Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE **Practical, Large-scale Detection of Obfuscated Malware Code via Flow Dependency Indexing**

PRESENTED BY **Wesley Jin**

ACCEPTED BY THE DEPARTMENT OF

Electrical and Computer Engineering

Priya Narasimhan — ADVISOR, MAJOR PROFESSOR — 4/26/14 — DATE

Jelena Kovacevic — DEPARTMENT HEAD — 4/26/14 — DATE

APPROVED BY THE COLLEGE COUNCIL

Vijayakumar Bhagavatula — DEAN — 4/26/14 — DATE

Practical, Large-scale Detection of Obfuscated Malware Code via Flow Dependency Indexing

Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
in
Electrical Engineering and Computer Science.

Wesley O. Jin
B.S., Computer Engineering, Massachusetts Institute of Technology

Carnegie Mellon University
Pittsburgh, PA 15213

May 2014

Keywords: Obfuscated Malware, Inverted Index, PDG

To my parents and everyone else who have refused to give up on a lost cause.

Abstract

Malware analysts often need to search large corpuses of obfuscated binaries for particular sequences of related instructions. The use of simple tactics, such as dead code insertion and register renaming, prevents the use of conventional, big-data search indexes. Current, state of the art malware detectors are unable to handle the size of the dataset due to their iterative approach to comparing files. Furthermore, current work is also frequently designed to act as a detector and not a search tool.

I propose a system that exploits the observation that many data/control-flow relationships between instructions are preserved in the presence of obfuscations. The system will extract chains of flow-dependent instructions from a binary's Program Dependence Graph (PDG). It will then use a representation of each chain as a key for an index that points to lists of functions (and their corresponding files). Analysts will be able to quickly search for instruction sequences by querying the index.

Acknowledgments

Thesis Committee: Priya Narasimhan, *CMU* (Advisor), Lujo Bauer, *CMU*, Stacy Prowell, *Oak Ridge National Laboratory*, Anthony Rowe, *CMU*.

I'd like to thank my family for their support and unconditional love. I'd like to thank my advisor Priya. You've given me a chance to pursue my interests. Finally, I'd like to thank Cory and Chuck. You've shown me what it means to do malware analysis in the real world. Finally, I would like to thank Priya's industry sponsors and Computer Emergency Response Team (CERT) for their financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, malicious binary code (i.e., malware) has grown exponentially in both number and sophistication. This phenomenon is particularly evident at organizations like CERT [8], who collect and analyze these files on a large scale. CERT's private collection more than doubled during the period from 2012 to 2013 [15]. Analysts often need to pinpoint files, containing specific instruction sequences (e.g., custom encryption implementations, command and control handlers, etc.), in these vast datasets. This ability is particularly valuable when studying a *family*, a group of malware that share a common codebase.

During the course of an investigation, analysts typically begin by analyzing one or two samples collected from an infected machine [15]. In order to understand more about the origins of the file, they try to identify other possible variants by searching for code patterns that they suspect to be unique to the malware's codebase (See Fig. 1.1). By studying *differences* among these files, analysts learn about the evolution of the group and new capabilities (and weaknesses that could be exploited for detection purposes) with each generation.

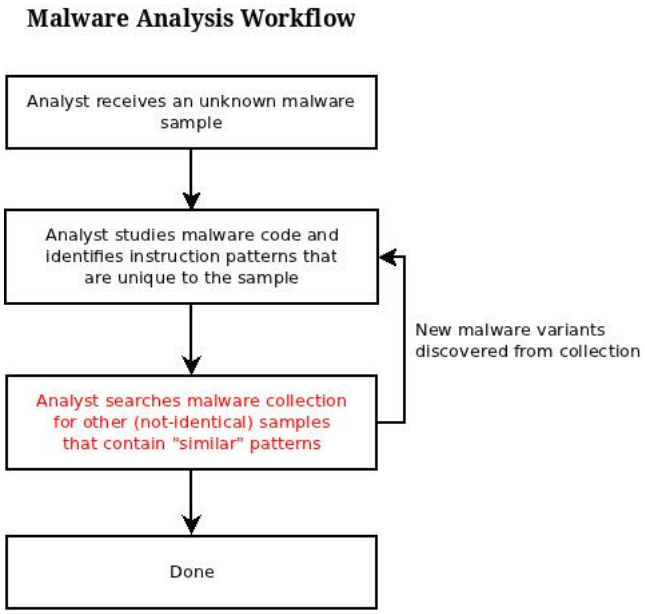


Figure 1.1: The focus of this thesis is on improving the search step, highlighted in red.

However, searching large collections of binaries is problematic for two reasons. First, the large number of files makes the use of common, iterative search tools, such as `grep`, impractical. For instance, a single `grep` over millions of files can take days to complete [15]. Second, malware authors often use obfuscation techniques that cause signature/regular expression matching to fail. Thus, the goal of this thesis is to give malware analysts the ability to quickly search for instruction sequences in large corpuses of obfuscated malware.

1.1 Related Work and Challenges in Detecting Instruction Patterns

Current work fails to address three issues that are crucial to meeting the needs of analysts:

1. **Scalability** of an approach is measured based on: 1) the speed at which queries return results (i.e., the latency of the search) and 2) any temporal or disk overhead associated with conducting the search.

2. **Search Granularity** refers to the level of detail that can be used to search a dataset. An example of a query with a fine granularity is one that would be able to identify files containing a byte-level pattern. An example of a query with a coarser granularity is one that would be able to identify files matching some representation of a function.
3. **Query Generation Costs** refers to the effort that must be invested by an analyst to generate a query, measured in time and complexity (i.e., lines of code, amount of tuning, etc.).

1.1.1 Scalability

State of the art malware detectors, such as those described in [11, 34, 37], are not designed to be search tools. Rather, they are built to determine whether a *pair* of obfuscated binaries could have come from a common codebase (i.e., belong to the same malware family). This assumption that only two files are involved leads to the following problems:

1. It allows them to use comparison techniques that would take too long when more files are involved. Furthermore, these techniques would have to be re-applied each time an analyst wishes to conduct a different query.
2. It allows them to choose a representation of malware that might require too much disk space for a large number of files.

Christodorescu et al. [12] introduce the notion of behavioral templates, graphs representing the high-level behavior of programs. Their approach represents the semantics of individual instructions as vertices, connected by edges that represent control-flow dependencies. Their detector tries to find an assignment for each node in a malware template to a node in the graph of an unknown program (subject to constraints generated by the def-use relationships between instructions), using an SMT solver. Unfortunately, from our own experiences with SMT solvers (e.g., Microsoft Z3 [41] and Yices [1]), solving millions of constraints per query does not scale.

Alzarouni et al. [34] introduce the notion of semantic signatures. Malware functions are analyzed and run in a semantic simulator, producing a set of input and output traces. Variables at the end of the output traces are *sliced* to produce shorter traces (consisting of only the portions of the program relevant those variables). A signature is a set of pairs of input values and their corresponding sliced traces. Similar to Christodorescu et al., this approach is able to overcome many simple of the obfuscation tactics like dead code and register renaming. However, the algorithm used to match semantic signatures is iterative. A mapping between sets of values would have to be established for each file, each time a query is to be performed. Furthermore, it is unclear how detailed traces, containing memory and register state information, could be maintained for millions of functions and files.

Leder et al. [37] propose a scheme that involves comparing Value Sets, the contents of registers and memory at 'points of interest', such as the beginning of loops, function calls, jumps, etc. However, as the authors admit themselves, generating Value-Sets, via static data-flow tracking, for large numbers of files runs into scalability issues. Furthermore, control-flow obfuscations that break apart basic blocks or introduce extraneous jumps would have a serious impact on points of comparison.

1.1.2 Granularity

Detectors, such as those described in [7, 9, 50], are designed to operate at a whole-program or function level granularity. Said differently, they compare properties that describe an entire program (i.e., inter-procedural control-flow graphs (CFGs) in [7], API call sequences in [50], function-call graphs in [28]) or a function i.e., intra-procedural CFGs in [9]. The idea is that by comparing programs at a higher-level of abstraction many lower-level obfuscations can be ignored. For example, by comparing CFGs, NOPs and *some* junk instructions become irrelevant. Unfortunately, this approach is problematic for two reasons:

1. Not all instruction-level obfuscations are eliminated at coarser granularities of comparison.

For example, in Section 5.1, we present a case study of a family of malware that inserts useless API calls (i.e., API calls that do not impact the overall semantic behavior of the program[1]) to obfuscate its control-flow graph and API call sequence. Indeed, our work takes the opposite approach. The idea is that is very difficult to eliminate *all* obfuscations. Therefore, it would be better to design a system that can operate at an instruction-level granularity despite the presence of obfuscations.

2. Malware analysts may only be interested in a short sequence of related instructions (not the entire program or function).

Bruschi et al. [7] propose a system that compares inter-procedural, control-flow graphs (CFGs), using the VF2 algorithm. First, pair-wise comparison with VF2, which has a best-case time complexity of $O(N^2)$, for large datasets would not scale. Furthermore, as discussed above, matching procedural-level, control-flow graphs may not suit the needs of analysts, and can be disrupted by obfuscations.

Cesare and Xiang [9] also propose an algorithm for determining program similarity by estimating isomorphisms between control-flow graphs. Specifically, they generate function signatures from the edges of the CFG (i.e., hashes of the control-flow dependencies between functions). Although this approach allows for quick and storage-efficient identification of functions with matching CFGs, the use of basic-block level hashes causes information about specific instruction sequences to be lost.

Shankarapani et. al [50] propose a system for extracting and storing sequences of API calls. The sequences of two programs are compared using metrics like Cosine similarity, Extended Jaccard, and Pearson correlation. Once again, using pair-wise comparison metrics does not scale. API calls sequences may not be precise enough to allow analysts to find what they are looking for.

[1]Determining that they are useless is challenging, because the return values of the API calls can be used by other junk instructions

1.1.3 Query Generation Costs

One of the primary complaints of analysts is that current detection tools 'are difficult to tune' [15]. More explicitly, it is often unclear how analysts would modify existing tools to perform instruction search, or it would take too long/be too complicated to do so. Again, this problem arises because most state-of-the art tools are not meant for search.

Analysts are most comfortable with assembly instructions, as they spend a significiant amount of time using disassemblers like IDA Pro [46]. Therefore, they naturally gravitate towards tools like YARA [58], a `grep`-like tool designed for malware analysis. YARA has many features that make it amenable for identifying binary patterns, such as regular expressions and rules that are applied to the structure of an executable. Unfortunately, simple obfuscations such as dead code are often sufficient to cause queries to fail (see Table 2.6). Furthermore, YARA performs pairwise comparisons, and takes days to return results when used on millions of binaries [15].

1.2 Thesis Statement

Focusing on the challenge of searching large collections of obfuscated malware for particular instruction patterns, I propose the following thesis statement:

Thesis. *A search index that maps unique data/control-flow dependencies between instructions to functions in which they can be found is a practical approach for quickly identifying obfuscated code in large datasets of binaries.*

In summary, the goal of this work is to develop a system to provide an analyst-intuitive search and query capability for obfuscated instructions. Specifically, I propose to create a search index for data and control flow dependencies between assembly instructions. The idea is that many obfuscations alter the syntax of code, but largely preserve dependency flows. The proposed

system will enumerate and extract instruction relationships by identifying unique subgraphs of a binary's Program Dependence Graph (PDG) [19], a data structure that represents instructions as vertexes and data/control dependencies as edges between vertexes. These subgraphs represent a sequence of flow-dependent instructions, which will be used as keys within the index. Upon completion, the index will contain a mapping between unique subgraphs to all files/functions that contain those relationships. Thus, analysts will be given the ability to quickly identify all files, containing instructions that have particular dependency relationships, by querying the index.

1.3 Contributions

This work makes the following contributions:

1. A technique for quickly identifying obfuscated code by indexing functions that share common sequences of flow-dependent instructions.
2. A representation of these instructions as keys for a search index, which minimizes query-time and storage overhead.
3. An implementation and evaluation of the index on a real world dataset from the CERT Artifact Catalog.
4. Case studies involving real-world, obfuscated malware.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows: Chapter 2 reviews background material about malware obfuscation strategies. Chapter 3 discusses the design and implementation of the index, PdgGrep. Chapter 4 analyzes the construction costs and query performance of PdgGrep. Chapter 5 presents three case studies on real world malware that demonstrate PdgGrep's utility in identifying obfuscated code. Chapter 6 discusses the limitations of the proposed system

and directions for future work. Chapter 7 concludes this thesis. The appendix **??** contains the user's testimonial.

Chapter 2

Background: Anti-detection and Analysis Techniques

In this chapter, we review some of the tactics employed by malware to evade detection and thwart analysis post-discovery. We conclude each discussion by explaining the relevance of the technique to current work and our own.

1. Code obfuscation techniques are program transformations that alter the structure of an executable, but leave its semantics *mostly* unchanged. Structural changes allow malware to evade static anti-virus signatures, and complicate code understanding. Many of these tactics have already been discussed extensively in the literature [7, 13, 44, 59].
2. Anti-disassembly and anti-emulation techniques are strategies employed specifically to delay analysis after the malware has been discovered. The first is intended to prevent the recovery of functions in a disassembler. The latter is intended to create problems for dynamic analysis by hiding the malware's true functionality. [6]
3. Packing and encryption alter the stored representation of malicious code (when the program is not being run) using compression or some invertible operation. They then reverse these changes just prior to execution. [5, 26, 32, 48]

2.1 Anti-disassembly

Before static analysis of a piece of malware can begin, analysts must obtain an accurate disassembly of the program's instructions. Two types of disassemblers can be used for this purpose: linear sweep or recursive traversal. Malware authors exploit difficulties in statically determining the control-flow of a binary to cause problems for these tools.

Linear sweep disassemblers produce instructions sequentially. They begin at a starting address, produce an instruction, and calculate its size. They then shift to the next address, the sum of the old address and the calcuated size, to produce the next instruction. This process repeats until the end of the program. Armed with this knowledge, a common anti-disassembly tactic is to insert junk/non-instruction bytes between legitimate instructions as in Table 2.1. These junk bytes are interpreted as instruction bytes (opcode, operand, etc.), leading to an incorrect disassembly as show in Table 2.2.

401040	jmp DST	Linear sweep tries to make code following jmp
401042	db 0xff	Bad byte
DST:	pop eax	Valid code
401044	add eax,1	

Table 2.1: A data-byte inserted into a sequence of assembly instructions

401040	jmp DST	
401042	call [eax+5]	Data byte 0xff misinterpreted as call
401047	add [eax], eax	Bad disassembly
401049	add [eax], al	

Table 2.2: Incorrect disassembly produced by linear sweep algorithm

Recursive disassemblers also produce instructions sequentially, but attempt to follow control-flow to find other code locations as well. (i.e., they will follow the branch target of a jump and begin making code at that address.) Therefore, a recursive disassembler would have produced the correct disassembly in the previous example. However, malware authors often write code

that make control-flow difficult to determine without actually evaluating the instructions. For example, in Table 2.3, the disassembler will follow the jump target at the `jnz` instruction and attempt to make code at `BAD`, even though that branch will never be taken during execution (because the preceding arithmetic operations will cause the `zero` flag to always be set).

