# IMPROVING PROGRAM REPRESENTATIONS FOR DYNAMIC CODE

by

**Jesse Bartels**

---

Copyright ©Jesse Bartels 2019

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

2019

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Master's Committee, we certify that we have read the thesis prepared by: Jesse Dalton Bartels
titled: Improving Program Representations for Dynamic Code

and recommend that it be accepted as fulfilling the thesis requirement for the Master's Degree.

_______________________________ Date: Oct 24, 2019
Saumya Debray

Katherine Isaacs
_______________________________ Date: Oct 28, 2019
Katherine Isaacs

Michelle Strout
_______________________________ Date: Nov 4, 2019
Michelle Strout

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to the Graduate College.

I hereby certify that I have read this thesis prepared under my direction and recommend that it be accepted as fulfilling the Master's requirement.

_______________________________ Date: Oct 24, 2019
Saumya Debray
Thesis Committee Chair
Department of Computer Science

## Acknowledgements

I feel that this (and any) acknowledgement section would be quite incomplete without first mentioning the folks who directly made my thesis possible in the first place. Thank you Dr. Debray. You started me on the rewarding path of research and these skills that you have taught me will serve me well in life. My life would be far poorer if you hadn't taken me on to do research. Thank you Jon Stephens, for keeping me going when things got rough and for showing me the ropes in program analysis and in how to do research. I think I eternally owe you a big slice of deep dish from Rocco's to repay you. A big shout out to my committee members, Dr. Strout and Dr. Isaacs, both of whom have given excellent feedback and attention to my thesis (thank you!).

I'd also like to thank the folks who supported me throughout all my endeavours that have lead to this point. Big thank you to my mom, dad, and step parents for all of the guidance and help throughout the years. I wouldn't have gotten this far without you. I also wish to thank my brothers. Thank you Trace, for sitting through all the practice renditions of my thesis defense and for being a solid big brother. Thank you Austin, I couldn't ask for a better twin brother (and yes, *finger guns*, it's been too long). Finally I'd like to thank all of my friends. Whether it was through tabletop games, football, or hiking, you guys always reminded me of the light at the end of the tunnel.

Thank you all, seriously, for your time and patience. I can't express enough how much of an impact you've all had on getting me through this chapter of my life. Thank you.

# Table of Contents

**Abstract** . . . . . . . . . . 6

**1 Introduction** . . . . . . . . . . 7

**2 Background** . . . . . . . . . . 9

2.1 Trace Based Analysis: . . . . . . . . . . 10

2.2 Dynamic Taint Analysis: . . . . . . . . . . 10

2.3 Backwards Slicing: . . . . . . . . . . 10

2.4 Dicing: . . . . . . . . . . 11

2.5 Symbolic Execution: . . . . . . . . . . 11

2.6 Control Flow Graphs and Basic Blocks: . . . . . . . . . . 11

2.7 Interprocedural Control Flow Graphs: . . . . . . . . . . 11

2.8 Trace Based Interprocedural CFG reconstruction: . . . . . . . . . . 11

**3 Approach** . . . . . . . . . . 12

3.1 Challenges and Motivation: . . . . . . . . . . 13

3.2 Definitions and Approach: . . . . . . . . . . 15

3.3 Implementation: . . . . . . . . . . 19

3.4 Improvements with Phase Overlap: . . . . . . . . . . 21

3.5 Slicing: . . . . . . . . . . 23

**4 Evaluation** . . . . . . . . . . 24

4.1 Experimental Methodology and Setup . . . . . . . . . . 25

4.2 Initial Results: . . . . . . . . . . 25

4.3 Results For Complex Real World Examples: . . . . . . . . . . 30

4.4 Experimental Results Summary: . . . . . . . . . . 35

**5 Related Work:** . . . . . . . . . . 36

**6 Conclusion** . . . . . . . . . . 38

6.1 DCFG and Code-Gen Dependencies . . . . . . . . . . 38

6.2 End-To-End Analysis . . . . . . . . . . 39

**7 References** . . . . . . . . . . 41

# List of Figures


1 Dynamic Code Example . . . . . . . . . . 14
2 Encountering Dynamic Code and Creating a New Phase . . . . . . . . . . 18
3 DCFG Representation . . . . . . . . . . 20
4 Overlapping Phases . . . . . . . . . . 23
5 First Synthetic Example . . . . . . . . . . 28
6 Second Synthetic Example . . . . . . . . . . 29
7 Environmental Triggers with Dynamic Modification . . . . . . . . . . 34
8 Implicit Flow Example . . . . . . . . . . 36


# List of Tables


1 Slicing: Precision on Synthetic Benchmarks . . . . . . . . . . 27
2 Slicing: Performance . . . . . . . . . . 32
3 Dicing: Performance . . . . . . . . . . 33

**Abstract**

Being able to properly represent dynamic code, or the notion of code that is created/modified at runtime, is crucial towards improving debugging capabilities, aiding software security analysis, and building a general understanding of the systems that make use of dynamic code. From malware to Just-In-time (JIT) compilers, dynamic code is surprisingly commonplace in today's software ecosystem. Yet despite the prevalence of dynamic code, we have found that the current state of the art program analysis tools are incapable of properly representing dynamic code as it is created/modified over a program's execution. This work aims to provide an improved program representation, allowing for what we call end-to-end analysis to help reason about dynamic code and its relationship with the code that generates it. Our results demonstrate using backwards slicing and forward taint analysis to perform exploit analysis, bug localization, and reasoning about environmental triggers not possible with other program analysis toolkits, providing useful insights from our end-to-end analysis as well as a foundation upon which to incorporate other program analysis techniques.

# 1 Introduction

When performing program analysis one often has to first reconstruct a working program representation, usually with the goal to gather insight on the program being analyzed. In this paper, we present a novel program representation for the case of dynamically modified/created code (hereby referred to as dynamic code), enabling what we call end-to-end analysis in cases of dynamic code. As we define it, end-to-end analysis is the idea that, given a suspect piece of dynamic code, one should be able to reason about the behavioral relationships between the originating code that created and/or modified that dynamic code and the dynamic code itself. As far as we are aware, existing program representations do not allow for end-to-end analysis, due to deficiencies in representing multiple versions of instructions in the control flow graph (CFG) and missing data flow representations between dynamic code and the code that generates/modifies it. These deficiencies mean that the possibility of performing end-to-end analysis, such as tracing backwards from an improperly generated section of JIT compiled code to the malfunctioning section of the JIT compiler, is impossible with existing program analysis frameworks. In this paper, we seek to address these concerns and put forth our solution, a modification to the traditional CFG structure and an explicit tracking of dependencies, to allow for end-to-end analysis in the case of dynamic code. The contributions of this paper are a novel program representation that allows for end-to-end analysis in the case of dynamic code. We demonstrate this with an implementation of backwards slicing as well as a modification to forward taint analysis that both use our program representation to perform several end-to-end analysis in examples of large scale real world software. In our evaluation we successfully perform several end-to-end analysis for exploit analysis and bug localization for Google's Javascript engine V8 as well as in LuaJIT, correctly including buggy JIT compiler code in slices taken from the exploit/crashed dynamic code. We also successfully detect the presence of environmental trigger based behavior in which dynamic code is used as a vehicle to launch the payload if the correct conditions are met.

There are many legitimate uses of dynamic code, which plays an important role in many large scale popular software systems. Internet browsers like Chrome or Edge often incorporate a Just-In-Time (JIT) compiler capable of generating and rewriting code at runtime to improve performance when executing javascript. Java, Ruby, and variations of Python all utilize a JIT compiler for performance benefits as well. Dynamic binary translators, such as QEMU, also make use of code generated at runtime for the sake of portability.

Despite its prevalence, we have noticed a lack of analysis capabilities for software that uses dynamic code. This presents issues for both software security analysis and traditional debugging. The ideal analysis capabilities would include the ability to easily work backwards from bugs or issues present in dynamically generated/modified code to the originating code that did the generation/modification, as often this is where the source of the bug and/or attack originates from[26][20]. This entails a more complex reasoning about control and data dependencies that as far as we are aware, is not present in existing work. End-to-end analysis capabilities, or the idea that one can, with an analysis starting at the dynamic code, work backwards towards the originating code through control/data dependencies, requires that one maintain a causal relationship between the code that does the generating/modification and the generated/modified code itself.

One key step towards recognizing these causal relationships entails obtaining a generic program representation for code that includes both the dynamic code and the code that does the dynamic generation/modification. Often this is done by building a control flow graph (CFG), but this only gets us part of the way towards building an analysis toolkit capable of end-to-end analysis in the presence of dynamic code. As we will show, existing methods of constructing CFGs prove inadequate for end-to-end analysis with dynamic code. We propose a modification to the traditional CFG representation that allows for better reasoning about dynamic code not possible with other CFG representations. In addition to the program representation provided by a CFG, one will need to maintain an explicit record of code-generation dependencies (hereby referred to as codegen dependencies), or the dependence introduced by one piece of code writing another during runtime. As far as we are aware, existing approaches in data flow analysis do not track this dependency. By combining both the idea of maintaining codegen dependencies along with our novel CFG representation, we provide the ability to perform end-to-end analysis. To demonstrate this, we used both our CFG representation along with the tracked codegen dependencies to implement backwards slicing. We then used our backward slicing implementation to analyze bugs and security exploits that appeared in dynamic code, tracing the issue back to the originating code in an end-to-end analysis that, as far as we can tell, is not possible with existing approaches. We also used our program representation along with forward taint analysis to examine cases of using dynamic code to hide environmental triggers, demonstrating the applicability of our analysis capabilities in a forward direction as well.

End-to-end analysis in the presence of dynamic code allows for several interesting applications wherein we believe our analysis toolset can serve as a solid foundation. One of the primary motivators in this project is analyzing JIT compilers when the resulting dynamic code produced is incorrect (either due to malicious exploits or because of bugs in the JIT compiler). End-to-end analysis allows one to work backwards from the buggy JIT-compiled code to the JIT compiler itself, allowing one to identify the root cause of the problem when performing debugging and/or security analysis. Data only exploits highlight such a case where one needs to be able to reason about corruptions that occur to bytecode before the bytecode gets JIT compiled into a malicious payload. Reasoning about obfuscated code that utilizes dynamic code unpacking would be greatly aided with an end-to-end analysis that kept track of the different phases of the program as it unpacked. Symbolic execution in the case of dynamic code can take advantage of end-to-end analysis by being able to reason about dependencies between the code generator and the dynamic code when one is attempting vulnerability detection, multi-path dynamic analysis, or test case/exploit generation. For instance, when attempting dynamic analysis of malicious code that uses environmental triggers to determine behavior, the payload can be hidden under dynamic code generation, with malware producing different samples of code based on the current environment.

