Pomona College
Department of Computer Science

Searching For Malware In Random Byte-Strings

Will Coster

May 04, 2013

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professor Kevin Coogan, advisor

Abstract

This paper presents the design and implementation of a metamorphic engine intended to be capable of evading current commercial classification methods, as well as several promising techniques from current anti-malware research. The resulting system is able to achieve high levels of variation among the instances generated while at the same time creating programs that share key statistical properties with benign programs. The engine achieves this by reframing malware obfuscation as a combined compiler and search problem and is heavily influenced by developments in the field of return-oriented programming. Given a semantic blue print of a program the engine pieces together blocks of random bytes to create the final executable. By controlling the distribution of the random bytes used to generate the executable, the engine is able to influence statistical properties of the resulting executables. To achieve the high degree of variation necessary to evade commercial detection techniques, the process of piecing together and choosing random bytes is performed nondeterministically.

Acknowledgments

I would like to thank Professor Coogan for advising me on this project. On everything from last minute paper suggestions to helping me organize and work through my presentation, his constant feedback and encouragement have been invaluable resources these past two semesters.

Contents

List of Figures

List of Tables

x

Chapter 1

Introduction

Modern malware relies on sophisticated obfuscation techniques to evade detection by antivirus software. In order to build better classifiers, it is useful to understand the limits and capabilities of these obfuscation techniques. This paper presents a new system for program obfuscation. There are two design goals for this system. The first is that the newly generated executables appear dissimilar to each other. The second is that the programs appear statistically similar to unobfuscated programs, thereby disguising the fact that any obfuscation was performed.

Historically malware detection has relied on various forms of static analysis, or techniques that attempt to identify malware prior to its execution. Static analysis has been favored because it has the desirable property that it is relatively quick, and when it works, it is able to stop malware before it is allowed to interact with the host. The classic example of a static method of detection, and one that still plays a major role in commercial antivirus software solutions, is byte signature matching. Byte signature matching classifies malware by checking if the file under consideration matches the byte signature of any known instances of malware.

Byte signature matching has the major drawback that it requires constant maintenance of a database of signatures in order to be effective. This means that newly minted malware enjoys a brief period where it is able to avoid detection before the databases have been updated to include its signature. In an attempt to thwart detection, malware authors have created systems that target this weakness. Modern malware typically uses various obfuscation schemes to scramble its byte signature every time a new host is infected. Because each infection has a unique byte signature it is not possible to create a universal byte signature that can classify each instance of a

specific piece of obfuscating malware. This type of malware poses a serious problem for antivirus vendors.

The classification of obfuscating, or metamorphic, malware is currently an open research problem. There are several emerging methods of detecting metamorphic malware that have been shown to perform well in lab settings. One category of techniques uses available knowledge of the host environment in order to statically classify obfuscated malware. In practice, the systems used to disguise malware are very good at creating new instances that appear very different from each other, however, as a side effect these instances also appear very different from normal benign programs. As a result, successful detection systems have been created that simply compare the opcode frequencies of a program under consideration to those of "normal" programs [RMI12].

In addition to the historically favored static methods, dynamic techniques that observe the execution of an instance of malware have shown promising results as well. These systems make the observation that while the byte level appearance of an obfuscated malware may change, how it interacts with the host largely remains the same. Examples of such methods include fingerprinting the behaviour of a program by the recording the sequence of syscalls made during execution.

This paper presents a new type of metamorphic system that is capable of producing semantically equivalent programs which exhibit a high degree of variation in their byte signatures while at the same time remaining statistically similar to benign programs. This new engine achieves this by reframing malware obfuscation as a combined compiler and search problem. Given a semantic blue print of a program the engine pieces together blocks of random bytes to create a working executable. By controlling the distribution of the random bytes used to generate the executable, the engine is able to influence statistical properties of the resulting executables. To achieve the high degree of variation necessary to evade byte signature detection, the process of piecing together and choosing random bytes is performed nondeterministically.

Chapter 2

Related Work

Malware classification, and more generally program analysis, fall under two umbrella categories: static and dynamic analysis. Static methods analyze a program using only the byte level representation as information. Being able to statically analyze a program is obviously desirable in the context of malware analysis as it allows for the classification of malware before it performs malicious actions on the host.

Modern antivirus software relies heavily on static analysis to classify malware. The current approach is to maintain a database of byte signatures (complex patterns of byte sequences) for every piece of malware ever encountered. If a program under investigation matches any of the byte signatures in the database then the program is classified as malicious. This technique is nice because it can be done relatively quickly which means that there is only minor disruption in the users work flow.

Dynamic analysis gathers information about a program as it executes. When dealing with potentially malicious programs, as in the case of malware analysis, it is common practice is to run it in a sandboxed or emulated environment. Dynamic analysis covers techniques that include computing a live trace of the instructions a program executes and anomaly detection which monitors a program interaction with the host for behaviour that deviates from normal program behaviour for some definition of normal. Sequences of syscalls or the api calls made over the lifetime of a program have been shown to be a good model of program behaviour [SYR$^+$10].