`401000`	`mov eax, 4`	
`401005`	`shr eax, 2`	
`401008`	`sub eax, 1`	
`40100d`	`jnz BAD`	Branch never taken, because eax is always 0
`401014`	`jmp GOOD`	
`BAD:`	`db 0xff`	Bad byte
`GOOD:`	`...`	Valid code

Table 2.3: Code snippet containing conditional jump that is never taken

2.1.1 Relevance

Obtaining correct disassemblies of malware binaries is a challenging problem [33, 36], and is one of the reasons for the popularity of dynamic techniques (used by [3, 14, 18, 28, 52]). Both approaches have strengths and weaknesses. In dynamic analysis, malware instructions are evaluated in a simulated environment. Therefore, by monitoring the flow of execution, the addresses of malicious code and behavioral traits, such as API call sequences, can be discovered without detailed reverse-engineering of code *in certain cases*. However, as will be discussed in the next section, dynamic analysis can be thwarted by anti-emulation techniques [10, 45]. It is also slow and potentially dangerous [38]. Static analysis can be hindered by a variety of techniques that will be discussed in this chapter. However, it is fast and safe[38]. Most importantly, it provides a greater level of detail of how malicious functionality is actually implemented.

For the latter reasons, our approach uses the ROSE framework [47] for disassembly. As will be discussed in Section 3.2, the disassembler combines the traditional recursive disassembly algorithm with limited emulation: The instructions in discovered basic blocks are evaluated in a ROSE's symbolic execution engine to help discover potential branch targets. API calls are not

simulated. The disassembler is also highly tunable, and will provide function fragments even when well-formed (i.e., stack-neutral) functions cannot be created.

2.2 Anti-emulation

Emulators simulate the behavior of one computer system (the guest) in another (the host) such that the emulated system closely resembles the behavior of the real guest. Emulators typically consist of modules that correspond to each of the guest's major subsystems: a CPU simulator, a memory subsystem, and I/O device simulators. They have become a crucial component in the malware analyst's toolkit, because of their ability to provide an enclosed environment in which malware can be safely run and monitored.

Anti-emulation techniques are designed to exploit differences between virtual environments and real ones. They can be categorized into the following groups: detectors, control-flow obfuscators, and delayed execution. The common idea among the three is that it is very challenging to *exactly* replicate the behavior of a real system [32]. For example, it is not uncommon for an operating system (OS), running on an emulator like Qemu, to be more than 4x slower than the same OS running on real hardware [27]. Therefore, a common strategy used by malware to detect the presence of a virtual machine is to compare the time required to complete a task with some expected time. If the measured time exceeds the expected period, the program reasons that it is being analyzed, and terminates without revealing its malicious behavior.

Another common strategy is to utilize obscure components of a computer system, which may not have been implemented by an emulator, to perform control-flow decisions. This strategy is particularly relevant to the work in this thesis as it is equally effective against emulators used for symbolic execution as it is for full-system emulators like QEMU. Table 2.4 depicts the anti-emulation code from the malware family Virut[39]. The code exploits the fact that many emulators do not implement the structured exception handling mechanism found in Windows.

Address	Instruction	
40FABD	pusha	
40FABE	call 0x40FACD	Set call back field in EXCEPTION_REGISTRATION
40FAC3	call 0x40FB52	Exception call back destination
40FAC8	jmp 0x40FB08	
40FACD	push DWORD fs[0]	Set previous pointer in EXCEPTION_REGISTRATION
40FAD3	mov fs[0], esp	Register EXCEPTION_REGISTRATION structure
...	...	
40FADF	xor ecx, ecx	
...	...	
40FB01	push ecx	Pass NULL parameter to wsprintfA
40FB02	call ds:wsprintfA	Cause exception
...	...	

Table 2.4: Exception-handler abuse by Virut

The code above is actually a complicated NOP. It could be reduced to the single call instruction found at `40FAC3`. The idea behind this code is to create a Windows EXCEPTION_REGISTRATION structure[40] with a callback address that points to `40FAC3`. The code registers that structure in the process's Exception Handler chain, and generates an exception to jump to the registered address. In more detail, the `call` instruction at `40FABE` pushes the address of the next instruction `40FAC3` onto the stack, which serves as the callback address. The branch target of the `call` pushes the pointer to the old exception registration structure (located at `fs[0]`) onto the stack to set the previous pointer of the structure. The instructions from `40FADF` to `40FB02` generate an exception by passing a NULL parameter (ECX is 0 and passed as a parameter) to `wsprintfA`.

In a real Windows environment, the exception handling mechanism would call the code pointed to by the callback address. However, in an emulator that does not support exception handling the program's behavior is not as clear. Similar to the strategies discussed in the previous section, a malware author can use this technique to confuse program understanding and to produce incorrect disassembly.

2.2.1 Relevance

Emulating instructions in a single basic block, using ROSE's symbolic execution engine, would not be affected by tactics like timing attacks. Furthermore, basic blocks are evaluated only once. Therefore, disassembly would not be affected by long loops used by malware to 'outlast' an

emulator[1]. However, since full-system emulation is not performed (API calls are not simulated), complex control-flow obfuscations, like the one used by Virut, could hinder disassembly. In these cases, ROSE also uses a number of heuristics to detect code fragments that could not be linked to the control-flow of larger functions [47] to provide partial results.

2.3 Packing and Encryption

Packing and encryption are techniques used to hide malicious code as it is being transmitted and as it resides on disk. An encrypted piece of malware typically consists of a decoder stub and the ciphered bytes of the original/unencrypted malware. When the malware is executed, the decoder stub decrypts the ciphered payload using a key, which may vary from one sample to another, and then transfers control to the decrypted code. For example, Fig. 2.5 depicts the decoder stub used by the malware family, Hidrag [49]. The register `ecx` is loaded with the decryption key, 0x233CF0. `esi` is loaded with the offset of the encrypted payload, 0x40A020. `edx` is loaded with the size of the encrypted payload, 0x6BC. The `xor`, `sub`, and `jnz` instructions form a loop that replaces each byte of the encrypted payload with the bitwise-xor of itself and the key.

Packing refers to the use of compression techniques to obfuscate malware instructions, and reduce the overall size of the program. Malware authors commonly employ both techniques in their code.

Address	Instruction	
0040A000	`mov ecx, 233CF0h`	Decryption key moved to ECX
0040A005	`mov esi, (offset loc_40A020)`	Started of encrypted payload moved to ESI
0040A00A	`push 6BCh`	Size of encrypted payload moved to EDX
0040A00F	`pop edx`	
0040A010	`xor [edx+esi], ecx`	XOR decryption
0040A013	`sub edx, 2`	Decrypt counter/shift pointer
0040A016	`sub edx, 2`	
0040A019	`jnz short loc_40A010`	Continue until EDX is 0
...	...	

Table 2.5: Decryption stub from Hidrag sample

[1]Malware authors know that it is common practice to run a piece of malware in an emulated environment using a timeout (5min). Therefore, they will often include delays in their code before performing any malicious activities

2.3.1 Relevance

Typically, analysts are interested in the code that has been obscured by packing and encryption. Therefore, the first step of our approach is to process executables with an unpack engine (see Section 3.2).

2.4 Code Obfuscation Techniques

Code obfuscations tactics are used by malware to evade anti-virus (AV) detection and to hinder analysis upon discovery. They alter the raw bytes of the binary in an unpredictable manner, allowing it to evade static signatures, but preserve the semantics of the original/unobfuscated program. Techniques can be categorized based upon whether they complicate the program's control-flow or not. Those that do not modify control-flow may add additional instructions that do not interfere with the original code (i.e., dead code, NOPs). Alternatively, they might also alter the original malware's instructions in a way that preserves their functionality (i.e., register permutation, semantically equivalent instructions), but changes the instruction bytes of the executable. Table 2.6 provides an example of two polymorphic variants camouflaged using dead code, register permutation and differing constant values. Both samples have the same basic functionality: compute a position-independent address relative to the current location in memory, and then jump to it. The `call +5` instructions push the address of the next instructions (i.e., the addresses of the `pop` instructions) onto the stack. The `pop` and `add` instructions retrieve this pushed value and add an offset to it (i.e., `1EEh` and `1EFh`). The computed address is then pushed onto the stack by the `push` instruction, and branched to by the `ret`. The use of different registers means that the bytes of each pair of instructions mentioned above do not match in both samples (i.e., the bytes corresponding to both of the `push` instructions in both samples

are different, because they use different registers). Furthermore, the insertion of junk instructions (those not highlighted in red in the figure) produces additional differences between the two samples. However, despite these differences, note that the data-flow relationships between the relevant (red) instructions are the same in both cases.

Address	Instruction	Bytes	Address	Instruction	Bytes
401040	pusha	60	401040	call $+5	E800000000
401041	call $+5	E800000000	401045	pop %ebp	5D
401046	pop %edx	5A	401046	add %ebp, 1EFh	81C5EF010000
401047	add %edx, 1EEh	81C2EE010000	40104C	push %ebp	55
40104D	push %edx	52	40104D	cmp %al, %bh	3AC7
40104E	repne add %edi, 0EF8029E8h	F281C7E82980EF	40104F	xchg %dl, %cl	86D1
401055	retn	C3	401051	retn	C3

Table 2.6: Obfuscated malware variants. Actual instructions are highlighted in red.

Control-flow obfuscations add unnecessary branches or subroutines to a program. These extraneous jumps and function calls repartition the original instructions into new basic blocks and/or functions. In addition to complicating overall program understanding, these techniques are particularly problematic for malware detectors that rely on blocks and functions as logical boundaries for comparison. For example, detectors such as the one described in [7] work by comparing the CFG of a known malware family to that of an unknown sample. The introduction of new jumps changes which instructions are found in each basic block, and the number of total blocks. Therefore, variants of a control-flow obfuscated malware family may be able to evade such detectors.

Address	Instruction
BLOCK1:	mov ecx, 233CF0h
	mov esi, (offset ENCRYPTED)
	push 6BCh
	pop edx
BLOCK2:	xor [edx+esi], ecx
	sub edx, 2
	sub edx, 2
	jnz BLOCK2
	jmp ENCRYPTED

Table 2.7: Hidrag decryption loop without control-flow obfuscation

Address	Instruction
BLOCK1:	mov ecx, 233CF0h
	mov eax, 22h
	cmp eax, 0
	jg BLOCK4
BLOCK2:	mov esi, (offset ENCRYPTED)
	push 6BCh
	jmp BLOCK5
BLOCK3	xor [edx+esi], ecx
	sub edx, 2
	sub edx, 2
	jnz short loc_40A010
	jmp ENCRYPTED
BLOCK4:	jmp BLOCK2
BLOCK5:	pop edx
	jmp BLOCK3
...	...

Table 2.8: Hidrag decryption loop with extraneous jumps

Table 2.7 depicts the two basic blocks found in the xor-decryption loop of Hidrag, shown before in the previous section. Table 2.8 shows the same loop, except obfuscated using unnecessary jumps. Note that the number of basic blocks and their contents have changed. However, like many of the obfuscations that do not complicate control-flow described above, the data-flow relationships between the relevant instructions remain the same.

2.4.1 Relevance

The main focus of this thesis is to allow analysts to search for instructions in the presence of obfuscations like those seen above. Two key observations that will motivate our approach, described in the next chapter, is:

1. Data-flow relationships between instructions are often preserved even in the presence of many obfuscations.
2. Although some data and control-flow dependencies may be changed, it is very difficult for malware authors to alter *all* instruction-level relationships. Therefore, if analysts can identify a characteristic sequence of instructions that share a data/control-flow relationship, they may still be able to identify an obfuscated function. For this reason, our approach

operates an an instruction-level granularity.

Chapter 3

Data and Control-dependency Indexing for Instruction Search

In this chapter, we present the design and implementation of PdgGrep, a search index for data and control-dependent instructions in sets of obfuscated malware.

3.1 Architecture

The core idea behind PdgGrep is to group functions/files by common sequences of instructions that share one of the following relationships: 1) One instruction defines a register or memory location used by the other. 2) The execution of one instruction is conditionally dependent on the result of another.

The index is built as follows: For each binary, we compute the dependency relationships for each of its instructions. We use this information to build chains of inter-dependent instructions, called *dependence paths*, from which we produce search keys using a *KeyHash()*. Each key maps to a list of functions/files which contain that particular code sequence. Search is done in an analogous fashion. Users provide a sequence, from which we generate dependence paths and keys to look for.

The design of the system is motivated by the following considerations:

1. **Empowering the analyst.** Given the time to analyze a malware sample, human analysts are very effective at zeroing-in on the aspects that make that program unique. The goal of the proposed system is to allow users to express this understanding in the form of a search query. We emphasize that our objective is *not to build another automated detector*. Rather, we are trying to enhance a person's ability to quickly search large collections of obfuscated code.

2. **Preservation of dependency relationships in obfuscated code.** Despite the presence of common, code obfuscation tactics like dead code, NOPs and register permutation, many of the data and control-dependency relationships between functional instructions (i.e., those that actually contribute to the behavior of the program) are preserved. Therefore, we hypothesize that search keys constructed from flow-dependent instructions should be largely unaffected by these techniques. Note, it is often not important that *all* dependency relationships be preserved. To identify a particular malware variant in a collection of binaries, an analyst need only identify a *single* sequence of unique instructions.

3. **Instruction-level granularity.** Malware detection systems operate by comparing various aspects of a program, such as its function-call graph[28] or the set of memory values at particular program points[37], to that of a known malware family. A more abstract aspect (with instruction and byte-level comparison being the least abstract) results in a coarser granularity of comparison. In contrast to previous work, we believe that a coarser granularity is *more succeptible* to obfuscation tactics. For example, dynamic API resolution, used in malware families such as Destory RAT, can easily alter the function-call graph of a piece of malware. We hypothesize that it is more difficult for a malware author to alter *all* of the instruction-level relationships in his code than it is to produce an obfuscation that would change some high-level aspect of the program. Furthermore, analysts are often in-

terested in code patterns that are at an instruction-level (i.e., a particular encryption loop, a call to a particular function with specific parameters). Therefore, our system must operate at this fine-level of granularity to be useful.

4. **Fast search in large datasets.** One of the limitations of current search tools, like YARA [58], is that they perform pairwise comparison. Said differently, they are designed to compare some aspect of a single program with that of a known malware family. Unfortunately, when there are many files to be examined, this approach does not scale. This problem is especially true if the method of comparison is expensive. For example, an approach, such as the one presented by Christodorescu et. al [12], that uses an SMT solver to compare components of two files might be effective for a small number of files. However, it would be too slow for repeated queries involving large datasets.

5. **Search flexibility.** Malware analysts are often interested in looking for variants of instruction patterns. For example, they might be interested in a particular sequence of instructions in which certain operands matter and others do not. Our system is designed to allow users to specify constraints on different instruction operands.

6. **Index construction overhead.** Although reducing query-time is our most important goal, we are also mindful of the time required to construct the index and its disk requirements. As will be discussed in later sections, the choice of *KeyHash* represents a tradeoff between query-speed and disk usage.

7. **Simplicity of search interface.** One of the major complaints of malware analysts is that current, academic tools are complicated and difficult to tune. They would like a simple interface to perform queries.