At this point we have described dynamic code and its many occurrences both in legitimate large scale software systems and in malware. Orthogonally, we have discussed the idea of end-to-end analysis and its value when applied to cases of dynamic code. Chapter 2 details some background followed by a discussion in chapter 3 of our approach to provide a program representation capable of providing end-to-end analysis. Our evaluation in chapter 4 entails examining several end-to-end analysis made possible through our program

representation. We conclude by looking at existing work in this area in chapter 5 and then summarizing the work and contributions this paper makes in chapter 6.

# 2 Background

In this section we discuss some of the fundamental program analysis techniques used in our approach, providing first a description of how an analyst would potentially use these techniques before going into more depth in each technique in the following subsections. Let us start with a scenario where analyst Alice has a buggy program that seems to crash randomly. Alice could first consider collecting a trace of a buggy execution, which allows her to deterministically replay the same crash without any elements of randomness. Gathering a trace also allows Alice to observe the program state at a fine grained level, with access to programs state such as registers in use and memory locations accessed/modified. At this point Alice could could take several different approaches towards debugging her program. She could first decide to rebuild the control flow graph (CFG) of the program, which allows her to observe how control transfers from instruction to instruction and potentially identify the path of instructions that leads to the buggy invocation (i.e. finding a point where control transfer is mistakenly not taken, such as at the end of loop that leads to a out of bounds access in an array). Alice could also observe control transfer at the procedure level, allowing for a high level view of which functions are taken before the program crashes. This higher level view could help with narrowing down where in the source the bug resides.

Alice could also attempt to work backwards from the bug. Once the control flow graph is built, Alice could use backward slicing to attempt to narrow down all of the instructions that play a role in causing the bug. One could imagine a buggy program with two completely independent functions. When attempting to debug either of the two functions, its safe to rule out having to investigate the other independent function since it plays no role in causing the bug. Slicing automates determining what is and is not important in causing the bug. Control dependencies are one aspect of how slicing works, allowing one to determine how instructions influence each other through control transfers. Backward slicing also takes data dependencies into account, allowing one to observe how one instruction influences the data used by another function. Data dependencies can be built up by tainting (marking) a given value and observing all the instructions that come into contact with the taint. Once an instruction is known to interact with tainted values, the taint can be propagated by tainting any of the other values used by our interfering instruction. Alice could try to slice backwards from the point of the crash, attempting to determine all of the instructions responsible for the crash, using both the control dependencies present in the control flow graph and the data dependencies built up from taint analysis. Finally the notion of taint analysis opens the door to symbolic execution, in which multiple program states can be explored by marking a value as symbolic and branching multiple executions on all possible paths that depend on that symbolic value. For instance, If Alice suspects that one of her program inputs is causing a crash, she could determine what the problematic input is by marking the input value as symbolic and observing which program state leads to a crash, looking at what the symbolic input's value was in that particular program state.

We have given a brief overview on how each of the program analysis techniques used in our approach could be used in a stand alone application by an analyst. We will now delve into more detail with each technique and discuss the role of each individual technique in the context of our entire approach.

## 2.1 Trace Based Analysis:

Our approach is based on trace-based dynamic analysis [2], where a trace of the execution of a given program is collected to perform the analysis upon. Trace based dynamic analysis entails instrumenting a program at runtime to collect information about the execution of a program, storing this information in a trace. Trace based approaches allow for one to record information like the sequence of instructions executed, registers written/read from, and information about the usage of memory during runtime. Potential downsides of trace based analysis include large trace sizes and overhead incurred in replaying the trace for analysis purposes. Often trace based analysis have to record substantial parts of the program state for each instruction executed/traced. While this leads to large space and time overheads, it does provide an incredibly detailed view of the program, with the entire program state easily accessible at any given point in the program. This highly detailed view of the program serves as the foundation to the rest of our work.

## 2.2 Dynamic Taint Analysis:

Another key aspect of our approach is dynamic taint analysis, which allows one to reason about data flow within a program [18]. Dynamic taint analysis is a method to "taint" (essentially mark) specific values within a program as it executes. Any other data values that come into contact with a tainted value will propagate that taint, thereby tracking the spread of data values as the program executes. Through taint analysis one can collect data dependencies within a program. Taint analysis comes into play both in our slicing approach and in how we determine when one piece of code writes/modifies another.

## 2.3 Backwards Slicing:

One of the methods we use to evaluate our program representation is backwards slicing, specifically because it allows us to demonstrate what end-to-end analysis can look like in the presence of dynamic code. Dynamic program slicing is a technique to reduce the size of a program while still maintaining its given behavior for an execution of that program [15]. A slice is a subset of all instructions that impact the instruction we are taking the slice from. Backwards slicing holds that for a given point in the program, all other previous instructions that play a role (i.e. control or data dependence) should be included in the slice. All other instructions can be disregarded, thus reducing the total amount of code one has to analyze. We use backward slicing to demonstrate the difficulties inherit with attempting to apply traditional program analysis techniques towards dynamic code and for demonstrating the value of end-to-end analysis.

## 2.4 Dicing:

Dicing, or differential slicing, is a technique that utilizes two (or more) slices taken from different executions of a program to help refine the precision of a given slice[12]. A dice allows one to reduce slice sizes even more by comparing redundant factors between two slices, one of which is of a test run where the program behaves as expected and the other of a faulty execution. By taking a dice, one can remove the portion of the slice that appears in both the normal execution and the faulty execution since it is not among the factors of interest in the slice containing the faulty execution. We use dicing to help increase the precision in our slice.

## 2.5 Symbolic Execution:

We also look at the results of symbolic execution during our evaluation, specifically in the case of when dynamic code is used to create trigger based behavior. Symbolic execution is a means to explore execution paths of a program through the use of symbolic values, or symbolic expressions that can represent any arbitrary value [13]. Often malware can attempt to hide its payload if certain conditions in its environment are not met, i.e. through the use of environmental triggers. Symbolic execution provides the means to explore these potential triggers, allowing one to circumvent the malware writer's attempt to hide the malware's payload from further analysis. We use symbolic execution for the same reasons as backwards slicing, to demonstrate the difficulties encountered with dynamic code and to highlight the utility of end-to-end analysis.

## 2.6 Control Flow Graphs and Basic Blocks:

Control flow graphs (CFGs) are a program representation composed of a graph of basic blocks, each connected by edges representing control flow. Basic blocks are straight line sequences of instructions, in that once control starts at the top of a basic block, it will continue down the sequence of instructions in the basic block until reaching the end, where control flow jumps/transfers to a different basic block (represented by an edge). Control flow graphs serve as the foundation to many program analysis techniques, such as building control/data dependency graphs and form the backbone to our approach after we have collected a trace.

## 2.7 Interprocedural Control Flow Graphs:

Interprocedural control flow graphs are based upon the standard definition of CFGs with the added capability of representing control flow between functions by using a pair of pseudo-blocks (entry/exit) through which all control flow into and out of the function (calls/returns) are routed. Call sites within a function are usually linked to return sites through link pseudo-edges. The interprocedural control flow graph adds additional capabilities to our end-to-end analysis, such as the ability to construct a call graph.

## 2.8 Trace Based Interprocedural CFG reconstruction:

The approach to build CFGs from static analysis is a well known procedure and as such will not be fully described here. These statically derived CFGs are inadequate towards building CFGs in the presence of

dynamic code (which occurs at runtime) and thus we resort to using dynamic analysis to build CFGs from the instruction sequence derived from an execution trace of the program. We will briefly discuss the difference in steps required to reconstruct an interprocedural CFG from a trace:

### 2.8.1 Building Basic Blocks:

Basic blocks, following the same definition as found in the original description of a CFG, hold sequences of instructions up until we find a control transfer. Thus building basic blocks from a sequence of executed instructions (i.e. as recorded in an execution trace) entails scanning over every instruction in the trace and appending each to the end of the current block being built as long as the previous instruction is not a control transfer instruction. We create a new basic block each time we see that the previous instruction is some sort of control transfer instruction, continuing construction with our new block and linking the two blocks with an edge. Control transfer to an existing block can split our existing block if control transfers to the middle of the existing block.

### 2.8.2 Handling Procedures:

By keeping track of call and return instructions, we can build an interprocedural control flow graph, wherein each block belongs within a function. Functions within interprocedural CFGs have an entry/exit pseudo-block where all edges from calls and returns into that function are routed. We link the call/return sites of functions with pseudo-edges (link edges) and by maintaining our own call stack.

### 2.8.3 Handling Multiple Threads and Exceptions:

Handling multi-threaded programs entails tracking the construction of the interprocedural CFG across all threads, with each update happening according to the state of the current thread. By state, we mean that we separately maintain the call stack, previous instruction seen, current function being reconstructed, etc. for each thread (i.e. 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). Handling exceptions entails delaying the reconstruction process until we get the first instruction after the exception. We do not include the exception itself alongside instructions within a basic block.

# 3 Approach

Here we will first discuss the challenges inherent to trying to build a program representation for dynamic code as well as some motivation for trying to build a program representation in the first place. We will then lay out our initial approach and define several of the terms used in our approach. Note that our initial approach follows a naive yet (hopefully more) intuitive course. We then seek to improve this approach, allowing for one to account in overhead incurred with overlapping parts of our program representation. Finally we discuss

slicing in the context of using it for end-to-end analysis of dynamic code, ending this section with a discussion on improvements to slicing precision.

### 3.1 Challenges and Motivation:

To better motivate the challenges behind trying to build a program representation in the presence of dynamic code, we will observe a simple case (see figure 1). The example entails repeatedly reading in some sensitive value before doing some operations and then clearing out the sensitive value. In this example, we can observe that there is a patching function that dynamically modifies the instructions within the function "zeroOut". If that patching function fails, either due to a bug in the code or because of some malicious attack, then the "zeroOut" function will not behave as expected. In the example below we will observe what happens when the second call to the patching function is buggy, wherein the instruction to zero out the given value in "zeroOut" is replaced with a NOP. If one was to try and observe the information flow to understand why the second call to "zeroOut" leaked sensitive data (i.e. failed to overwrite a password that the user had entered and expected to be cleared out) we can note several things.

First, we can see that there are multiple versions of the function "zeroOut", each with differing instructions after the result of patching. When building a program representation for this program one cannot just overwrite the previous program representation built for the function "zeroOut" with the newly patched version, as this erases any chance of being able to reason about how the previous program from before the modification interacts with the new version of the program from after the dynamic modification/generation. In this case the final version of the code for "zeroOut" happens to be the correct version, after "patchCorrect" gets called. Any program representation that erases previous versions of zeroOut upon finding a new version from dynamic modification/generation will no longer have the buggy version. If we wanted to observe all instances in the program representation where the user's password was cleared out (both properly and improperly) then both versions of the "zeroOut" function must be present. The program representation for this code should be able to account for the arbitrary many different versions of the instructions encountered within "zeroOut".