Due to modern commercial antivirus software's reliance on static analysis, a large portion of the evasion efforts staged by malware authors have been in the attempt to thwart byte signature detection by creating malware that obfuscates its byte signature for each host it infects.

The earliest class of malware capable of automatically altering its byte signature is polymorphic malware. The defining attribute of polymorphic malware is that it uses encryption to randomize the appearance of the byte level representation of the program [Bea07]. The simplest case of polymorphism partitions the malware into two segments, a small decryption routine and the payload. The payload is encrypted with a randomly chosen encryption key and upon execution the small decryption routine decrypts the body in memory and then transfers control to the main portion of the program. New generations of the program are created by altering the encryption key and applying minor syntactic variations to the decryption routine. The result is that the new binary has a widely different byte signature than it's predecessor. More sophisticated approaches to polymorphism include modularized encryption schemes where portions of the malware body are decrypted on the fly as they are needed and then re-encrypted.

This form of obfuscation has largely been defeated through the use of hybrid detection schemes that use aspects of both dynamic and static analysis [RHD$^+$06, MCJ07, KPY07]. In such schemes, the suspected malware is allowed to execute in a sandboxed environment while the antivirus program monitors execution for behaviour characteristic of a packed program, e.g. passing control to a section of memory marked as data. The monitoring program can then dump the portion of memory that was jumped into, oftentimes revealing the entire unpacked contents of the malware. Once the unpacked contents of the malware has been revealed it is then vulnerable to common byte signature detection.

The next generation of obfuscating malware, metamorphic malware, achieves its obfuscation by applying semantics-preserving transformations to the bytecode representation of its instructions. The result is a program that executes a unique sequence of instructions while performing the same semantic operation. There are several existing metamorphic engines of varying complexity though most implement similar transformations which include: movement and interleaving of logical sequences of instructions, insertion of random non-executed instructions, and swapping of registers [Bea07].

Theoretically, metamorphic obfuscation is very strong. Reliably identifying a piece of malware that has undergone metamorphic transformations has been shown to be NP-complete [Spi03]. Further, there are known methods of control flow obfuscation that make simply determining the control flow graph of a program NP-hard [WHKD01]. These proofs pertain to static analysis and provide a bound on the ability to statically classify metamorphic malware. In practice however, there are static techniques being researched that have been shown to reliable classify modern metamorphic malware.

The key observation that makes classifying current metamorphic malware tractable is that while the sequences of bytes exhibited between various generations of metamorphic malware differs significantly, as a whole they exhibit certain statistical characteristics that allow them to be identified. For instance, the opcode frequency distributions of metamorphic variants has been shown to be relatively constant between variations. Comparing these opcode histograms have been shown to be robust enough in some cases to reliably classify real world metamorphic malware samples [RMI12].

In addition to failing to disguise certain commonalities, current metamorphic engines have also been shown to introduce properties that distinguish the generated malware from benign programs. Hidden Markov model based classifiers that capitalize on this observation have been shown to be particularly successful in classifying metamorphic malware [SW06].

Given this shortcoming with regard to statistical fingerprints in current metamorphic engines, recent work has explored creating metamorphic engines that are able to statistically blend in with benign applications. In some instances it has been shown that simply embedding whole subroutines taken from benign applications randomly through out a malware sample is enough to fool certain classifiers [LS11].

More sophisticated classification models that take into account the opcode instruction frequencies for the entire executable are more robust to these sorts of evasion tricks. Chi-squared tests have been shown to be more sensitive to the frequency histograms and thus are more robust to the minor alterations that occur when embedding portions of benign programs [TS12].

More recent advances in obfuscation have focused on altering the entire frequency histogram of a metamorphic variation to more closely resemble those of benign software. One particular system, which the authors referred to as mimimorphism, employs mimic functions to generate each variation [WGXW10]. The mimic functions create Huffman forests to represent the frequency distribution of opcodes. Given the initial version of a program and a pseudo-random number generator, the mimimorphic engine walks the Huffman forest to generate a new obfuscated version.

Other techniques build on the idea of incorporating benign code in each generation. Instead of simply using benign code fragments as dead code, these engines scan benign executables to generate a database of repurposable code fragments from which subsequent generations of the program are pieced together [MH12]. The intellectual roots for this type of metamorphic engine come from a recently discovered shellcode execution attack known as return-oriented programming.

Classic examples of buffer overflow attacks involve the attacker over-

flowing a buffer allocated on the stack, filling it with instructions that will spawn a shell [Ale96]. The portion of the content copied into the buffer that overwrites the return address of the stack frame contains an address that points into the overrun buffer. The result is that when the current function exits by executing the "ret" instruction, control will pass to the injected code thus giving the attacker the ability to execute arbitrary code.

Since the publication of [Ale96] there have been several advances that make classic buffer overflows much more difficult to execute. One of the most notable examples of these measures is the introduction of $W \oplus X$ memory which marks every segment of memory allocated to a process as either writable or executable but not both. The result of this is that anywhere the attacker may be able to inject malicious code will have permissions set that disallow it from being run.