Figure 3.1 illustrates the architecture of PdgGrep. Logically, the system can be divided into a back-end and a front-end. The back-end is responsible for populating the index, and is composed of the following parts: 1) A generic unpacking engine that detects and automatically unravels packed code, 2) ROSE [47] disassembler that extracts functions and code fragments from the

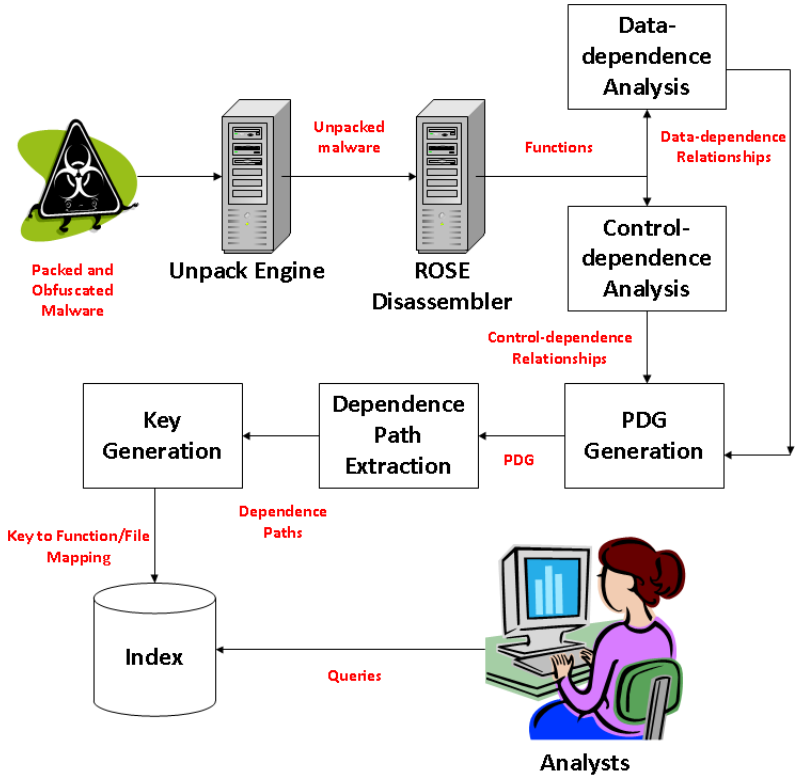


Figure 3.1: PdgGrep Architecture

unpacked executable, 3) Data and control-dependence analysis modules that compute the dependency relationships for each function's instructions, 4) PDG construction module that combines the information from both of the previous modules, 5) Dependence Path extraction that builds dependence paths for each of the instructions, and 6) Key generation that converts each dependence path into a key that points to a list of all functions which contain that path. The front-end is responsible for processing users' search requests. Each request is comprised of a sequence of instructions and constraints on how their operands should be compared. The front-end generates a set of keys from the instructions, retrieves the corresponding entries from the index, and returns a list of functions containing the sequence.

3.2 Unpacking and Disassembly

Each binary to be inserted into the index is first processed by an automated unpacking engine and the ROSE disassembler. Similar to the automated systems proposed by Kang et. al [32] and Bohne et. al [5], the engine uses various heuristics to detect packed code, such as branching to a memory address that was previously written to. The engine produces a memory dump, containing unravelled instructions, when necessary.

Unpacked executables are processed by the ROSE disassembler [47]. This disassembler is well suited for malware analysis for a couple of reasons. First, the disassembly algorithm uses symbolic execution to help determine potential locations to produce code. Consider the example that was previously discussed in the Anti-disassembly section, shown again here in Figure 3.1 . At the end of each basic block, ROSE emulates the instructions contained within the block to help determine potential successor addresses. In this example, ROSE would evaluate the three instructions upon arriving at the `jnz` instruction, and determine that the branch could never be taken (i.e., the Sign flag is always set, as `eax` will always be 0). Therefore, the disassembler would not produce code beginning at `BAD`, and is less susceptible to simple, anti-disassembly techniques.

401000	mov eax, 4	
401005	shr eax, 2	
401008	sub eax, 1	
40100d	jnz BAD	Branch never taken, because eax is always 0
401014	jmp GOOD	
BAD:	db 0xff	Bad byte
GOOD:	...	Valid code

Table 3.1: Code snippet containing conditional jump that is never taken

Second, the disassembler is flexible. It gives users the ability to implement their own function discovery algorithms, subroutines that take an ordered list of instructions and determine the starting addresses of procedures within that list. We added two heuristic-based detectors to ROSE.

3.2.1 Detector 1: Register Save and Restore

Disassemblers such as ROSE and IDA Pro use various heuristics to detect the boundaries of procedures in binary code. For example, successfully identifying the function-prologue sequence[30], `push ebp mov esp, ebp`, in a disassembly is a good indicator that a new procedure begins at that point. Another commonly seen sequence is the 'register save and restore' pattern.

Many functions will preserve registers prior to overwriting their contents at the beginning of the routine. Binary calling convention dictates that the procedure must save the contents of the registers, as subsequent code may make use of these values. For example, a function that uses/overwrites the values in `ebx` and `esi` will often contain the instruction sequence `push ebx push esi` in its first basic-block. The inverse of this sequence will be found just prior to the exit of the function to restore the original state, `pop esi pop ebx ret`.

Our first detector operates after ROSE's standard detectors have completed. It examines instructions that were not assigned to any function, whose addresses lie between the boundaries of two other functions (i.e., If function 1 ends at address X and function 2 begins at X+Y, the detector will examine instructions between X and X+Y.) In most well-formed binaries, consecutive

procedures follow directly after one another or are separted by padding-bytes such as `cc` and `90`. It is suspicious for there to be valid instructions between the two.

The detector iterates through the instructions in address order, and marks the first `push REGX` instruction it encounters. It then continues iterating, recording the last occurrence of a `pop REGY` instruction, until it hits a function return, `ret`. If REGX and REGY are the same register, the address of the `push` is fed to ROSE as the potential start of a new function. ROSE will attempt to create a procedure at that point (i.e., discover other instructions and basic blocks).

3.2.2 Detector 2: Finding Functions Mislabeled as Data

Once ROSE has completed it's initial analysis, it marks any disassembled instructions that were not assigned to a function as being data. It also assigns a code likelihood score to those bytes. Disassemblers can erroneously produce instructions from bytes that are actually data. ROSE uses statistical analysis to produce a probability score that measures the chance that bytes are actually valid instructions [47] (see Partitioner::CodeCriteria).

Our second detector iterates through groups of unassigned instructions that can be collected into basic blocks and also labeled as data. It considers only those that have a high likelihood score (0.7+). Using ROSE's symbolic execution engine, the detector determines if control-flow from any of these basic blocks can reach another. In more detail, we emulate the instructions and determine the value of the instruction pointer `eip` at the end of the block. If the value in `eip` could match the address of another block, control-flow from the first may reach the second. Once we have determined interconnected blocks, we mark the starting address of the first block from each group as being a potential function start. We feed those addresses back to ROSE for further analysis.

3.3 Dependency-analysis and PDG Generation

3.3.1 Definitions

Once an executable has been disassembled, each of its functions are processed by PdgGrep's data and control-dependency modules. Before going into their details, we provide definitions that will be used in their discussion. For additional information, we direct the reader to work by Kiss, Jász, and Gyimóthy [35].

A computer system can be defined as $C = \langle P, M, R \rangle$, where P is a program; M and R are memory locations and registers that are available for use by P. Each program is composed of a set of functions, F, which can be further divided into sequences of instructions (i.e., $\forall f \in F, f = \langle i_0, i_1, i_2, ... \rangle$). Let I be the set of instructions, and V be the set of values they manipulate.

Instructions read from and write to parts of M and R. Let $\mathrm{Use} : I \mapsto 2^{(V \times (M \cup R))}$ be a mapping such that $\mathrm{Use}(i)$ is the set of all pairs $\langle v, a \rangle$, where v is the value read by i and a is either a memory address in M or a register in R that stores v:

$$\mathrm{Use}(i) = \left\{ \langle v, a \rangle \mid a \in M \cup R, i \xleftarrow{v} a \right\}$$

Simply stated, $\mathrm{Use}(i)$ is a data structure that maps instructions to values read in particular registers and memory locations. Similarly, let $\mathrm{Def} : I \mapsto 2^{(V \times (M \cup R))}$ be a mapping between an instruction and the locations it writes to:

$$\mathrm{Def}(i) = \left\{ \langle v, a \rangle \mid a \in M \cup R, i \xrightarrow{v} a \right\}$$

An instruction is said to be *data-dependent on* another instruction, if it reads a value that has been set by the other. We define the function $\mathrm{DataDepOn}$ to be a mapping between an instruction i and the set of triplets $\langle v, a, j \rangle$, where v is the value written to the register or memory

location a by j.

$$\mathrm{DataDepOn}(i) = \{\langle v, a, j\rangle \mid \langle v, a\rangle \in \mathrm{Use}(i) \cap \mathrm{Def}(j)\}$$

An instruction is said to be *control-dependent on* another instruction, if the result of executing the latter instruction determines whether the former will be run or not. We define the function $\mathrm{ControlDepOn}$ to be a mapping between an instruction i and the set of instructions $\langle c\rangle$. c is an instruction that conditionally write one of two possible values to the program counter, one of which is the address of i.

$$\mathrm{ControlDepOn}(i) = \{\langle c\rangle \mid \mathrm{BranchCondition}(c)?eip \leftarrow \mathrm{AddressOf}(i) : eip \leftarrow addr2\}$$

A basic block A is said to post-dominate block B, if the execution of B implies that the program must also execute A to reach its conclusion.

3.3.2 Data-dependency analysis

To construct $\mathrm{DataDepOn}$, we extended the work-list algorithm [17, 31, 35] for data-flow analysis to work with x86 instructions, and implemented it on top of ROSE's symbolic execution engine [47]. The pseudocode is shown in Procedure 3.3.1. It maintains a list of symbolic expressions (called $states$) that capture the contents of registers and memory after each basic block is executed. For each basic block B, and for each instruction i of B in flow order, the algorithm: (i) symbolically executes i; (ii) updates the register and memory contents of B's state with the result r; and (iii) adds i to the list of "modifiers" of r. This list records the addresses of all instructions that have contributed to the value up to this point. For example, processing the instruction `add [eax], 5`, located at address 0x00405630, updates the memory contents at the address pointed to by *EAX* with 5, and adds 0x00405630 to the value's list of modifiers. Therefore, when a different instruction reading this same memory location is processed later (i.e., `cmp [EAX],`

`0`), a dependency relationship with the `add` is established by reading the list of modifiers.

The state of each basic block, before any instructions are executed, is composed of the 'merged' states of each of the block's predecessors. In more detail, if control-flow can reach a basic block from multiple locations, the contents of registers and memory at block entry may have different symbolic values and modifiers, depending on the specific path taken. Thus, the merged state combines the information from each possible entry path by performing a union across all possible entry states. Explicitly, if the contents of a register or memory location is the same in two different entry states, the symbolic value for that location in the merged state is the same. If they are different, the merged state reflects that the value is unknown, and the resulting list of modifiers is the combination of the lists from each entry state.

The state of each basic block, after all instructions are executed, is compared with the its previous state in $states$. If any of the registers or memory contents have changed, $states$ is updated and all the block's successors (those that the block can flow into) are marked for processing. The algorithm terminates when the states of all blocks stop changing.

3.3.3 Control-dependency analysis

To construct $\mathrm{ControlDepOn}$, we extended the algorithm presented by Ferrante et. al [19] to operate on x86 instructions. We first compute the control-dependencies for each basic block, $\mathrm{BlockDeps}$. The instructions within a block A that is control-dependent on another block B are marked as being control-dependent on the conditional jump found in B to populate $\mathrm{ControlDepOn}$.

Procedure 3.3.2 illustrates how $\mathrm{BlockDeps}$ is built. The procedure takes a single parameter, the control-flow graph (CFG) of a function produced by ROSE. Each vertex of the graph represents a basic block, and each edge represents a possible path of execution. The code visits each $block$ and determines if any of its successors (basic blocks that would be executed immediately following this one) post-dominate it. If the successor does not post-dominate $block$ (i.e., the successor's execution depends on the result of executing $block$), then the successor and all

blocks that it post-dominates up to the least-common post-dominator shared with $block$ are also control-dependent on $block$.

Procedure `buildDependencies()`

Input: Func: A binary function composed of assembly instructions

Input: EntryState: Symbolic state of system, storing register and memory contents, upon function entry

Result: Uses, Defs, and DataDepsOn are populated for each instruction

```
1  foreach block ∈ getBasicBlocks(Func) do
2      states[block] ← initSymbolicState();
3      queue[block] ← true;
4  changed ← true;
5  while changed do
6      foreach block ∈ getBasicBlocks(Func) do
7          if queue[block] then
8              if isFirstBlock(block) then
9                  curstate ← EntryState;
10             else
11                 foreach pred ∈ getPredecessorBlocks(block) do
12                     curstate ← mergeStates(curstate, states[pred]);
13             foreach instr ∈ getInstructions(block) do
14                 curstate ← symbolicExec(instr, curstate);
15                 foreach aloc ∈ getRegsAndMemRead(instr) do
16                     symval ← getRegOrMemValue(aloc, curstate);
17                     Uses[instr] ← ⟨symval, aloc⟩;
18                     foreach definer ∈ getModifierList(symval) do
19                         DataDepsOn[instr] ← ⟨symval, aloc, definer⟩;
20                 foreach aloc ∈ getRegsAndMemWritten(instr) do
21                     symval ← getRegOrMemValue(aloc, curstate);
22                     Defs[instr] ← ⟨symval, aloc⟩;
23             if not regsAndMemEqual(curstate, states[block]) then
24                 changed ← true;
25                 foreach successor ∈ getSuccessorBlocks(block) do
26                     queue[successor] ← true;
27                 states[block] ← curstate;
28             queue[block] ← false;
```

Procedure `buildBlockControlDeps()`

Input: CFG: A control-flow graph of a function, where vertexes are basic blocks and edges represent flows between them.

Result: BlockDeps is populated for each basic block

```
1 foreach block ∈ CFG do
2     foreach successor ∈ getBlockSuccessors(block) do
3         if not isPostDominatedBy(block,successor) then
4             d1 ← getBlocksPostDominatedBy(block);
5             d2 ← getBlocksPostDominatedBy(successor);
6             ControlDepOn[successor].append(block);
7             foreach b ∈ d2 do
8                 if b ∉ d1 then
9                     ControlDepOn[b].append(block);
```

3.3.4 Program Dependence Graph

Once the data and control dependencies for each instruction have been determined, the information is combined in a Program Dependence Graph (PDG). First introduced in [19], the PDG is a directed graph that features instructions as vertexes. There are two types of edges: A data-dependence edge between one vertex and another implies that the destination vertex uses some piece of data defined by the source. A control-dependence edge between one vertex and another implies that the destination vertex is control dependent on the source.

Fig. 3.3 depicts an example PDG for the simple decryption loop in Fig. 3.2. Data-dependence edges are marked with the letter 'D'. Control-dependence edges are marked with 'C'.

```
004010A3   pop %ebx                                  | Move the address of the encrypted payload into ebx
004010A4   mov %ecx, 0x00000049e                     | Size of the payload is 0x49e
004010A9   xor BYTE PTR ds:[%ecx + %ebx], 0x20       | Perform XOR decryption with constant key 0x20
004010AD   loop 0x004010a9                           | Repeat 0x49e times
```

Figure 3.2: Simple decryption loop

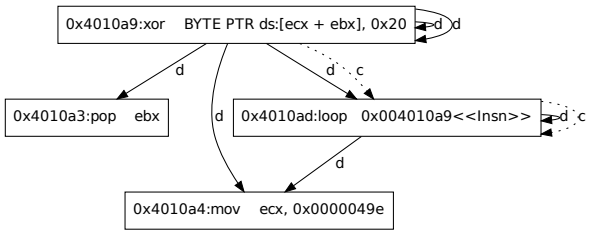


Figure 3.3: An example PDG

3.4 Dependence Path Extraction

Once the PDG for each function has been generated, we generate dependence paths beginning at each of its vertexes.