Next, we observe that traditional control/data dependencies do not always catch a relationship between dynamic code and the code that modifies/generates the dynamic code. We can see in this scenario there are no control dependencies between the modified code and the code that does the dynamic modification, as neither of the two "patch" functions have to be called for the "zeroOut" function to run. Furthermore, since traditional data dependencies only check the set of memory locations/registers read from and written to by instructions, we can see that there are no data dependencies between the modified code in "zeroOut" and the patching code (i.e. both the NOP and the clearing of a memory location can execute without having any data dependencies). If one wanted to work backwards to uncover that it was the patching code that ultimately leads to the leak of the user's password then there needs to be some sort of relationship between the dynamic code and the code that does the modification and/or generation. Since traditional control/data dependencies do not always catch a relationship between dynamic code and the code that does

the generation/modification thereof, we instead need some explicit handling of the dependencies that form from once piece of code modifying/generating another, such that dynamic code is considered dependent on the code that generates it.

These two observations highlight two main ideas unique to trying to build a program representation in the presence of dynamic code. First among these is the notion that one's existing program representation can change drastically in the presence of dynamic code. The challenge arises upon encountering dynamic code of what to do to update an existing program representation such that both versions of the program, from before and after the dynamic modification, are properly represented. On top of dealing with the fundamental changes dealt by dynamic generation/modification to an existing program representation, there arises the concept of linking the code that does the dynamic modification/generation to the dynamic code itself in some sort of relationship, i.e. a dependency. It is these two challenges of handling changes to an existing program representation and handling dependencies from code generation that inform our approach towards building a program representation to handle dynamic code. Now that we have presented the challenges motivating an improved program representation for dynamic code we will discuss/define the underlying concepts behind our approach in more detail.

**Pseudo-code:**

```
function patchGood(){
   //code to correctly modify instruction bytes at point A in ZeroOut (i.e. mov mem_location, 0)
}

function patchBuggy(){
   //code to incorrectly modify instruction bytes at point A in ZeroOut (i.e. to a NOP)
}

function zeroOut(int *arg1){
   *arg1 = 0      // Point A: patchBuggy turns this to a NOP
}

function main(){
   int secret = input()
   zeroOut(secret)
   print(secret)  // No Leak

   secret = input()
   patchBuggy()
   zeroOut(secret)
   print(secret)  // LEAK

   secret = input()
   patchGood()
   zeroOut(secret)
   print(secret)  // No Leak
}
```

Figure 1: Dynamic Code Example: Pseudo code to demonstrate our motivating example of dynamic code

## 3.2 Definitions and Approach:

The key concept behind our approach of a program representation capable of properly representing dynamic code is the need to preserve multiple versions of a given program's representation, as dynamic code can lead to any number of arbitrary changes to an existing program representation, even if the dynamic code is supposed to be semantics preserving (i.e. in the case of bugs in a JIT compiler). Furthermore, some relationship between any two consecutive program representations must be maintained to aide in reasoning how we transitioned from one program representation to the other. Therefore, if one wants to reason about two different versions of the program representation, they must keep information about both program representations along with an indication of how the instructions in one program representation influence the instructions in the other program representation. We will now lay out our initial approach and define the concepts that will be put in place towards solving the challenges listed above.

### 3.2.1 Phases:

To properly represent all versions of a given program upon discovering dynamic modification we preserve the existing program representation and construct a new instance of the program representation to house the dynamic code, resulting in the notion of 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 $\varphi_i$, none of the locations written by the instructions in $\varphi_i$ is part of any instruction executed by $\varphi_i$. Each $\varphi_i$ is referred to as a *phase*. Execution begins in phase $\varphi_0$ with the program's initial code. When the first dynamic instruction is encountered, we switch to $\varphi_1$. Execution continues in $\varphi_1$ (including other instructions that may have been created or modified in $\varphi_0$) until an instruction is encountered that was modified in $\varphi_1$, at which point we switch to $\varphi_2$, and so on. 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 $S$:

$$instr_locs(S) = \bigcup_{I \in S} instr_locs(I)$$
$$write_locs(IS = \bigcup_{I \in S} write_locs(I)$$

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 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*
  $write_locs(T[m:j]) \bigcap instr_locs(T[m:j]) = \emptyset\}$.

A program with no dynamic code will appear as a single phase ($\varphi_0$) and will only have one instance of whichever program representation one is using for analysis. By using phases we handle the fundamental changes to an existing program representation wrought by dynamic code by keeping all versions of a given program's representation throughout all cases of dynamic modification/generation. Phases allow for one to reason about different versions of the program, each separated by acts of dynamic code generation/modification.

### 3.2.2 Codegen dependencies:

Along with the creation of new phases, one must define some sort of relationship between dynamic code and the code that does the dynamic generation/modification. We introduce a new type of dependency, which we call a codegen dependency. Codegen dependencies will be necessary as a primitive to relate together the code that does the generation/modification to the dynamic code itself, as one can see in the patch example above where control/data dependencies were inadequate. By observing where an instruction writes to and where instructions are executed from, one can determine both the code that does generation of dynamic code and the dynamic code itself. Code that executes from a location that was written to is considered dynamic code, whereas the code that does the writing is considered the generating code. By linking together this dynamic code as being codegen dependent on the generating code, one can build up a complete collection of codegen dependencies across all phases. Formally we define codegen dependencies, or one instruction being codegen dependent on another, as follows:

**Definition 2** *Given an execution trace $T$ for a program, 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., $J$ is the instruction that most recently modifies loc before $I$ is executed.*

While codegen dependencies resemble data dependencies in some ways, it is different 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 $I$, then $J$ can change the nature of the computation performed by $I$. For example, in the x86 ISA the arithmetic instruction *bitwise or* (opcode: or; encoding: 0x0c) can, by flipping a single bit, be changed to the control transfer instruction *jump if equal* (opcode: je; encoding: 0x0f).

### 3.2.3 DCFGs:

This process of discovering when an instruction has written/modified another instruction and the resulting data structure from that discovery builds the core concept of our program representation. For instance,

one way to implement our program representation is with what we call the DCFG (Dynamic Control Flow Graph). The DCFG is a modified interprocedural CFG. More formally:

**Definition 3** *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 $\phi_i$. 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 $\varphi_i$ and the first instruction of $\varphi_{i+1}$.*

Traditionally, CFGs identify instructions with the address at which those instructions occur. With dynamic code, multiple versions of an instruction can live at the same address. The traditional definition of a CFG cannot account for both versions of an instruction. For our initial approach to building a program representation for dynamic code, the discovery of dynamic code entails creation of another DCFG, wherein the original instruction lives in the first DCFG and the new version of the instruction lives in the newly created DCFG. These two DCFGs then are linked together in a manner that indicates the discovery of dynamic code leading from one phase to the next. The DCFG forms the basis for how we implemented our program representation and we will first briefly discuss an example, followed by a discussion on our implementation of DCFGs. Later we will discuss the improvements that one can make, such that redundancy between phases is reduced.

For this example, we will be looking at the DCFG that forms from the case above of code that does a several rounds of self-modification to a single function "zeroOut", using the functions "patchGood" and "patchBuggy". Upon encountering the dynamic code in "zeroOut", we must deal with the fact that we have an entirely different function being executed than the function we had seen before. Our initial program representation handles this case of self-modification as three phases, one phase with a DCFG as it existed up until we encountered the first case of dynamic code (in this case up until we discover that "zeroOut" was modified), one phase with a DCFG for after we detect dynamic code to house the rest of the now modified program (in this case, the incorrectly patched version of "zeroOut"), and one final phase with a DCFG for the correctly patched version of "zeroOut". See figure 2 for an illustration of the process when we find the first case of dynamic code (when zeroOut is incorrectly patched).

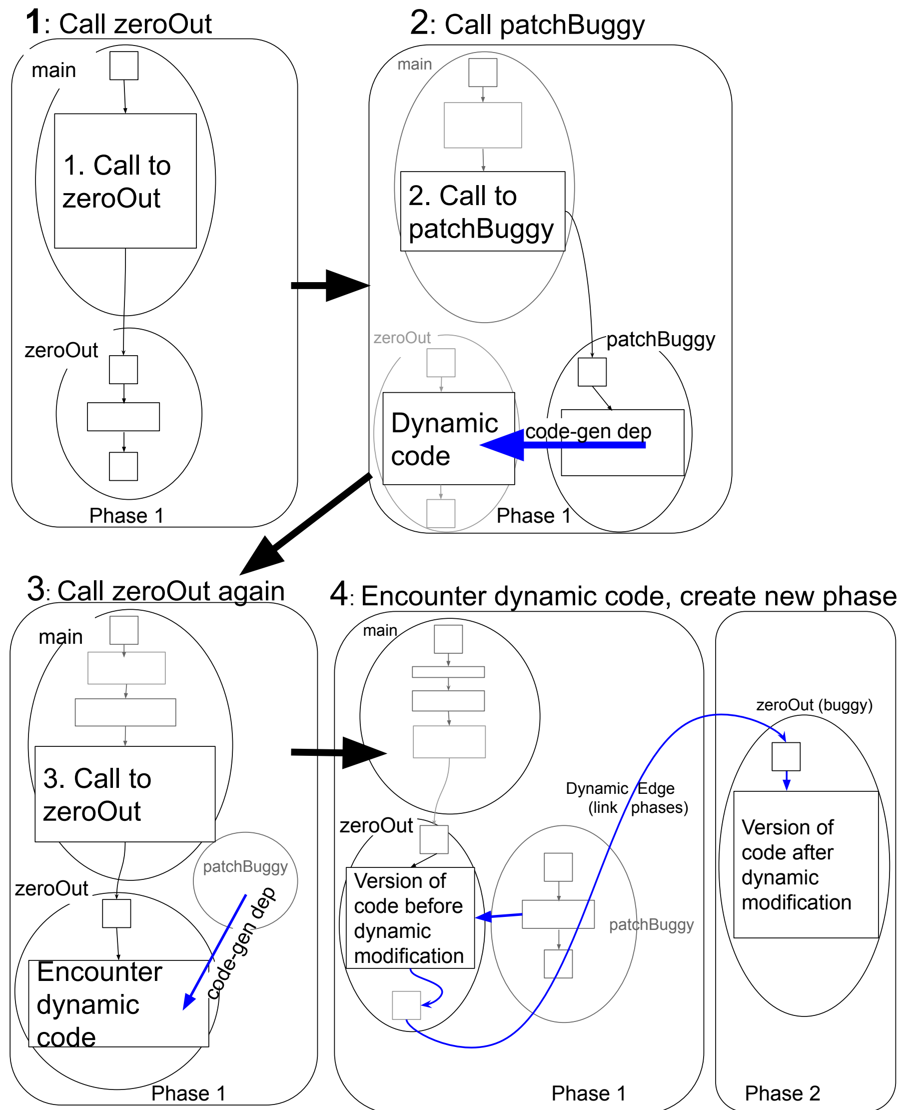


Figure 2: Encountering Dynamic Code and Creating a New Phase: Steps 1-4 each show the DCFG as a sequence of instructions are executed (i.e. per each step). Step 1 is the intial call to zeroOut. Step 2 has the dynamic modification to zeroOut. Step 3 has us discover the dynamic modification in zeroOut. Step 4 shows the final DCFG, now with two phases upon encountering the dynamic code in zeroOut.

