A system for dynamic slicing and program visualization

by

Ray Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
February 8, 2019

Signature redacted

Certified by.......... ....
Tim Leek
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Signature redacted

Accepted by ....
Katrina LaCurts
Chairman, Department Committee on Graduate Theses

A system for dynamic slicing and program visualization

by

Ray Wang

Submitted to the Department of Electrical Engineering and Computer Science
on February 8, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

Dynamic analysis, which involves gathering information about a program as it is executing, is becoming increasingly common as reverse engineers attempt to more efficiently analyze complex software systems. In this thesis, I implement a technique called dynamic slicing, which determines how values in a program's register or memory, such as strings, are computed as a function of the program's initial state. This technique is then evaluated on sets of programs of interest to reverse engineers: string generation and string encoding/decoding algorithms.

A common source of these algorithms is malware, which often employs these mechanisms to obfuscate and to make human-driven reverse engineering more difficult. In particular, malware will use Domain Generation Algorithms (DGAs) to construct seemingly randomized domain names to contact their control servers. We demonstrate that metrics and graphs produced with dynamic slicing can be successfully employed on these algorithm classes, elucidating interesting features from different families of malware and reducing the manual workload of malware reverse engineers.

Thesis Supervisor: Tim Leek
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

I would like to thank Professor Brendan Dolan-Gavitt for providing advice and assistance with this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Those who reverse engineer software systems have the goal of expending a minimum of man-hours to gain a maximal understanding of the piece of software they are analyzing. As software systems grow larger, human-driven program understanding becomes increasingly difficult, and automated techniques become essential.

There are two broad categories of automated program analysis — dynamic and static — each with certain strengths and weaknesses. Static analysis is a class of methods to understand the program without executing it, ex. manual disassembly of a compiled binary or static source code analyzers that look for violations of safe programming patterns. Dynamic analysis, as the name suggests, involves instrumenting the execution of the program itself, gathering information from the program as it is running.

One advantage of static analysis is that even rarely-used branches of code or possible exceptional cases can be detected and analyzed statically, that would otherwise never be tested dynamically. Static analysis often involves manual understanding by humans that are better at synthesizing information about complex code than machines. However, dynamic analysis is more powerful overall, as it is a much richer source of program understanding, enabling access to call stacks, execution traces, code coverage, memory information, etc. Dynamic analysis can also handle types of reversing tasks that static methods cannot — in particular, obfuscated code. Dynamic analysis does have its disadvantages — it can be more computationally expensive and have a higher rate of false positives and negatives. Often, a combination of both approaches is used to understand an unfamiliar piece of software or codebase.

This thesis explores a dynamic analysis technique called dynamic slicing. Dynamic slicing takes as input a list of executed instructions (a "trace") and some internal quantity of interest in a program's state, such as a variable. As output, it produces the subset of the instructions that computed that quantity (the "slice"), as well as the variables that were modified as part of the slice. Dynamic slicing is essentially a reverse engineering tool — it works *backwards* from the end of the instruction trace, eventually determining what initial state of the system was important to the computation of some final state.

We first design and implement the dynamic slicing algorithm at the full operating system level, meaning that we operate in terms of memory and registers as part of our internal state, and assembly instructions as the slice. This is unique in that most other work on dynamic slicing has been performed at the level of source code. Working on the CPU level with zero higher-level information allows us to avoid any abstractions about libraries, kernel, etc. and produce a pure slice based on data computation alone. We then evaluate the performance and effectiveness of dynamic slicing for facilitating reverse engineering on a set of string encoding and string generation algorithms, as commonly employed by malware writers.

The contents of this thesis are organized as follows:

Chapter 2 discusses prior related work on the several areas that are explored in this thesis: dynamic slicing theory and implementations, data dependency graphing, and string manipulation algorithms used by malware.

Chapter 3 details the design and implementation of our dynamic slicing, which is built upon the PANDA full-system emulator developed by Lincoln Lab [7]. We also discuss technical challenges that were encountered over the course of implementing the system and their solutions.

Chapter 4 introduces various metrics and visualizations devised to better understand and evaluate the benefits of slicing. In particular, the goals of this chapter are to demonstrate how we can extract useful information from slice output to facilitate the task of reverse engineers when facing an unfamiliar algorithm in a complex piece of software.

Chapter 5 documents the results of applying the above techniques to the most challenging reversing task, malware. Malware is, by nature, heavily obfuscated to deter reverse engineers. One way in which it attempts to circumvent security researchers is by using custom string encoding and decoding algorithms for two main purposes: first, Domain Generation, or producing large numbers of randomized domain names that can be used as potential command and control servers, and secondly, static string obfuscation, or concealing the presence of interesting strings within the binary, such as filepaths. We apply the tools and techniques discussed in this thesis to some characteristic examples of Domain Generation Algorithms used by malware observed in the wild, describing the benefit afforded by our work on malware understanding.

1.1 Objectives of Dynamic Analysis

There are many forms of dynamic analysis that have been developed to aid in program understanding. In all cases, the objective is to reduce the manual workload of a reverse engineer when debugging or analyzing a particular program — even as programs get more complex, the human workload should scale at a much slower pace. By contrast, manual static analysis that involves visually inspecting source code or disassembled binary code, while much more precise and less prone to false positives/negatives, requires great skill on the part of an experienced researcher, and quickly becomes infeasible as programs increase in size. Thus, where dynamic techniques seek to outperform static ones is scalability and throughput of program testing.

For instance, consider several common techniques in dynamic analysis: fuzzing and taint analysis. Taint analysis is essentially backward slicing in reverse — labels are applied to some input state, such as user data or the bytes of an input file, and then computations propagate those labels elsewhere into the program. This can determine, for example, whether a program is exploitable via user input [15]. Fuzzing involves mutating program input in a way that increases code coverage and potentially triggers exceptional branches or rarely-touched areas of code where bugs might lie. Both of these techniques afford reversers a huge benefit — otherwise, a reverser would need to manually construct mutations and manually trace all possible uses of user input.

In addition, dynamic analysis can circumvent many of the issues that static analysis struggles with, particularly in the case of obfuscated code. Malware will often employ techniques such as disassembler confusion and code packing to make static analysis orders of magnitude more difficult. In order to be effective, dynamic analysis should be able to succeed where static methods fail entirely.

Chapter 2

Related Work

2.1 Dynamic Slicing in Theory and Practice

Dynamic slicing is a well-known technique in the program analysis space, and there have been several papers on both theory and implementations of it.

The concept was first introduced in a static context on high-level programs by Weiser [17]. There, a static slice was defined as the subset of all statements or instructions in a program that could potentially affect a certain variable in question.

Then, Korel and Laski extended this idea to a dynamic slice on a program, in which the slice makes use of information about a particular execution of the program and contains the subset of statements/instructions that 1) is executable and 2) computes the same result as the full program [9].

Agrawal and Horgan further developed the theory of slicing by introducing techniques for generating a non-executable *Dynamic Dependence Graph* that is smaller in size than a full executable slice, because it does not contain all occurrences of a statement that is involved in the computation of the target value, but rather only the specific occurrences (say, one iteration out of many loop iterations) that affect the value. Later, Agrawal demonstrated the first practical application of slicing by incorporating it into an application to debug high-level code [4, 3].

In a relatively recent advance, Sahoo et. al implemented Giri, a dynamic program slicer that works on the LLVM Intermediate Representation (IR) [14]. LLVM is a compiler framework and instruction set with an API to instrument its IR, allowing information to be collected during runtime [10]. This is useful for slicing because such instrumentation can augment an instruction trace with data that is not available statically. In addition, LLVM is a good choice because it has more straightforward semantics than source code or machine code, simplifying the task of slicing with a smaller instruction set.

This most closely resembles our system, discussed in the next chapter, which is also LLVM-based. However, the major difference between this system and ours is that Giri requires high-level source code, as it uses the LLVM compiler along with debugging symbols to generate the LLVM IR bitcode that it slices on. PANDA, and thus, our dynamic slicer, operates on a much lower semantic level — the full CPU, running pure assembly code. Kernel and library code is handled implicitly, as well as unusual or complex machine instructions — all machine instructions are translated to a small subset of the LLVM IR, and we need no higher-level information to generate a complete slice.

2.1.1 Data dependency graphing

Dynamic slicing has generally gone hand-in-hand with program graphing techniques based on data dependencies. Here we discuss various related work in program graphing.

A common type of graph is a Program Dependence Graph (PDG), in which nodes represent variables within a program and directed edges represent a dependence of one value on another. This dependence is referred to as "data flow dependency", or just "data dependency", and the graphs are sometimes referred to as "data flow graphs".

Early theory on program dependency graphing was centered around its usefulness in code optimization — that is, generating PDGs that could be used to detect parallelism and vectorize code, loop fusion, constant expression folding, etc. [8].

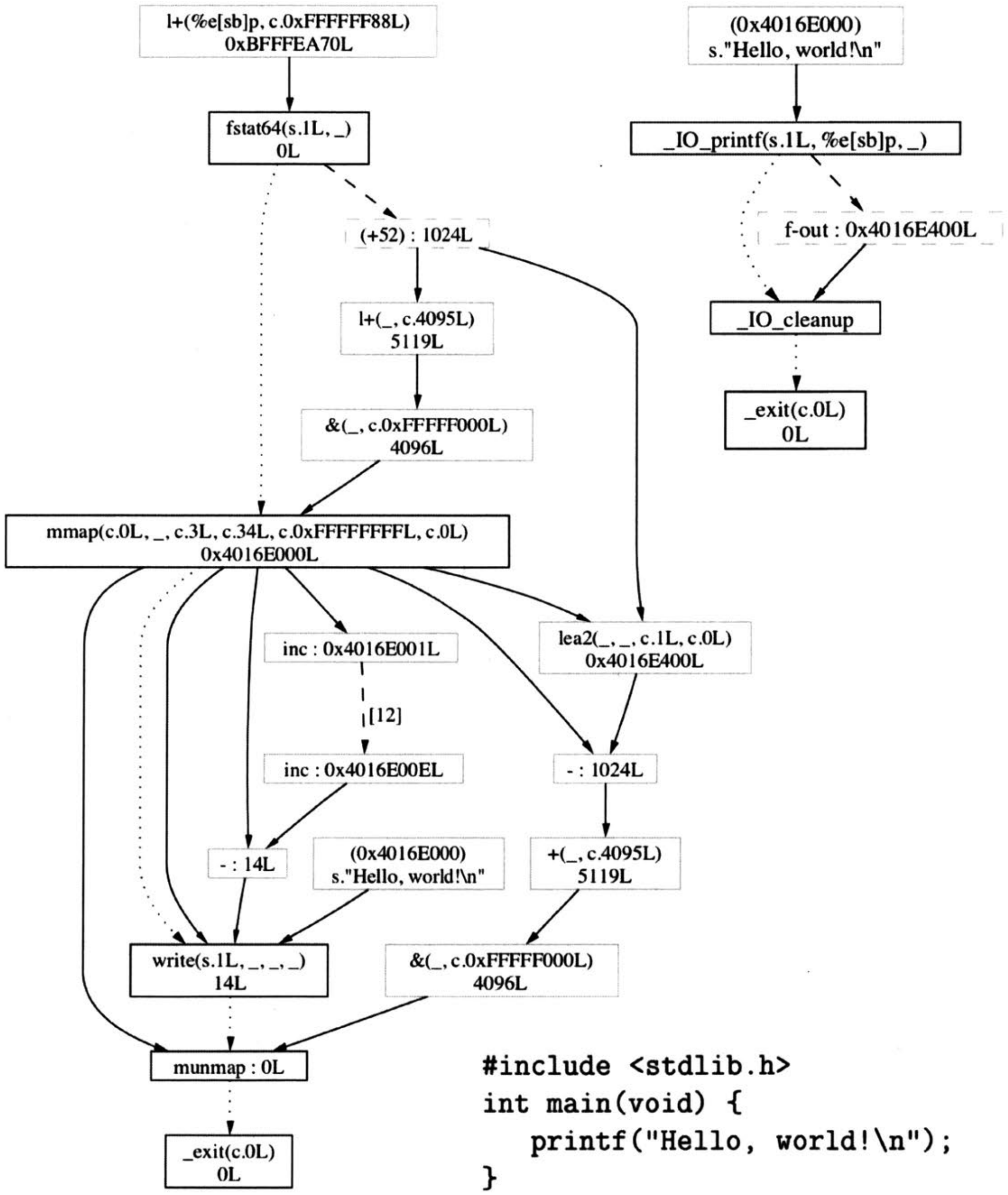
More recently, Redux [12], was created by the same author as the Valgrind memory profiling and binary instrumentation toolkit. Like our system, it can operate without source code, converting binary code to its RISC-like IR called UCode to construct a dependence graph. The fundamental method of constructing the graphs is very similar to ours — Redux generates new graph nodes for each system call or arithmetic instruction and connects them via directed edges when data is transferred.

