-up for each exploit to determine the bugs responsible for each exploit, identifying the buggy code generator portions in the execution traces recorded for each exploit. We then checked the slice to determine whether the buggy generator code is present in the slice.

The first security exploit we consider entails an OOB write to the JIT code pages within Google's V8 JavaScript engine [9]. The exploit is a result of array type ambiguity that allows the attacker to write and execute arbitrary shellcode. We constructed a DCFG from an execution trace of the buggy V8 code and computed a backward dynamic slice from the first nop shellcode instruction in the nop sled in the attack code. Our backward slice correctly included both the buggy code within V8 that led to the array type ambiguity along with the exploit code that generated the shellcode at runtime.

The second exploit we examined is discussed in detail by Rabet [38]. It arises out of a bug in V8's escape analysis and causes some variable initializations in the JIT-optimized code to be incorrectly optimized away when performing load reduction. The proof-of-concept code provided causes V8 to crash while executing the optimized dynamic code due to an OOB read. The write up provided by Rabet proceeds to use this OOB read as a stepping stone towards demonstrating arbitrary code execution. For our analysis of this example, we built our DCFG from the execution trace recorded by Pin and then we computed a backward dynamic slice from the dynamic instruction prior to the exception that is thrown due to the OOB read. We found that the resulting slice correctly included the buggy portions of the load reducer in the escape analysis phase of V8's JIT compiler, whose optimizations cause the OOB read.

Our final example in this category was with malicious Lua bytecode being used to escape a sandbox in LuaJIT [8]. The proof of concept malicious program corrupts bytecode with the goal of writing shellcode which prints a message. We followed an approach similar to the one we used to slice the V8 OOB write, starting our slice at the beginning of the NOP sled used in the attack. We found that the backward slice computed by our tool correctly picks up the Lua code that generates the shellcode.

The role of codegen dependencies. For each exploit example discussed, we computed slices starting at a `NOP` instruction in the NOP sled generated as part of the shellcode. To assess the role of codegen dependencies, we recomputed these slices ignoring codegen dependencies. We found that, in each case, the resulting slice consisted of just the `NOP` instruction and nothing else. By contrast, when codegen dependencies were considered, the relevant JIT-compiler code was included in the slice. This demonstrates that codegen dependencies are fundamental to reasoning about the relationship between dynamically generated code and the dynamic code generator that created that code.

	Test program	Original $DCFG_{orig}$	Original $slice_{orig}$	Dicing $DCFG_{mk}$	Dicing $slice_{mk}$	Improvement (%) Δ_{DCFG}	Improvement (%) Δ_{slice}	Improvement (%) Δ_{mk}
Exploit analysis	V8 OOB to JIT Code Pages	191,613	81,986	90,736	42,317	52.6	48.4	53.4
	V8 Escape analysis bug	245,935	120,885	157,847	89,307	35.8	26.1	43.4
	LuaJIT Exploit	18,248	5,139	10,354	1,808	43.2	64.8	82.5
Bug localization	OOB Read	150,115	61,511	35,261	10,460	59.0	83.0	70.3
	JIT Type Confusion	158,849	67,765	188	103	99.9	99.8	45.2
	Scoping issue	99,378	47,023	14,896	7,721	85.0	83.6	48.2

Key:

$DCFG_{orig}$: No. of instructions in original DCFG
$slice_{orig}$: No. of DCFG instructions in original slice
$DCFG_{mk}$: No. of instructions in DCFG with marker
$slice_{mk}$: No. of DCFG instructions in slice with marker
Δ_{DCFG} : Improvement in DCFG size due to dicing $= (DCFG_{orig} - DCFG_{mk})/DCFG_{orig}$
Δ_{slice} : Improvement in slice size due to dicing $= (slice_{orig} - slice_{mk})/slice_{orig}$
Δ_{mk} : Fraction of $DCFG_{mk}$ removed due to dicing $= (DCFG_{mk} - slice_{mk})/DCFG_{mk}$

Table 3: Dicing: Performance

6.3.2 Bug Localization. We consider three JIT compiler bugs from Google's V8 JavaScript engine that were posted to `bugs.chromium.org` and classified as "Type: Bug-Security."