Definition 1. *A dependence path is a sequence of at most L unique instructions, in which each adjacent instruction shares either a data or a control-dependency.*

For example, suppose L is five. The dependence paths for `xor` instruction from Figure 3.2. are shown in Figure 3.4. `-D->` denotes a data-dependence relationship (i.e. `A-D->B` denotes A uses data defined by B). `-C->` denotes a control-dependence relationship.

Note that all paths have fewer than five instructions, and instructions are never repeated in the same sequence. Also, note that although the third and fourth paths contain the same instructions, they capture different dependencies. The third path captures the control-dependence relationship between the `xor` instruction and the `loop`. It also captures the data-dependence relationship between the `mov` and the `loop`, as the `loop` instruction reads and writes `ecx`. The fourth path captures the fact that the `xor` instruction reads `ecx`, which is also decremented by the `loop` instruction.

Procedure 3.4.1 is invoked on each vertex of the PDG. The algorithm branches out through the instruction's data and control-dependence edges to build a list of of dependence paths, $Paths$.

Procedure `extractDependencePaths()`

Input: insnVertex: A vertex in the PDG

Input: curpath: A list of instructions in the current dependence path

Input: maxlen: l, maximum path length

Output: Paths: List of all paths generated for this instruction

Result: Paths is populated for insnVertex

```
1 if insnVertex ∉ curpath and length(curpath) < maxlen then
2     curpath.append(insnVertex);
3     foreach dinsn ∈ getDataEdges(insnVertex) do
4         extractDependencePaths(dinsn, curpath, maxlen, Paths)
5     foreach cinsn ∈ getControlEdges(insnVertex) do
6         extractDependencePaths(cinsn, curpath, maxlen, Paths)
7 if length(curpath) > 1 then
8     Paths.append(curpath);
9     return
```

```
xor BYTE PTR ds:[%ecx + %ebx], 0x20 -D-> mov %ecx, 0x00000049e
xor BYTE PTR ds:[%ecx + %ebx], 0x20 -D-> pop %ebx
xor BYTE PTR ds:[%ecx + %ebx], 0x20 -C-> loop 0x004010a9 -D-> mov %ecx, 0x00000049e
xor BYTE PTR ds:[%ecx + %ebx], 0x20 -D-> loop 0x004010a9 -D-> mov %ecx, 0x00000049e
```

Figure 3.4: Dependence paths for Xor

3.5 Index Key Generation

Once all unique, dependence paths have been generated for each instruction in a function, they are transformed into search keys by a `KeyHash`. The design of this search key is essential to the performance of the system. When analysts issue queries (described in Section 3.6), the front-end of PdgGrep will attempt to lookup keys that match those generated from the query. Therefore, keys must have the following properties:

1. Allow for fast lookup.
2. Contain sufficient detail to satisfy an analyst's constraints on instruction operands (Section 3.6).

3. Be storage efficient.

We experimented with the two implementations of `KeyHash` described below.

3.5.1 KeyHash 0.1

Our first implementation of `KeyHash` was motivated by Giugno et. al [22], and prioritized search speed. The basic idea is to generate a hash representation of a dependence-path string. This hash is used as a key in the index, and points to a list of functions for which that path is present. When the user issues a query, PdgGrep's front-end generates a parallel set of hashes from the instructions in the query to search the index.

`KeyHash 0.1` simply takes the `md5` hash of a dependence-path string, similar to those seen in Section 3.4. However, two issues complicate matters. First, a user may be interested in searching for a particular sequence of instructions in which certain operands matter and others do not. For example, she may not care that a particular instruction uses register `eax` or that a particular constant is $0x123$. Second, a user may only provide a subset of the instructions in a dependence path. For example, consider the dependence paths that include `xor` and `loop` from Figure 3.4. Those paths also include the `mov` instruction. However, a user could issue a query that only includes the `xor` and `loop` instructions.

To address the first problem, `KeyHash` generates the hashes of variants of a dependence-path string, substituting abstract placeholders for different concrete operands. In more detail, `KeyHash` produces a set of *abstract instructions* for each instruction in a dependence path. The abstract form of an instruction is simply the string representation of the instruction with one or more of its operands written as a description of the operand type. For example, the concrete instruction `add [eax+10], 5` would produce the abstract instructions `add [REG+10], 5`, `add [REG+CONSTANT0], 5` and `add [REG+CONSTANT0], CONSTANT1`, `add[REG+10], CONSTANT1` and finally the instruction itself `add [eax+10], 5`. 'REG' is used to denote some general purpose register. 'CONSTANT' is used to denote some integer value. Therefore,

when a user issues a query where she does not care about the exact register or constant used, the front-end produces a dependence path with that particular operand replaced with the appropriate high-level descriptor, hashes it, and queries the index.

To address the second problem, `KeyHash` also generates the hashes of all subpaths shorter than the dependence path. Therefore, for a dependence path containing x instructions, `KeyHash` also generates hashes of the abstracted forms for paths of length $x-1$, $x-2$, ... m. m is the minimum subpath length.

For example, the set of strings produced by `xor BYTE PTR ds:[%ecx + %ebx], 0x20 -D-> loop 0x004010a9 -D-> mov %ecx, 0x00000049e` with $m = 1$ are:

```
xor [%ecx + %ebx], 0x20 -D-> loop 0x004010a9 -D-> mov %ecx, CONSTANT0
xor [%ecx + %ebx], 0x20 -D-> loop 0x004010a9 -D-> mov REG0, 0x00000049e
xor [%ecx + %ebx], 0x20 -D-> loop 0x004010a9 -D-> mov REG0, CONSTANT0
xor [%ecx + %ebx], 0x20 -D-> loop CONSTANT0 -D-> mov %ecx, 0x00000049e
xor [%ecx + %ebx], 0x20 -D-> loop CONSTANT0 -D-> mov %ecx, CONSTANT1
...
xor [%ecx + %ebx], 0x20 -D-> loop 0x004010a9
xor [%ecx + %ebx], 0x20 -D-> loop CONSTANT0
...
xor [%ecx + %ebx], 0x20
...
```

Figure 3.5: Dependence paths for Xor

The `md5` of each string produces a key that points to a list of function identifiers. Each of these IDs is associated with a function that contains the path used to build the key. For example,

`md5('xor [%ecx + %ebx], 0x20 -D-> loop CONSTANT0 -D-> mov %ecx, CONSTANT1')` $\longrightarrow \{\langle FuncIDX, FuncIDY, ...\rangle\}$

As will be discussed later, although this approach generates a large number of keys and has a significant storage overhead, hashing dependence paths allows for very fast lookups.

3.5.2 KeyHash 0.2

One of the major limitations with `KeyHash 0.1` is that a hash destroys the semantic information contained within the original dependence-path string. Therefore, it is necessary to also

hash all possible operand permutations and subpaths as well. Our second design creates a key suitable for a B+ tree, and reduces the total number of keys generated. The basic idea is to group dependence paths that are different only because of registers or constants.

The key is a variable-length byte array composed of a sequence of high-level descriptors for each instruction in a dependence path. Each descriptor encodes the instruction's mnemonic and type identifiers for each of its operands (i.e. register = 0, immediate = 1, memory dereference with base and index register = 2, etc.). The order of the descriptors corresponds with the order of the instructions in the dependence path.

```
[Insn1Mnem][OpndTypeDst][OpndTypeSrc][Insn2Mnem][OpndTypeDst][OpndTypeSrc]...
```

Figure 3.6: Key is composed of abstract instruction descriptors

Each key points to a list containing entries that encode: 1) a function identifier, 2) the size of the entry, and 3) concrete descriptors that specify the exact registers or immediates/constants for each of the operands in the key. Therefore, two dependence paths that are different because of immediate values or registers will share the same key. However, the different immediates or registers will be reflected in their corresponding list entries.

```
[FunctionID][SizeOfEntry][RegOrImm1][RegOrImm2]...
```

Figure 3.7: An entry in the list pointed to by a KeyHash 0.2 key

For example, the key generated for the dependence path `xor BYTE PTR ds:[%ecx + %ebx], 0x20 -D-> loop 0x004010a9 -D-> mov %ecx, 0x00000049e` and `xor BYTE PTR ds:[%ecx + %edx], 0x54 -D-> loop 0x00403bb8 -D-> mov %ecx, 0x000000123`, from function 1 and 2, is shown in 3.8:

```
[XorOpCode][MemBaseIndex][Imm][LoopOpCode][Imm][MovOpCode][Reg][Imm]
```

Figure 3.8: Key generated for Xor dependence path

The key above points to the list shown in 3.9.

```
[1][10][ECX][EBX][0x20][0x4010a9][ECX][0x49e]
[2][10][ECX][EDX][0x54][0x403bb8][ECX][0x123]
```

Figure 3.9: List of function IDs and operand details

The index is a B+ tree, implemented on top of Google's LevelDB [23]. The index orders keys using a byte-comparator from left to right (see Figure 3.10). This setup allows users to find keys with matching abstract instruction descriptors quickly: The front-end produces a set of descriptors from the instructions in a query, and seeks to the first key that matches. It then iterates through each of the entries pointed to by the key, decodes the concrete operands, and checks to see if they meet the users' requirements (discussed in Section 3.6). The process repeats itself with each key that has a prefix that matches the abstract descriptors in the query.

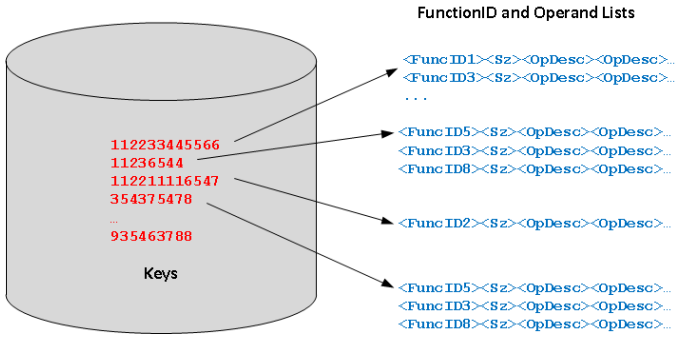


Figure 3.10: B+ tree index using KeyHash 0.2

3.6 Query Generation

Users supply queries to PdgGrep's front-end that closely resemble instructions. Except, special symbols are permitted to describe operand constraints. For example, Figure 3.11 depicts a possible query for the Xor loop shown above.

```
            pop REG0
            mov REG1, 0x00000049e
DECRYPT:    xor BYTE PTR ds:[REG0 + REG1], <?>
            loop DECRYPT
```

Figure 3.11: User query for Xor loop

The special symbol `REGx` implies that the user does not care which particular register is used, only that it is used consistently across the instructions. The `<?>` symbol means that the user does not care what particular immediate value is used. Other, symbols include predicates for greater than, less than, etc. In the first `KeyHash` implementation, the only immediate symbol permitted was `<?>`.

The front-end generates a PDG and a list of keys for each instruction in the query, analogous to the way files are inserted into the index. In `KeyHash 0.1`, the front-end simply performs a hash lookup for each of these keys. It then returns the intersection of the function ID lists. (i.e., It finds functions in which all of the query paths are present.)

The query algorithm for `KeyHash 0.2` is shown below:

1. From the user's query, produce a PDG.
2. Using the PDG, extract all dependence paths for each instruction.
3. Select the minimal set of paths such that each instruction is included in at least one of the paths, prioritizing the longest paths.
4. For each path in this subset, produce the sequence of high-level descriptors that correspond with the instructions in the path.
5. For each generated sequence, seek to the first key with a matching prefix. Fetch the list pointed to by the key. For each list entry, decode the operands and check to see if they match the constraints specified in the query. If a match is found, record the function ID. Repeat for all keys that have a matching prefix.
6. Perform a set intersection for each group of function IDs returned by each of the sequences.

Chapter 4

Scalability Analysis

In this chapter, we discuss the variables that influence the disk overhead and query performance of PdgGrep. Sec. 4.1 provides an analysis of the upper-bound on the disk used by both `KeyHash` implementations. The following subsections, Sec. 4.1.1 and Sec. 4.1.2, provide experimental data in support of this analysis. In a similar fashion, Sec. 4.2 gives an upper-bound on the search times for both implementations. The subsequent subsection provides data in support of the `KeyHash 0.2` calculation. As will be discussed in Sec. 4.1.2, we stopped experimenting with `KeyHash 0.1` after it became clear that this approach would require too much disk.

4.1 Disk Overhead

The disk overhead of both `KeyHash` implementations is directly proportional to the total number of keys, the size of the keys, and the sizes of the lists that the keys point to.

In the case of `KeyHash 0.1`, the size of a key is constant (i.e., the size of an `md5` hash). The number of keys is dependent on several variables: 1) the number of unique dependence paths per instruction D, 2) the maximum and minimum dependence path lengths, l and m respectively, 3) the average number of abstract instruction permutations r, 4) the total number of instructions, I, across all functions, and 5) the total number of unique functions F.

In order to give users the ability to ignore the exact registers or immediates used by certain instructions, `KeyHash 0.1` hashes every permutation of a dependence path with different combinations of actual operands and abstract descriptors (see Sec. 3.5.1). Also, to make partial paths searchable, `KeyHash 0.1` hashes every subpath of a dependence path. Thus, an upper-bound on the number of keys for `KeyHash 0.1` is $O(ID\sum_{i=m}^{l} r^i)$. (The ID term approximates the total number of unique dependence paths. Each path has r^i permutations, where i is the number of instructions in the path. Keys are generated for path lengths from m to l.) Assuming a worst-case scenario that every key points to a list containing every function in the index, an upper-bound on the disk requirement is $O(FID\sum_{i=m}^{l} r^i)$.

In the case of `KeyHash 0.2`, the size of the key is proportional to the length of the dependence path, and bounded by the maximum possible length l. It's associated list has a size that is dependent on the number of functions for which the key's dependence path is found and the number of instruction operands. Assuming a worst-case scenario that every list contains every function and that all lists are of maximum length, the disk requirement is bounded by $O(FIDl^2)$.

4.1.1 `KeyHash 0.2` Experiments

Using a dataset of 310 malware files, downloaded from websites blacklisted by *http://lists.clean-mx.com*, we constructed `KeyHash 0.2` indexes for varying values of l. We selected these files because they are current, state-of-the art malware being distributed in the wild (according to the heuristics used by *clean-mx*). Although we could have used a larger or smaller collection, the objective of these experiments was to demonstrate how metrics like disk space, query-time and index construction time would change as a function of l. Therefore, although the raw measurements would be scaled up or down respectively, the trends observed between the variables would remain the same regardless of the size of the dataset.

Indexes were built on machines running Red Hat 5 Linux 2.6.32 with 48Gb RAM, with a 15-minute timeout set on each file to simulate a production environment. Figure 4.1 shows the

polynomial growth in disk requirement as l increases that was predicted above.

Figure 4.2 shows the number of files that were completely processed before the timeout occurred. A larger value of l results in a greater fan-out from each node in the PDG (see Section 3.4) and more keys (see Figure 4.3). Therefore, the time required to fully construct an index increases with l (i.e., A greater amount of time is spent constructing keys and committing them to disk. Therefore, the number of files that are processed in a fixed period of time decreases with l). The exact time is also dependent on the particular dataset, and the average number of dependencies per instruction (i.e., the number of outgoing edges from each node of the PDG, which we measured to be 2 for the 310 files).