These measures spurred a new form of attack known as "return to libc." In this scenario, the attacker capitalizes on the fact that most programs are linked to libc, which contains functions that can be used to perform potentially malicious operations (notably the "system" method which executes a string argument in a shell) [Sha07]. When the attacker overruns a buffer they are able to control the contents of the stack frame. In $x86$ architectures, the return address and function parameters are passed on the stack. By specially crafting the contents of the stack, the attacker can set the return address to the libc "system" method and place a malicious command as the parameter thereby executing an arbitrary shell command when the current function exits.

Return oriented programming builds on the key idea seen in return to libc attacks of reusing existing instructions in a malicious manner. Return oriented programming generalizes the idea by noting that because the return address of a method is stored on the stack, and that the attacker can control the contents of the stack, they can chain together code fragments (gadgets) from the victim executable that end in a "ret" opcode. As each code fragment "returns" the address of the next fragment is obtained from the tainted stack frame. It has been shown that the libc library alone contains sufficient and varied gadgets to constitute a Turing complete system [Sha07].

The "Frankenstein" metamorphic engine in [MH12] combines the principles of return oriented programming with the observation that programs statistically similar to benign programs are more difficult for static analysis techniques to classify. Their metamorphic system builds a collection of repurposable code fragments by scanning through a selection of all the available benign programs found on the host. Using these fragments, the

metamorphic system generates programs by taking a semantic blueprint of the desired program and searching for sequences of fragments that fulfil those semantics.

Chapter 3

Methodology

This section describes the theoretical underpinnings and the implementation of a gadget-based metamorphic engine. The engine maintains a large library that maps from discrete operations to bytes that perform those operations when executed. Leveraging this library and a semantic blueprint of a desired program, the engine generates an executable by nondeterministically piecing together bytes from the library in a way that fulfils the semantics of the program.

3.1 Gadgets

The idea of stitching together programs out of existing building blocks originated in the field of return-oriented programming [Sha07]. Return-oriented programming chains together small snippets of instructions ending in return statements by manipulating the return address on the stack. These snippets of instructions are called gadgets in the return-oriented programming literature and each gadget typically performs a single simple operation.

The traditional formalization and use of gadgets has been heavily influenced by the constraints present in return-oriented programming. Many of these imposed constraints are not present in the context of a metamorphic engine. For instance, the requirement that a gadget end in a return statement is necessary in return-oriented programming because that is the mechanism through which gadgets can be composed. However, given that this engine directly generates the text section of an executable this requirement is no longer necessary.

Therefore, it is useful to reformalize gadgets in the context where they are free of the constraints imposed by return-oriented programming. For the

Gadget Type	Parameters	Semantics
NoOp	—	—
LoadReg	out, in	$out \leftarrow in$
LoadConst	$out, value$	$out \leftarrow value$
LoadMemReg	$outReg, in, value$	$outReg \leftarrow [in + value]$
StoreMemReg	$outReg, value, inReg$	$[outReg + value] \leftarrow inReg$
Plus	out, in_1, in_2	$out \leftarrow in_1 + in_2$
Minus	out, in_1, in_2	$out \leftarrow in_1 - in_2$
Times	out, in_1, in_2	$out \leftarrow in_1 \times in_2$
Xor	out, in_1, in_2	$out \leftarrow in_1 \oplus in_2$
RightShift	$reg, value$	$reg \leftarrow reg >> value$
Compare	in_1, in_2	$eflag \leftarrow compare(in_1, in_2)$
Jump	$offset, reason$	$eip \leftarrow \begin{cases} eip + offset & \text{if } eflag \& reason \\ eip & otherwise \end{cases}$

Table 3.1: Enumeration of gadgets and their semantic definitions.

purposes of this engine, gadgets are defined as a byte-string and corresponding semantics given as a symbolic description of the effect on the execution state the byte-string would have upon execution.

To simplify the process of reasoning about a large collection of gadgets, a given byte-string is restricted to matching several of a collection of predefined gadget types. A Turing complete set of gadget types is given in [MH12], and serves as the basis for the gadget types in this implementation. Each type of gadget is parameterized over a list of hardware locations[1]. A complete listing of gadgets and their definitions is given in table 3.1.

Along with the core semantics presented in table 3.1, each byte-string also has an associated "clobber set." This set contains the set of hardware locations that the gadget overwrites in addition to performing its core semantics. Allowing gadgets to have limited side-effects describable by this clobber set allows for junk instructions to be present in the byte-strings for any given gadget type. This is a desirable feature for a metamorphic engine because it leads to greater variety in the byte instances for each gadget type and through that more variety in the generated programs. Concrete examples of gadgets along with their clobber sets are shown in table 3.2.

---
[1] Memory addresses and registers

Bytes	Instructions	Gadget Type	Clobber Set
01CB A100 0000 00	add ebx, ecx mov eax, $10	Plus(*ebx*,*ebx*,*ecx*)	*eax*
330D 0000 0000 8998 0000 0000 29CA	xor ecx, $42 mov [eax+$10], ebx sub edx, ecx	StoreMemReg(*eax*, 10, *ebx*)	*ecx*, *edx*

Table 3.2: Concrete examples of gadgets along with their associated clobber set

3.2   Semantic Blueprints

Gadgets as they are used in this engine occupy a role similar to assembly instructions in traditional code generation. They perform simple operations and are parameterized over hardware locations. Just as writing a program in pure assembly would require manually maintaining such things as stack frames and mappings of registers to variables, so too would attempting to write a program in pure gadgets.