(1) Empty jump tables generated by the bytecode generator leading to out-of-bound reads that crash the generated JIT-compiled code [17].
(2) A type confusion bug that leads to a crash after the dynamic code has been generated [18].
(3) Arrow function scope fixing bug, where certain constructs involving a single line arrow function cause a crash [19].

For each of these bugs we proceeded as follows. To identify the problematic code in the JIT compiler, we examined the corresponding GitHub commits, together with any relevant information in the bug report, to determine the code that was changed to fix the bug. We delineated the problem code so identified using small "marker code snippets"—i.e., small easily identifiable code snippets that do not affect the operation of the JIT compiler—and confirmed that the behavior of the buggy JIT compiler was unaffected. We then used the example code submitted with the bug report to obtain an execution trace demonstrating the bug, and used this trace, together with the DCFG constructed from it, to compute a backward dynamic slice starting from the instruction that crashed. Finally, we analyzed the resulting slice to determine whether the problematic code, as identified above, was included in the slice.

The results of our experiments are summarized in Table 1. Our end-to-end analysis was able to successfully pick up the buggy code for each of the bugs mentioned above in the slice, allowing one to narrow down the functions involved in V8 that lead to the crash.

6.3.3 Performance. Table 2 shows the performance of our prototype DCFG-based slicing implementation on our real-world test inputs (the environmental trigger example is omitted because it does not use backward slicing). These input programs all involve computations of substantial size: the smallest, LuaJIT exploit, has a trace of 464K instructions, while the remaining execution traces range from almost 7.9M instructions (V8 scoping issue bug) to 135M instructions (V8 escape analysis bug). The time taken to read the traces (and do nothing else) is roughly 1M instructions/sec.[2]

The DCFGs constructed typically range in size from about 22K basic blocks and 62K edges (V8 scoping issue bug) to about 41K blocks and 117K edges (V8 OOB exploit), with a low of 4.6K blocks and 12K edges for the LuaJIT exploit and a high of about 53K blocks and 154K edges for the V8 escape analysis bug. Most of our test programs have 2 − 4 phases, with the V8 JIT type confusion example an outlier with 9 phases. DCFG construction incurs an overhead of roughly 15× over simply reading a trace: most of the test inputs take roughly 2 − 3 minutes, with the lowest time being 7.5 seconds for the LuaJIT exploit and the highest being about 30 minutes for the V8 escape analysis bug. Since DCFG construction involves processing each instruction in the execution trace, the time taken depends on the sizes of both the instruction trace and the DCFG.

The overhead incurred by slicing relative to the time taken for DCFG construction ranges from 1.04× for the LuaJIT exploit to 9.5× for the V8 scoping issue bug, with most of the test programs ranging from 3× to 6×. In absolute terms, most of the programs take about 2 − 10 minutes for slicing, with a low of about 8 secs for the LuaJIT example and a high of about about 2.8 hours for the V8 escape analysis bug. Slicing is able to remove about 50%–60% of the instructions in the DCFG, with a high of 71% of the instructions removed for the LuaJIT exploit. These results indicate that our approach is both practical (in terms of time) and useful (in terms of the amount of code removed from the DCFG). Since our approach does not fundamentally alter the slicing algorithm, but rather augments it to work over DCFGs and use codegen dependencies, it is not difficult to adapt our approach to other slicing algorithms with different cost-precision characteristics.

6.3.4 Focusing the analysis: markers and dicing. Given our objective of localizing problems in the JIT-compiler code, it is useful to examine the extent to which our approach is able to reduce the amount of actual JIT-compiler code that has to be considered. To

[2] Our implementation uses Pin to collect an instruction trace that is written to a file on disk. The numbers reported here refer to the time required to read such instruction trace files; the time taken to record the traces and write the trace files, which depends on the tracing tool used and is independent of the ideas described here, is not included.