Figure 4.3 shows the total number of keys created for each of the five indexes. Noting the similarities between this Figure and Figure 4.1, we see a direct correlation between the number of keys generated and the disk-overhead of the index as expected.

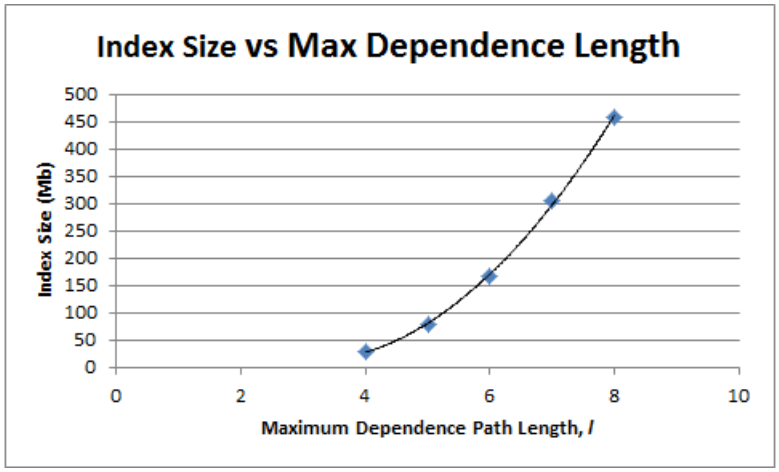


Figure 4.1: KeyHash 0.2 Index Size vs. Max Dependence Length

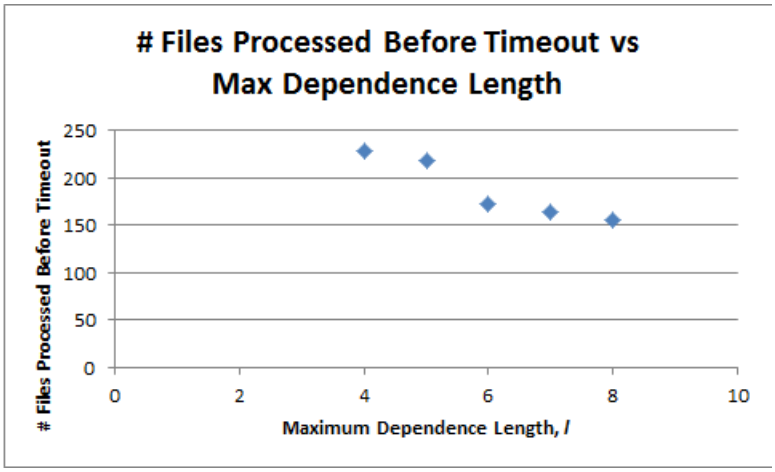


Figure 4.2: # Files processed using 15-min. timeout/file vs. max. dependence length

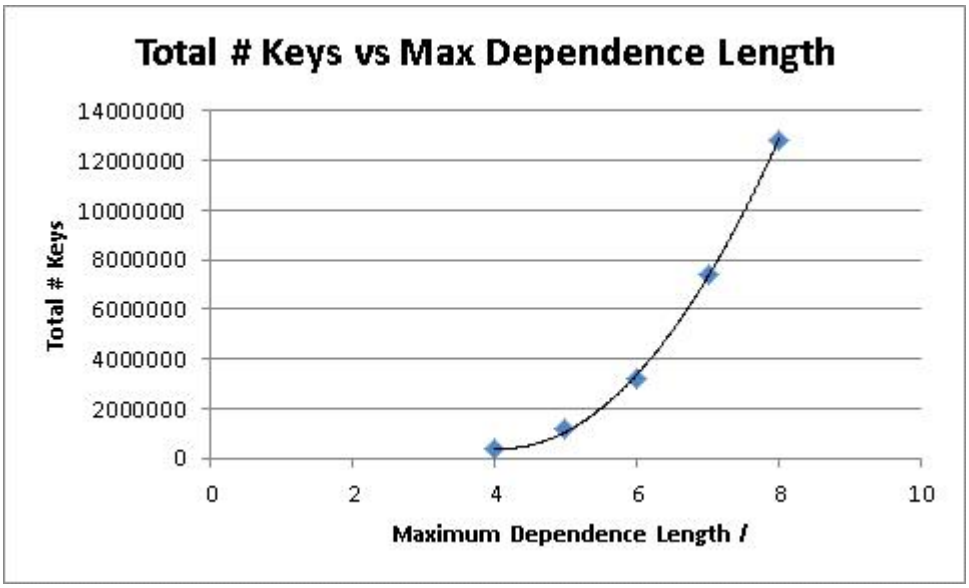


Figure 4.3: Total # of keys vs. max. dependence length

4.1.2 `KeyHash 0.1` Experiments

We only constructed two indexes using `KeyHash 0.1`, because it quickly became apparent that the approach would use too much disk. `KeyHash 0.1` was meant to be a proof-of-concept, designed to demonstrate that our approach could be used to detect instructions in obfuscated

malware datasets. Scalability was not the focus.

The first index contained 166 binaries from the CERT collection: 69 non-malware files, 93 randomly selected malware files, and a member of four known polymorphic malware families (Virut, Alman, Allaple and Sality)[42]. The second contained 390 randomly selected binaries from the CERT collection. The purpose of choosing arbitrary files was to see how the index and queries would fare against a mix of packed, non-packed, malicious and benign programs.

Using members of Virut, Alman, Allaple, and Sality not present in either of the indexes, we constructed queries from instructions that we suspected to be characteristic of those families, and attempted to identify other variants in both datasets. We used an l of 5 and an m of 2. The results of the queries are summarized in Table 4.1

Dataset	**Sality**	**Virut**	**Allaple**	**Alman**
1	Sample found. No False positives.	Sample found. No False positives.	Sample found. No False positives.	Sample found. No False positives.
2	6 samples identified. 3 samples identified from other families, but containing the query pattern.	4 samples found.	11 samples found.	0 samples found.

Table 4.1: KeyHash 0.1 query results

Table 4.2 shows the number of keys and sizes of both indexes. Note that both of these quantities are extremely large for datasets that containing so few files. The large number of keys is to be expected, because all permutations of abstract instructions from lengths of 2 to 5 are being hashed to produce keys (an expontential relationship). Since the disk overhead is directly related to the number of keys, it is clear that the size of the index is also exponentially related to l and m. It is interesting to note that the smaller dataset actually occupied a greater amount of disk than the larger one. We found that the number of dependence relationships in the first

dataset was greater than in the second, leading to a greater number of keys being generated, and more disk usage.

Dataset	# Unique Functions	# Keys	Size (Gb)
1	20909	221167075	2.3
2	33169	201172999	2.2

Table 4.2: KeyHash 0.1 index statistics

4.1.3 Summary

In this section, we saw that the number of keys plays a crucial role in determining the size of indexes generated by both `KeyHash` implementations. An important distinction between `KeyHash 0.1` and `KeyHash 0.2` is that the number of *keys* is expontentially-related to the dependence path length in the former and proportionately in the latter.

As discussed in Sec. 3.5.1, `KeyHash 0.1` produces a key by taking the MD5 hash of a dependence path. Therefore, the semantic information (i.e., instruction and operand type) contained with the path is lost. To allow users to search for paths containing the same instructions but different register and immediate values, `KeyHash 0.1` hashes all permutations of the dependence path (i.e. all possible combination of concrete and abstract operands)[1], and results in the exponential-relationship between keys and path length.

`KeyHash 0.2` does not perform an exact hash of a dependence path when generating a key. Therefore, it does not destroy the semantic information (i.e., instruction and operand type) contained within the path. As a result, unlike with `KeyHash 0.1`, it is not necessary to generate keys for all permutations of abstract instructions in dependence paths and their corresponding subpaths.[2] Although the size of an individual `KeyHash 0.2` key maybe large than an MD5

[1]Recall that search is performed in a similar way as indexing. The query front-end takes a dependence path that the user wishes to search for, including which operands are relevant/concrete and which ones are not/abstract, and produces an MD5 hash. In order for a hash collision to occur, the same hash must be in the index.

[2]Recall that search involves finding a key with a sequence of matching abstract instruction descriptors, and verifying their concrete operand values.

hash (the maximum size is proportional to l), the relationship between keys and path length is directly proportional. Disk usage is polynomially-related to maximum path length, since each entry in the list pointed to by a key contains a sequence of concrete operands that is once again bounded by l.

4.2 Query Performance

The greatest advantage to using `KeyHash 0.1` is search speed. Hash lookup allows queries to have a $O(log(N))$ latency, where N is the number of unique keys in the index.

In `KeyHash 0.2`, search speed is dependent on how unique a particular dependence path is. Recall the way that queries are matched. The front-end produces a sequence of abstract instruction descriptors and seeks to the first matching key. On average, this operation is $O(log(N))$. Next, the front-end iterates over each entry in the list pointed to by the key, unpacks its operands and checks to see if they match the user's constraints. This operation is linear over the number of entries, P, and the number of operands in the query's instructions, Q. The process is repeated for each key with a matching prefix, W. Therefore, the operation is $\approx O(WPQ)$. The overall, average run-time for a `KeyHash 0.2` query is $\approx O(log(N)) + O(WPQ))$.

4.2.1 `KeyHash 0.2` Experiments

Figure 4.4 shows the latency times for randomly-selected queries of lengths 4-8 instructions, issued over indexes with l from 4-8. Indexes were produced from the same 310 files described in Sec. 4.1.1.

In general, query times were higher for greater values of l due to a larger value of N (recall Figure 4.3). For short queries of lengths 4, the number of keys with matching prefixes W is large and the search time is high. W is worst for increasing values of l, because a greater number of keys with the same prefix as the query exist. As the query length increases to 5, we see a drop

in the query time, because fewer keys have a matching prefix (i.e., the query has become more specific).

The shortest time occurred with the index of length 5 and query of length 5. In this case, for the index of length 4, the front-end was forced to split the query of length 5 into 2 sub-queries of lengths 4, and take the intersection of the results. The time required to perform these two searches exceeded the time required to conduct a single query for the index of length 5 (even though there are more keys in the index of length 5).

As the query length increases, the number of operands that must be unpacked also increases. As a result, search times get progressively worse. As mentioned above, as the query length exceeds the maximum dependence length of each index, the front-end is forced to split queries into multiple parts. Multiple queries also increase overall search time.

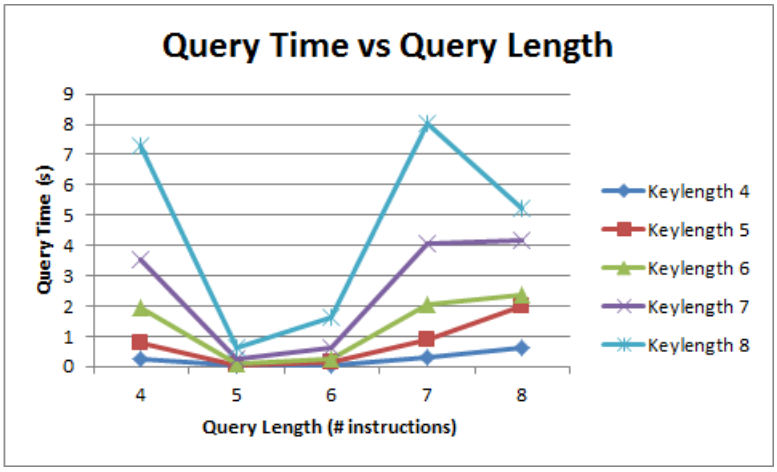


Figure 4.4: Search times for queries with lengths 4-8

It is important to note that the times recorded in Figure 4.4 are specific to the particular set of queries that were used in that experiment. Different queries might lookup keys with different values of P. Figure 4.5 shows P plotted for each of the indexes described above. As l increases, the number of entries associated with each key decreases (i.e., there is a better distribution of entries amongst the keys). However, as observed in 4.4, this benefit is often overcome by the large value of N that results from a larger value of l.

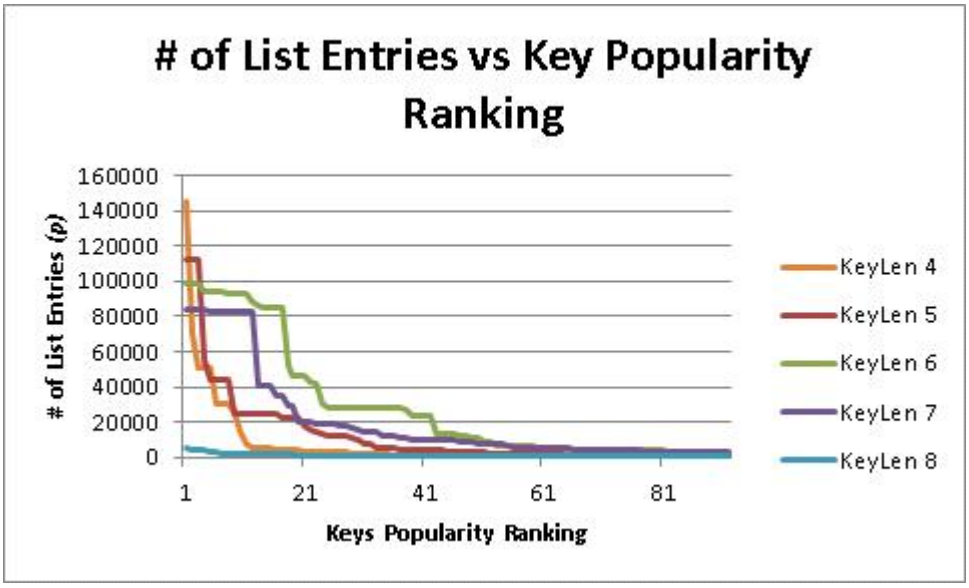


Figure 4.5: # of entries per key in descending order

We found the most popular key to be an alternating series of pushes and pops (i.e., saving and restoring a register, seen in Figure 4.6):

```
push REG-D->pop REG-D->push REG-D->pop REG...
```

Figure 4.6: Most popular key

The least popular key was:

```
aaa-C->jb IMM-D->rol [BASE+DISP], REG-D->loop IMM-D->loop IMM
```

Figure 4.7: Least popular key

The `aaa` instruction, ASCII Adjust After Addition instruction, is very rarely used. In this particular context, it's presence may be the result of incorrect disassembly or obfuscation.

4.2.2 Summary

In this section, we see the tradeoff between query performance and index size between the two `KeyHash` implementations. Although the use of an exact hash by `KeyHash 0.1` leads to many more keys being generated and a significantly larger index, queries have short latencies (i.e., seconds). The use of abstract instruction descriptors by `KeyHash 0.2` reduces the number of keys and results in smaller indexes. However, it also leads to slower queries.

The query performance of `KeyHash 0.2` is influenced by a number of dependent variables, most notably l and the popularity of various keys P. In general, a large value of l results in a very large number of keys that decreases query performance. However, very short values of l causes the popularity of individual keys, p, to be very high, and forces queries to be split into multiple sub-queries that also degrades perforance. Ideal values of l will change from dataset to dataset, because of changing key popularities. However, for a production environment where new files are constantly being inserted into the index, a value of l between 4-7 appears to be a reasonable choice.