Redux then takes the extra step of performing many compaction and "prettifying" passes on the produced graph, substantially reducing the number of nodes and edges. This results in PDGs like that of a simple "Hello World" program in Figure 2-1. In this graph, system calls and arithmetic instructions are each given a node. Arrows from one node to another denote that the target node uses the value in the source node. Note that the Redux authors have heavily post-processed the graph to prettify it: chains of "increment" nodes are abbreviated, with the dashed edges indicating elided nodes, and non-shared constants are inlined into nodes. The figure in the top-right is an even more abstracted version of the "Hello World" program, obtained by condensing the larger graph into the `_IO_printf` and `_IO_cleanup` symbols, which are found by looking in the `glibc` source code.

While Redux produces very visually appealing and simple graphs due to its aggressive pruning and graph rewriting, it does nothing of note with it. Redux raises the ideas of some potential applications of its dependency graphing, including de-obfuscation and program equivalence, but does not go as far as to explore them. The PDGs that we produce are a direct product of a dynamic slice, and thus have direct value tied to the reverse engineering task, whether it is de-obfuscation or program understanding, at hand.

2.2 Malware

The field of malware analysis is perhaps the greatest challenge to program analysis techniques because of the heavily obfuscated nature of most malware observed in the wild. For instance, malware might obfuscate itself to hide itself from static analysis, to thwart antivirus software, to evade sandboxes that attempt to analyze it, etc. This often involves a custom combination of techniques including code packing, multi-stage payloads, self-modification and mutation, etc. [18].



```
#include <stdlib.h>
int main(void) {
   printf("Hello, world!\n");
}
```

Figure 2-1: Hello World PDG produced by Redux

The common malware features that we study in this thesis are the Domain Generation Algorithms (DGAs) described previously, used by malware in generating seemingly random domains for C&C servers. The malware authors register some small subset of all possible domains, and the malware will attempt to contact some of the random domains, ensuring that at least some small percentage of connections succeed. If a malware is able to contact its C&C server, it can be updated or commands to further compromise the system can be received.

Attempts have been made to classify and compare DGAs [13]. Plohmann et. al. analyzed the DGAs of several dozen malware families and variants, proposing a taxonomy based on the seed — time-based deterministic or non-deterministic — and the generation scheme — arithmetic, hash-based, wordlist, or permutation-based. More on this is given in Chapter 5, when these categorizations are used to guide our own understanding of different DGA malware.

An example of using static and dynamic analysis techniques on malware is from Barabosch et. al., who employ data flow techniques to extract the DGA functions from malware [6]. As part of their "extraction phase", the authors claim to apply a use-defs algorithm similar to what we use in Chapter 3 for data flow analysis. A use-def chain of some variable consists of a target variable and all the variables that contributed data to it, or "defined" it.

However, the authors do not describe their algorithm except at a high level, leaving many questions as to how they perform the data flow analysis. What is said is that they require an external database of knowledge about Windows API calls. By contrast, our dynamic slicer does not need or use high-level information about API libraries, relying solely on pure assembly data flow analysis. Also, the architecture of the framework requires that the underlying disassembler, in this case the popular IDA Pro, correctly disassembles the code and recognizes all functions, which is a very tenuous assumption for anything more than simple, nonobfuscated malware. By contrast, we don't require such a static disassembler because we work on emulated code which disassembles a basic block at a time, on demand, and cannot be fooled by tricks that break static tools like IDA Pro.

Chapter 3

Dynamic Slicing

3.1 Slicing Algorithm

This section presents the variation of the backwards dynamic slicing algorithm that we implemented. The *inputs* to the dynamic slicing algorithm are, at a high level, an "instruction trace" plus some internal machine quantities on which to slice. These quantities can be registers or memory addresses, in the context of a full OS. This set is termed the *worklist.* The slice will operate *backwards* through the instruction trace from the internal machine quantities, outputting the initial state of the system that must be specified to produce those internal machine quantities, as well as the subset of instructions that are part of the computation of the worklist. The initial machine values are, essentially, "free variables" of the slice with respect to the worklist.

At a high level, the algorithm is as follows:

Algorithm 1 Usedefs algorithm

```
1: procedure SLICETRACE($instruction\_trace$)
2:     worklist ← {$initial\_state$}
3:     for all $inst \in reversed(instruction\_trace)$ do
4:         uses, defs ← GETUSESANDDEFS($inst$)
5:         for all $def \in defs$ do
6:             if $def \in worklist$ then
7:                 worklist ← $worklist \setminus def$
8:                 worklist ← $worklist \cup uses$
       return worklist;
```

The exact nature of the inputs and outputs of this procedure, as well as implementation details, are described in the following sections.

3.2 Design and implementation

3.2.1 PANDA and LLVM translation

Much of this work is built upon PANDA, the Platform for Architecture-Neutral Dynamic Analysis, which is itself based on the QEMU full-system emulator. Emulation is the ability for a host operating system to run a guest system of a potentially different architecture. The guest being emulated is often referred to as a *virtual machine* (VM), since the emulator creates virtual objects to represent a physical CPU, memory layout, hardware peripherals, etc. QEMU executes the VM by means of "lifting" guest assembly instructions to an intermediate representation called TCG (Tiny Code Generator), which is then translated to the host machine's assembly set [2].

PANDA extends QEMU to enable a programmer to make a recording of a guest VM doing any variety of tasks, then replay it back deterministically. This is done by saving all sources of nondeterminism, such as hardware interrupts or timestamp reads, in a *nondeterminism log*. For the purposes of a slice, record/replay is crucial because a slice must be performed on a single execution of the program, which PANDA can guarantee will exist in a reliable replay. However, the true power of PANDA is that the replay can then be augmented with *plugins* that can be written to perform a variety of dynamic analyses.

For dynamic slicing, we employ another translation from TCG to the LLVM IR [10], which is the representation that the slicing algorithm operates on. All the guest architectures that PANDA supports (currently x86/64, ARM, and PowerPC) can be lifted to a common set of *LLVM bitcode*, composed of only binary operators (`Add`, `Sub`, `Mul`, etc.), bitwise operators (`And`, `Not`, `Shl`, etc.), control flow instructions (`Call`, `Branch`, etc.), and conversion operators (`IntToPtr`, `Trunc`, `ZExt`, `SExt`). The LLVM

Language contains dozens more opcodes [1], but only a small subset suffices to fully emulate guest systems.

Again, the ability of PANDA to translate guest assembly to LLVM is essential to slicing. The LLVM IR allows us to ignore the complex and wildly varying semantics of machine code on different architectures and potentially unusual instructions that would otherwise be unhandled — all machine code is translated into LLVM in the form of LLVM basic blocks, and, for special instructions, LLVM helper functions. Also, the full instrumentation toolkit provided by LLVM allows us to collect all the necessary information that the slice requires during runtime.

After lifting to the LLVM IR, there are two phases to dynamic slicing: in the first, dynamic information about the execution of the program (the exact requirements of which are described below) is recorded in a log file that we term the *trace log*. This is implemented as a plugin that operates on the PANDA recording during a replay, using an LLVM *instrumentation pass* that allows the insertion of logging functions that are called during runtime. The dynamic information collected in the trace log is important because it exactly specifies the updates to the system state, such as memory and registers, made during a single execution of a program. By having a log of all computation done on all machine state, we can then isolate which instructions are part of our slice.

Second, the trace log and the generated LLVM bitcode are passed to the dynamic slicing algorithm above, along with an initial set of elements in the worklist. The slicer *aligns* the log and the bitcode, then performs the backwards dynamic slice.

LLVM Trace Logging

One requirement of dynamic slicing is having access to runtime information about dynamic values, such as the source/destination and values of loads and stores. This involves instrumenting certain LLVM instructions with functions that, when executed at runtime, record dynamic values to a log file, that can later be recalled during the dynamic slicing.

[1]LLVM Language Reference — `https://llvm.org/docs/LangRef.html`

For example, the non-instrumented LLVM IR for the x86 instruction **push ebp** may be represented by

```
%tmp2_v = sub i32 %esp_v, 4
%10 = add i64 %2, 20 // Offset in QEMU's X86CPUState structure
%ebp_ptr = inttoptr i64 %10 to i32*
%ebp_v = load i32* %ebp_ptr
call void @helper_le_stl_mmu_panda(%struct.CPUX86State* %0, i32 %tmp2_v,
    i32 %ebp_v, i32 1, i64 3735928559)
```

As can be seen in the vanilla LLVM code, the **push ebp** is translated into the following sequence: the value of **ebp** is loaded from a QEMU CPU object. Then, the value is stored at the top of the stack, **esp**, by way of a QEMU helper function to write emulated memory.

The corresponding instrumented version would look like:

```
%tmp2_v = sub i32 %esp_v, 4
%ebp_v = load i32* %ebp_ptr
call void @recordLoad(i8* %ebp_ptr, i64 %ebp_v, i64 4), !host !0
ttoptr i32 %tmp2_v to i8*%ebp_v80 = zext i32 %ebp_v to i64
call void @recordStore(i8* %tmp2_v, i64 %ebp_v, i64 4)
call void @helper_le_stl_mmu_panda(%struct.CPUX86State* %0, i32 %tmp2_v,
    i32 %ebp_v, i32 1, i64 3735928559)
```

In the instrumented version, both the load and store are instrumented with calls to logging functions, that are passed the location and value loaded/stored. As described above, we must record any updates to system state, such as memory loads/stores, so that the dynamic slicer can correctly evaluate whether or not a certain load or store is part of the use-defs chain of the worklist.

In addition to memory loads and stores, several other dynamic values require such logging — those having to do with control flow. Because we are isolating a single execution path of a system, the trace log must contain information on exactly which path was taken at every branch. The LLVM instructions that determine control flow, such as `Select`, `Branch`, and `Switch`, are instrumented with a function that records the condition that determines which branch is taken.

The log is serialized using Google's Protocol Buffers [16], which takes a schema for a serialization format, similar to a C struct, and provides an API for generating and consuming a stream of serialized entries. A full description of all the information that must be logged can be seen in the Protobuf schema for the log [2].

Slice alignment and slicing

The inputs to the slicer are specified in a `criteria` file — this contains the instruction range that we want to slice within, the virtual memory areas (VMAs)[3] that are relevant to us, and the memory addresses/registers that are part of our initial worklist.

An example criteria file is shown in Listing 3.2.1. As part of the string slicing workflow described in the next chapter, this file will usually be automatically generated by another plugin.

```
rr_start:975034
VMA:single-byte-xor
MEM_804a03c-804a04a
rr_end:1034893
```

As a first step during slicing, the LLVM instructions must be aligned with the trace log entries. Each executed LLVM instruction is represented in the generated LLVM bitcode file, and certain instructions willl also have a corresponding trace log entry that it must be matched with. This combined array of trace entries, consisting of instructions + their corresponding dynamic log data, is then passed to the slicing algorithm.

3.2.2 Performance

Trace generation

The overwhelming majority of time is spent in the dynamic trace logging step. LLVM translation and execution causes a significant performance hit during replay. The

[2]https://github.com/panda-re/panda/pull/207

[3]A VMA refers to a region of named mapped memory in a program; ex. text section of program, shared library, stack, heap

simple act of running a replay incurs about a 10x reduction in speed from the real-time execution. Turning on LLVM execution, by itself, incurs a slowdown of around 40x over the replay. If the LLVM tracing instrumentation is added to the code, this could increase the slowdown to nearly 300x over the replay, or 3000x over real-time execution. Thus, performance is a large consideration in the usability of the system (see Table 3.1). Note that this is a one-time cost — once the trace is generated, the slicing algorithm can be performed offline on any part of the system's state with no overhead.