Therefore to provide a more convenient programming experience it is necessary to be able to express program semantics using a higher level construct. The solution used in this implementation is "abstract statements." Abstract statements, or simply statements, are similar to gadgets in that the scope of operations they perform but abstract away the hardware locations into variables. A full listing of the statements supported by the engine is given in table 3.3.

3.3   Gadget Discovery

In order to create programs out of these gadgets, it is first necessary to actually find byte-strings that match the described gadget types. Therefore the first stage of the metamorphic engine is to pregenerate a large library that maps from parameterized gadgets to byte-strings that implement them.

11

Statements	Parameters	Semantics
noop	—	—
move	out, in	$out \leftarrow in$
add	out, in_1, in_2	$out \leftarrow in_1 + in_2$
sub	out, in_1, in_2	$out \leftarrow in_1 - in_2$
mul	out, in_1, in_2	$out \leftarrow in_1 \times in_2$
xor	out, in_1, in_2	$out \leftarrow in_1 \oplus in_2$
jump	$label, reason$	$eip \leftarrow \begin{cases} eip + offset(label) & \text{if } eflag \& reason \\ eip & otherwise \end{cases}$

Table 3.3: Enumeration of abstract statements and their semantic definitions. Note that parameters here are either variables or constants and that each statement has a clear correspondence with a gadget in 3.1

3.3.1 Byte Source

The first thing required to begin matching byte-strings to gadgets and inserting them into the library is to determine where the bytes are going to come from. The method proposed by [MH12] is to scan the host computer and read bytes from existing executables. The assumption is that because the gadgets are taken from existing executables, they share the same byte frequency distribution. Unfortunately, there is a potentially fatal flaw with this approach. This method of sampling bytes requires constant interaction with the host environment and in the processes performs an uncommonly large number of disk seeks. The result is that this process of sampling bytes is susceptible to detection through behavioural analysis [MDL$^+$12].

The naive alternative to this approach would be to randomly generate bytes from a uniform distribution. This solution successfully eliminates the suspicious behaviour of the previous approach. However, this benefit is achieved at the cost that the bytes being sampled no longer have any connection to the distribution of bytes found in benign programs.

A natural improvement to the naive approach is to precalculate the distribution of bytes found in benign programs and randomly sample bytes from this calculated distribution. This approach maintains all of the desirable benefits of the naive approach while avoiding the shortcomings of both the naive approach and the approach proposed in [MH12]. In addition, there is also the additional benefit that since the distribution is calculated in advance, it is no longer tied to the host architecture, which opens the door to cross-platform code generation.
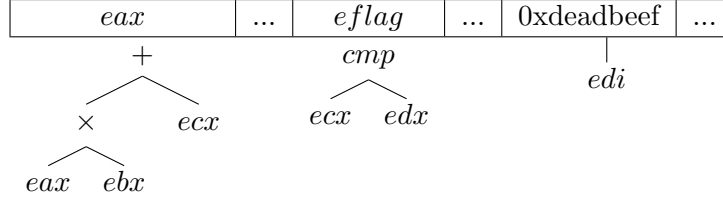
Figure 3.1: Example of a state forest produced by abstract evaluator.

3.3.2   Symbolic Evaluation

In order to match a given byte-string with the various gadget types that it may implement, it is necessary to be able to represent the effect on hardware that the execution of the byte-string will have. Modelling the effect on hardware is done by disassembling the byte-string and symbolically evaluating the corresponding instructions in sequence. The result is a series of execution-state forests that represent the progressive effect on hardware that executing the byte-string would have.

Each state forest contains a tree for each altered hardware location. In turn, each tree in the forest is an expression tree where nodes are algebraic operations and leaves are the initial values of hardware locations. An example of a state forest is given in figure 3.1.

The types of algebraic operations that are allowed in the expression trees are limited to effects which are unconditional and deterministic. Further each tree in the forest is required to correspond to a unique hardware location. This eliminates possible ambiguity which would further complicate the matching process and allows gadgets to be composed without worrying about unforeseen side effects.

3.3.3   Gadget Matching

Each type of gadget recognized by the engine has a corresponding general state tree that describes it's core semantics. Examples of what these state trees look like are given in figure 3.2. The wild card nodes are constrained to only match the leaf nodes which correspond to the parameters of the gadget types described above.