The following is a summary of the major tradeoffs present in the system and how they are influenced by different variables. The summary focuses on the final `KeyHash 0.2` design:

- The choice of `KeyHash` is crucial to the performance of the system, and often represents a tradeoff between query speed and index size.
- A greater number of functions/files, N, results in larger index and longer search times.
- Longer dependence lengths, l, result in more keys being generated. A greater number of keys results in a larger index, and slower search times in a B+ tree.
- Very short dependence lengths, l, cause user queries to be split into multiple search queries that can also degrade search times.
- Key popularity, P, is directly controlled by l. A short dependence length means that individual keys map to more entries.

- Key popularity is also heavily dependent on the particular dataset (i.e., certain paths are more popular than others in one dataset to the next).
- Key popularity impacts query time, because each entry must be checked to see if it matches the search criteria.
- W, the number of keys with matching prefixes, depends on the length of the query. It is also controlled by the dependence length l: Shorter lengths means fewer keys and a greater W.
- Larger W results in a greater search time, since each key and its corresponding entries must be checked.
- Longer queries with more instructions and operands, Q, result in slower queries, since each operand requirement needs to be verified.

Chapter 5

Case Studies

In this chapter, we present a series of case studies to demonstrate the utility of PdgGrep for identifying obfuscated, instruction patterns. Sec. 5.1 discusses a typical scenario where analysts wanted to identify a function obfuscated using techniques discussed in Sec. 2. Sec. 5.2 discusses a case, where we were able to identify function variants that contained implementation changes as well as simple obfuscations. Sec. 5.3 discusses a malware family that poses significant challenges to automated analysis and disassembly.

5.1 Case study: Destory RAT string decryption algorithm

For malware analysts, a common problem that arises when studying a family of malware is the identification of particular functions and their variants. In this section, we present a case study in which we used PdgGrep to look for variants of a string decryption function used by the Destory RAT. This function is used to restore encrypted .dll and library names, so that the corresponding libraries and functions can be loaded by the malware dynamically. A more detailed discussion of this import resolution technique can be found here [43]. Obfuscations used by the malware make identifying functions such as this one problematic for existing techniques. Additional information about attacks conducted using this RAT can be found here [24, 25].

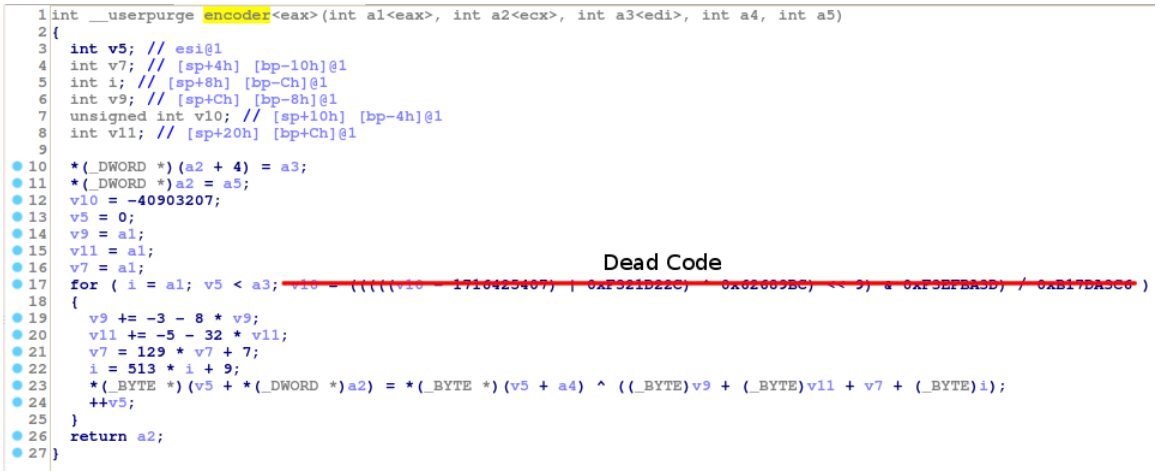
5.1.1 Background: Decrypting .dll and library names

Destory employs a number of anti-detection and anti-analysis techniques: 1) dynamic API resolution, 2) encryption algorithms for ciphering strings, network traffic and configuration files, 3) dead code designed to mimic legitimate code, and 4) numerous control-flow obfuscation techniques. Dynamic API resolution is a technique for loading .dlls and finding exported functions at run-time. Run-time resolution allows malware authors to avoid linking their code with a list of explicit imports, making it more difficult for analysts and AVs to determine what function is being called and why. It is typically implemented using two API calls: 1) `LoadLibrary()`, which loads a .dll at runtime, and 2) `GetProcAddress`, which is used to resolve the address of a function within the .dll. Both methods are passed string parameters corresponding to a .dll name and a function name respectively. Figure 5.1 illustrates the use of `LoadLibrary` to load `ws2_32.dll` and `GetProcAddress` to find the address of `WSASend`.

```
dllHnd = LoadLibrary('ws2_32.dll');
sendFunc = GetProcAddress(dllHnd,'WSASend');
...
sendFunc(socket, &buf,...);
```

Figure 5.1: Loading ws2_32.dll at runtime and finding WSASend

Therefore, to use this technique, programs must contain strings for all needed .dlls and library functions. A common analysis technique is to search for these strings. In response, some variants of Destory RAT also perform run-time, string decryption (i.e. The malware contains encrypted .dll and function names, and decrypts them only before the API calls.) A disassembly of the decryption function is shown in Figure 5.2.We extracted this function from the sample with the MD5 hash 06f3755ffd3ed38e573c591da489d735 (VirusTotal [53]) .




```
int __userpurge encoder<eax>(int a1<eax>, int a2<ecx>, int a3<edi>, int a4, int a5)
{
  int v5; // esi@1
  int v7; // [sp+4h] [bp-10h]@1
  int i; // [sp+8h] [bp-Ch]@1
  int v9; // [sp+Ch] [bp-8h]@1
  unsigned int v10; // [sp+10h] [bp-4h]@1
  int v11; // [sp+20h] [bp+Ch]@1

  *(_DWORD *)(a2 + 4) = a3;
  *(_DWORD *)a2 = a5;
  v10 = -40903207;
  v5 = 0;
  v9 = a1;
  v11 = a1;
  v7 = a1;
  for ( i = a1; v5 < a3; v10 = (((((v10 - 1716425407) | 0xF521D22C) ^ 0x02669BC) << 5) & 0xF3EFBA5D) / 0xB17DA3C6 )
  {
    v9 += -3 - 8 * v9;
    v11 += -5 - 32 * v11;
    v7 = 129 * v7 + 7;
    i = 513 * i + 9;
    *(_BYTE *)(v5 + *(_DWORD *)a2) = *(_BYTE *)(v5 + a4) ^ ((_BYTE)v9 + (_BYTE)v11 + v7 + (_BYTE)i);
    ++v5;
  }
  return a2;
}
```

Figure 5.2: Disassembly of string-decryption routine in Destory RAT. Dead code has been redacted. (VirusTotal [53]

The routine is a simple linear-congruential generator (LCG). It takes an initial salt and a key to produce a keystream that is xor'ed with ciphered bytes. Note the particular constant values used in the loop: 513, 129, 9, 7, -32, -8, -5 and -3.

5.1.2 Problem Description

When studying families like Destory, analysts are often interested in understanding how the code has evolved over time. For example, it is not unusual for malware authors to make small changes (i.e., encryption algorithm constants, keys, etc.) to thwart detection temporarily[15]. By identifying samples with small differences, analysts can then do a thorough investigation of more significant changes that might exist between two code generations.

To simulate this scenario, we built an index containing 223 suspected Destory samples. The samples were chosen from the CERT catalog, based upon analysts' heuristics. These heuristics alone were insufficient at confirming the presence of the string decryption-function due to obfuscation and other decoy functions. Thus, our objective was to 1) verify the presence of the routine in each sample and 2) record any constants that are different from those listed above.

5.1.3 Script-based Detection

Current code identification techniques rely heavily on the use of specialized scripts such as those written in IDAPython[29] and YARA[58]. For example, the blog entry by ASERT[2] describes how their analysts use IDAPython scripts to identify 'magic values' in the Gh0st RAT family. These scripts are based upon analyzing raw instructions and/or bytes. Therefore, taking the same approach as the one used for Gh0St, analysts might try to write a similar script for Destory's decoder. Figure 5.3 provides an example of what such a script might look like. The script iterates through each of a program's functions. It then tries to identify one containing characteristic instructions known to be present in Destory's decoder (i.e., multiply by 513 and add 9).

```
# Iterate through all of the functions
for f in Functions():
    func = get_func(f)

    # Booleans to record presence of characteristic instructions
    mul513found = False
    add9found = False
    ...

    # Iterate through all instructions
    for head in Heads(func.startEA,func.endEA):
        # Look for muliply with a register as its first parameter and an immediate of 513 as its second
        if GetMnem(head) == "imul" and GetOpType(head,0) == o_reg and GetOpType(head,1) == o_imm and GetOperandValue(head,1) == 513:
            mul513found = True
        # Look for add with a register as its first parameter and an immediate of 9 as its second
        if GetMnem(head) == "add" and GetOpType(head,0) == o_reg and GetOpType(head,1) == o_imm and GetOperandValue(head,1) == 9:
            add9found = True
        ...

    # If all characteristic instructions found => possible decoder found
    if mul513found and add9found and ...:
        print 'Possible decoder function found @' + hex(func.startEA)
```

Figure 5.3: Example IDAPython script that looks for multiple 513 and add 9 instructions

This approach suffers from a number of problems:

1. It's slow. Each time a query must be performed. The binary must be loaded into the disassembler, or an image of the full binary must be loaded into memory. Raw instructions or bytes must then be searched iteratively. Furthermore, this process must be repeated for each new query.

2. It's prone to false positives in the presence of obfuscations like the dead code seen in Figure 5.2. Previous authors [7] have suggested using compiler optimizations, such as

the elimination of *dead stores*[1], as a means of removing junk. Indeed, we tried that approach using our own deobfuscation tool. However, Destory RAT's authors carefully designed their chaff instructions to make it difficult to distinguish from legitimate code. Figure 5.4 depicts a more heavily obfuscated version of the decoder function (sample MD5: 2744c6dcc9edce0bd4ce8c6fdd5ec2ea, VirusTotal [54])). Note that junk instructions have been interleaved between the actual instructions (compare with the version seen in Figure 5.2). Furthermore, note that the instructions always read and then write a memory location. Therefore, dead stores elimination does not work.

3. It's labor intensive. The relationship between instructions must explicitly defined in a script for each unique query.

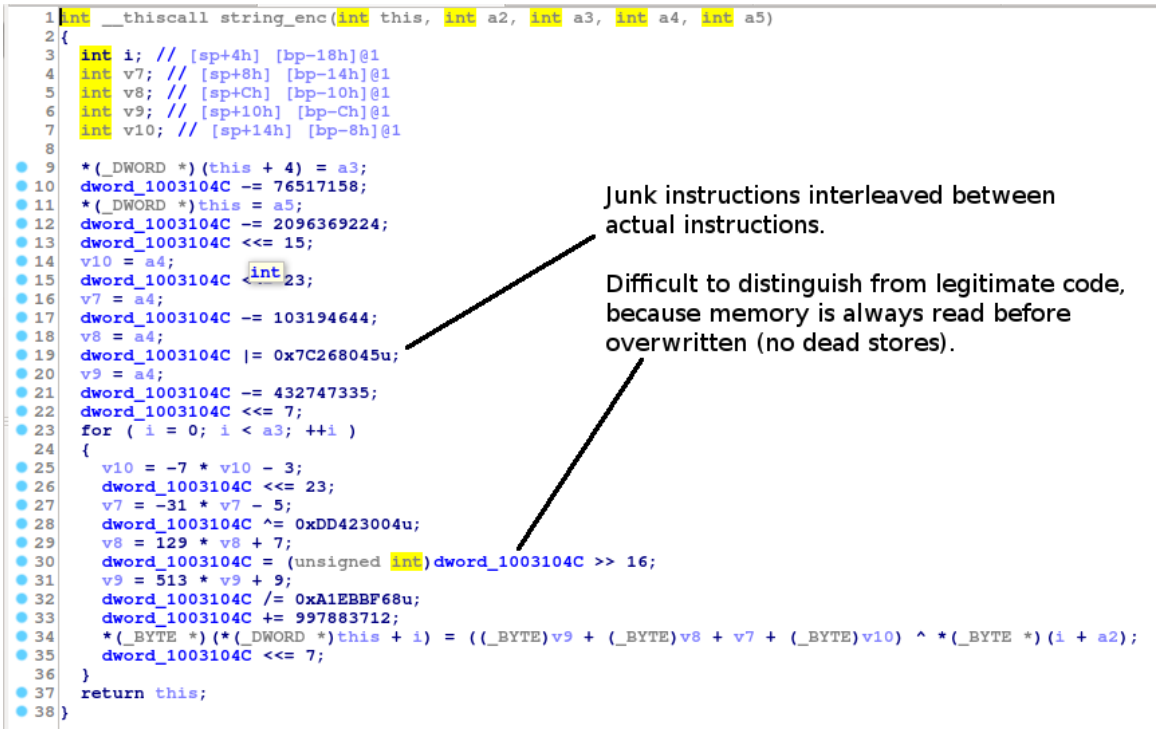


Figure 5.4: Decoder with junk instructions interleaved between every actual instruction. VirusTotal [54]

[1]A dead store refers to the situation where one instruction writes to a memory/register location, followed by another instruction that immediately overwrites the previous value without using it first. Compilers recognize this situation as being sub-optimal, and eliminate the first instruction

Another potential approach at identifying the decoder is by examining the program's function-call graph, as suggested by Hu et. al [28]. For example, knowing that the string decoder is invoked prior to a call to `LoadLibrary` or `GetProcAddress`, it might be possible to identify the decoder by examining functions that are called just before those two routines. Unfortunately, Destory RAT's authors have also incorporated numerous control-flow obfuscations, such as spurious API calls and functions that serve no purpose, which would complicate that type of analysis.

5.1.4 Our approach

After creating an index containing the 223 samples, we issued a series of queries to answer the following questions: 1) How many of the samples contained decoders using the already known constants? 2) Of the samples in which the known decoder was not found, how many of these samples used different constants or were entirely different implementations?

Queries to answer the first question captured the following properties of the decoder:

1. The function follows the `thiscall` convention. (i.e., The function is most likely a member function of an object, and is passed a reference to this object via `ecx`).
2. The function stores the length of the decrypted string and the location of the decrypted string within this object.
3. The function takes parameters that serve the following roles: a seed for the LCG, a decryption key, a length for the encrypted string and pointers to an input and output buffer.
4. The function performs an xor of each byte pointed to by the input buffer with the sum of four variables.
5. The variables are modified using a combination of bit-wise shifts, adds and subtractions. The value in each variable is not influenced by any other, except its previous value in each loop-iteration.

Figure 5.5 shows part of a query that captures the high-level statement `x = k1*x + k2`. Note that the instructions from the query are the same as those found in known Destory RAT samples, except variables have been introduced to capture potential variants (i.e., different stack offsets, registers, $k1$ and $k2$ values). A register followed by ! means that the instruction must use the specified register. $< V0 >$ is an integer variable that must be used consistently across the instructions in the query. (i.e. It doesn't matter what the integer is, as long as it is the same in all instructions that use the variable.) $<? >$ is a wild-card that will match any numeric value. rx means a register that must be used consistently across all instructions.

```
...
1     mov r0, [ebp!  + <V0>]    Stack variable moved into a register
2     shl r0, <?>               Register is left-shifted to perform k1 multiplication
3     mov r1, [ebp!  + <V0>]    Same stack variable is moved into another register
4     lea r2, [r1 + r0 + <?>]   Shifted result is added with original value and some constant k2
5     mov [ebp!  + <V0>], r2    Result is stored back into the same stack location
...
```

Figure 5.5: Portions of a query that captures the modification of a variable within the decoder's loop

With our first set of queries, we were able to determine that 170 of the 223 samples contained the already known decoder pattern. Furthermore, we verified that they all used the known constants. (i.e. We generated queries with exact values of $k1$ and $k2$.)