We use several techniques to alleviate the cost of generating a trace log: First, noting that most of a program's execution time is spent inside the kernel handling hardware interrupts, page faults, etc. We do not translate to LLVM inside these sections, under the assumption that there is no data dependency to these portions of the system and they are thus irrelevant to the slice. Our claim is that page faults and hardware interrupts are essentially bookkeeping operations by the kernel that should be transparent to the program being analyzed, and thus should not contribute to the data flow of the program. We have not verified this to always be the case, but we expect that there are only rare instances where this assumption does not hold. We can identify when the CPU is executing an interrupt inside the kernel via architecture-specific techniques — on x86, for instance, we can insert a callback inside the QEMU hardware interrupt handler code to mark when we have entered an interrupt, and another callback on the `iret`, or "interrupt return" instruction, to know when we have exited the interrupt.

Second, we allow the user to specify a certain range of addresses or instruction counts that they would like to generate a trace for. The *instruction count* is simply a counter for the number of guest assembly instructions executed at any point in the PANDA replay. Typically, for a single program, the starting address or instruction count would be the beginning of the `main` function of a program, and the end would be at the return of `main`. If a range is specified, LLVM translation and trace generation is only activated for this set of instructions. This subset of instructions is likely to be a tiny fraction of the whole system's execution. In our use case, which is

Table 3.1: Replay execution times

# of instructions	plain replay	w/ LLVM	w/ LLVM + trace logging
10.2 million	1.9s	76s	551s

string encoding/decoding algorithms, we know that the program will be small enough to merit this level of truncation. For larger programs that may involve millions of instructions or many system calls, we would need better ways of determining whether this sort of truncation does not suffice to capture trace log, which we leave for Future Work.

Trace generation

In the next chapter, we discuss evaluating the slicing system on representative string decoding functions that provide a first look at the effectiveness of dynamic slicing.

Chapter 4

Slice Visualization and Evaluation

The slicing algorithm, by itself, does not produce results that are very interpretable. In this chapter, we discuss various methods that were devised to visualize and understand the output of our dynamic slice.

4.1 Evaluation

4.1.1 String decoding/encoding

One set of test cases we use to evaluate a slice is from FireEye Labs, which has compiled a set of string generation programs originally for evaluating a malware deobfuscation tool [1].

These tests perform a range of string encode/decode operations, from simple single-byte XOR to RC4 and Base64 decoding. Each produces a known string ("Hello world") that is located at some memory address, potentially on the stack or the heap.

The general testing workflow is as follows: For each test, we create a PANDA recording of the program running on a 32-bit Debian QCOW. Then, we replay the recording along with a plugin called `stringsearch`, which monitors memory accesses and outputs all locations where a certain string was observed. The address and instruction count of the first observed occurrence of the "Hello world" string is saved

[1]https://github.com/fireeye/flare-floss

to a file, which will later be given to the dynamic slicer as the initial worklist. The assumption is that the first instance of the string is the point immediately after it has been decoded, which would be the natural starting point to slice from.

The server for all tests contains an Intel Xeon E5 CPU at 3.10GHz with 63GB of RAM and 24 CPU cores running Ubuntu 16.04.

4.2 Slice analysis

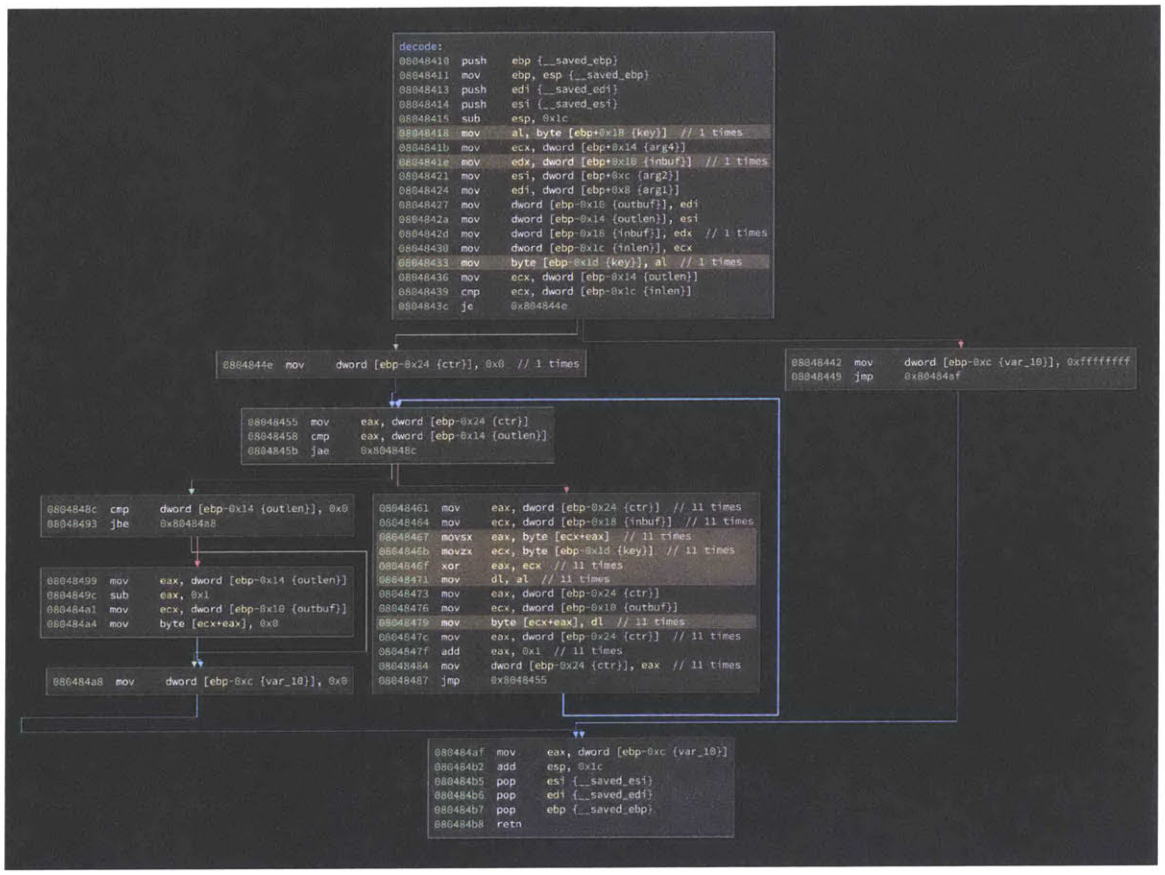
The results of the slicing algorithm are of multiple varieties. A bitarray is generated for the LLVM instructions, where a bit is 1 if the instruction is marked as part of the slice, and 0 if not.

4.2.1 Disassembler Visualization

The most immediately useful method of visualizing the slice is via drawing the reverse engineer's attention to the sliced instructions from within a disassembler. Parsing the bitarray of marked instructions leads directly to slice visualization in this fashion for the popular disassemblers Binary Ninja and IDA Pro. To do this, we add *metadata* to each LLVM instruction that corresponds to its guest opcode bytes.

Figure 4-1 shows the slice visualization for a simple single-byte xor operation on a string of length 11. The corresponding C code for the `decode` function can be found in Listing A.1. The `decode` function takes five arguments on the stack: `out_buf`, a pointer to the decoded string, `out_len`, `in_buf`, a pointer to the encoded string, `in_len`, and the XOR `key`. As seen in the figure, the highlighted instructions are deemed relevant to the slice output, and the number of times each basic block is executed is labeled in a comment. The dataflow slice on the `out_buf` correctly deduces the dependency on `in_buf` and `key` arguments, as well as the XOR operation.

As another example, we look at the source code (Listing A.2) and corresponding slice disassembler visualization (Figure 4-2) for a custom substitution cipher. The cipher decode routine indexes each byte of the input string into a lookup `key` using the `memchr` function, then performs a series of conditional checks on the byte to



```
decode:
08048410  push    ebp {__saved_ebp}
08048411  mov     ebp, esp {__saved_ebp}
08048413  push    edi {__saved_edi}
08048414  push    esi {__saved_esi}
08048415  sub     esp, 0x1c
08048418  mov     al, byte [ebp+0x18 {key}]  // 1 times
0804841b  mov     ecx, dword [ebp+0x14 {arg4}]
0804841e  mov     edx, dword [ebp+0x10 {inbuf}]  // 1 times
08048421  mov     esi, dword [ebp+0xc {arg2}]
08048424  mov     edi, dword [ebp+0x8 {arg1}]
08048427  mov     dword [ebp-0x10 {outbuf}], edi
0804842a  mov     dword [ebp-0x14 {outlen}], esi
0804842d  mov     dword [ebp-0x18 {inbuf}], edx  // 1 times
08048430  mov     dword [ebp-0x1c {inlen}], ecx
08048433  mov     byte [ebp-0x1d {key}], al  // 1 times
08048436  mov     ecx, dword [ebp-0x14 {outlen}]
08048439  cmp     ecx, dword [ebp-0x1c {inlen}]
0804843c  je      0x804844e
```

```
0804844e  mov     dword [ebp-0x24 {ctr}], 0x0  // 1 times
```

```
08048442  mov     dword [ebp-0xc {var_10}], 0xffffffff
08048449  jmp     0x80484af
```

```
08048455  mov     eax, dword [ebp-0x24 {ctr}]
08048458  cmp     eax, dword [ebp-0x14 {outlen}]
0804845b  jae     0x804848c
```

```
0804848c  cmp     dword [ebp-0x14 {outlen}], 0x0
08048493  jbe     0x80484a8
```

```
08048461  mov     eax, dword [ebp-0x24 {ctr}]  // 11 times
08048464  mov     ecx, dword [ebp-0x18 {inbuf}]  // 11 times
08048467  movsx   eax, byte [ecx+eax]  // 11 times
0804846b  movzx   ecx, byte [ebp-0x1d {key}]  // 11 times
0804846f  xor     eax, ecx  // 11 times
08048471  mov     dl, al  // 11 times
08048473  mov     eax, dword [ebp-0x24 {ctr}]
08048476  mov     ecx, dword [ebp-0x10 {outbuf}]
08048479  mov     byte [ecx+eax], dl  // 11 times
0804847c  mov     eax, dword [ebp-0x24 {ctr}]  // 11 times
0804847f  add     eax, 0x1  // 11 times
08048484  mov     dword [ebp-0x24 {ctr}], eax  // 11 times
08048487  jmp     0x8048455
```

```
08048499  mov     eax, dword [ebp-0x14 {outlen}]
0804849c  sub     eax, 0x1
080484a1  mov     ecx, dword [ebp-0x10 {outbuf}]
080484a4  mov     byte [ecx+eax], 0x0
```

```
080484a8  mov     dword [ebp-0xc {var_10}], 0x0
```