It would be theoretically possible to match gadgets on entire subtrees instead of just leaf nodes. This would allow gadgets to be discovered that performed intricate compound operations, and potentially increase the variability of the programs the engine is capable of producing. However this flexibility would come at a significant price, which is that the number of

Plus	LoadReg	StoreMemReg

| ... | *out* | ... | | ... | *out* | ... | | ... | $[out + value]$ | ... |

$$+$$
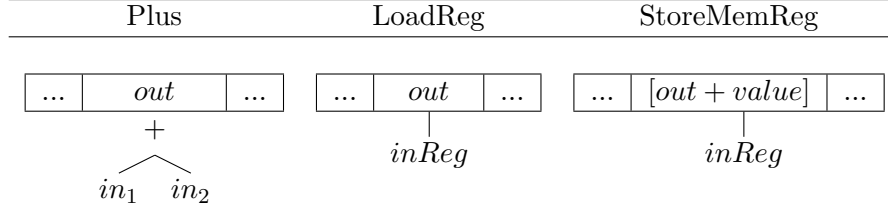$$\overset{\frown}{in_1 \quad in_2}$$

$inReg$

$inReg$

Figure 3.2: Example of general state trees for different types of gadgets.

unique parameterizations of any given gadget type would be unbounded as each parameter could be any valid expression tree.

Matching a byte-string and a corresponding sequence of instructions to gadget types is done by attempting to match each node of the gadget's general tree with any tree in the byte-string's state forest. The hardware locations that correspond to the unmatched trees in the state forest make up the clobber set of the matched gadget.

In order to ensure that the clobber set is able to entirely account for all of the side effects of executing the byte-string there are restrictions that are placed on the unmatched expression trees which must be fulfilled for the match to be accepted. Namely, that the expression trees must be "safe" to execute regardless of the current execution state. To achieve this the unmatched trees are only allowed to correspond to writes to registers, and are not allowed to include any memory accesses as it is not possible to guarantee which locations in memory will be in the resulting program's address space.

3.4   Code Generation

Generating executables from a semantic blueprint is done by taking a representation of a program composed of a collection of methods, each described as a sequence of statements, and resolving the statements into gadgets and then resolving each gadget into actual bytes.

In order to translate statements into gadgets it is necessary to first assign variables to hardware locations. This requires keeping track of which registers and memory locations are free and which have already been assigned to variables. This paper uses a naive approach and assigns all local variables and method arguments as memory addresses offset from the *EBP* register. There are more efficient variable mapping algorithms [CAC$^+$81, GA96], however adapting these techniques to the nondeterministic nature of this

Statements	Gadgets
move($[ebp+8]$,10)	LoadConst($[ebp+8]$,10)
sub($[ebp+4]$,$[ebp+8]$,$[ebp+16]$)	Minus($[ebp+4]$,$[ebp+8]$,$[ebp+16]$)
mul($[ebp+4]$,$[ebp+4]$,$[ebp+4]$)	Times($[ebp+4]$,$[ebp+4]$,$[ebp+4]$)

Figure 3.3: Example of compiling statements directly to gadgets.

engine is beyond the scope of this paper.

Once a mapping of local variables to hardware locations has been determined it is possible for statements to be translated in to gadgets. Generating the bytecode code for each method can then be done by sequentially retrieving byte-strings from the gadget library for each gadget.

3.4.1   Variable Mapping & Statement to Gadget Translation

Because gadgets can only be parameterized over hardware locations and not local variables, a variable mapping must be generated before statements can be translated into gadgets. Once a mapping exists, it is possible to directly compile statements into gadgets by substituting gadgets for equivalent statements. However while this approach may theoretically work, in practice this is highly inefficient.

Recall that in this implementation, gadgets correspond to short sequences of $x86$ instructions typically on the order of 1-4 instructions long. Most $x86$ instructions only operate on registers. If we were to naively translate statements into a single gadget, the gadget would have to include the instructions that load values from memory into registers and then if applicable store the value back in memory. The odds of a random sequence of instructions cooperating in this manner appearing randomly is very small. Therefore to make search a practical method of code generation, it is necessary to prune search paths that are unlikely to be fruitful.

To increase the likelihood that any given gadget will be present in the library, statement translation explicitly generates gadgets that load variables from memory addresses into temporary registers. An example of a possible translation is given below. Note that each gadget is tagged with the registers that were bound at the moment of translation. This information will be required later when translating gadgets to bytes to ensure that no temporary registers are overridden as a side effect of the gadget.

15

Statements	Gadgets
move([$ebp + 8$],10)	LoadConst(eax,10) [eax] StoreMemReg(ebp,8, eax) [eax]
sub([$ebp + 4$],[$ebp + 8$],[$ebp + 16$])	LoadMemReg(eax, [$ebp + 8$]) [eax] LoadMemReg(ebx, [$ebp + 16$]) [eax, ebx] Minus(ecx, eax, ebx) [eax, ebx, ecx] StoreMemReg(ebp,8,ecx) [ecx]
mul([$ebp + 4$],[$ebp + 4$],[$ebp + 4$])	LoadMemReg(eax, [$ebp + 4$]) [eax] Times(eax, eax, eax) [eax] StoreMemReg(ebp,4,eax) [eax]

Figure 3.4: Example of compiling statements to gadgets with explicit memory accesses. The registers in square brackets correspond to registers that are not allowed to be clobbered while translating that gadget.

3.4.2   Gadget to Byte Translation

Translating bytes to gadgets is relatively straight forward in most cases and can be accomplished by randomly selecting a byte-string from the gadget library for any encountered gadget.