### 3.3 Implementation:

We collect instruction level execution traces of a given program as it executes, wherein our tracer keeps track of information like the sequence of instructions executed along with the register/memory contents accessed by each executed instruction. We use this collected trace to go from list of instructions executed to our program representation, i.e. the DCFG with codegen dependencies. We will discuss our approach in terms of interprocedural control flow graphs, but our initial approach can be used in program representations independent of interprocedural CFGs.

Our initial program representation is based on a series of phases, each with one DCFG. Each DCFG appears in exactly one phase. Each phase has a reference to the first function encountered when constructing the DCFG for that phase. The DCFGs in two consecutive phases are linked together with a pseudo-edge that indicates where in each interprocedural CFG we encountered dynamic code and where in the next phase we ended up. This structuring of phases allows for one to keep both the code that does the dynamic creation/modification and the dynamic code itself within one overall program representation. It also allows us to reason backwards through a series of dynamic modifications, keeping things like control and data dependencies intact. Our implementation of building DCFGs follows the approach used to build standard interprocedural CFGs from a trace (see the Background section). Where things differ is when we encounter dynamic code and how we handle constructing a new phase. See figure 3. for a visualization of our DCFG representation. Algorithm 1 lays out the process for building DCFGs.

For the purposes of this paper, we observe two types of dynamic code, code created at runtime and code modified at runtime. We label code created at runtime as dynamic generated, whereas we label code that is modified at runtime as dynamically modified. We define the following predicate:

**Definition 4** *is_phase_transition(Ins, DCFG): is true in two cases:*

- *Case 1: We have seen Ins in our DCFG before but the instruction bytes for Ins have changed*
- *Case 2: We find that the address range from Ins.address to Ins.address + Ins.lengthOfInstructionBytes is tainted (has been written to).*

The difference between the two types of dynamic code boils down to whether we have previously executed the instruction before observing that it was dynamically changed, as dynamically generated code will not have existed before whereas self-modifying code will entail alteration of instructions we had previously seen. When building our DCFGs, we are able to detect both types through examining both the existing DCFG and through using taint analysis. Code that is modified at runtime will result in an instruction whose address has already been seen/incorporated into the CFG, but whose instruction bytes have changed. Dynamically generated code can be found by tainting memory writes when replaying the execution trace and observing whether any of the instruction's bytes are tainted when being read from the trace. One can check for taint among the instruction's bytes by looking at the range of memory from the instruction's address to the instruction's address plus the length of the instruction's bytes. Both methods of detection occur during the reconstruction of the DCFG from the execution trace. Theoretically an analyst could cut down on potential

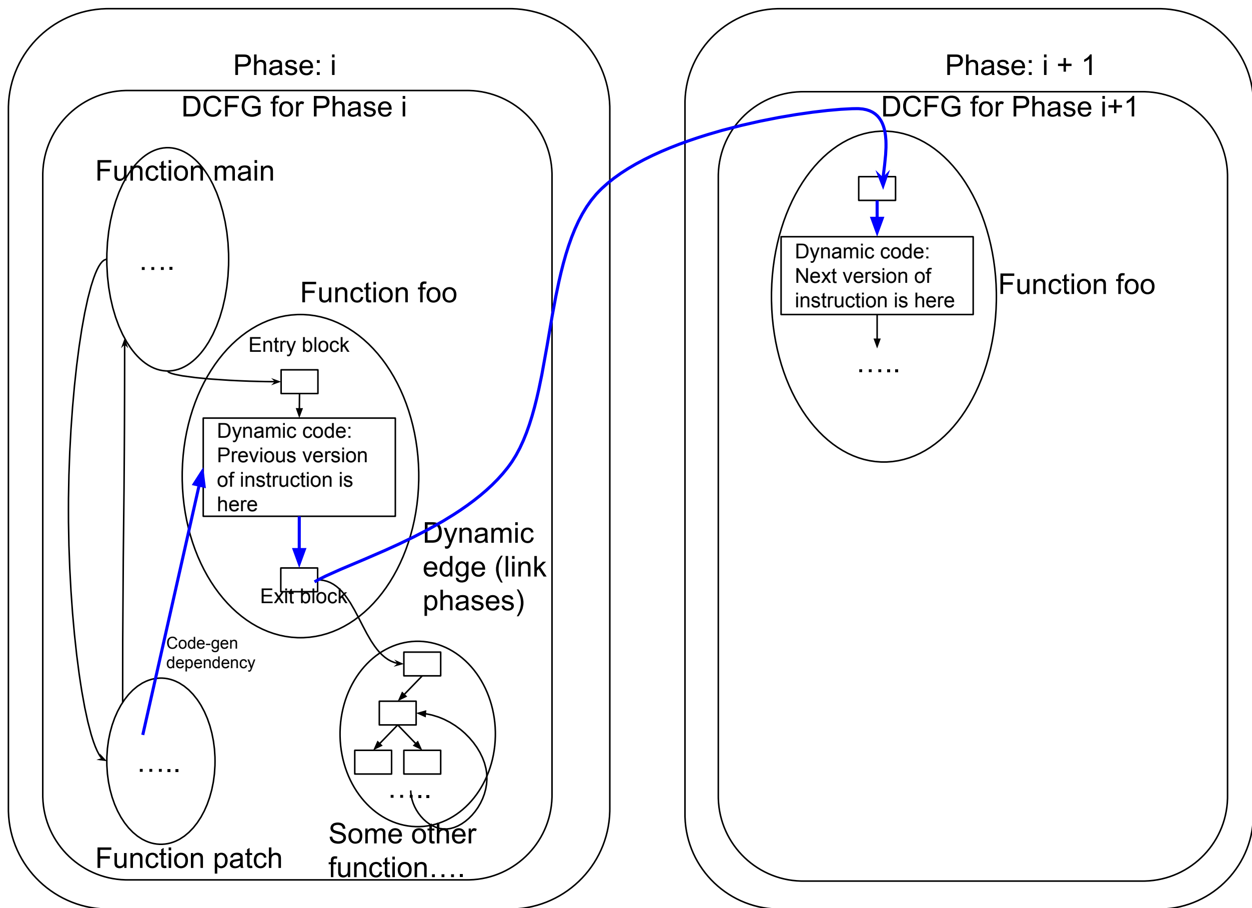


Figure 3: DCFG Representation: Two phases are shown here, with the first phase demonstrating the dynamic code in function "foo" and the dynamic modifier code in the function "patch". The second phase demonstrates how we represent both versions of foo (from before and after the dynamic modification)

overhead if they knew which type of dynamic code they were attempting to analyze, as dynamically modified code can be processed without resorting to taint analysis. In practice we handle dynamically modified and dynamically generated code the same, and as such both terms will be used interchangeably throughout the rest of the paper.

Finding that is_phase_transisition (Definition 4) is true for a given instruction during the construction of a DCFG from a trace leads to the creation of a new phase, which terminates building up the current DCFG and entails creating a new DCFG and linking the two DCFGs. Within our initial approach that link appears as an edge (marked as "dynamic" to differentiate it from other edge types) that links blocks within two different interprocedural control flow graphs. A new function is created for the first instruction added into a new phase. In the case of creating a new phase for dynamic code, the entry block of this new function is where the linking dynamic edge between phases connects to. The linking dynamic edge's source starts from the block where the dynamic code is found and is routed through the exit block of the function where we found the previous version of the dynamic code (creating a new exit block if one doesn't yet exist).