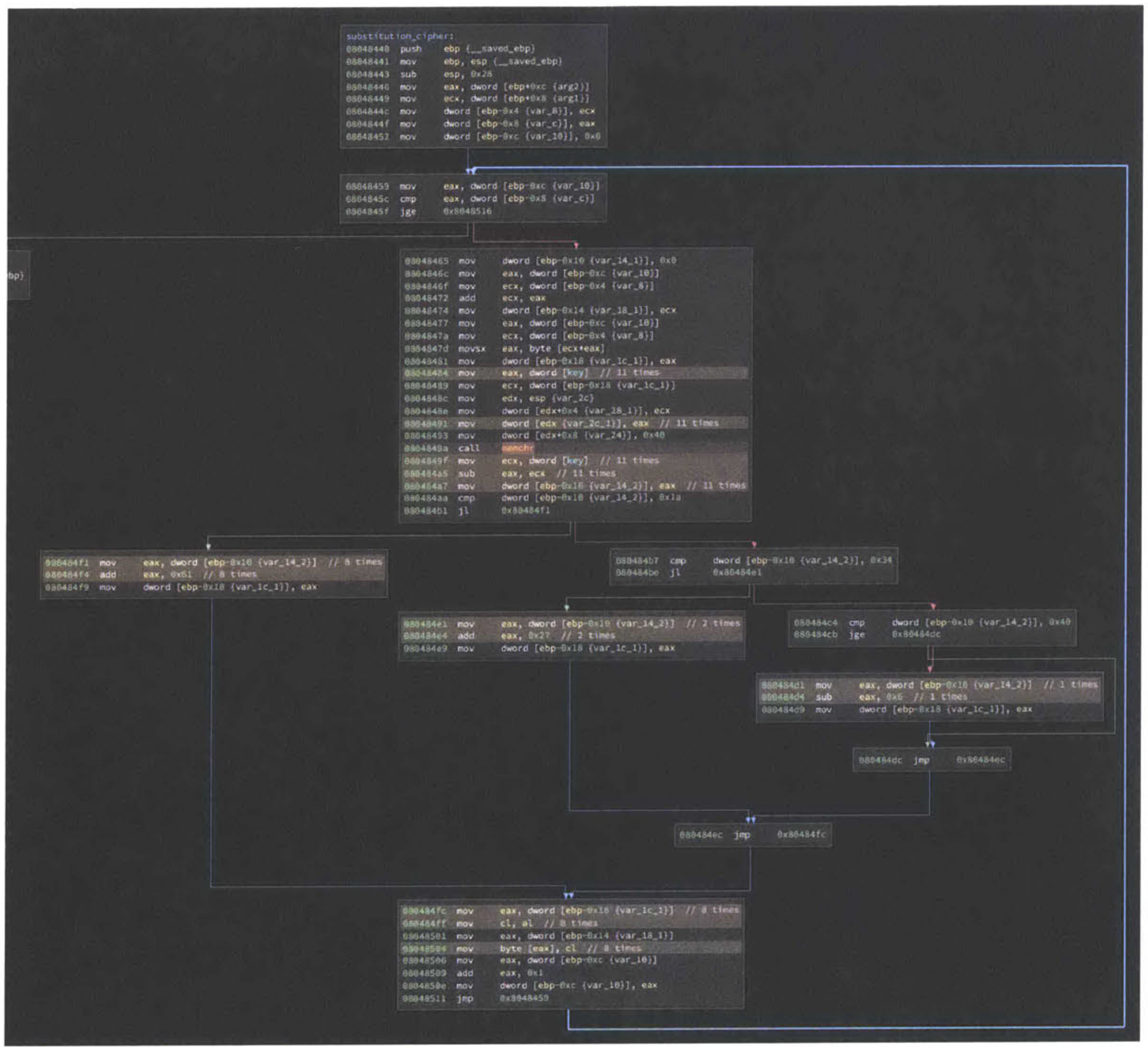
```
080484af  mov     eax, dword [ebp-0xc {var_10}]
080484b2  add     esp, 0x1c
080484b5  pop     esi {__saved_esi}
080484b6  pop     edi {__saved_edi}
080484b7  pop     ebp {__saved_ebp}
080484b8  retn
```

Figure 4-1: Single-byte XOR slice visualization in Binary Ninja

determine the final transform into the output byte. The `memchr` function, from the `string.h` header in the C standard library, simply returns the location of the first instance of a byte in the lookup key.

In the slice for this program, the dataflow analysis identifies that the argument to `memchr`, `key`, is the source of the bytes in the decoded string. We also detect how many times each branch of the cipher decode routine is observed in the slice — for the 11-byte input string `xzhhKDmKjhT`, 8 bytes take the outermost else branch, 2 take the middle else branch, and 1 takes the innermost if branch. The number of times each branch is executed is annotated in a comment that we add to the disassembly via our slice visualization plugin.

It is intended that reverse engineers can incorporate offline slicing into common disassembly or static debugging workflows — they can generate a slice on memory, registers, or other program values of interest in a piece of unfamiliar software for which there is no source available, and quickly focus on the portions of the disassembly that are part of the slice. In Future Work (6.1), we discuss more complex and featureful applications of the slice, such as reverse debugging.



```
substitution_cipher:
08048440  push    ebp {__saved_ebp}
08048441  mov     ebp, esp {__saved_ebp}
08048443  sub     esp, 0x28
08048446  mov     eax, dword [ebp+0xc {arg2}]
08048449  mov     ecx, dword [ebp+0x8 {arg1}]
0804844c  mov     dword [ebp-0x4 {var_8}], ecx
0804844f  mov     dword [ebp-0x8 {var_c}], eax
08048452  mov     dword [ebp-0xc {var_10}], 0x0
```

```
08048459  mov     eax, dword [ebp-0xc {var_10}]
0804845c  cmp     eax, dword [ebp-0x8 {var_c}]
0804845f  jge     0x8048516
```

```
08048465  mov     dword [ebp-0x10 {var_14_1}], 0x0
0804846c  mov     eax, dword [ebp-0xc {var_10}]
0804846f  mov     ecx, dword [ebp-0x4 {var_8}]
08048472  add     ecx, eax
08048474  mov     dword [ebp-0x14 {var_18_1}], ecx
08048477  mov     eax, dword [ebp-0xc {var_10}]
0804847a  mov     ecx, dword [ebp-0x4 {var_8}]
0804847d  movsx   eax, byte [ecx+eax]
08048481  mov     dword [ebp-0x18 {var_1c_1}], eax
08048484  mov     eax, dword [key]  // 11 times
08048489  mov     ecx, dword [ebp-0x18 {var_1c_1}]
0804848c  mov     edx, esp {var_2c}
0804848e  mov     dword [edx+0x4 {var_28_1}], ecx
08048491  mov     dword [edx {var_2c_1}], eax  // 11 times
08048493  mov     dword [edx+0x8 {var_24}], 0x40
0804849a  call    memchr
0804849f  mov     ecx, dword [key]  // 11 times
080484a5  sub     eax, ecx  // 11 times
080484a7  mov     dword [ebp-0x10 {var_14_2}], eax  // 11 times
080484aa  cmp     dword [ebp-0x10 {var_14_2}], 0x1a
080484b1  jl      0x80484f1
```

```
080484f1  mov     eax, dword [ebp-0x10 {var_14_2}]  // 8 times
080484f4  add     eax, 0x61  // 8 times
080484f9  mov     dword [ebp-0x18 {var_1c_1}], eax
```

```
080484b7  cmp     dword [ebp-0x10 {var_14_2}], 0x34
080484be  jl      0x80484e1
```

```
080484e1  mov     eax, dword [ebp-0x10 {var_14_2}]  // 2 times
080484e4  add     eax, 0x27  // 2 times
080484e9  mov     dword [ebp-0x18 {var_1c_1}], eax
```

```
080484c4  cmp     dword [ebp-0x10 {var_14_2}], 0x40
080484cb  jge     0x80484dc
```

```
080484d1  mov     eax, dword [ebp-0x10 {var_14_2}]  // 1 times
080484d4  sub     eax, 0x6  // 1 times
080484d9  mov     dword [ebp-0x18 {var_1c_1}], eax
```

```
080484dc  jmp     0x80484ec
```

```
080484ec  jmp     0x80484fc
```

```
080484fc  mov     eax, dword [ebp-0x18 {var_1c_1}]  // 8 times
080484ff  mov     cl, al  // 8 times
08048501  mov     eax, dword [ebp-0x14 {var_18_1}]
08048504  mov     byte [eax], cl  // 8 times
08048506  mov     eax, dword [ebp-0xc {var_10}]
08048509  add     eax, 0x1
0804850e  mov     dword [ebp-0xc {var_10}], eax
08048511  jmp     0x8048459
```

Figure 4-2: Substitution cipher slice visualization in Binary Ninja

4.2.2 Program Dependence Graphing

Another means of visualization enabled by dynamic slicing is data flow, or program dependence graphing. We can leverage the Boost graphing library [1] to construct program dependence graphs concurrently with the slicing algorithm.

Each vertex is represented by the following struct:

```
struct Vertex {
    std::string slice_var; // string representation of slice variable
    int slice_var_type; // type of slice variable
    unsigned opcode; // LLVM Instruction opcode
    uint64_t value; // value that is loaded or stored
}
```

This is the complete explanation for reading the graphs: First, the vertices — each vertex in the graph is associated with a *slice variable*, which has one of several defined types: LLVM intermediate values (essentially an infinite set of registers, denoted by `LLVM_*`), target register values (denoted by `TGT_*`, ex. `TGT_EAX`), guest memory addresses (`MEM_0xxxx`), etc. The integer `opcode` of the LLVM Instruction (`Load`, `Store`, `Xor`, etc.) is recorded in the vertex as well, along with the hex-encoded `value` loaded from or stored to that vertex. Note that constants are not shown on this graph; therefore, binary operators that have only one input should be assumed to have a constant as the second input.

Each directed edge (an arrow in the graph) connects a *source* vertex to one *target* vertex. This edge means that the target value depends on the source value — in the terminology of dynamic slicing, the source is the *use* value that defines the target *def* value. Each edge usually corresponds to one byte of data — thus, 11 bytes of data that pass through a vertex would be represented by 11 edges exiting the vertex. When the slicing algorithm identifies that a certain instruction is part of the use-defs chain, the Boost library adds a graph edge from the use vertex to the def vertex.

The Boost library outputs the graph in Graphviz DOT format, which can then be rendered with the `dot` command-line tool or Python's `graphviz` package.

The program dependence graph for the single-byte XOR example (slightly cropped to fit) is rendered in Figure 4-3. In this graph, the inputs to the slicer (the outputted decoded bytes) are colored green, guest registers yellow, and memory addresses are colored gray. Red nodes are discussed in the following section, and correspond to nodes that are part of the slice compute number (SCN) chain. To interpret the graph, one can follow the path of input bytes at the top of the graph to output bytes at the bottom, tracing values through memory loads/stores and arithmetic instructions. For instance, at the bottom of the graph, we can see individual bytes being stored to the decoded buffer in Figure 4-4. At the top of the graph, the ultimate source of data is the input buffer from which the encoded bytes are loaded (Figure 4-5).

From these several graphs, it's clear that LLVM is much noisier than guest assembly, generating up to 10x more intermediate registers, and hence vertices, for each instruction. However, these graphs, while overlarge, do contain complete information on data flow dependencies.

There are ways to refine the graph further — one can remove some unnecessary edges by condensing those nodes that are simply LLVM conversion operations (**Trunc**, **SExt**, **ZExt**), which do not have any effect on the data computation flow. This generates slightly more condensed graphs, as seen in Figure 4-6.

There are several other pruning methods that could be employed to compact the graph and make it more readable, but these ideas are not implemented in this paper. For instance, one could condense straight-line data flow that passes through a linear sequence of node-edge-node into a single node. One could also rewrite nodes based on how many times each is used as input to other nodes — the goal would be that rarely used nodes be aggressively pruned to reduce superfluity. As for the edges, one could reduce the overall number of edges by replacing multiple parallel edges with a single edge that is either labeled with the number of bytes transferred along that edge, or a thickness that corresponds to how frequently it is used.

Similarly, the PDG for the substitution cipher is show in Figure 4-7. The same color conventions apply as above. In addition, the intermediate values corresponding to the different decoding branches taken by each byte are highlighted in blue, to show where the control flow branching dictates the data flow graph.

Even for relatively simple programs, it is already starting to get difficult to interpret these results in the form of full data dependence graphs.