The primary constraint is ensuring that the chosen byte-string does not have any unintended side effects. Recall that effects on hardware that are outside the scope of the core semantic of a given gadget are allowed and that such effects are encapsulated in each gadget's "clobber set." The clobber set contains a set of hardware locations that will be in an undefined state after execution of the gadget.

Therefore to ensure that a gadget does not have unintended effects on the execution state it is sufficient to require that the gadget's clobber set be a subset of the unallocated registers. It is only necessary to consider registers because in the creation of the gadget library memory accesses were disallowed from being present in the clobber set.

In the case that there are no byte-strings available for a given gadget, the algorithm backtracks and tries another translation of the current statement using different temporary registers. If every combination of temporary registers is tried and a translation is still impossible, the algorithm continues backtracking all the way to the top of the search tree to where variables are mapped. If backtracking this far is still insufficient the engine termi-

nates with the conclusion that the gadget library is too small and does not provided adequate coverage.

3.4.3 Edge Cases

The above works well for most types of gadgets but there are several edge cases that require further optimizations to make the engine practical. The first is the assignment of constants to variables and the second is forward jumps.

Constants

The naive approach to gadget assignment would be to directly translate an assignment statement into a corresponding *LoadConst* gadget. However, for such a gadget to be found during the construction of the library it is necessary that a particular 32 bit value be present in the byte-string at the correct offset. This is a very unlikely and it would also be impractical to store each 2^{32} possible *LoadConst* gadget in the library.

A natural solution is to attempt to construct the equivalent of a *LoadConst* gadget out of gadgets that exist in the library. There are some caveats with this approach however, namely that it is reminiscent of the subset sum problem[2] which is NP-complete.

The following is a novel solution whose runtime is independent of the number of bits in the constant being loaded. Loading an n-bit constant is done using two gadgets. The first loads a constant where the most significant bits match the lower n bits of the desired constant. The second gadget performs an arithmetic right shift of $32 - n$ on this value to generate the desired constant.

The $x86$ shift instruction takes as an operand a 8 bit number for the shift amount. This approach significantly improves the odds of successfully compiling an assignment statement to bytes because it relies on the odds of an 8 bit and an n bit number being generated independently rather than a specific 32 bit integer.

Jumping Forward

The second edge case that requires special handling is forward jumps. The problem arises because jump statements are written with relative offsets

---

[2]The subset problem aims to find a subset of a set of integer whose sum is equal to a target integer

jump(2,*always*) ⌐————————┐
sub([*ebp* + 4],[*ebp* + 4],1)   │
add([*ebp* + 4],[*ebp* + 4],1) ◄————┘

Figure 3.5: Example of a sequence of statements that contain a forward jump. Note that this sequence is not translatable using a naive single pass algorithm.

in terms of statements while gadgets have offsets expressed in bytes. This means that in order to resolve jump statements to gadgets it is necessary to know the byte offset of the statement that the jump is trying to pass control to.

This becomes a problem when dealing with code that attempts to jump forward, for example see the example code in figure 3.5. In order to know the byte offset of the target statement, the current statement must be compiled first which results in a circular dependency.

The proposed method of resolving the circular dependency is to perform compilation in two phase. During the first phase non-jump statements are translated into bytes as described above. However, when a jump gadget is encountered, the size of its corresponding byte-string is guessed and a placeholder of the appropriate size that notes the target offset is generated. After which, following statements are compiled as above.

After each statement has been translated into either bytes or place holders, the second phase translates the place holders into appropriate byte-strings. If there are no corresponding bytes for the gadget in the library, then the search backtracks potentially up to the translation of the problematic jump statement in the first phase and guesses a different size. The backtracking process also considers different statement to gadget and gadget to byte translations on the way back to retranslating the problematic jump. This allows for the target byte offset to change without having to throw away all of the translations after the problematic jump.

3.5   Scaffolding

The code generation algorithm given above handles turning statements into bytes, but it does not handle linking or any of the other requirements of generating actual executables.

As this is a proof of concept system and not intended for production use, this implementation uses handwritten assembly code and tools from the GNU Compiler Collection to create object files for each method, and link them to form actual executables [RMS12].

This approach is ideal as it allows for easy inspection of the generated code in the object files, along with providing a mechanism for linking generated methods against C code to facilitate unit testing.

Chapter 4

Evaluation

There are two general criteria that metamorphic engines are generally evaluated on. The first is that the programs produced by the engine perform the same high level function as the original seed program. The second, is that all of the programs produced by the metamorphic engine exhibit a high degree of dissimilarity among each other at the byte level. The underlying goal here is the evasion of classical byte signature based malware detection algorithms.

In addition to evading traditional detection techniques, the second goal of this engine is to evade newer detection algorithms which rely on comparing the opcode histograms of "normal" programs with the histogram of the program being classified [RMI12].

4.1   Functional Equivalence

For a metamorphic engine to be useful it must claim to preserve the semantics of the seed program. In this implementation, each method of the seed program is encoded as a sequence of abstracted assembly-like statements. Each of these statements has a clear and well defined mapping to sequences of gadgets.

During code generation, each method's statements are compiled down to a sequence of gadgets. In the process, local variables are mapped to memory locations and registers. The resulting byte-string for the method is generated by translating each gadget into bytes. This is done choosing an appropriate instance of the gadget in the library.