**Algorithm 1:** DCFG Construction

**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 = ∅ // Start with an empty DCFG
5  |   φ ⟵ ∅ // Start with an empty first phase
6  |   W = ∅ // Written locations starts empty
7  |   G_φ = (∅, ∅); add G_φ to G // DCFG for first phase is empty
8  |   // Walk through trace
9  |   for i = 0 to len(T) − 1 do
10 |   |   // If we find a new phase then initialize it
11 |   |   if instr_starts_new_phase(T[i], W) then
12 |   |   |   φ += 1 // Next phase id
13 |   |   |   G_φ = (∅, ∅); add G_φ to G // Add new phase's empty DCFG to G
14 |   |   // See Sec. 2.7 for how to process T[i]
15 |   |   process T[i] in the context of G_φ
16 |   |   // Connect DCFG's across phases
17 |   |   if instr_starts_new_phase(T[i], W) then
18 |   |   |   add a dynamic edge from last block of G_{φ−1} to first block of G_φ
19 |   |   |   W = ∅ // Reset write locations
20 |   |   W ⟵ W ∪ write_locs(T[i]) // Update write locations with current instruction
```

The dynamic edge is then routed through the entry block of the new function to the block where the newer version of the dynamic code resides. Finding dynamic code in the middle of a block entails splitting that block and linking the two new blocks with a fall through edge. See figure 2 for a visualization of this process.

Upon building a new phase we erase the taint that we have built up and start over with tainting memory operands. Phases only get built for each instance of dynamic generation/modification we encounter, hence the resetting of the dynamic taint we have applied. Two dynamic instructions next to each other do not warrant two separate phases if both were dynamically created/modified in the same phase.

### 3.4 Improvements with Phase Overlap:

One may note that DCFGs, as they are currently described, lead to duplication/overlap of components between phases. Our final implementation merges phases through the use of phase IDs, such that every component in a DCFG is labeled with the phase(s) it belongs to. Traversal through a DCFG then can only

occur through the components labeled with the same phase ID, or through a dynamic edge. Multiple versions of an instruction at a single address are stored together. Blocks are thus traversed by checking the phase ID of instruction iterated over, ensuring that one stays within the correct phase during a given traversal. Merging phases leads to several complications, which we will now discuss:

### 3.4.1 Overview:

Our initial approach of creating a new DCFG to house each phase leads to overlap between phases (i.e. shared basic blocks/functions that could be combined). We reduce this redundancy by instead adding onto the same DCFG across phases, allowing for common components across phases to be shared. This entails handling several parts of the construction process differently.

### 3.4.2 Instructions:

Instructions are still identified through instruction addresses. For two instructions to be saved together, they must both occupy the same range of instruction addresses. Instructions that share the same address range can both be saved within the same basic block at the same location. Instructions that do not share the same range of instruction addresses (i.e. one extends further than the other) will be separated into two separate blocks (see section 3.3.4.2), one for each phase the instruction occurs in. Each instruction has a phase identifier associated with it, such that when iterating over a basic block, one can also iterate all instructions saved at the same location within a block to determine the path of instructions taken during a given phase.

### 3.4.3 Blocks and Edges:

Every block/edge contains a list of phases in which that block/edge resides. Every instruction within a given basic block must exist in one of the phases that the block resides in. This informs how we handle the splitting of a block when the instruction address range between two instructions (from two different phases) no longer lines up. Since the block would normally be considered whole in the two separate DCFGs formed in our initial approach, we utilize a new pseudo edge, which we call a ghost edge, to indicate that a block was split due to the mismatch of instruction address ranges between phases for a given block. An example of splitting a block due to address range mismatching between dynamic instructions is given below in figure 4. A ghost edge indicates that a block is whole for any of the phases found in that ghost edge's phase identifier list. Any time a block/edge is traversed in a new phase, we update the list of phase identifiers for the block/edge. A phase that splits a block normally (i.e. control flow to the middle of a block) produces a ghost edge for all other phases in which the block resides (since the block would be whole in the other phases had this phase not split the block). Finally, dynamic modification can cause the instructions within a block to continue past where the original block ends (i.e. by changing the control flow instruction at the end of a block to a series of nops). For this case we create a new block for the extra instructions appended to the block in the new phase and link the two blocks with a ghost edge.

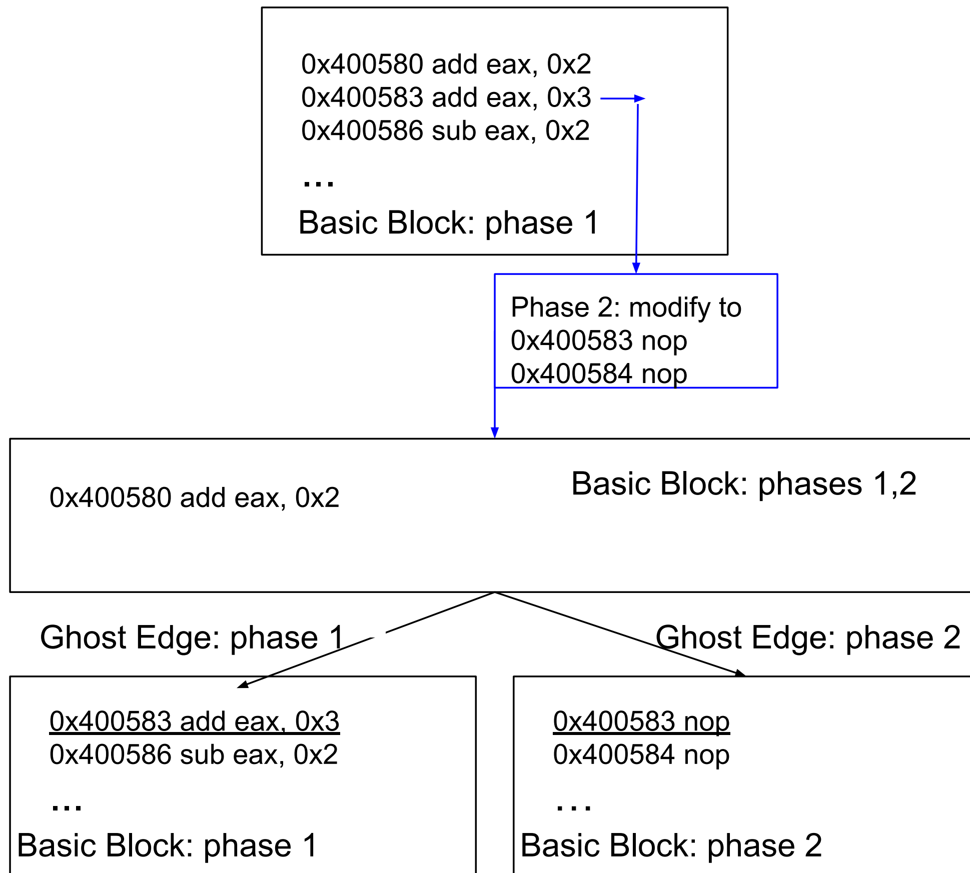


Figure 4: Overlapping Phases: Demonstration of how multiple phases can be merged with ghost edges, or edges that represent a block being whole for whichever phases are included by that edge. In this case the basic block has the instructions "add, add, sub, ..." for phase 1 and the instructions "add, nop, nop, ..." for phase two

## 3.5 Slicing:

We implemented backwards slicing for use in end-to-end analysis to demonstrate the effectiveness of our program representation in the presence of dynamic code, using both our modified interprocedural CFG and code-gen dependencies in our backwards slicing algorithm. The basis for our backwards slicing algorithm comes from Korel's work on computing dynamic slices for unstructured programs [15]. Korel's algorithm iterates over the instructions of a given execution, categorizing each as either contributing or noncontributing. A core component of Korel's algorithm is the notion of "last definition", wherein data dependencies are used to find last definitions of instructions. Contributing status comes from either being considered as part of the last definition chain formed from a previously discovered iteration of found contributing actions (the initial contributing action is the instruction one wishes to slice from) or from being unable to definitively rule out an instruction as noncontributing due to control flow. To update the slicing algorithm for end-to-end analysis we modified the computation of last definitions for a given instruction and made the use of our constructed DCFG to determine control flow information across phases. Our modification of last definition computation

makes use of codegen dependencies, with the code-generator being considered a last definition of the dynamic code itself. The pseudo-code for determining whether an instruction is a last definition of another instruction appears below:

```
LastDef(X, Y):
  if(Y writes to a source of X) or //
  if(Y writes to the instruction bytes of X)
    return true
return false
```

It is important to note that the notion of DCFGs plays 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, are unable to capture the effects of control dependencies and therefore are unsound with respect to slicing.

### 3.5.1 Improving our slicing results:

We had two primary strategies to increase the amount of instructions considered removable in the slices obtained from Korel's algorithm. The first was to implement the extended approach described by Korel, which improves the logic for classifying actions as noncontributing, leading to better slicing percentages. The second was to utilize the logic behind dicing to remove portions of the slice that are not of interest towards the faulty execution/bug. We do this through the placement of markers, in which we specify a point in the source code being examined (i.e. Javascript or Lua) at which to start recording slicing statistics in terms of actions to consider. The placement of the markers depends on where the user believes the bug to reside (i.e. placement of the marker closer to the potential bug will yield better slices). Often JIT bugs occur after a function has been optimized. As such, one can place a marker before the function get optimized to get a slice that contains the optimizing instructions within the JIT compiler that lead to the buggy function. We found that these two improvements helped drastically improve our slice sizes.

# 4 Evaluation

To evaluate our DCFG we need to show its value in end-to-end analysis of dynamic code. We chose backwards slicing as well as a modification to forward taint analysis to demonstrate our DCFG's applicability in several examples of end-to-end analysis. We compared our backwards slicing capabilities to those provided by three state of the art dynamic analysis toolkits when slicing on multiple examples of dynamic code. We also compare our modified forward taint analysis in a similar matter, using environmental triggers to setup more examples of end-to-end analysis with which to evaluate our DCFG. To demonstrate the value of our DCFGs, we need to show that, without the improvements to program representation and the notion of code-gen dependencies, one is unable to properly reason about dynamic code in a successful end-to-end analysis. We

will first discuss how we set up our evaluation, including the specific steps we took to construct our DCFG. We then discuss the experiments themselves.

## 4.1 Experimental Methodology and Setup

The first step in building a DCFG is obtaining an execution trace using our tracing library, which uses Pin to instrument a given executable and write out the corresponding trace. We used Intel's Pin software (version 3.7)[16] for program instrumentation and Intel's XED[11] for disassembly (version 8.20.0). This trace is then accessed through a trace reader, which is used to iterate over the instruction trace with our DCFG construction tool. We also use our own tainting library during construction of our DCFGs. The tainting library is used to determine code-gen dependencies by tainting writes to memory and then, for each instruction we add into the DCFG, checking for taint in the memory region starting at that instructions address and ending at that instructions address plus the size of the instruction. At this point the DCFG is ready for use in aiding end-to-end analysis involving dynamic code. Our evaluations use backwards slicing and forward taint analysis with our program representation to demonstrate end-to-end analysis.

We ran our experiments on a machine with 32 cores (Intel(R) Xeon(R) CPU E5-4627 v2 @ 3.30Ghz) and 1 terabyte of RAM, running Ubuntu 16.04 with GCC version 5.4.0. Our initial experiments are based on simpler, easy to discuss cases of dynamic code. We then move to analyzing more complex real world examples of dynamic code, looking at security exploits and bugs within popular JIT compilers like Google's V8 or LuaJIT. In each case involving backwards slicing we tested three other dynamic analysis toolkits with backwards slicing implementations PinPlay (revision 1.29), angr (commit bd3c6d8 on githu), and Triton (build number 139) to see whether slices of dynamic code include the generating code as well. For these experiments, 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.

## 4.2 Initial Results:

The first set of tests we ran were to test end-to-end analysis in the case of simple dynamic code, i.e. very small synthetic cases of self modifying/patching code. These examples, while small, highlight the core concepts behind our program representation and will aid in further discussion of more complex examples. The two examples of dynamic code we used for our initial evaluation appear below in 2 forms; as source code and as a text representation of the x86 execution trace we recorded using PIN (see figures 4, 5). The first example has code that dynamically modifies the offset for a jump, turning a loop (negative offset) into a jump whose target is fall through code. In the second example, the function "AddTwo" has been modified while the code is running to instead add three to a given argument instead of the value two [3]. In both cases we have multiple phases produced (due to encountering dynamic code) along with code gen dependencies.

In the first example we find a case of dynamic code generation followed by multiple occurences of self modification wherein the bytes for the instruction are different from the first time we saw the instruction.

Looking at the text trace below we can see the dynamic code is at address 0x401017, which will now be referred to by its label "Dyn1". Instruction Dyn1 is codegen dependent upon the generating/modifying instruction at address 0x401011, which will now be referred to by its label "Gen1". Examining the trace reveals this dependency, where reads/writes are indicated by "R/W" respectively and memory operations are prefixed with a "M". Gen1 writes to the second instruction byte of the Dyn1, which is the offset for the jump. Each time we encounter Dyn1 after it has been written to produces a new phase and reveals a new codegen dependency between the Gen1 and Dyn1.

The second example demonstrates a single case of self modification, where the code being written to has already been executed before. Looking at the ascii trace for this example reveals the instruction at address 0x400580, which we will call Dyn2, getting written to by the instruction at address 0x4006d1 in the patching function, which we will call Gen2. Gen2 which adds one to the previous constant used for the addition with eax in Dyn2. As such, Dyn2 is codegen dependent upon Gen2. Our representation has two phases for this example, with a new phase being generated upon encountering the new version of Dyn2 after it is patched.

We used both of these examples to do an initial evaluation of our program representation. Using our backwards slicing implementation we were able to, in both cases, pick up the generating code when creating a backwards slice from the dynamic code (from the last instance of Dyn1 and Dyn2 for each example respectively). In both cases, the instructions that modify/write dynamic code can be directly reasoned about in our backwards slicing implementation from codegen dependencies. The presence of these codegen dependencies allows us to pick up the generating code in our backwards slice. For instance, the patching instruction Gen2 in the second example is explicitly found in the slice as part of a dependency on the patched instruction Dyn2 due to the codegen dependency, despite there being no data or control dependencies between the instructions Gen2 and Dyn2. Traditional dependencies alone are not enough to allow for backward slicing to pick up the code that does the generation of dynamic code when slicing backwards from the dynamic code. Furthermore, our slicing algorithm is able to use our DCFG representation to reason about the different versions of the instructions Dyn1 and Dyn2 for each example, respectively. For instance, in the second example, the original unpatched version of the Dyn2 is not found in the slice as its not relevant in terms of data/control/code-gen dependencies, whereas in the first example all the versions of Dyn2 are found in the backward slice due to it being a control dependency. In this first example, our slice would be incomplete without having all versions of dynamic code present (i.e. we must have each version of Dyn2, as it leads to the computation of the next version of itself).

We evaluated PinPlay, angr, and Triton with these two smaller examples of dynamic code. Based on our experiments, none of the other approaches were able to pick up the generating code within the backwards slices of the dynamic code for either of the two examples. For the first example we found that without the notion of code-gen dependencies and multiple representations of differing phases of dynamic code that all three other backward slicing implementations were unable to pick up Gen1, which modifies the bytes for Dyn1. These three slicing implementations did pick up all of the other correct non code-gen dependent related instructions in the slice. For the second example we found similar results, with the patching code not getting found in the slice produced by PinPlay/angr. For angr we found that their CFG data structure

| | Our Approach | PinPlay | angr | Triton |
|---|---|---|---|---|
| Example 1 | Y | N | N | N |
| Example 2 | Y | N | N | N |

**Key:**

Y: Picks up dynamic modification (generator) code from backwards slice of dynamic (generated) code.

N: Does not pick up dynamic modification (generator) code from backwards slice of dynamic (generated) code.

Table 1: Slicing: Precision on Synthetic Benchmarks

does not handle self modifying code, as there is no updates to the basic block even when Dyn2 is patched. PinPlay's CFG does seem to update the bytes for Dyn2, as the previous version of Dyn2 is no longer included in the list of basic blocks for the "AddTwo" function. As far as we can tell, the approach taken by PinPlay seems to overwrite the previous block when updating for self modifying code.

Note that slicing is one method we chose to demonstrate using our program representation for an end-to-end analysis in the presence of dynamic code. We have found that for simpler examples (i.e. cases without dynamic code), our slicing results tend to be comparable to the results from angr/PinPlay. Once we move onto more complex examples (i.e. Google's V8), we see that our slices are fairly conservative, often times only removing around 50% of the original program. There are several reasons for this. Firstly, we include library code within our end-to-end analysis, which is one area that greatly increases the overall size of the program retained in the slice. Another reason for the conservative nature of our slices is that adding the notion of code-gen dependencies into our slicing algorithm means a massive increase in number of instructions to consider in these more complex examples, as portions of the JIT compiler and interpreter that make up these complex pieces of software must now be reasoned about in our slice. For the purposes of our evaluations, we found that using our code-gen dependencies along with our slices was enough to narrow down and streamline our end-to-end analysis, i.e. identifying the source of malicious attacks or buggy JIT compiled code. The addition of markers/dicing further improved our results in terms of raw number of instructions contained in the slice.

Source code:

```
main:
.LFB0:
  .cfi_startproc
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  movq %rsp, %rbp
  .cfi_def_cfa_register 6
  xor %rax, %rax
L0:
  leaq L2(%rip), %rcx
  leaq L0(%rip), %rdx
  sub %rdx, %rcx
  imul %rax, %rcx
  leaq L1(%rip), %rdx
  add %cl, 1(%rdx) ###  <<< Gen1
  inc %rax
L1:
  jno L0 ###            <<< Dyn1
L2:
  nop
  .cfi_def_cfa 7, 8
  movq $60, %rax
  movq $0, %rdi
  syscall
  .cfi_endproc
```

Trace (truncated): Format: Address Function Dissassembly

```
        # Code that occurs before dynamic modification
        400fee main push …
        …….
        # Code that does dynamic modification
Gen1:  401011 main add … MW[401018]=dc
        …….
        # Dynamic code
Dyn1:  401017 main jno 0xffffffffffffffde
        # Repeated loop where offset changes in 401017
        …….
        # Final modification of 0x401017
Gen1:  401011 main add … MW[401018]=48
        …….
Dyn1:  401017 main jno 0x2
        …….
```

Figure 5: Example 1: Slice on last instance of 401017. Notice both versions of the instruction at 401017 from before and after the modification in the trace.

Source:
// Disclaimer: Source of this code: https://www.quora.com/C-programming-language-Can-you-write-a-C-program-to-demonstrate-a-self-modifying-code

```
// Code source: https://www.quora.com/C-programming-language-Can-
// you-write-a-C-program-to-demonstrate-a-self-modifying-code
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
int AddTwo(int input) {
  return input + 2;                           //<<<  Dyn2
}

void patchFunction() {
  int pageSize = sysconf(_SC_PAGE_SIZE);
  // unprotect the two pages of memory containing the code for
  // AddTwo() by rounding down the pointer to the nearest page;
  // set their permissions to read, write, and execute
  mprotect((void*) ((((long) &AddTwo) / pageSize) * pageSize),
           pageSize * 2,
           PROT_READ | PROT_WRITE | PROT_EXEC);
  // for each byte in the code for AddTwo(): if the byte is 2,
  // change it to 3
  unsigned char* addInstrPtr = ((unsigned char*) &AddTwo);
  // Change the byte used in addition
  *(addInstrPtr+12) = 3;                      //<<<  Gen2
}


int main() {
  int out = AddTwo(2);
  patchFunction();
  AddTwo(2);
  return 0;
}
```

Trace (truncated): Format: Address Function Dissassembly