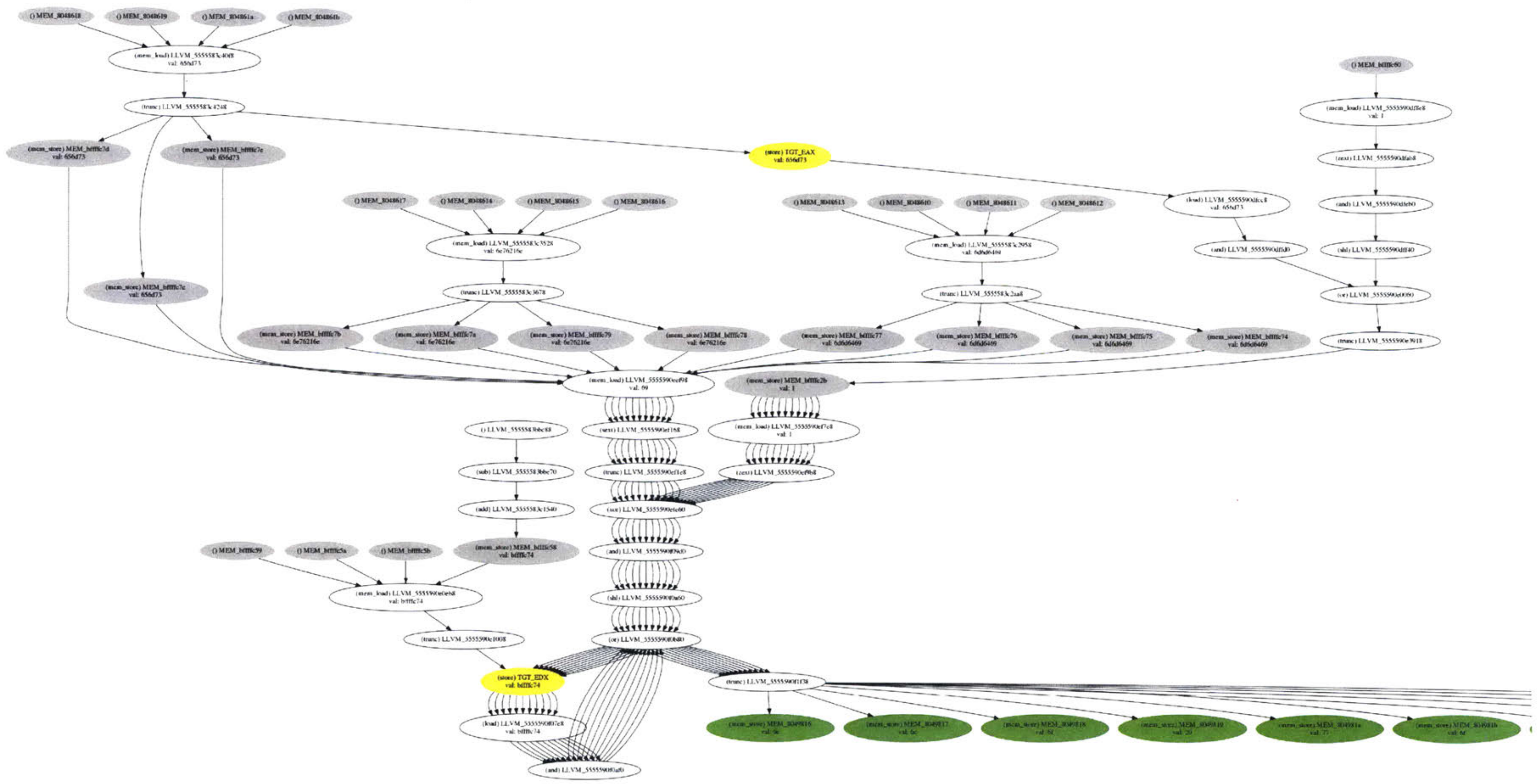


Figure 4-3: Program Dependence Graph for single-byte XOR

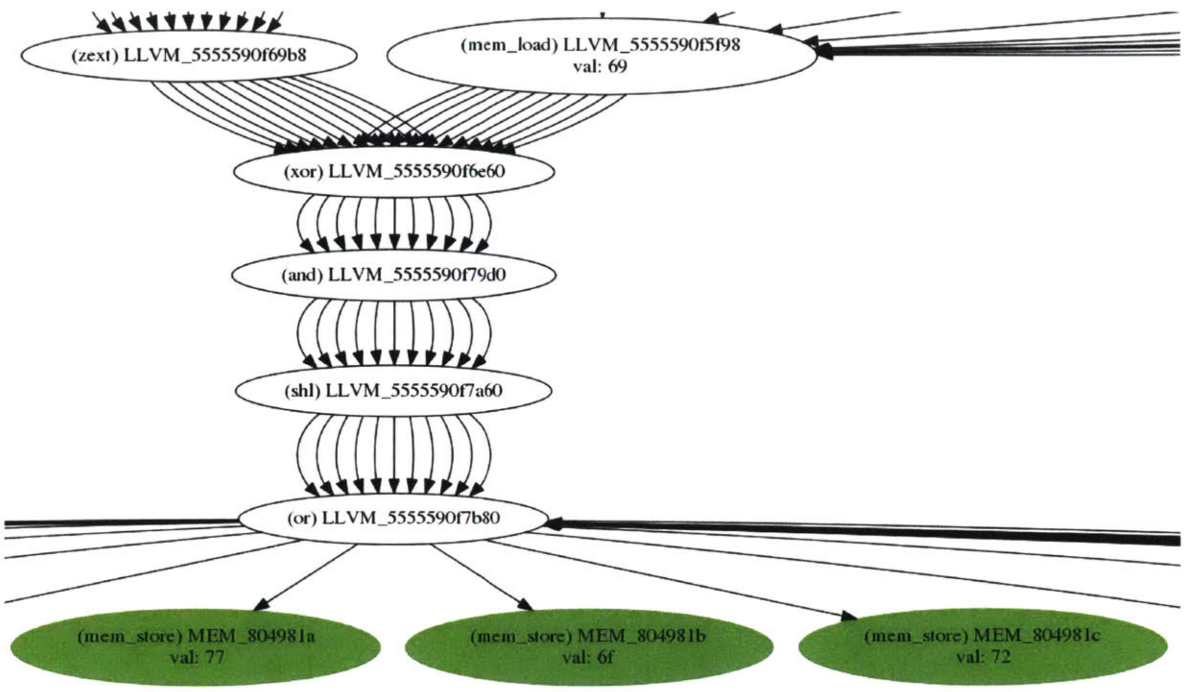


Figure 4-4: Storing decoded bytes at output of single-byte XOR

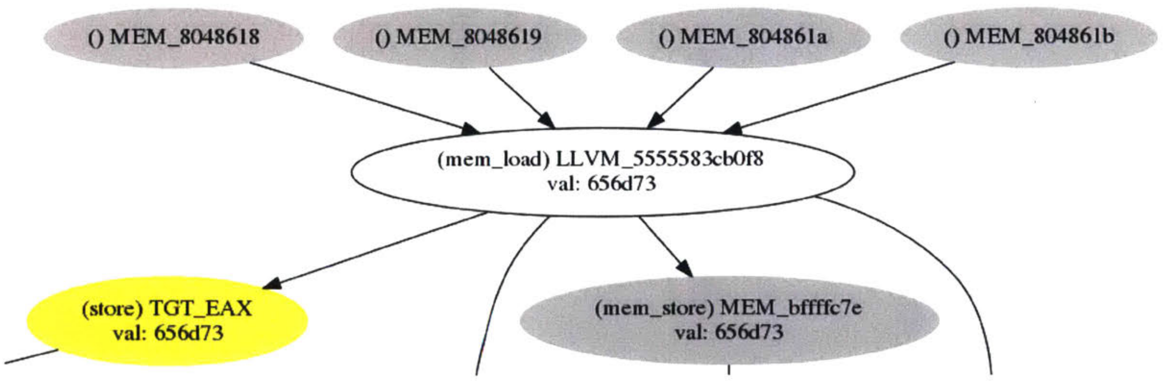


Figure 4-5: Loading encoded bytes at input of single-byte XOR

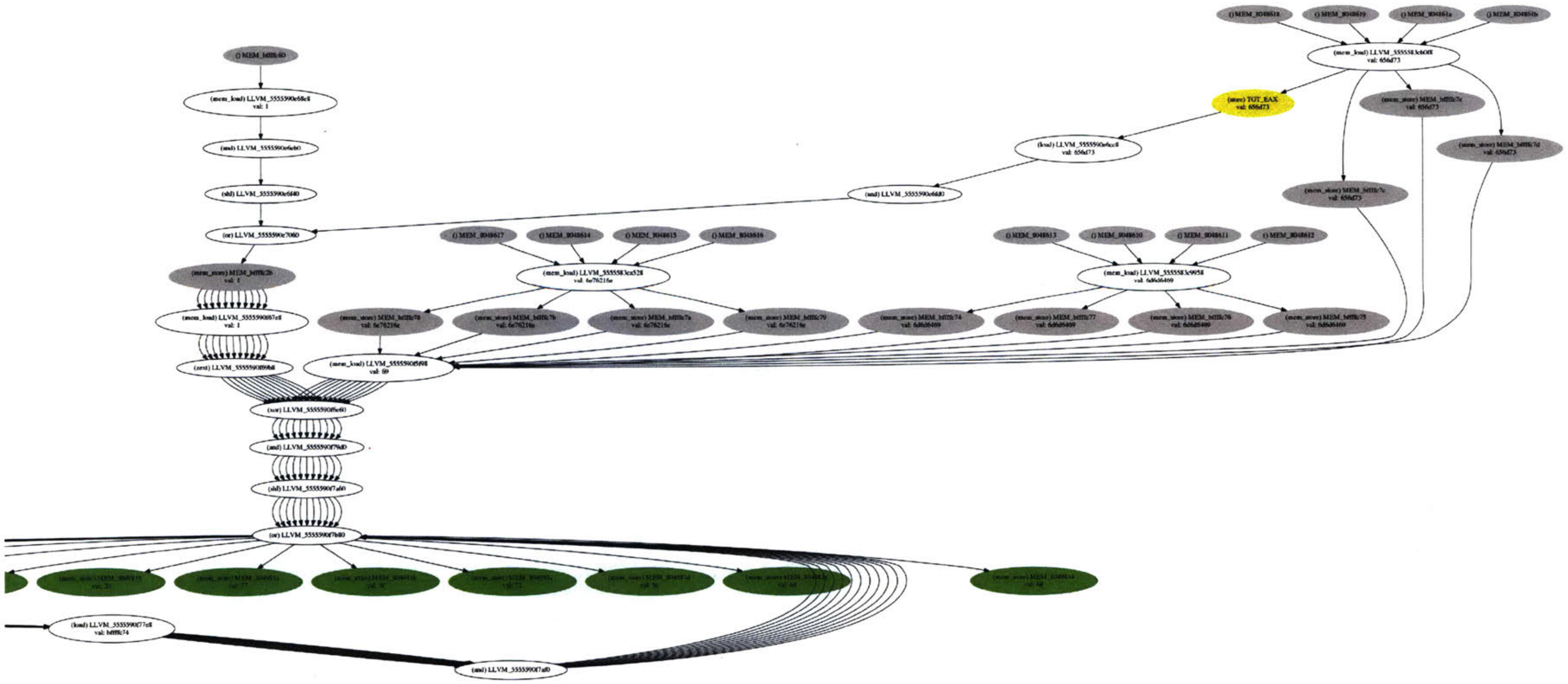


Figure 4-6: PDG for single-byte XOR, after removing LLVM conversions

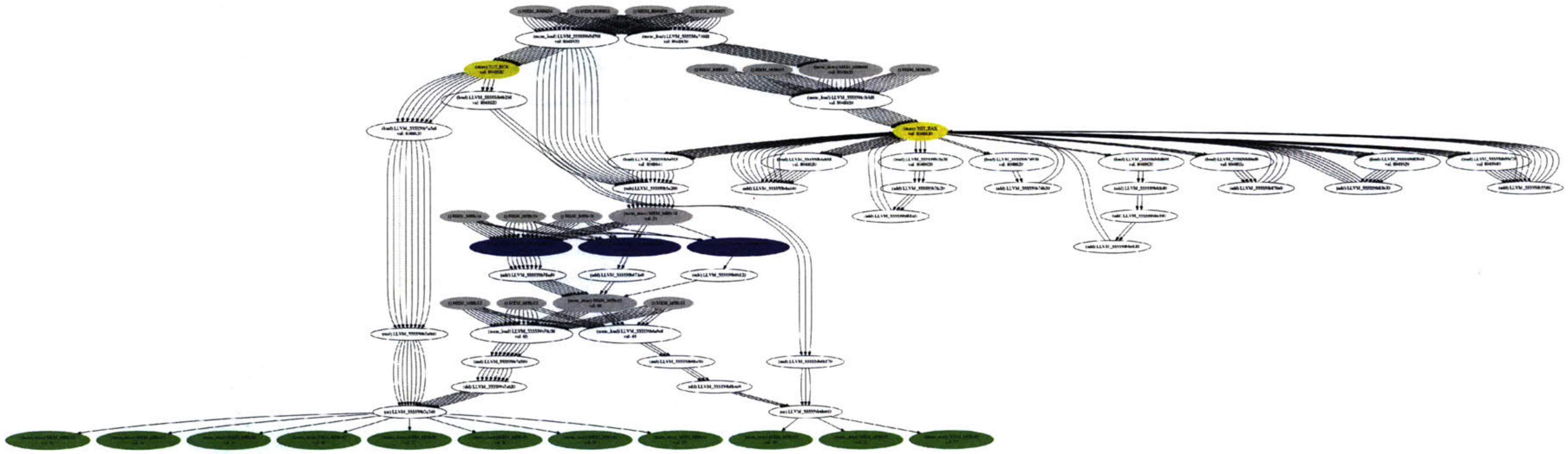


Figure 4-7: PDG for substitution cipher

4.2.3 Slice Compute Number

Given the program dependence graph generated during a slice, we can compute a metric for understanding and comparing computation that we term a *slice compute number* (SCN). The use-defs chain of computation can be represented as a tree, in which the root of the tree is an element in the initial worklist, and the leaves of the tree are the "free variables" associated with the computation of that element.