	Test program	No. of Instructions Orig	Opt	Δ(%)	No. of Basic Blocks Orig	Opt	Δ(%)	No. of Edges Orig	Opt	Δ(%)
Performance benchmarks	base64	781,404	308,748	60.5	167,925	64,095	61.8	308,748	197,042	36.2
	crypto-sha1	1,158,366	319,098	**72.5**	245,758	65,634	**73.3**	719,114	202,096	**71.9**
	date-format	453,177	324,279	28.4	94,417	67,611	28.4	278,666	101,902	63.4
	nbody	394,264	284,973	27.7	81,617	58,054	28.9	239,080	174,498	27.0
	poker	595,329	366,571	38.4	125,709	78,485	37.6	365,978	236,562	35.4
	str-unpack	372,862	251,121	32.7	75,164	50,899	32.3	215,716	151,980	29.5
Security benchmarks	V8 OOB to JIT Code Pages	193,339	152,723	21.0	41,302	32,205	22.0	117,158	94,568	19.3
	V8 Escape analysis bug	247,264	212,800	13.9	52,929	46,201	12.7	153,922	137,974	10.4
	LuaJIT Exploit	21,389	19,436	9.1	4,584	4,153	9.4	12,606	11,624	7.8
	OOB Read	151,773	133,134	12.3	31,469	27,268	13.3	92,254	82,046	11.1
	JIT Type Confusion	160,526	128,188	20.1	32,536	25,441	21.8	93,132	76,110	18.3
	Scoping issue	101,193	89,675	11.4	22,394	19,910	11.1	62,204	56,382	9.4

Key:
Orig : Value in original-representation DCFG
Opt : Value in optimized-representation DCFG
Δ : Improvement = $(Orig - Opt)/Orig$

Table 4: Impact of representation optimization on DCFG size

do this, we placed markers—i.e., small code snippets that are unambiguously identifiable and semantically neutral—in the code as close as we were able to the invocation of the JIT compiler. During analysis, we excluded the portion of the execution trace before the marker. This effectively computed a program dice that excluded the front-end parser, byte-code generator, and interpreter.

Table 3 gives the results of these experiments. The two columns labeled 'Original' refer to the size of the DCFG and the backward slice computed without markers, i.e., as shown in Table 2; the columns labeled 'Dicing' refer to the size of the DCFG and slice when markers are used; the columns labeled 'Improvement' show the percentage improvement due to dicing. The columns labeled Δ_{DCFG} and Δ_{slice} show, respectively, the reductions in the size of the DCFG and the slice when irrelevant code is excluded. These are in the range 35%–85% for DCFG size and 26%–84% for slice size. The JIT Type Confusion bug sample is an outlier, with almost all of the original DCFG and slice eliminated. The final column, labeled Δ_{mk}, shows the effects of slicing focusing only on the DCFG resulting from dicing: these range from about 43% to about 82%. Overall, these results show that (1) our approach is effective in focusing on the relevant portions of the JIT compiler; and (2) the use of code markers to identify entry into the JIT compiler can be helpful in zeroing in on the relevant portions of the code being analyzed.

6.4 Detecting Environmental Triggers

We use two test programs to evaluate the detection of environmental triggers based on dynamic code: one is shown in Figure 5, the other is a variant of this program that uses implicit flows to further disguise the influence of environmental values on the trigger code.

We built two detectors to demonstrate the utility of DCFGs and codegen dependencies for this purpose. In the first case, we taint the input source and propagate the taint forward in the execution trace. If there is a codegen dependency from an instruction with tainted operands to an instruction that is later executed, an input-dependent value may be influencing the instruction bytes of some dynamic instruction, and we report that there is dynamic input-dependent program behavior. In the second case, we compute a backward dynamic slice with the slicing criterion being the dynamically modified code location at the point where it is executed.

Our implementations correctly detect that environmental values influence dynamic program behavior for our benchmarks. To assess the state of the art, we tested these programs using two widely used analysis tools: S2E, a widely used symbolic execution engine [10], and angr. In each case, we found that the input values used to patch the function `hide()` in Figure 5 are silently concretized and only the false path is explored. As a result, these tools are unable to identify the environment-dependent aspect of the program's behavior.