```
       # Code before patch
       400576 AddTwo push ...
       .....
Dyn2:  400580 AddTwo add eax, 0x2
       ......
       # Place where patching occurs
Gen2:  40060e patchFunction mov … MW[400582]=03
       ......
       # Patched code
       400576 AddTwo push ...
       ......
Dyn2:  400580 AddTwo add eax, 0x3
       ......
```

Figure 6: Example 2: Slice on last instance of 400580. Notice both versions of 400580 from before and after the modification

## 4.3 Results For Complex Real World Examples:

While useful in illustrating how our program representation can aid end-to-end analysis with smaller examples of dynamic code, these smaller examples don't fully capture what analysis on real world software with dynamic code entails. In reality programs like V8 or Chakra both produce traces that are considerably larger. The first case of end-to-end analysis will entail exploit analysis, seeking to find the source of malicious shellcode. Our second case of end-to-end analysis will deal with bug localization in systems that utilize dynamic code. The final end-to-end analysis we will explore deals with detecting when dynamic program behavior is dependent on input, which can be used to detect trigger based malware. The goal for these end-to-end analysis will be to be to reason about dynamic exploits/buggy code in a manner that includes portions of the generating code that originate the bugs/security exploits. The results are shown in table 2.

### 4.3.1 Exploit Analysis:

The first security exploit we used to evaluate using our program representation for end-to-end analysis in real world software entails an Out-Of-Bounds (OOB) write to the JIT code pages[7] within Google's V8 javascript engine. The exploit allows for arbitrary execution of shell code, which the original author used to demonstrate launching xcalc. We modified the example so that the shellcode encounters a divide-by-zero exception and then collected an x86 execution trace using PIN. This modification was made to allow for the creation of an easily identifiable point to slice from (i.e. the divide by zero exception). We could have sliced from the point where xcalc launched but for the purposes of keeping our examples more concise we had the shellcode crash instead and sliced from that point. This reduces the size of the trace and the cuts down on the time in takes to find a point to slice from, but for all intents and purposes we could have left the example unmodified and done our analysis in the same manner. Once we had a trace we then constructed a DCFG and backwards sliced from the shellcode instruction immediately before the divide-by-zero exception. We used D8 to invoke the V8 javascript engine from the command line. Through the use of codegen dependencies, our backwards slice was able to correctly pick up the correct Javascript instructions that wrote the shellcode before it executed. Our backwards slice allows us to successfully narrow down the source of the exploit shellcode and to extract all of the instructions necessary to craft and execute the exploit shellcode.

The second exploit we examined is discussed in detail by Rabet[22]. 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.

The final example we investigated was with malicious LUA bytecode being used to escape a sandbox in

LUAJIT[19]. As in the previous example, the malicious program ends up writing shellcode, this time printing a simple message to the screen. We followed a similar approach as before, building a backwards slice using our DCFG and slicing from the shellcode itself (starting with the beginning of the NOP sled used in the attack). Our backward slice correctly picks up the code that generates the shellcode, again using code-gen dependencies along with our DCFG to successfully allow one to narrow down the responsible instructions for the exploit in an example of an end-to-end analysis.

We compared our backwards slicing results for these three real world examples with those derived from PinPlay and angr. Pinplay, Triton, and angr, all previously evaluated for their slicing capabilities, crashed when evaluating V8 and did not pick up the exploit code in the backwards slice for LuaJIT. At this point the backwards slice could be used for simply tracking down where and how the shellcode for this exploit is crafted, or for extending the analysis into other approaches, such as information flow analysis. Further analysis at this point could entail constructing a call graph from our DCFG/backward slice to understand the flow of functions that lead to the exploit or using information garnered from the backwards slice to constrain the given runs of symbolic execution down a specific path in the program. In any case, successfully obtaining these correct backwards slices (i.e. including the malicious generating code) for our exploit examples demonstrates performing end-to-end analysis for real world complex examples in a manner that is not possible with other dynamic analysis toolkits.

One concern that could be raised over these slicing results comes from trying to determine how much of the JIT compiler is actually included within the slices. A slice could be realtively small compared to the rest of the program, and yet contain all of the JIT compiler. This situation demonstrates a potential issue with our slicing metrics. To address this point we implemented markers, small code snippets that are unambiguously identifiable and semantically neutral, to help focus where we collected our slicing statistics. We placed the markers within the source code, roughly before the point where the JIT compiler lead to buggy dynamic code (i.e. before an optimization call that leads to a buggy function). By cutting out sections of the program which we know to not be of interest to the bug, we remove the potential for these sections to falsely inflate our slicing statistics. These markers help produce statistics that both indicate how much of the JIT compiler we include within our slices and help us produce more conservative slices, in terms of number of DCFG instructions within the slice. Our use of markers is akin to the practice of dicing, only here we exclude portions of the slice that are not of interest to us, i.e. everything before the buggy invocation of the JIT compiler. The results can be seen in table 3, in which the reduction in DCFG and slice size is shown for when markers are used to exclude parts of the program that are not interest. We generally get a 35-85% decrease in DCFG size and a 26-84% decrease in slice size, with slicing ratios that are fairly close to before we placed the marker (i.e. roughly 50%).

#### 4.3.2 Bug Localization:

We also used our backwards slicing implementation on our program representation to do end-to-end analysis for bug localization on three separate bugs [8][9][10] found within V8. Unlike the analysis of the security

| | Test program | Tracing | | DCFG Construction | | | | | Slicing | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $N_{trace}$ | $T_{read}$ | $N_{instrs}$ | $N_{blocks}$ | $N_{edges}$ | $N_{phases}$ | $T_{DCFG}$ | $N_{slice}$ | $T_{slice}$ | $\Delta_{slice}$ |
| *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)
$\Delta_{slice}$ : Fraction of DCFG removed from slice $= (N_{instrs} - N_{slice})/N_{instrs}$.

Table 2: Slicing: Performance

exploits mentioned above, these cases end-to-end analysis entail finding portions of the V8 source code that lead to buggy code generation, whereas the security exploits entailed working backwards to the Javascript source code that either wrote the shell code used in the exploit. To determine the effectiveness of our bug localization, we first manually analyzed each of the fixes prescribed in the bug reports to narrow down where in V8 the problematic code was. The next step was to identify where these buggy regions of code exist within the x86 trace recorded for a crashed execution. Finding the corresponding buggy code in the x86 traces is not a trivial task, as each trace can hold millions of instructions and trying to match up V8's source code to places within the x86 trace can prove quite arduous. Through manual analysis we are able to identify which portions of the x86 instruction trace correspond to the buggy V8 source code ahead of time.

With the corresponding buggy sections of the trace identified we then performed backward slicing with the goal of trying to pick up the buggy regions that incorrectly generated the dynamic code for each of the three separate bugs. The first bug arises due to OOB read produced by the bytecode analyzer when observing/optimizing switch statements with no case statements, leading to a crash when one executes the dynamic code. The second issue demonstrates a type confusion issue during optimization that leads to a crash after the dynamic code has been generated. The final bug has scoping issues with Javascript arrow functions that leads to a crash while executing dynamic code. In each case we sliced with the dynamic instruction preceding the exception/crash and then manually analyzed the resulting slice to ensure that the problematic areas described in the bug reports were found in our slice.

We 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. Testing the three dynamic analysis toolkits mentioned above produced similar results as to when we tried analysis on the V8 security exploit and thus we were not

| | | Original | | Dicing | | Improvement (%) | | |
|---|---|---|---|---|---|---|---|---|
| | *Test program* | $DCFG_{orig}$ | $slice_{orig}$ | $DCFG_{mk}$ | $slice_{mk}$ | $\Delta_{DCFG}$ | $\Delta_{slice}$ | $\Delta_{mk}$ |
| *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

$\Delta_{DCFG}$ : Improvement in DCFG size due to dicing $= (DCFG_{orig} - DCFG_{mk})/DCFG_{orig}$

$\Delta_{slice}$ : Improvement in slice size due to dicing $= (slice_{orig} - slice_{mk})/slice_{orig}$

$\Delta_{mk}$ : Fraction of $DCFG_{mk}$ removed due to dicing $= (DCFG_{mk} - slice_{mk})/DCFG_{mk}$

Table 3: Dicing: Performance

able to get the correct slices from these three other dynamic analysis toolkits. Table 2 and table 3 summarize the slicing results for the bug localization experiments, with and without markers respectively.

#### 4.3.3 Detecting Dynamic Program Behavior Dependent on Input:

Our final example of end-to-end analysis with our program representation entails trying to determine if dynamic program behavior is dependent on input. Often the task of checking whether program behavior is dependent on input is one of the first steps towards determining whether one is observing malware which hides its payload behind environmental triggers. Malware that use environmental triggers to hide malicious behavior works by only triggering an attack if the correct environmental conditions are met. Well known examples of this behavior can be found in malware that only executes its payload on a certain date or when certain keys are typed. Previous work [4] is based on analyzing static environmental triggers, wherein the instruction bytes that check for the correct environmental criteria are not dynamically influenced by the input. These approaches often utilize techniques like symbolic execution to try and trigger the correct environmental conditions obscuring the execution of malware. For our evaluation, we decided to look at the case where environmental conditions are used directly in the generation of dynamic code. We took the motivating example from David Brumley's MineSweeper, modifying it so that the input values are used to dynamically modify the check for executing the payload. The resulting code can be seen in figure 7. Note that the conditional that determines whether the "ddos" function gets executed is not dependent on the input in our example. Instead there is a code-gen dependency between the conditional for determining whether the payload executes and the patching instructions, which are data dependent on the input.

```
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 7: Environmental trigger source code based on a dynamic modification technique

Our implementation to detect whether dynamic program behavior is input dependent takes advantage of this key difference. We first collect an execution trace and build our program representation. Next, we taint the input source and then forward propagate the taint. Note that our taint propagation has been modified to account for code-gen dependencies, such that any writes to an instruction's bytes can get tainted. If, at any point, we discover that the taint propagates to an instruction that is code-gen dependent, we report that there is dynamic program behavior dependent on user input. Note that in our example the code hidden by the environmental trigger doesn't actually have to execute for us to determine an attempt at evasion. The reason for this has to do with our technique of using dynamic modification to hide the payload. Even if the conditions for the environmental trigger aren't met there is still dynamic modification to write 0 to the conditional hiding the payload. Note that this approach can be expanded to include control dependencies when trying to determine the impact of user input on dynamic program behavior, but for the purposes of this evaluation we stuck to data and code-gen dependencies. Our implementation is able to correctly determine if user input influences dynamic program behavior for the modified MineSweeper example. When tested with S2E, a symbolic execution engine, we found that the input values used to patch the "check" function get silently concretized and thus only the false path gets explored. This is an expected result, since exploring all possible writes to an instructions bytes could result in an explosion of possible paths. We also tested this

example with angr, and again we found that only the false path gets explored. We believe that symbolic execution alone is not enough to fully explore that paths in this program.

At this point we could extend our methodology described above to improve the total amount of paths explored by S2E by delaying symbolic execution when a symbolic value is used to write to a region of code. The key point here is that rather than trying to symbolically solve/explore all possible instruction bytes that can be written when generating dynamic code, we can instead mark the sources of the created/modified dynamic instructions for later exploration. For instance, in the MineSweeper example, rather than explore all possible values that could be written to modify the instruction bytes within the "check" function, we could instead mark the sources of the dynamically modified conditional (the "if(0)" in this case) for later exploration when we actually execute the "check" function. In this manner one could extend symbolic execution, using our program representation and code-gen dependencies to detect when user input influences dynamic behavior and then explore that dynamic code at a later stage after it has been written/modified. This example, along with being able to detect when user input influences the dynamic behavior of a program, demonstrates another case of valuable end-to-end analysis made possible by our program representation and code-gen dependencies.

As mentioned in Section 3.5, data dependencies alone may not suffice for reasoning about information flow in dynamic code. We demonstrate this using a variant of the previous example but uses an implicit flow to propagate the value of the environmental trigger; the implicit flow code, shown in Figure 9, is based on an example from Cavallaro [6]. To analyze this example, we compute a backward dynamic slice with the slicing criterion being the dynamically modified code location at the point where it is executed. This slice correctly includes the environmental triggers, indicating that the code modification is influenced by environmental triggers. The slice also includes the relevant portion of the implicit flow code, thereby demonstrating the utility of the control flow information obtained from the DCFG. In practice, one might use forward slicing from the point where triggers are evaluated to determine whether any dynamically generated code is in the slice. However, our point here is not so much one of the directionality of the computed slice, but rather to demonstrate the sophistication of analyses possible when reasoning about dynamic code using control flow information, available in DCFGs, together with codegen dependencies.

## 4.4 Experimental Results Summary:

To summarize, our evaluation seeks to demonstrate the utility of our program representation and code-gen dependencies when performing end-to-end analysis on dynamic code. Our initial evaluation results involve simpler examples of dynamic code, showing how backwards slicing on dynamic code, one potential end-to-end analysis, could benefit from both DCFGs and code-gen dependencies. We compared our results to three other dynamic analysis toolkits that implement backwards slicing, showing that, without the capability to reason about multiple versions of instructions or code-gen dependencies, one cannot obtain a correct backwards slice for end-to-end analysis in these simpler cases of dynamic code. We then demonstrated three cases of complex real-world end-to-end analysis, namely identifying the source of shellcode within security exploits

```
void patch() {
          ...
    int final = day_bits & mth_bits;
    // Implicit flow here
    volatile int x = 0, y = 0, z = 0;
    if(final){
        x = 1;
    } else {
        y = 1;
    }
    if(x == 0){
        z = 0;
    }
    if(y == 0){
        z = 1;
    }
    unsigned char* addInstrPtr = ((unsigned char*) &hide);
    *(addInstrPtr+11) = z;
}
```

Figure 8: Implicit Flow Example: Version of hiding an environmental trigger with both dynamic modification and implicit flow

dealing with remote code execution, bug localization within the V8 javascript engine, and detecting when user input influences dynamic program behavior. We made use of our backwards slicing implementation for the first two cases, seeking to show how both our program representation and code-gen dependencies allow for end-to-end analysis for these complex real-world examples that is not possible with other dynamic analysis toolkits. For the third and final case we implemented a tool to check for when user input influences dynamic program behavior by using forward taint propagation as well as our notion of code-gen dependencies. This translates directly to trying to determine if one is analyzing trigger based malware. We demonstrated that stock symbolic execution is not enough to fully detect or explore all possible paths when user input is used directly to create trigger based behavior. In each of these cases we discuss how one can expand the initial end-to-end analysis to provide even more valuable insight on software that makes use of dynamic code. We will now describe related work in this area.