The maximal depth of the tree is the slice compute number. A distinguishing feature of this computation tree from the PDG is that only a small subset of instructions actually increase the depth of the tree. Non-computation-related instructions, such as memory loads and stores, LLVM conversion operators, etc. do not affect the computation tree or the SCN, effectively distilling down the graph to just the significant vertices. This metric is influenced by the taint compute number that is used by the taint analysis of PANDA [7], which is an analogous computation in the forward direction.

The SCN is implemented as follows: Each element in the worklist is associated with an SCN of 0 initially. Each trace entry is also associated with a *depth* that corresponds to its depth in the computation tree, which is initialized to 0. When the slicing algorithm is updating the worklist by replacing a definition of some variable with its inputs, it checks whether the instruction it is processing is a significant computation instruction such as an XOR. If so, it updates the SCN to be `max(def.depth+1, previous_scn)`. Then, the depth of each of the inputs is updated to this new SCN.

For instance, in the case of the single-byte XOR, the relevant computation instructions that contribute to the SCN are highlighted in red from the PDG in Figure in 4-8. The red vertex at the bottom of the graph is the LLVM intermediate value that is eventually stored into the decoded string `out_buf`, and the vertices at the top of the graph are the two inputs to the XOR operation — the `key` and `in_buf`. The slice compute number for each byte is 4. The reason the SCN is not 1 is a consequence of how single-byte operations are translated from x86 to TCG to LLVM

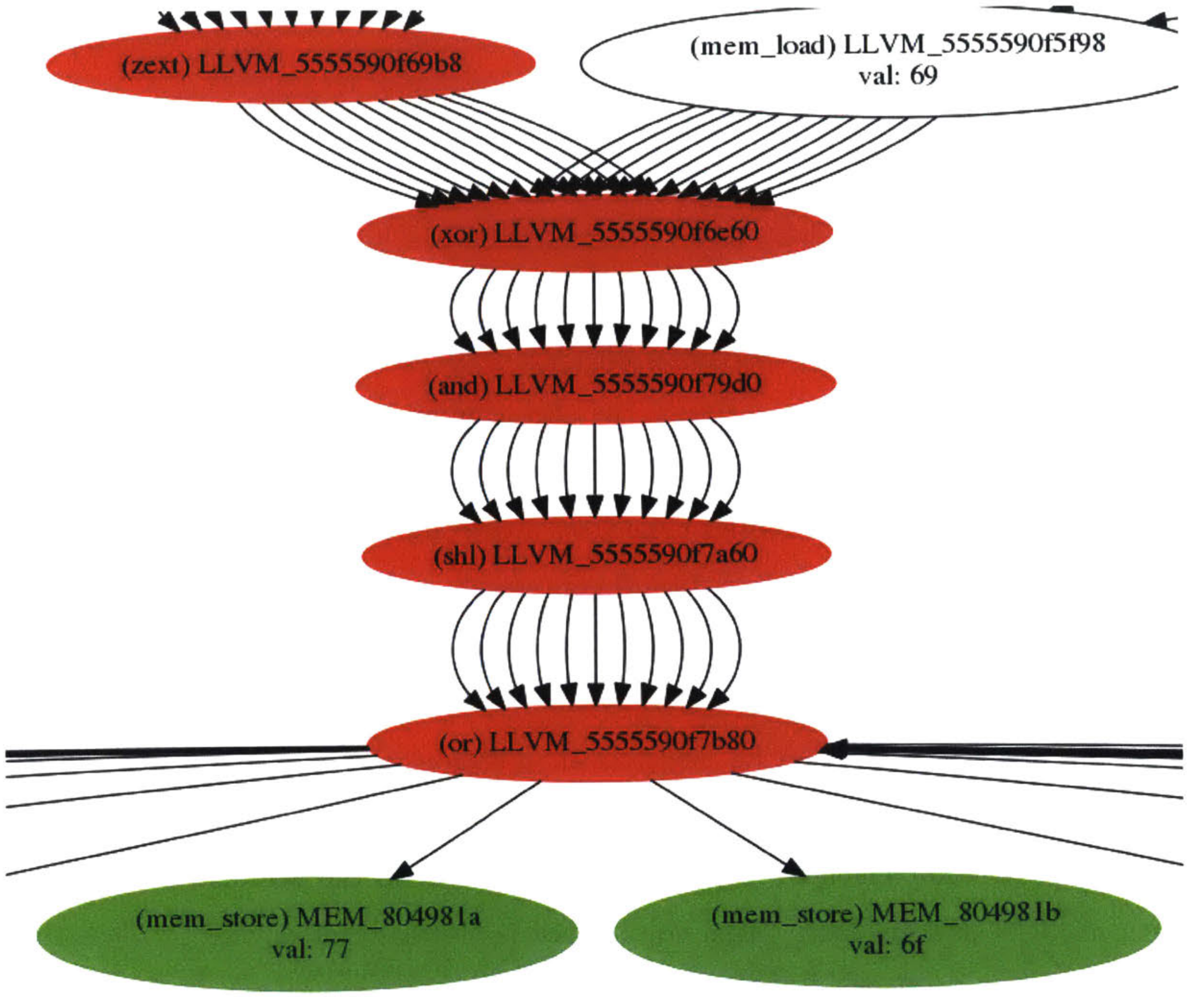


Figure 4-8: Computation tree for single-byte XOR

— the x86 instruction `movb` or `mov %reg, byte` is encoded as a sequence of several bitwise operations with constant operands (not shown).

The slice compute numbers for several other operations in the FireEye test set are shown in Table 4.1. The number of instructions executed in the program is obtained with the `callgrind` code profiler, and the number of instructions executed in the replay is obtained from the PANDA log. What ths table shows is that, while there are a large number of instructions executed by a full system, the relevant portion to a reverse engineer seeking to understand string decoding can be reduced to several dozen instructions as part of a slice. Furthermore, if considering only the instructions that are computationally significant, it can be seen that in each of these cases, the

Table 4.1: Slice compute numbers

program name	median SCN	# guest assembly instructions executed in program	# guest assembly instructions in decode routine	# guest assembly instructions in slice
RC4	4	212,023	13,952	42
Base64	5	194,092	383	63
substitution cipher	9	199,422	426	28
single-byte XOR	4	195,464	227	18

SCN is less than 10, demonstrating that very little actual computation on the bytes is actually being performed as part of the decoding.

While it serves as an overall measure of complexity, the SCN turns out to not be a very useful point of comparison for the purposes of understanding string algorithms. The SCN doesn't reveal anything about the specific structure of the algorithm — two algorithms with the same SCN may have vastly different graphs and different modes of computation. Only when two algorithms vary greatly in how much computation they actually perform would an SCN be able to distinguish them. In the next chapter, we do not rely on the SCN to help analyze malware domain generation algorithms, instead focusing solely on slicing and program dependence graphing.

4.2.4 Implementation details

All slicing code is written in C++ as a PANDA plugin or standalone binary. The LLVM trace plugin is 1200 lines of code, the standalone dynamic slicer is 1400 (not including the graph generation subset), and the slice analyzer that produces the slice disassembly is 250 LoC.

All visualization, such as slice disassembly highlighting and graphing, was performed in a combination of C++ and Python. Constructing the `dot` graphs involved 300 LoC in C++ using the Boost AdjacencyList graphing class. The IDA Pro and Binary Ninja plugins to highlight a slice are around 100 lines of Python each. The code can be found in the PANDA repository on Github in the footnote [2].

[2] https://github.com/panda-re/panda

Chapter 5

Malware

Malicious software tends to pose significant challenges to reverse engineers. Malware employs many obfuscation techniques to deter and confuse security researchers. For example, most malware will attempt to hide any nontrivial strings from the reverser by employing custom-made string encoding and decoding algorithms.

There are also many classes of malware that communicate with command-and-control (C&C) servers as part of their functionality; for instance, to exfiltrate information from an infected system, or to receive updates and activation/deactivation commands. Malware authors must combat not just reverse engineers attempting to understand their program, but also researchers and law enforcement trying to take down the network by disrupting communications with C&C servers, often by *blacklisting* them. To do this, they generate large amounts of possible domain names for their server, of which only a small subset are registered as C&C servers. However, for law enforcement to be able to preempt the malware authors, they must register every possible domain name first, which is an infeasible task. These *Domain Generation Algorithms* are employed by many classes of malware — among them, ransomware (Cryptolocker), banking trojans (Zeus, Pushdo, Torpig, Tinba, Ranbyus), computer worms (Conficker/Kido), and bootkits (Rovnix). The original malware to use a DGA was the Kraken botnet in 2008 [5], followed closely by Conficker A.

Just ransomware alone caused $5 billion in damages in 2017, and that number is expected to increase to $11.5 billion this year [11]. Thus, it is of great interest to security researchers to develop a better understanding of DGAs and create new tools to combat them.

5.0.1 Domain Generation Algorithms

We focus on using the capabilities afforded by dynamic slicing to analyze different families of DGA malware.

5.0.2 Classification

As a first step, we attempt to identify DGA features and families via dynamic slicing. As sources of ground truth, we leverage several existing repositories of domain generation algorithms extracted from malware [1] [2] [3], as well as prior work from Plohmann et. al. on classifying DGAs [13]. There are several distinguishing traits of DGAs that we can attempt to categorize: origin of the seed and generation scheme for producing domain names; namely, the alphabet or arithmetic algorithm employed.

We choose several pieces of malware represented in the DGA database for testing.

DGA seeds

The first, Ramdo, is a click-fraud malware that, once installed, silently clicks online advertisements for financial gain. It uses a DGA to generate some number of 16-character domains with the `.com` TLD. The corresponding C++ code for the DGA is given in Listing A.3.

In the case of Ramdo, we can perform a slice on a single domain name that is output from the program. When running live malware under PANDA, one would have to capture the domain names using the `network` plugin, which dumps observed network traffic to a PCAP file.

A slice on this modest DGA is already quite large, with the program dependence graph containing 216 nodes and . The full graph is shown in Figure 5-1, following the

[1] https://github.com/pchaigno/dga-collection
[2] https://github.com/baderj/domain_generation_algorithms/
[3] https://github.com/andrewaeva/DGA

same color scheme as the graphs in Chapter 4. While the computation of the graph is difficult to grasp in its entirety, inspecting the topmost portion of the graph reveals the root of the data flow tree: the 4-byte hard-coded seed of the DGA, `0xdeadbeef` (Figure 5-2).

In reality, the 4-byte seed of Ramdo is configurable and predefined by the malware author, so the PDG of a sample of Ramdo seen in the wild would have similar structure, but different values along its computation tree.

In this case, the malware can be identified as deterministic and time-independent. Furthermore, the seed itself is never modified during the execution of the program.

As another analysis, we look at Ranbyus, a banking trojan that collects financial information and personal data from an infected victim's computer.

Unlike with Ramdo, it's not immediately obvious what the seeds for the DGA are, just by visual inspection of the PDG. Instead, the PDG of Ranbyus shows us why that might be: its most prominent feature distinct from Ramdo is the presence of several large loops. Loops in the PDG signify that an algorithm has some internal state that it is reusing/updating. These types of loops are also observed in common hash functions such as MD5 and SHA1, which are calculated by repeatedly operating on an internal state that is 16 bytes and 20 bytes, respectively.

There are four large cycles in this graph — this can be determined by a graph analysis library such as Python's `networkx`. The cycles are boxed in red in Figure 5-3 and enlarged in the following figures.. In each case, the initial value of the state is displayed at the root of the loop — `0` (Figure 5-4), `0xb` (Figure 5-5), `0x74` (5-6), and `0xdeadbeef` (Figure 5-7). These values correspond to the local day, month, and year (offset from 1900) on the guest VM, as well as the initial seed to the DGA. Without knowing what the guest system's time is, it would be difficult to determine that these bytes are time-dependent, however. With this information, it's possible to see that Ranbyus is time-dependent, but also contains a configurable 4-byte seed. Time-dependence is perhaps the most common type of malware DGA — 25 of 43 malware families studied by Plohmann et. al. were of this form.