6.5 Space Optimization: The Impact of Merging

To evaluate the effect of the space optimization discussed in Section 4, we used a collection of benchmarks from the Jetstream 2 suite of Javascript workloads [5]: *base64* [37], *crypto-sha1* [26], *date-format* [49], *nbody* [16], *poker* [1], and *str-unpack* [25]. The results are shown in Table 4. These benchmarks have significantly larger DCFGs than the security benchmarks described earlier. This is not surprising, since the security benchmarks were submitted as demo code for bug reports and thus aimed to quickly manifest the bug and crash or exit the program. The performance benchmarks yielded significantly higher performance improvements than the security benchmarks, with improvements ranging from 27% to 72%.

We also found that the amount of improvement increases with the size of the unoptimized DCFG. This is shown in Figure 6. This indicates that there is a significant amount of overlap in the code executed by different phases (e.g., library code, the interpreter and JIT compiler), and also that our merged DCFG representation is effective in optimizing away the resulting redundancies.

We did not see a significant difference in execution speed between the DCFG implementations with and without this optimization. The version using space-optimization was slightly faster on average, possibly due to fewer calls to allocate/free routines and improved memory locality.

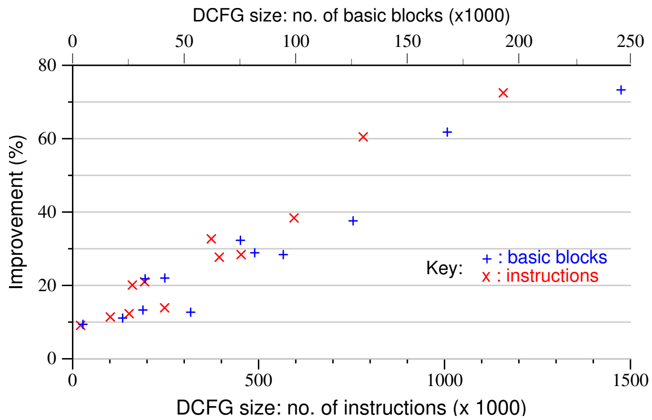


Figure 6: Space optimization improvements vs. DCFG size

7 SUMMARY AND DISCUSSION

Our design goals, in Section 3.1, were to devise a program representation that naturally and scalably generalizes existing representations; allows existing analyses to be extended to dynamic code in a simple and natural way; and is precise enough to distinguish between conceptually distinct dynamic code modifications. DCFGs provide a natural generalization of the well-known notion of control flow graphs to dynamic code and thus satisfy the first goal. Section 5.1 shows how we extend slicing to dynamic code in a straightforward way, thereby satisfying the second goal. For the third goal, DCFGs allow us to distinguish the code structure of individual JIT-compiled functions by separating out the different code modifications in different DCFG phases, with the space optimizations of Section 4 ensuring scalability; codegen dependencies then make it possible to identify and reason about the code components and value flows in the dynamic code generator relevant to the code modifications in each such phase. As far as we know, no other existing system can do this.

8 RELATED WORK

Anckaert *et al.* describe a program representation for dynamic code that is capable of representing multiple versions of the code as it is modified during execution [4]. However, this work does not have a notion of codegen dependencies and as a result is of limited utility for applications that involve reasoning about causal relationships between the dynamic code generator and the dynamic code.

Debray and Yadegari discuss reasoning about control dependencies in interpreted and JIT-compiled code [52]. While the goals of this work are similar to ours, its technical details are quite different. In particular, it does not aim to provide a program representation capable of supporting arbitrary dynamic code, but instead is narrowly focused on control dependency analysis in interpretive systems. It also makes assumptions, such as the ability to map each dynamically generated instruction to a unique byte-code instruction it originated from, that render it inapplicable to contexts not involving interpreters, such as the dynamic-code-based environmental triggers discussed in Sections 5.2 and 6.4.