To show that the code generation maintains the original semantics of the seed program, it is sufficient to show the following:

1. All of the byte-strings that a given gadget maps to have the same effect on the execution state.

2. The sequences of gadgets corresponding to each statement preserve the semantics of that statement.

3. Choice of registers and memory locations do not have any outwardly observable side effects.

4.1.1 Functional Equivalence of Byte-Strings

The guarantee of functional equivalence of byte-strings for a particular gadget in the library is a direct result of the conservative design of both the abstract evaluator and the gadget discovery algorithm.

The abstract evaluator only successfully evaluates $x86$ instructions that have a deterministic effect on the execution state. Meaning that the resulting state of every hardware location[1] can be described algebraically, without the use of conditional operators, in terms of the original values of the hardware locations prior to execution. Examples of states that cannot be represented in this model include the effect of instructions following a conditional branch, and any sequence of instructions that perform multiple indirect writes to memory.

Confining the space of allowed execution states in this manner allows the resulting execution state to be represented as a collection of algebraic expression trees, with exactly one tree for each modified location. Matching bytes to gadgets is done by attempting to match one of the resulting expression trees to the general trees defined for each class of gadget. Simply matching a byte-string to a gadget does not account for the effects of the expression trees not included in the match. To remedy this, the locations whose values correspond to each unmatched tree is recored in the "clobber set."

The intent is that the only side effects of a gadget is the overwriting of the locations in its clobber set. In order to guarantee this, restrictions are placed on the expression trees of the locations placed in the clobber set. Because the expression trees only include algebraic operators, there is no possibility that evaluating a tree will result in a write to another location. Therefore the only potential side effects are indirect ones caused by exceptions like division by zero or accessing unmapped memory. To prevent these indirect

---

[1]Hardware locations are defined as registers, flags and memory addresses.

side effects, locations in the clobber set are not allowed to have expressions trees that contain any memory accesses or the division operator.

4.1.2 Gadget/Statement Equivalence & Irrelevance of Specific Registers

Every local variable defined by a method is mapped to memory locations at positive and negative offsets from the EBP register corresponding to method input and local variables respectively. The mapping of local variables to memory locations is static and does not change over the course of the program.

Each statement has a core semantic that maps to specific gadgets. In the case of this implementation each statement corresponds to a single core gadget operation[2]. Given this it is possible to define the operational semantics of a statement as the gadget that implements it.

To fully execute a statement the values or constants that constitute the statement's parameters must be loaded into temporary registers so that they may be used with the corresponding gadget[3]. This is done through the *LoadMemReg* and *LoadConst* gadgets. After the necessary values have been loaded into temporary registers, another temporary register is allocated for the result of the operation if applicable and the core gadget is performed on the temporary registers. If the result is intended to be saved the temporary register corresponding to the results is placed back into the abstract location through a *StoreMemReg* gadget.

Temporary registers are allocated for the duration of the statement's execution and are temporarily removed from the pool of unallocated general registers. When translating gadgets into concrete bytes, only those instances whose clobber list is a subset of the current unallocated register pool are considered valid translations. Because the side-effects of gadgets are limited to altering values in unallocated general purpose registers, the observable[4] effect on the execution state can be described by the core semantics and the clobber sets of the sequence of gadgets.

---

[2]For instance the *add* statement's core semantic corresponds to the *Plus* gadget.

[3]Unless required by the semantics of the gadget, each gadget only operates on register locations. The only gadgets in this implementation that operate on values other than registers are *LoadMemReg*, *StoreMemReg*, and *LoadConst*

[4]An observable effect being one that alters the value of an allocated temporary register, the value of a memory location, or altering the instruction pointer

4.2 Test Suite

The next two sections describe experiments intended to measure the performance of the metamorphic engine against detection techniques resembling real world algorithms.

A gadget library was pregenerated using each of the described byte sources. The method presented in [MH12] is referred to as "Frankenstein", the naive approach of sampling bytes from a uniform distribution is referred to as "Uniform Random", and the proposed method of sampling bytes from the observed byte frequency distribution of windows programs is referred to as "Observed Random."

Three simple test programs are provided as seeds for the engine. Descriptions of the sample programs are given below, and the source code for each is given in Appendix A. A collection of 100 programs was then generated for each combination of test program and gadget library.

4.2.1 Sample Programs

Max

The max program is a very simple method that takes as input two signed integers and returns the greater of the two. The purpose of the max program is to test value returning and forward conditional jumps (if statements).

Factorial

The factorial program takes as input a signed integer and returns a signed integer corresponding to the factorial of the input. The calculation is done inside a loop. The factorial program is intended to test various arithmetic gadgets along with backward conditional jumps.