The full DGA is given in Listing A.4, where it can be seen that the local time and

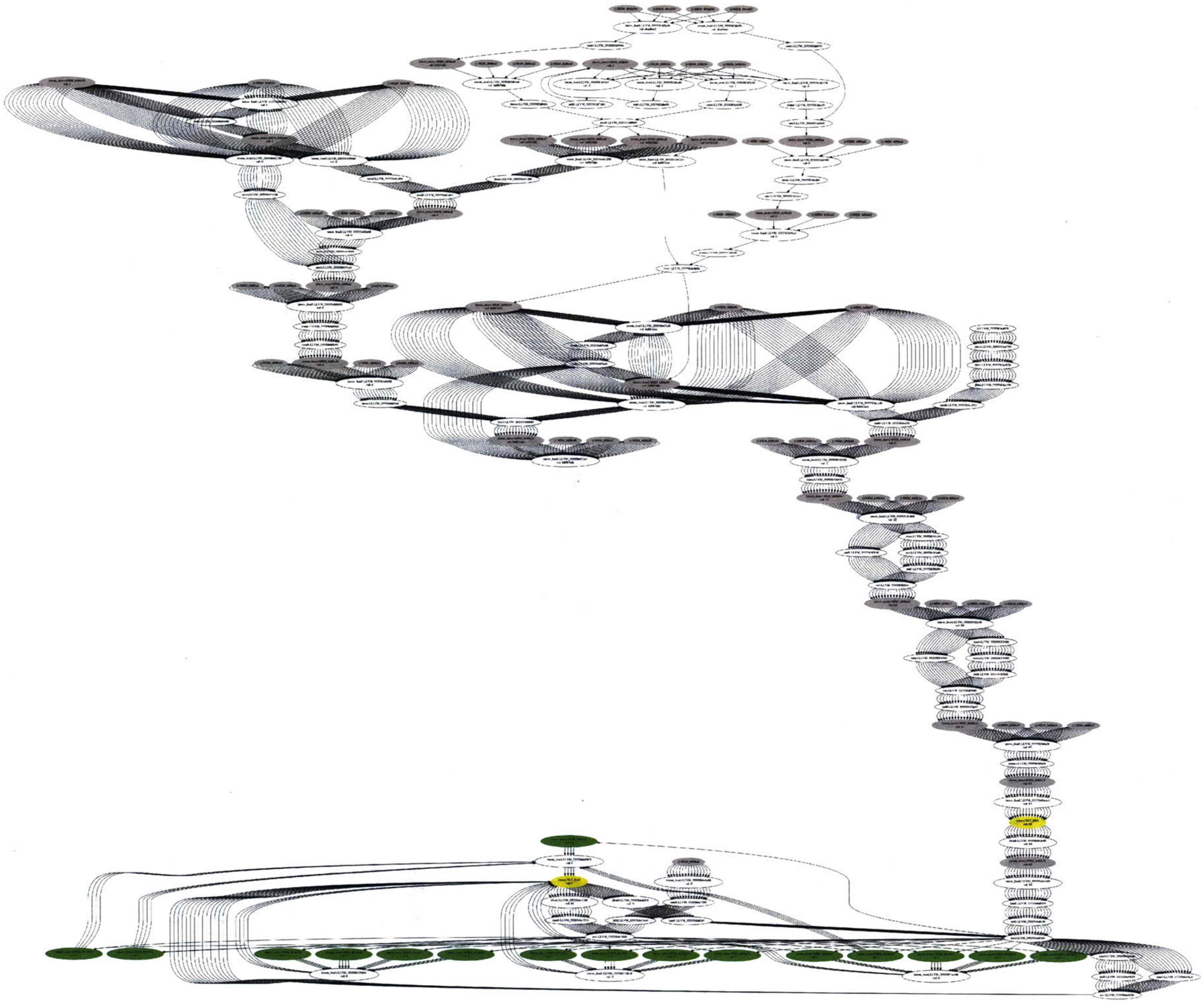


Figure 5-1: Ramdo domain generation algorithm PDG

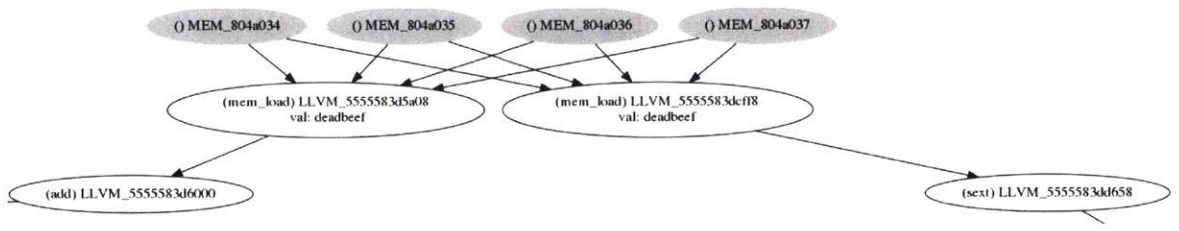


Figure 5-2: Ramdo PDG input seed

Table 5.1: Time spent to produce slice

malware name	generate recording	generate trace	perform slice + graph generation	total time
Ramdo	3s	55s	16s	74s
Ranbyus	4s	82s	10s	96s

seed are updated upon producing each byte of the domain. This leads to a heuristic for identifying DGA internal state from data flow graphs: the presence of loops in the PDG that are repeatedly involved in the computation of the output domain.

The total time to produce the PDG from the malware DGA sample trace is under two minutes. Table 5.1 shows the time taken to execute each phase of slicing for Ranbyus and Ramdo — for Ranbyus, 4s to generate the recording, 82 seconds for trace generation, and 10s to perform the slicing algorithm/graph generation.

By comparison, manually reversing each piece of malware is much more difficult. Plohmann et. al., presumably very experienced reverse engineers/malware analysts, spent one day on each piece of malware. From past experience, it's expected that much of this time is spent defeating malware's layers of static obfuscation — code packers, disassembler confusion, etc. With a dynamic slicing system, all of the difficulties in static reversing can be bypassed, as well as many dynamic obfuscation techniques such as dead code insertion. Thus, the manual effort of reversing DGAs can be drastically reduced.