Korczynski and Yin discuss identifying code reuse/injections using whole-system dynamic taint analysis [29]. While this work captures codegen dependencies, it does not propose a program representation that can capture the code structure for the different phases that arise during execution. As a result, this approach is not suitable for analyses, such as program slicing, that require information about the control flow structure of the code. Dalla Preda *et al.* describe a notion of phases to characterize the semantics of self-modifying code [12], however this work was never implemented and the technical details are very different from ours.

There is a large body of literature on program slicing (e.g., see [30, 39, 46, 50, 55]), but all of this work focuses on static code. There is a lot of work on dependence and information flow analyses (e.g., see [20, 27, 35]), but these typically do not consider end-to-end analysis of dynamic code. Several authors have discussed taint propagation in JIT-compiled code, but focusing on taint propagation in just the application code rather than on end-to-end analyses [13, 28, 41]. Whole-system analyses [11, 13, 21, 53, 54] focus on issues relating to dynamic taint propagation through the entire computer system. Such systems provide end-to-end analyses but typically consider only explicit information flows ($\simeq$ data dependencies), not implicit flows ($\simeq$ control dependencies); they are thus of limited use for reasoning about behaviors, such as conditional dynamic code modification (i.e., where the dynamic code generated may depend conditionally on input and/or environmental values), which are common in applications such as JIT compilers.

There are a number of systems that reason about program behavior using dynamic analysis, and therefore are able to perform some kinds of analysis on dynamic code [36, 42, 44, 47]. Our experiments indicate that these systems do not keep track of multiple versions of code resulting from dynamic code modification, and so cannot fully capture the dependencies arising from runtime code changes.

Cai *et al.* [7] and Myreen [33] discuss reasoning about dynamic code for the purposes of program verification using Hoare logic. We have not seen any implementations to apply their work towards modern software that utilizes dynamic code (i.e. a javascript engine). Furthermore, our work is more specific in that we seek to provide a program representation capable of representing dynamic code.

9 CONCLUSIONS

Dynamic code is ubiquitous in today's world. Unfortuntely, existing approaches to program analysis are not adequate for reasoning about the behavior of dynamic code. This paper discusses how this problem can be addressed via a program representation suitable for dynamic code as well as a new notion of dependencies that can capture dependencies between the dynamic code and the code that generated it. Experiments with a prototype implementation of backwards dynamic slicing based on these ideas show, on a number of real-world examples, that these ideas make it possible to work back from the faulty code to the JIT compiler logic that led to the generation of the faulty code.

ACKNOWLEDGMENTS

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

REFERENCES

[1] [n.d.]. *Uni-poker Javascript source code.* https://browserbench.org/JetStream/RexBench/UniPoker/poker.js

[2] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *ACM SIGPlan Notices*, Vol. 25. ACM, 246–256.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. 1985. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

[4] B. Anckaert, M. Madou, and K. De Bosschere. 2006. A Model for Self-Modifying Code. *LNCS* 4437, 232–248.

[5] Saam Barati. 2019. *Introducing the JetStream 2 Benchmark Suite.* https://webkit.org/blog/8685/introducing-the-jetstream-2-benchmark-suite/

[6] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 65–88.

[7] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified self-modifying code. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 66–77.

[8] Peter Cawley. 2015. *Malicious LuaJIT bytecode.* https://www.corsix.org/content/malicious-luajit-bytecode

[9] Oliver Chang. 2017. *Exploiting a V8 OOB write.* https://halbecaf.com/2017/05/24/exploiting-a-v8-oob-write/

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 265–278.

[11] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium.* 321–336.

[12] M. Dalla Preda, R. Giacobazzi, and S. Debray. 2015. Unveiling metamorphism by abstract interpretation of code properties. *Theoretical Computer Science* 577 (April 2015), 74–97.

[13] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS* 32, 2 (2014).

[14] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX. In *Proc. 2017 ACM Conference on Computer and Communications Security.* 2405–2419.