Next, we manually disassembled some of the samples that did not contain the known decoder. It soon became obvious that these samples did not use a string decoder at all, because we found unencrypted .dll names and function names being passed to `LoadLibrary` and `GetProcAddress` directly. We verified this belief by doing a simple string search for .dll names across the 53 samples, finding the strings in each case.

Therefore, we concluded that early variants of Destory RAT did not use string obfuscation. Most likely, the malware's authors added this functionality to increase their code's resiliency later.

5.1.5 Comments and Limitations

PdgGrep is an architecture for instruction search. Although it gives users the ability to identify variants of non-sequential, data/control-dependent instructions, it was not designed to perform search for semantically equivalents sets of instructions. We acknowledge that we would not have been able to find implementations of the decoder with different instruction patterns initially. However, by giving analysts the ability to quickly encode variants of known patterns, we allow them to focus on the aberrant samples that do not conform to known patterns. Furthermore, as will be seen in the next two case studies, variants of malware functions are often *not semantically equivalent*. Rather, they are similar in the sense that share significant portions of the same code.

5.2 Case study: Enumerating CNDrop's Command and Control servers

In this section, we present a case study that demonstrates PdgGrep's utility in finding functions that contain implementation changes as well as those from obfuscation (i.e., the functions are similar, but are *not semantically equivalent*).

5.2.1 Background: 'Phone-home' procedure

The malware family involved in this study is classified as a *dropper*. For the remainder of this section, we shall refer to this group as CNDrop[2]. CNDrop is the first stage in a multi-step attack, that converts a victim into a remotely-controllable zombie. When a host becomes infected, the malware hooks itself into the system by manipulating registry keys and creating hidden services. It then tries to contact a Command and Control (C2) server, from which it can download additional malware and instructions. For analysts, identifying C2 addresses is an important step for eliminating the root of the problem.

[2]Commercial anti-virus tools did not produce a consistent label for the family

Members of CNDrop contain a 'phone-home' subroutine, shown in Figure 5.6, which decrypts a string that is used as the address of a C2 server. Pinpointing this function allows analysts to quickly enumerate active servers (i.e. the strings are encrypted using a simple algorithm, found within the file).

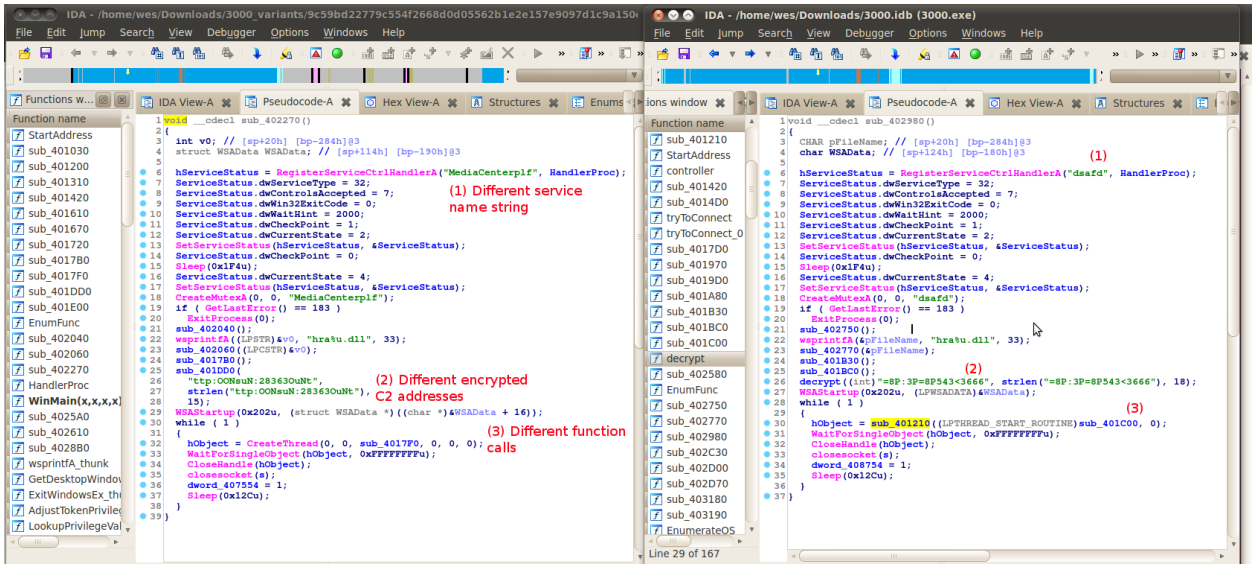


Figure 5.6: Function that sets up connection to C2 server (i.e., *Phone home* procedure)

The 'phone-home' procedure is obfuscated using randomly inserted NOPs, which limits the effectiveness of exact string matching (see Figure 5.7). Furthermore, different variants of the function exist amongst CNDrop members. For example, in Figure 5.6, the version on the left spawns a new thread directly by calling `CreateThread` (marked as (3) in red). The version on the right calls a helper routine, which manages threads.

To summarize, although the procedure is similar amongst all members of CNDrop, instruction-level differences arise from both obfuscation and implementation changes.

Figure 5.7: Randomly inserted NOPs used to thwart exact, signature matching

5.2.2 Problem Description

Starting with two samples of CNDrop (downloaded from websites blacklisted by `http://lists.clean-mx.com`), find other variants within the CERT collection, and identify their 'phone-home' procedures.

5.2.3 Our approach

Our approach was divided into two stages: 1) Identify and index suspected CNDrop binaries within CERT's collection of (mostly) unlabeled binaries. 2) Identify functions that are *similar* to the 'phone-home' procedures found in the first two samples.

Definition 2. *A function F can be thought of as a bag of features, $F = \langle K_1, K_2, ..., K_n \rangle$, where features, K_x, are unique index keys. (i.e., a function is a set of index keys that would be generated for its constituent instructions). Given a pair of functions A and B, B is similar to A if $\frac{|A \cap B|}{|A|} \geq T$ for some threshold T.*

Using BigGrep [57], we identified 854 possible CNDrop files by searching for two strings found within the starting samples: 'DOWNFAIL' and 'BAIDU'. BigGrep is a probabilistic exact N-gram search system[3]. There is no guarantee that the files actually contain the strings. Furthermore, there is no guarantee that the files are actually CNDrop members. After constructing an index from the candidate pool, we identified similar functions as follows:

1. We generated index keys for each of the 'phone-home' functions from the initial pair, and queried for them in the index.
2. For each set of returned function, we maintained a count of the number of times each had been seen across all of the queries.
3. We computed the similarity of each function by dividing the counts with the total number of keys generated in Step 1.
4. We recorded all functions with a similarity greater than 70%.

We identified 57 'phone-home' procedures, divided into 4 different groups. Groups were similar to the initial pair by: 99.39%, 87.73%, 84.05%, and 83.44%. (results are summarized in Figure 5.8) We verified that the functions were actually 'phone-home' in a disassembler. We identified C2 servers in both China and the US. Figure 5.9 shows an IP lookup of one of the discovered addresses.

Index size	854 binaries
Functions found	57
Function variants found	4

Figure 5.8: CNDrop results summary

[3] At the time of writing, a PdgGrep index has yet to be built for the entire CERT collection. However, this N-gram search step could be avoided in the future, when sufficient resources are allocated for an index spanning the entire collection

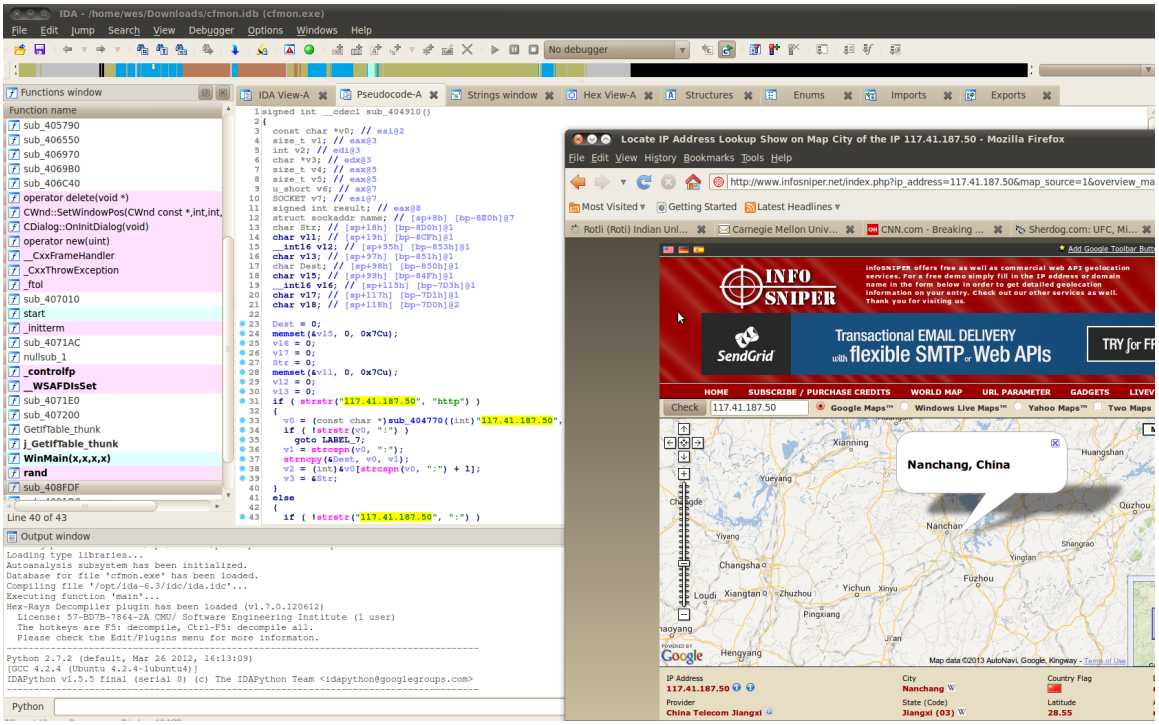


Figure 5.9: IP lookup of C2 server address

5.3 Case study: 9002

This section discusses a malware family, known as 9002, that poses a number of significant challenges to automated analysis systems: 1) anti-emulation, 2) self-modification and 3) code that is executed in memory, but never written to disk. We discuss how we were able to detect members of this family from a candidate set. We then discuss how we processed memory dumps from the identified samples to help enumerate variants of the family's command and control functionality. For additional information about 9002's C2 protocol and disk-less execution technique, we direct the reader to FireEye [20, 21].

5.3.1 Background: Anti-emulation and self-modifying code

Unlike the previous two case studies, 9002 modifies its own instructions at run time. Therefore, analysis systems that rely solely on static-analysis are presented with a different view than one after self-modification has completed. This strategy is also effective at helping the malware evade signature detection, as the modification code is different in each sample. The malware also employs a number of tactics to thwart dynamic-analysis and automated unpacking: 1) Floating-point code that is not handled by all emulators and 2) Multi-threaded code that confuses emulators [51].

Therefore, to successfully study 9002 samples, a system must identify the 1) self-modification code/thread, 2) restore the actual malware instructions by directing an emulator to the correct point, and 3) analyze a memory dump of the deobfuscated code.

5.3.2 Problem description

Using a set of 542 candidate binaries, selected from the CERT collection based upon their use of a compression algorithm known to be employed by the 9002 family and other non-malicious programs:

1. Identify the actual 9002 samples, specifically the code that performs the self-modification.
2. After obtaining a memory dump of the valid instructions, identify a particular C2 function containing a protocol version number.

5.3.3 Our approach

9002 identification

9002 samples contain a subroutine that spawns a thread that performs the self-modification. Fig. 5.10 shows two variants of this procedure (MD5s: 8627a63559f1632c523e7270e7fb8f14 and 2cf2320be1e5e4bd6603c88ab1529207, VirusTotal [55, 56]). Note that the functions are similar: They both allocate memory buffers using `VirtualAlloc`. They then spawn the self-

modification thread by calling `CreateThread`. However, the functions also contain significant differences: For example, the sample on the left contains anti-emulation code that causes the malware to sleep for 1000 seconds if a particular registry key is set.

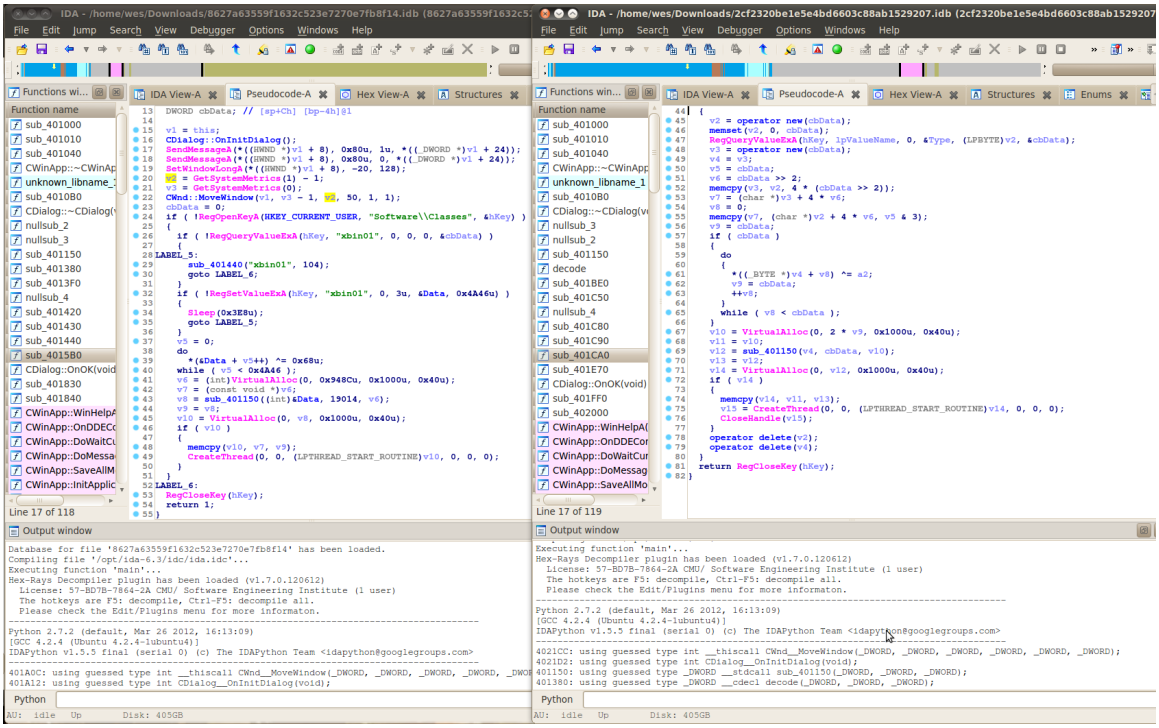


Figure 5.10: Two variants of the function used to spawn the self-modification thread (VirusTotal [55, 56])

After building an index of the 542 files, we identified 9002 samples by conducting a similarity search for the thread-spawning routine in the same way that we searched for CNDrop's 'Phone Home' procedure (see Sec. 5.2.3). We identified 111 samples. We confirmed that 108 of these files were variants of previously seen 9002 implementations. The other 3 samples were also confirmed to be 9002, and had not been encountered before.