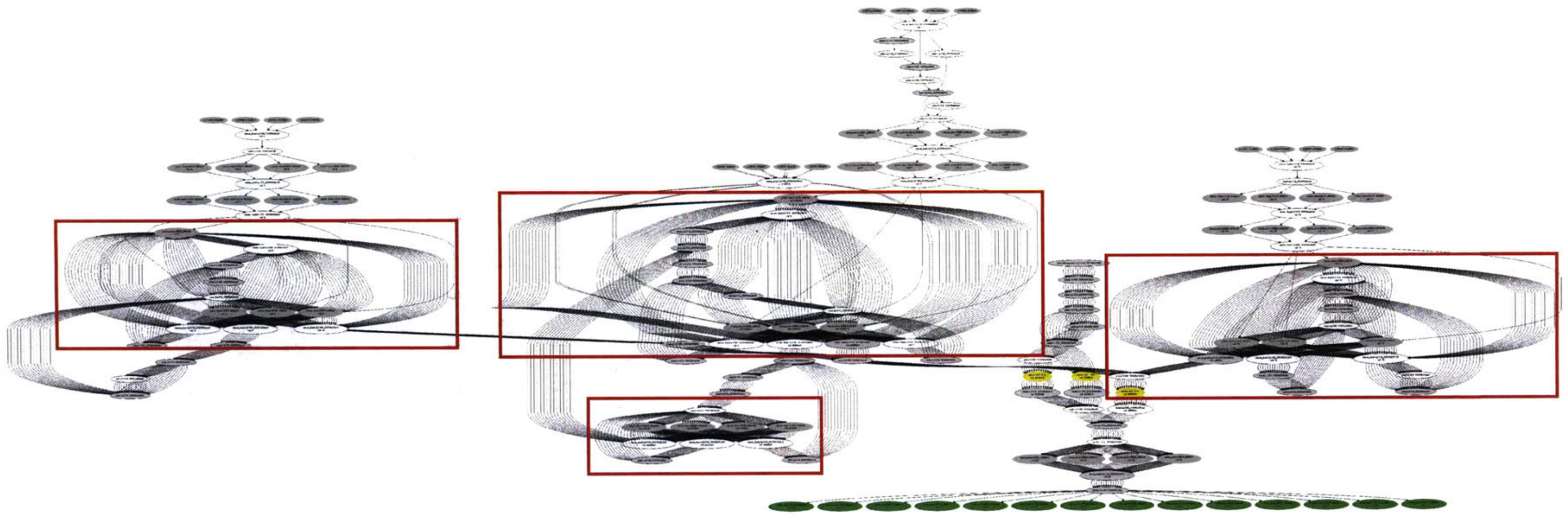


Figure 5-3: Ranbyus domain generation algorithm PDG

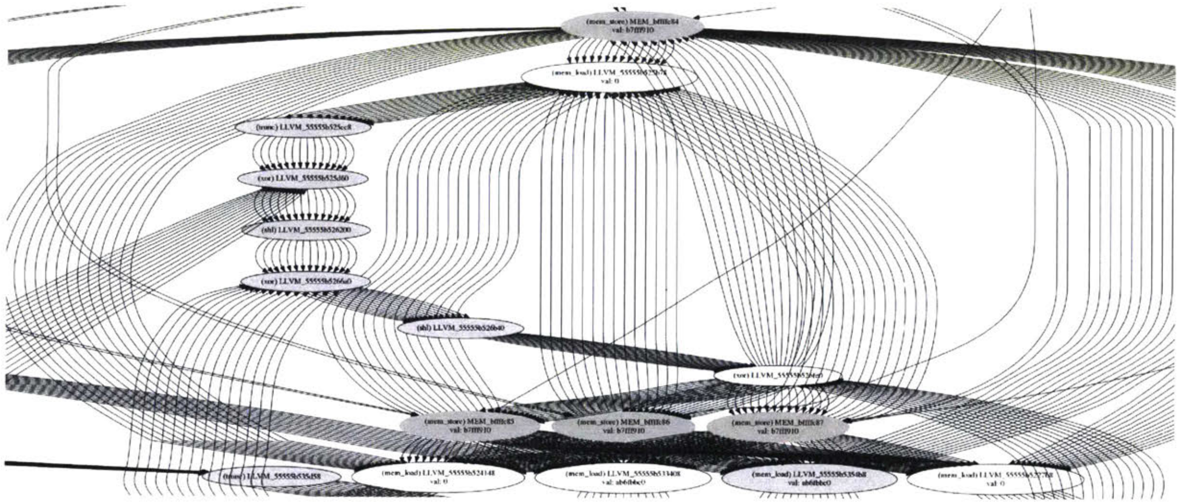


Figure 5-4: Leftmost cycle: byte containing day

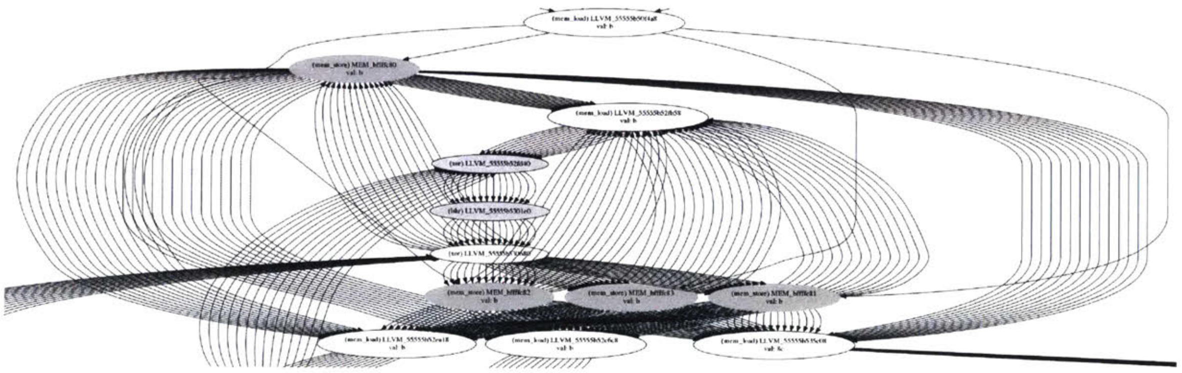


Figure 5-5: Middle cycle: byte containing month

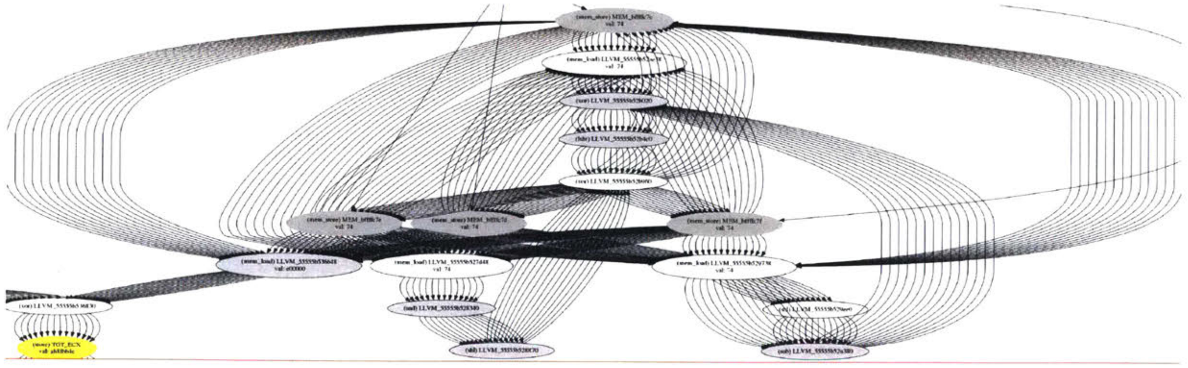


Figure 5-6: Rightmost cycle: byte containing year

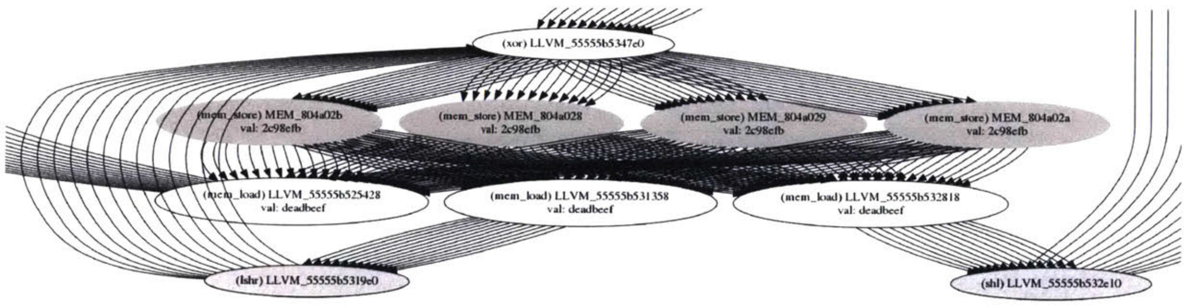

(xor) LLVM_55555b5347e0
(mem_store) MEM_804a02b val: 2c98efb
(mem_store) MEM_804a028 val: 2c98efb
(mem_store) MEM_804a029 val: 2c98efb
(mem_store) MEM_804a02a val: 2c98efb
(mem_load) LLVM_55555b525428 val: deadbeef
(mem_load) LLVM_55555b531358 val: deadbeef
(mem_load) LLVM_55555b532818 val: deadbeef
(lshr) LLVM_55555b5319e0
(shl) LLVM_55555b532e10

Figure 5-7: Bottom cycle: byte containing seed

Chapter 6

Conclusion

In this thesis, we have demonstrated a compelling use of dynamic analysis for program understanding in the form of dynamic slicing. We implemented a dynamic slicer using data collected from a PANDA full-system recording and LLVM instrumentation. Then, we take steps to analyze this output via traditional disassembler tools, as well as Program Dependence Graphing. We first apply this analysis to string decoding routines, such as common substitution ciphers and Base64. Then, we investigate the Domain Generation Algorithms employed by malware to produce Internet domain names for C&C, attempting to determine the structure and state of DGAs based on a dynamic slice of the observed domain names.

The results demonstrate that dynamic slicing captures a full picture of a system's computation of some result state, but requires the correct visualizations to be useful to reverse engineers. The goal is always to reduce the human effort that malware analysts and reverse engineers need to manually understand complex programs, and dynamic slicing is just one powerful tool in the toolkit of automated program analysis.

6.1 Future work

6.1.1 Further exploration of malware

The DGAs studied in this paper are far from the most complex observed in the wild. Malware have been known to use very unique sources of entropy for its seed, such as foreign exchange rates and Twitter trends, etc. [13]. In addition, some malware draw from a distinctive alphabet, such as text files downloaded from NASA, the Ninety-Five Theses by Martin Luther, and the DNS RFC (Gozi). While we did not perform slicing or program visualization on these samples, it would be interesting to see what benefit slicing can provide in these unique cases.

Another rich area of study for malware string manipulation is obfuscation of functionally significant strings in the program, such as filepaths, to deter reverse engineers from determining the true functionality of all or part of the binary. There are many techniques that malware employ to this effect — custom string decoding algorithms, or redundant computations that drown out significant instructions, etc. It's expected that dynamic slicing would be able to both locate key features of such decoding algorithms and pinpoint the relevant code among the chaff.

6.1.2 Expanding dynamic analysis

Dynamic analysis is a fast-growing field in the security space.

There are many possible further applications of dynamic slicing, particularly in conjunction with other types of dynamic analysis. A large challenge for fuzz testing, a common dynamic vulnerability discovery technique, is creating a test harness from which to begin exploration of the program by mutating inputs. Dynamic slicing could be used to determine the initial system state to create the test harness, allowing faster fuzzing with less overhead.

Another potential application of a slice could be in dynamic reverse debugging. PANDA supports time-travel debugging in which the user can step backwards, as well as forward, when debugging a replay in a debugger such as GDB. Setting temporary breakpoints at the instructions that are part of a dynamic slice would allow the reverse engineer to reverse-continue the system to points of interest and inspect live system state.

While we didn't find the plain SCN metric useful for malware, we imagine that it has applications elsewhere, if it were combined with other information gathered from the system. The SCN would benefit from including a temporal component, in which one can see whether the heavy SCN computation is spread out across the whole program, or clustered in a small region at the beginning or end, which may help differentiate different algorithms.

6.1.3 Program Dependency Graphing

Much work can be done in reducing the clutter of our current graphs and extracting useful information in both a visual and automated fashion. Increased graph pruning and compaction would make the graphs more pleasing to the eye and easier to follow manually. More graph analysis enabled by graph libraries such as `networkx` could also help to distill these large graphs into a more readily comparable set of metrics, such as connectedness and vertex cover, which can help detect free variables, DGA seeds, etc.

While there is still a vast amount to explore, this thesis lays the preliminary groundwork for slicing-driven program understanding.

Appendix A

Code Listings

```
int decode(void *out_buf, size_t out_len, const void *in_buf, size_t
    in_len, unsigned char key) {
    if (out_len != in_len) {
        return -1;
    }

    for (unsigned int i = 0; i < out_len; i++) {
        ((char *)out_buf)[i] = ((char *)in_buf)[i] ^ key;
    }
    if (out_len > 0) {
        ((char *)out_buf)[out_len - 1] = 0;
    }

    return 0;
}
```

Listing A.1: Single-byte XOR

```
char *key = "dRSTzVefgHIhUJK2tjpLqru.y0is1vwOPxEF3l4kcWXn/
   o7Ym9BCD5MN6GQa8AbZ";

void __cdecl substitution_cipher(char * encoded, int len) {
  int i;
  int keyoffset;
  char *pEncodedChar;
  int c;

  for (i = 0; i < len; i++)
  {
   keyoffset = 0;
   pEncodedChar = &encoded[i];
   c = encoded[i];

   keyoffset = ((char *)memchr(key, c, 64) - (char *)key);

   if ( keyoffset >= 26 ) {
    if ( keyoffset >= 52 ) {
     if ( keyoffset < 64 )
      c = (keyoffset - 6);
    }
    else {
     c = (keyoffset + 39);
    }
   }
   else {
    c = (keyoffset + 97);
   }

   *pEncodedChar = c;
  }
}

int main(int argc, char **argv){
  char in_out[] = "xzhhKDmKjhT";
  substitution_cipher(in_out, strlen(in_out));
}
```

Listing A.2: Substitution cipher

```
char domain[17];

int main(int argc, const char * argv[]) {
    unsigned int initial_seed = 0xDEADBEEF;
    int domain_iterator = 0;
    int numdoms = 0;

    while (numdoms < NUM_ITERATIONS) {
      unsigned int xor1 = 0;
      unsigned int shl = initial_seed << 1;
      domain_iterator += 1;
      unsigned int step1 = domain_iterator * shl;
      unsigned int step1b = domain_iterator * initial_seed;
      domain_iterator -= 1;
      unsigned int iter_seed = domain_iterator * initial_seed;
      unsigned int imul_edx = iter_seed * 0x1a;
      xor1 = step1 ^ imul_edx;
      int domain_length = 0;
      while(domain_length < 0x10)
      {
          unsigned int xor1_divide = xor1 / 0x1a;
          unsigned int xor1_remainder = xor1 % 0x1a;
          unsigned int xo1_rem_20 = xor1_remainder + 0x20;
          unsigned int xo1_step2 = xo1_rem_20 ^ 0xa1;
          unsigned int dom_byte = 0x41 + (0xa1 ^ xo1_step2);
          char dom[3];
          sprintf(dom, "%c", (uint8_t) dom_byte);
          strcat(domain, dom);
          unsigned int imul_iter = domain_length * step1;
          unsigned int imul_result = domain_length * imul_iter;
          unsigned int imul_1a = 0x1a * imul_result;
          unsigned int xor2 = xor1 ^ imul_1a;
          xor1 = xor1 + xor2;
          domain_length += 1;
      }
      strcat(domain, ".com");
      printf("%s\n", domain);
      domain_iterator +=1;
      numdoms += 1;
    }


    return 0;
}
```

Listing A.3: Ramdo DGA

```
char domain[15];
unsigned int seed = 0xdeadbeef;

int genDate(unsigned int date[]){
    unsigned int year;
    unsigned int month;
    unsigned int day;

    time_t currTime = time(NULL);
    struct tm *localTime;
    localTime = localtime(&currTime);
    year = localTime->tm_year & 0xff;
    month = localTime->tm_mon + 1; // Just plain ol' month - Number of
        months since January + 1
    day = (localTime->tm_mday / 7) * 7;

    date[0] = day;
    date[1] = month;
    date[2] = year;
    return 1;
}

char* dga(unsigned int day, unsigned int month, unsigned int year,
        unsigned int nr)
{
   printf("%d,␣%d,␣%d\n", day, month, year);
    char *tlds[] = {"in", "me", "cc", "su", "tw", "net", "com", "pw", "
        org"};
    int d;
    int tld_index = day;
    for(d = 0; d < 1; d++)
    {
        unsigned int i;
        for(i = 0; i < 14; i++)
        {
            day = (day >> 15) ^ 16 * (day & 0x1FFF ^ 4 * (seed ^ day));
            year = ((year & 0xFFFFFFF0) << 17) ^ ((year ^ (7 * year)) >>
                11);
            month = 14 * (month & 0xFFFFFFFE) ^ ((month ^ (4 * month))
               >> 8);
            seed = (seed >> 6) ^ ((day + 8 * seed) << 8) & 0x3FFFF00;
            int x = ((day ^ month ^ year) % 25) + 97;
            domain[i] = x;
        }
        printf("%s.%s\n", domain, tlds[tld_index++ % 8]);
    }
}

int main (int argc, char *argv[])
{
    unsigned int date[3];

    genDate(date);
    //dga(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), 1);
    dga(date[0], date[1], date[2], 1);
    return 0;
}
```

Listing A.4: Ranbyus DGA

Bibliography

[1] The boost graph library (bgl). `https://www.boost.org/doc/libs/1_66_0/libs/graph/doc/`.

[2] Documentation/tcg. `https://wiki.qemu.org/Documentation/TCG`.

[3] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6):589–616, 1993.

[4] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation - PLDI 90*, 1990.

[5] Pieter Arntz. Explained: Domain generating algorithm, Dec 2016.

[6] Thomas Barabosch, André Wichmann, Felix Leder, and Elmar Gerhards-Padilla. Automatic extraction of domain name generation algorithms from current malware. 2012.

[7] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, Jan 1987.

[9] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] Steve Morgan. Global ransomware damage costs predicted to hit 11.5 billion by 2019, Oct 2018. `https://cybersecurityventures.com/ransomware-damage-report-2017-part-2/`.

[12] Nicholas Nethercote and Alan Mycroft. Redux a dynamic dataflow tracer. *Electr. Notes Theor. Comput. Sci.*, 89:149–170, 10 2003.

[13] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A comprehensive measurement study of domain generating malware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 263–278, Austin, TX, 2016. USENIX Association.

[14] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS 13*, 2013.

[15] Frederick Ulrich. Exploitability assessment with teaser, 2017.

[16] Kenton Varda. Protocol buffers: Google's data interchange format. Technical report, Google, 6 2008. `http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html`.

[17] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[18] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.