# 5 Related Work:

Our approach makes use of several well known program analysis techniques. First among these is reconstructing a program representation, often a CFG, to help aide further techniques in deriving an understanding of the program being analyzed. We begin this section with a discussion on work that relates to this notion of constructing a program representation, comparing and contrasting existing work with what we have done. We then discuss related work in the fields of dynamic taint analysis, backward slicing, and program verification. We conclude this section with a review of related work in symbolic execution and in analysis of environmental triggers.

Our work in producing a program representation capable of reasoning about dynamic code is not the first to recognize the challenges in representing code that can change at runtime. The SE-CFG[1] demonstrates the capability to handle multiple versions of instructions through the use of their code-bytes data structure,

which allows for a single instruction to represent having multiple different versions of its own instruction bytes. While useful in producing a CFG that can represent self modifying code, we believe that the SE-CFG has some downsides that make end-to-end analysis with dynamic code not easily achievable. For instance, SE-CFG has no notion of code-gen dependencies, which means that dependency analysis where the only link between dynamic code and the generating code is the code-gen dependency would not be trivial with an SE-CFG. Furthermore no distinction is given to code that is created dynamically, whereas our use of phases to represent each subsequent discovery of dynamic code incorporates and makes explicit both dynamically created and self modified code. This means that the SE-CFG representation will not account for cases where the only instances of dynamic code are dynamically created. Finally, SE-CFGs can only be built for binaries created with specially patched tool chains. Exploit analysis often can only be performed on stripped binaries where one doesn't have access to source code. Tasks like trying to implement backwards slicing from dynamic code, i.e. for exploit analysis, or attempting to detect trigger based behavior hidden by dynamic generation/modification, are not easily achieved with an SE-CFG alone and for these reasons we deem that the SE-CFG is not enough to perform end-to-end analysis in the presence of dynamic code.

The approach taken in "Hybrid Analysis and Control of Malware"[23] also discusses building a program representation in the presence of dynamic code. The approach taken here identifies both self modifying code and dynamically created code during analysis, but in both cases the basic blocks in the CFG produced by K.A. Roundy and B.P. Miller are overwritten with the updated versions to include the new dynamic code. This differs from our goal, wherein we seek to provide a program representation that is capable of representing all versions of dynamic code across multiple phases of dynamic generation/modification. Furthermore, code-gen dependencies are also not accounted for in this approach. This, along with the lack of representing multiple versions of instructions, leaves this approach unsuitable for end-to-end analysis in the presence of dynamic code.

Korczynski and Yin discuss identifying code reuse/injections using whole-system dynamic taint analysis[14]. 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.

PinPlay[21], angr[25][27], and Triton[24] are all dynamic binary analysis frameworks that allow for CFG reconstruction. PinPlay's DCFG shares a similar approach in how basic blocks are constructed from an execution trace, but it appears that self modified code is overwritten in their DCFG representation and that there are no changes to PinPlay's DCFG data structure in the presence of dynamically created code. The data structure built by angr, CFGEmulated, is as of the time of writing, not capable of representing dynamic code. Triton also makes no mention of support for dynamic code. None of these approaches take code-gen dependencies into account and, as demonstrated in our evaluation, do not obtain the correct backwards slice for end-to-end analysis with dynamic code. TEMU/DECAF, two other binary analysis platforms, allow for whole system taint analysis. We attempted to test these systems to see if code-gen dependencies were accounted for when taint was propagated and found that they were not. We believe that none of these binary

analysis frameworks support end-to-end analysis with dynamic code.

Both Cai et al.[5] and Myreen[17] 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. Program verification is one of many potential goals for reasoning about dynamic code

S2E and angr both provide symbolic execution to allow for automated exploration of potential program paths. One potential use for path exploration is in detecting trigger based behavior in malware, as seen in Brumley's work[4]. We have found that using dynamic code to obscure trigger conditions, i.e. using the environmental values to directly write the code that performs the check for whether the trigger conditions are met, makes default path exploration with symbolic execution untenable, as one would have to explore all possible writes to instruction bytes. We believe that our approach towards detecting dynamic trigger based behavior by using code-gen dependencies can be used to extend, rather than compete with, symbolic execution.

# 6 Conclusion

Dynamic code has become commonplace in today's world, finding widespread use within JIT compilers, dynamic binary translators, and malware. Being able to reason about dynamic code for the purpose of program analysis becomes more and more vital as both the usage and potential security threats within software that utilizes dynamic code continues to grow. Often the first step towards program analysis is building a program representation.

We have found existing program representations are lacking when it comes to representing dynamic code, or code that is written to/modified at runtime, due to two main challenges that arise. First, one must be able to represent multiple versions of instructions after self modification or risk not properly representing all the instructions that execute for a given program. Second, there needs to be a direct dependency link between dynamic code and the code that generated/modified it, such that one can always determine where the source of a given piece of dynamic code is.

These two challenges have informed our approach, resulting in a modified CFG which we call the DCFG and a new type of dependency which we call a code-gen dependency.

## 6.1 DCFG and Code-Gen Dependencies

The DCFG incorporates the notion of phases, wherein each phase is a given program representation (in our case an interprocedural CFG) built up until an occurence of dynamic code that was generated by the current phase. A DCFG then is a series of phases, each an interprocedural CFG, linked together by pseudo-edges at the points where dynamic code is discovered. All versions of a dynamic instruction can be found across these phases, allowing multiple different versions of an instruction to exist within one overall program representation

and helping us solve the first challenge. We also use tainting during the construction of DCFGs to be able to both detect when we encounter an instruction that is writing to and/or generating another instruction. Detecting when an instruction generates another instruction setups a code-gen dependency, which explicitly links dynamic instructions to the instructions that generate them. Code-gen dependencies help us solve the second challenge, allowing us to track which instructions generate other instructions as a dependency. With both code-gen dependencies and DCFGs we are able to address the two challenges that come from trying to build a program representation for dynamic code.

## 6.2 End-To-End Analysis

We use code-gen dependencies and our DCFG program representation to directly aide in reasoning about dynamic code in what we call end-to-end analysis, or analysis where one can work backwards from dynamic code in a manner that incorporates the generating code as well. We built a backwards slicing implementation that makes use of our DCFGs and code-gen dependencies to evaluate our program representation's capability for end-to-end analysis. We demonstrated how existing dynamic binary analysis frameworks with backward slicing implementations were incapable of end-to-end analysis with dynamic code due to issues with underlining program representations or lacking code-gen dependencies. We first performed backwards slicing for the purposes of exploit analysis, attempting to narrow down the source of malicious shellcode executed through flaws in the JIT compilers for Google's javascript engine V8 and for LuaJit. Our backwards slices successfully contained the locations in the execution trace corresponding to the Javascript source code that writes the shellcode for both examples, whereas existing slicing implementations did not. We also evaluated our program representation for use in bug localization, successfully picking up backward slices containing the buggy regions of the JIT compiler that lead to a crash in the execution of V8 for three separate bugs. Finally we evaluated our program representation by trying to detect trigger based malware that directly utilizes environmental values when generating dynamic code to hide trigger conditions. We found that symbolic execution alone, used in previous approaches towards detecting trigger based behavior in malware, is ill suited towards this problem, due to potentially having to explore all possible writes to dynamic code. We instead used forward taint analysis and our program representation/code-gen dependencies, tainting input values and detecting when these input values are used to write to dynamic code (code that has a code-gen dependency).

The approaches used in our evaluations can be extended to even more complex and powerful analysis, i.e. aiding symbolic execution in exploring dynamic code paths or constructing call graphs for the series of functions that lead to an exploit/bug in software that uses dynamic code. In a similar fashion to how CFGs allow for more complex analysis techniques so too does our DCFG, as made evident by our extension of backwards slicing and forward taint analysis. By building a program representation capable of representing dynamic code we provide a foundation for future applications of end-to-end analysis. As far as we are aware, existing program analysis frameworks are unable to perform the end-to-end analysis described in our evaluation.

Our successful experiments with bug localization/exploit analysis on real world Javascript engines demonstrate our program representation's capability to allow for reasoning between dynamic code and the code that does the dynamic modification/generation. We show that DCFGs and code-gen dependencies enable one to discover the source of the vulnerability in the Javascript JIT compiler that allows for malicious shell-code to be executed. We are also able to demonstrate how our DCFGs and code-gen dependencies enable one to perform bug localization when a bug in the JIT compiler leads to buggy dynamic code. Our modification to forward taint analysis along with our program representation allows us to reason about environmental triggers that are constructed as dynamic code. Through our experiments we show that our program representation allows for complex end-to-end analysis to aid in the understanding and exploration of software which uses dynamic code.

# 7 References

[1] B. Anckaert, M. Madou, and K. De Bosschere. A model for self-modifying code. volume 4437, pages 232–248, 07 2006.

[2] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163. ACM, 2006.

[3] Abhilash Bhoi. Can you write a c program to demonstrate a self modifying code?, 2018.

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

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

[6] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[7] Oliver Chang. Exploiting a v8 oob write, 2017.

[8] Loki Hardt. Issue 794825: Security: V8: Empty bytecodejumptable may lead to oob read, 2015.

[9] Loki Hardt. Issue 794822: Security: V8: Jit: Type confusion in getspecializationcontext, 2017.

[10] Loki Hardt. Issue 807096: Security: Arrow function scope fixing bug, 2018.

[11] Intel Corp. Intel XED. `https://intelxed.github.io`.

[12] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *2011 IEEE Symposium on Security and Privacy*, pages 347–362. IEEE, 2011.

[13] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

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

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, June 2005.

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

[18] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. Citeseer.

[19] Mike Pall. Luajit, 2017.

[20] T. Park *et al.* Bytecode corruption attacks are real—and how to defend against them. In *Proc. DIMVA*, pages 326–348. Springer, 2018.

[21] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 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*, pages 2–11. ACM, 2010.

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

[23] Kevin A Roundy and Barton P Miller. Hybrid analysis and control of malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 317–338. Springer, 2010.

[24] Florent Saudel and Jonathan Salwan. 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*, pages 31–54. SSTIC, 2015.

[25] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[26] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. 2015.

[27] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.