Processing memory dumps and functions recovery

As described earlier, 9002 modifies its own code at runtime, and the malware's actual malicious functionality is never written to disk. Therefore, after positively identifying the 111 samples,

we produced 132 memory snapshots (i.e., memory dumps captured using an emulator) of their payloads.

From these raw memory dumps, we had to identify valid functions (i.e., starting and ending addresses) so that the code could be analyzed. For this purpose, we used our modified version of ROSE and the industry standard decompiler, IDA Pro [46], for this purpose. The following is a head-to-head comparison of the results:

- 132 memory snapshots processed.
- IDA Pro discovered a total of 6887 functions.
- ROSE discovered all of these functions as well as 22828 additional functions.

We validated the additional functions by writing an IDAPython script that invokes `MakeProcedure` [46] at each of their entry addresses. `MakeProcedure` is a routine that instructs IDA to try and create a procedure at a given address. IDA perform various checks, such as a stack neutrality test, to verify that one can be made. If the tests fail, IDA will return an error.

- 20279 were successfully validated. (i.e., 20279 of the function boundaries identified by ROSE, but not initially by IDA, were subsequently validated as functions by IDA once fed their starting and ending points.)
- 3191 could not be validated. (i.e., IDA was unable to confirm that the boundaries marked by ROSE actually corresponded to functions in these cases.)

In some cases, the additional functions could not be validated, because they terminated with `calls` that never return. However, they were actually valid procedures.

C2 protocol enumeration

9002 variants communicate with a command and control server to download further instruction, additional malware, and to exfiltrate data (see [21] for a thorough discussion of 9002's C2 protocol). Packets used between the client and server are identified by a 32-bit version number that

corresponds to a particular date (i.e., 0x20111209, December 9, 2011). The procedure responsible for handling communications contains a hard coded check for this number. Fig. 5.11 shows a version of this routine. Note, the check for 0x20111209 that causes the code to exit if a received message does not contain the same number.

```
52      return result;
53    v27 = 0;
54    v22 += 4;
55    sub_3057(v25, 0x401360u, &v27);
56    v26 = v22 - v23;
57    result = sub_3A17(v14 + 192) - (v22 - v23);
58    if ( result < v27 )
59      return result;
60    if ( (signed int)v27 > 2 )
61      sub_38D7(v22, v27);
62    sub_3B2A(v14 + 192);
63  }
64  else
65  {
66    sub_38D7(a3, a4);
67  }
68  v33 = 4;
69  v34 = 76;
70  while ( 1 )
71  {
72    result = sub_3A17(v14 + 140);
73    if ( result < v34 )
74      break;
75    v8 = v34;
76    v9 = sub_3BFF(v14 + 140, v7);
77    sub_DE04(&v28, v9, v8);
78    sub_D219((char *)&v28 + v33, v34 - v33, v28);
79    if ( v32 != 0x20111209 )
80      return sub_5010(v14);
81    v21 = v34 + v30;
82    result = sub_3A17(v14 + 140);
83    if ( result < v21 )
84      return result;
85    if ( v31 && v29 == 1 )
86    {
87      v20 = sub_62D3(v31);
88      v19 = v31;
89      v10 = *(_DWORD *)(v14 + 180);
90      v11 = v20;
91      v12 = v30;
92      v13 = sub_3BFF(v14 + 140, v30);
93      sub_3D02(v13, v12, v11, &v19, v10);
94      v18 = sub_62D3(8);
```

Figure 5.11: Communication subroutine containing check for 0x20111209

At an assembly level, this comparison is implemented using the `cmp` instruction. Given this knowledge and the fact that the protocol version number corresponds to a date, we decided to encode a query that identifies methods containing a `cmp` instruction with an integer operand greater than 0x20090000 (i.e., `cmp [ebp! + <?>], <UGT0x20090000>`).

From the dumps, we identified 4 distinct protocol versions. Enumerating and understand-

ing 9002's communication protocol allows administrators to create network signatures for the malware.

Chapter 6

Limitations and Future Work

In this chapter, we discuss the limitations of PdgGrep. We also provide suggestions for improvements and future work.

6.1 Malware disassembly and complex obfuscations

PdgGrep was designed to allow users to search for data/control-dependent instructions in the presence of simple obfuscations. The system assumes that a binary can be disassembled correctly, so that dependency relationships can be established. However, in practice, revealing malicious code using either static or dynamic analysis is often a non-trivial task. Indeed, a significant amount of work has been done on this topic alone [4, 16, 32, 33]. Malware authors are constantly devising new techniques to delay/thwart analysis. Therefore, we believe that flexibility is the key. We chose ROSE as our disassembler, in part, because it is highly-configurable and programmable. As malware authors devise new tactics, we hope that the users of PdgGrep can adapt the disassembly front-end to meet these new challenges.

It is certainly the case that malware authors can use techniques, such as the substitution of semantically-equivalent instructions, to destroy particular instruction relationships. However, in many cases (including malware that use semantically-equivalent instructions), *some* instruction

relationships are preserved. PdgGrep allows analysts to quickly encode queries to perform fast search. In order to pick out a sample from a data set, an analyst need only find one characteristic instruction pattern to search for.

6.2 Index construction overhead

Index construction is expensive from both a time and disk point-of-view. As was discussed in Sec. 4.1 and Sec. 4.2, the implementation of the index (in particular, the choice of `KeyHash` and its parameters) is a key determinant of just how expensive this upfront cost is. The issue is complicated, because different implementations often represent a tradeoff between query speed and storage. A direction for future work is to design better filters for instruction relationships. For example, the ubiquitous index key composed of the prologue instruction pair `push ebp` and `mov ebp, esp` could certainly be excluded from the index. Disk overhead and construction time could be further reduced by the addition of other filters. Also, PdgGrep can easily be parallelized. Indexes could be built on multiple nodes simultaneously. Future work includes writing client/server software that would allow users to query such a cluster.

Chapter 7

Conclusion

In this dissertation, we sought to design a practical system that would enable users to search for instruction patterns in the presence of many common obfuscations. We propose that in the presence of these simple tactics, many data/control-dependent instruction relationships are still preserved. The core idea of this work is to index binaries by these relationships.

A distinguishing aspect of this work is that it was designed for search in mind, and not as a detector. In other words, our goal is to provide a means for analysts to codify their intuition (i.e., unique instruction patterns) about malware families, and query a large set of files with this understanding. It is *not* our goal to replace the analyst.

The case studies on real-world malware demonstrate that PdgGrep can be an effective tool at helping to find variants of malicious code. Our scalability experiments demonstrate that indexes can be built at a reasonable cost in terms of disk and time.

Bibliography

[1] Yices Website. http://yices.csl.sri.com. 1.1.1

[2] ASERT. ASERT MindshaRE: Finding byte strings using IDAPython. http://www.arbornetworks.com/asert/2013/07/asert-mindshare-finding-byte-strings-using-idapython/. 5.1.3

[3] Michael Bailey, Jon Oberheide, Jon Andersen, Z.Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 178–197. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74319-4. doi: 10.1007/978-3-540-74320-0_10. URL http://dx.doi.org/10.1007/978-3-540-74320-0_10. 2.1.1

[4] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1871–1878, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi: 10.1145/1774088.1774484. URL http://doi.acm.org/10.1145/1774088.1774484. 6.1

[5] L Bohne and T Holz. *Pandora's bochs: Automated malware unpacking*. PhD thesis, University of Mannheim. http://archive.hack.lu/2009/PandorasBochs.pdf, 2008. 3, 3.2

[6] Rodrigo Branco. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-virtual machine technologies. http://research.dissect.

pe/docs/blackhat2012-paper.pdf. 2

[7] D. Bruschi. Detecting self-mutating malware using control-flow graph matching. In *Lecture Notes in Computer Science*, volume 4064, pages 129–143. Springer-Verlag, 2006. 1.1.2, 1.1.2, 1, 2.4, 2

[8] CERT Website. CERT Website. http://www.cert.org. 1

[9] Silvio Cesare and Yang Xiang. A fast flowgraph based classification system for packed and polymorphic malware on the endhost. In *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE Computer Society, 2010. 1.1.2, 1.1.2

[10] Xu Chen, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008. 2.1.1

[11] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn X. Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005. 1.1.1

[12] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn X. Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005. 1.1.1, 4

[13] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005. 1

[14] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior, 2007. 2.1.1

[15] Cory Cohen. Personal communication with cert malware researcher. 1, 1, 1.1.3, 5.1.2

[16] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 167–176. IEEE, 2009. 6.1

[17] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative data-flow analysis, revisited. Technical Report TR04-432, Rice University, 2004. 3.3.2

[18] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012. 2.1.1

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041. URL `http://doi.acm.org/10.1145/24039.24041`. 1.2, 3.3.3, 3.3.4

[20] FireEye. FireEye's analysis of the sunshop campaign. `http://www.fireeye.com/resources/pdfs/fireeye-malware-supply-chain.pdf`, . 5.3

[21] FireEye. FireEye operation ephemeral hydra. `http://www.fireeye.com/blog/technical/cyber-exploits/2013/11/operation-ephemeral-hydra-ie-zero-day-linked-to-deputydog-uses-diskless.html`, . 5.3, 5.3.3

[22] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 2, pages 112–115 vol.2, 2002. doi: 10.1109/ICPR.2002.1048250. URL `http://dx.doi.org/10.1109/ICPR.2002.1048250`. 3.5.1

[23] Google LevelDB. Google LevelDB. `http://code.google.com/p/leveldb/`. 3.5.2

[24] Command5 Research Group. Command and Control in the Fifth Doman. `www.`

commandfive.com/papers/C5_APT_C2InTheFifthDomain.pdf,. 5.1

[25] Command5 Research Group. Command 5 SKHack Analysis. http://www.commandfive.com/papers/C5_APT_SKHack.pdf,. 5.1

[26] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008. 3

[27] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259030. URL http://doi.acm.org/10.1145/2259016.2259030. 2.2

[28] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653736. URL http://doi.acm.org/10.1145/1653662.1653736. 1.1.2, 2.1.1, 3, 5.1.3

[29] IDAPython. IDAPython. https://code.google.com/p/idapython. 5.1.3

[30] Intel. IA-32 Developer Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. 3.2.1

[31] Harold Johnson. Data flow analysis for 'intractable' system software. In *SIGPLAN Symposium on Compiler Construction*, pages 109–117, 1986. 3.3.2

[32] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM. ISBN

978-1-60558-780-6. doi: 10.1145/1655148.1655151. URL http://doi.acm.org/10.1145/1655148.1655151. 3, 2.2, 3.2, 6.1

[33] A. Kapoor. An approach towards disassembly of malicious binaries. Master's thesis, University of Louisiana at Lafayette, 2004. 2.1.1, 6.1

[34] David Clark Khalid Alzarouni and Laurence Tratt. Semantic malware detection. Technical Report Technical Report TR-10-03, University College London, London, UK, 2010. 1.1.1, 1.1.1

[35] Ákos Kiss, Judit Jász, and Tibor Gyimóthy. Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Control*, 13(3):227–245, September 2005. ISSN 0963-9314. doi: 10.1007/s11219-005-1751-x. URL http://dx.doi.org/10.1007/s11219-005-1751-x. 3.3.1, 3.3.2

[36] Nithya Krishnamoorthy, Saumya Debray, and Keith Fligg. Static detection of disassembly errors. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 259–268. IEEE, 2009. 2.1.1

[37] Felix Leder. Classification and detection of metamorphic malware using value set analysis. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on Malware*, pages 39–46, 2009. 1.1.1, 1.1.1, 3

[38] Kirti Mathur and Saroj Hiranwal. A survey on techniques in detection and analyzing malware executables. *International Journal*, 3(4), 2013. 2.1.1

[39] McAfeeVirut. W32.Virut Family. http://download.nai.com/products/mcafee-avert/documents/combating_w32_virut.pdf. 2.2

[40] Microsoft. Win32 Structured Exception Handling. http://www.microsoft.com/msj/0197/exception/exception.aspx. 2.2

[41] Microsoft Z3 Website. Microsoft Z3 Website. http://z3.codeplex.com. 1.1.1

[42] OpenMalware. Open Malware. http://www.openmalware.org. 4.1.2

[43] Portcullis. Destory RAT Revealing the Hidden Data. http://www.portcullis-security.com/destroy-rat-revealing-the-hidden-data/. 5.1

[44] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5):25:1–25:54, September 2008. ISSN 0164-0925. doi: 10.1145/1387673.1387674. URL http://doi.acm.org/10.1145/1387673.1387674. 1

[45] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In *Information Security*, pages 1–18. Springer, 2007. 2.1.1

[46] Hex Rays. IDA Pro. http://www.hex-rays.com/idapro. 1.1.3, 5.3.3

[47] rose. ROSE website. http://www.rosecompiler.org. 2.1.1, 2.2.1, 3.1, 3.2, 3.2.2, 3.3.2

[48] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 289–300. IEEE, 2006. 3

[49] SecureListHidrag. W32.Hidrag. https://www.securelist.com/en/descriptions/old20627. 2.3

[50] Madhu K. Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *J. Comput. Virol.*, 7(2):107–119, May 2011. ISSN 1772-9890. doi: 10.1007/s11416-010-0141-5. URL http://dx.doi.org/10.1007/s11416-010-0141-5. 1.1.2, 1.1.2

[51] Peter Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005. 5.3.1

[52] A. Vasudevan and R. Yerraballi. Cobra: fine-grained malware analysis using stealth localized-executions. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15 pp.–

279, 2006. doi: 10.1109/SP.2006.9. 2.1.1

[53] VirusTotal06f3. Virus Total Analysis results for Destory sample 06f3. https://www.virustotal.com/en/file/14ea22f4374e1c03018a94ec828cf4ea74fe4bf8e09a009dc73b8bd96fd71ade/analysis/. (document), 5.1.1, 5.2

[54] VirusTotal2744. Virus Total Analysis results for Destory sample 2744. https://www.virustotal.com/en/file/7728a6e379978c98ba83a20b1d0a1a1a8e3e13bb6848092c269ffdf955245fad/analysis/. (document), 2, 5.4

[55] VirusTotal2cf2. Virus Total Analysis results for 9002 sample 2cf2. https://www.virustotal.com/en/file/a6b84eb6c0b29172d1b51fc473a05e15bb5b47a20004ad901f62288554ed42c1/analysis/. (document), 5.3.3, 5.10

[56] VirusTotal8627. Virus Total Analysis results for 9002 sample 8627. https://www.virustotal.com/en/file/9f60d73cfdb1452c95ae6dc4ea0f6b93661ca145ca4ec54eeaaaf5721f8c89b6/analysis/. (document), 5.3.3, 5.10

[57] Sagar Chaki Wesley Jin. BigGrep: A Scalable Search Index for Binary Files. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on Malicious and Unwanted Software*, October 2012. URL http://www.malware2011.org/. 5.2.3

[58] YARA. YARA website. http://code.google.com/p/yara-project/. 1.1.3, 4, 5.1.3

[59] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, BWCCA '10, pages 297–300, Washington, DC, USA, 2010.

IEEE Computer Society. ISBN 978-0-7695-4236-2. doi: 10.1109/BWCCA.2010.85. URL http://dx.doi.org/10.1109/BWCCA.2010.85. 1