[15] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proc. PLDI 2009.* 465–478.

[16] Isaac Gouy. [n.d.]. *nbody Javascript source code.* https://browserbench.org/JetStream/SunSpider/n-body.js

[17] Loki Hardt. 2015. *Issue 794825: Security: V8: Empty BytecodeJumpTable may lead to OOB read.* https://bugs.chromium.org/p/chromium/issues/detail?id=794825

[18] Loki Hardt. 2017. *Issue 794822: Security: V8: JIT: Type confusion in GetSpecializationContext.* https://bugs.chromium.org/p/chromium/issues/detail?id=794822

[19] Loki Hardt. 2018. *Issue 807096: Security: Arrow function scope fixing bug.* https://bugs.chromium.org/p/chromium/issues/detail?id=807096

[20] Christophe Hauser, Frederic Tronel, Ludovic Mé, and Colin J. Fidge. 2013. Intrusion detection in distributed systems, an approach based on taint marking. In *Proc. 2013 IEEE International Conference on Communications (ICC).* 1962–1967.

[21] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 248–258.

[22] David Van Horn and Harry G. Mairson. 2007. Relating complexity and precision in control flow analysis. In *Proc. 12th ACM SIGPLAN International Conference on Functional Programming (ICFP).* 85–96.

[23] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2012. Adaptive Multi-level Compilation in a Trace-based Java JIT Compiler. In *Proc OOPSLA 2012.* 179–194.

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

[25] Bob Ippolito. [n.d.]. *str-unpack Javascript source code.* https://browserbench.org/JetStream/SunSpider/string-unpack-code.js

[26] Paul Johnston. [n.d.]. *crypto-sha1 Javascript source code.* https://browserbench.org/JetStream/SunSpider/crypto-sha1.js

[27] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*.

[28] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Information flow tracking meets just-in-time compilation. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 38.

[29] David Korczynski and Heng Yin. 2017. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1691–1708.

[30] Bogdan Korel. 1997. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering* 23, 1 (1997), 17–34.

[31] 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.

[32] Florian Martin. 1999. Experimental comparison of call string and functional approaches to interprocedural analysis. In *International Conference on Compiler Construction*. Springer, 63–75.

[33] Magnus O Myreen. 2010. Verified just-in-time compiler on x86. In *ACM Sigplan Notices*, Vol. 45. ACM, 107–118.

[34] N. Nethercote and J. Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI).* 89–100.

[35] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*.

[36] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* ACM, 2–11.

[37] Martijn Pieters and Samuel Sieb. [n.d.]. *base64 Javascript source code.* https://browserbench.org/JetStream/SunSpider/base64.js

[38] Jordan Rabet. 2017. Browser security beyond sandboxing. Microsoft Windows Defender Research. `https:cloudblogs.microsoft.com/microsoftsecure/2017/10/18/browser-security-beyond-sandboxing`.

[39] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B Dwyer. 2007. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 5 (2007), 27.

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 49–61.

[41] Tiark Rompf, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *Acm Sigplan Notices*, Vol. 49. ACM, 41–52.

[42] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015.* SSTIC, 31–54.

[43] M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones (Eds.). Prentice-Hall, 189–233.

[44] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).

[45] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes.* Elsevier.

[46] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation.* 112–122.

[47] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. (2016).

[48] F. Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3 (1995), 121–189.

[49] Svend Tofte. [n.d.]. *date-format Javascript source code.* https://browserbench.org/JetStream/SunSpider/date-format-tofte.js

[50] Mark Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering* 10, 4 (July 1984), 352âĂŞ357.

[51] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proc. SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95).* 1–12.

[52] Babak Yadegari and Saumya Debray. 2017. Control Dependencies in Interpretive Systems. In *International Conference on Runtime Verification*. Springer, 312–329.

[53] Heng Yin and Dawn Song. 2010. Temu: Binary code analysis via whole-system layered annotative execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3* (2010).

[54] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 116–127.

[55] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2004. Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. In *Proc. 26th International Conference on Software Engineering*. 502–511.