Xor

The xor program defines a method that receives as input a pointer to an array of signed integers, the length of the array, and an integer valued key. The method iterates through the array xoring each value with the supplied key. Along with performing a task commonly seen in polymorphic malware, this method also demonstrates the ability to dereference memory locations, and the ability to control flow through conditional jumps.
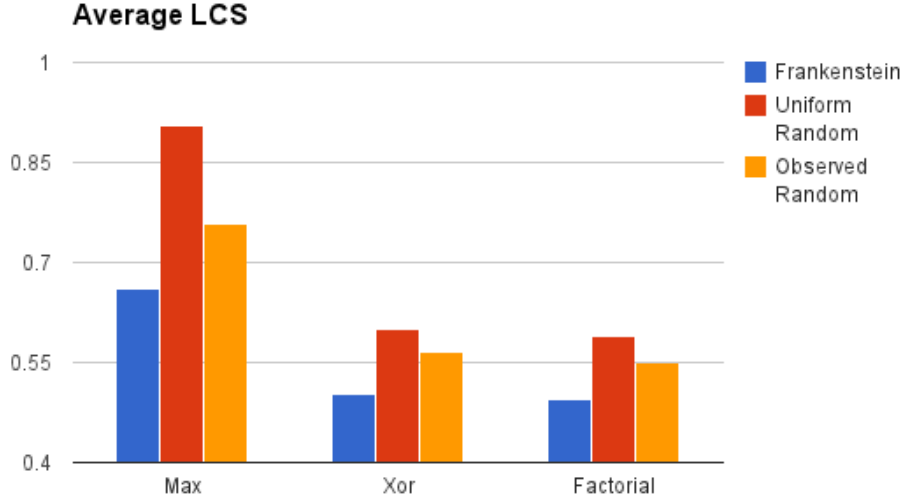
Figure 4.1: Similarity scores for each combination of test program and gadget library. Lower scores are more desirable, a score of 1 indicates that the programs are identical, while 0 indicates that there is no overlap.

4.3 Metamorphic Variety

The following experiment is intended to evaluate the performance of the engine in a scenario closely resembling traditional byte signature based detection techniques. In order to evade these techniques, the generated programs must exhibit a high degree of variation in their opcodes.

Accordingly, the evaluation metric for this experiment is modelled after the system presented in [VLG+09] which uses the longest common subsequence (LCS) as its basis. The metric described in [VLG+09] is concerned with the sequences of opcodes in the basic blocks of the control flow graph and yields a value from 0 to 1 corresponding to the percent of overlap in corresponding basic blocks in the control flow graph.

Given that our metamorphic engine does not support basic block reordering as one of it's obfuscation techniques it follows that corresponding basic blocks in each variant will appear in the same relative ordering in the generated code. Therefore the metric has been adapted to take advantage of this simplification.

The similarity score between two methods is defined to be the percent of the total opcodes among the two methods that are present in their longest

common subsequence. The formula is defined as follows, where A and B are the lists of opcodes making up the two methods, $LCS(A, B)$ is the length of their longest common subsequence, and $|A|$ and $|B|$ are the length of A and B:

$$Sim(A, B) = \frac{LCS(A, B)}{|A| + |B| - LCS(A, B)}$$

The score for each combination of gadget library and test program is calculated by averaging each pairwise similarity score among the 100 generated samples. The resulting scores are shown in figure 4.1. The proposed method of sampling from an observed byte distribution clearly out performs the naive approach, but doesn't perform as well as the approach given in the "Frankenstein" paper [MH12].

To understand why the proposed approach doesn't perform as well as the "Frankenstein" approach it is informative to consider why it out performs the naive approach.

In order for a given sequence of instructions to be placed in the gadget library they must be interpretable by the abstract evaluator, which is only capable of evaluating some of the most common $x86$ instructions. Under the naive scheme, bytes are sampled from a uniform distribution. Considering that common instructions are typically shorter than rare instructions it follows that common instructions are only slightly more likely to be generated than rare instructions in this scenario. As a result, the length of the gadgets that are placed in the library are likely to be relatively short, consisting of only one or two instructions on average.

By biasing the probability of specific bytes from being generated, the proposed approach of sampling bytes effectively increases the likelihood of common, and thus interpretable, instructions being generated. Therefore the likelihood of generating a sequence of interpretable instructions is greater than the naive approach, which in turn leads to longer gadgets in the library. The reason that longer gadgets lead to a better similarity score is that there are more opportunities for junk instructions to be placed in the resulting executable thus increasing the denominator of the similarity metric and improving the score.

To understand why the "Frankenstein" approach outperforms the proposed approach, consider that the proposed approach makes use of a unigram byte model to model the distribution of opcodes in normal programs. As such, it does not fully capture the dependencies between bytes that occur in instructions that span more than one byte. Therefore it is logical that the proposed method is not as likely to generate common instructions as directly

26

Opcode	Freq	Opcode	Freq	Opcode	Freq	Opcode	Freq
add	0.2184	or	0.0256	outsd	0.0137	popad	0.0090
mov	0.1188	inc	0.0205	dec	0.0137	jae	0.0089
push	0.1060	lea	0.0204	jb	0.0136	ret	0.0082
int3	0.0447	sub	0.0202	jmp	0.0129	sbb	0.0073
call	0.0343	and	0.0176	jnz	0.0121	insb	0.0068
pop	0.0309	xor	0.0168	imul	0.0110	invalid	0.0062
jz	0.0286	adc	0.0159	outsb	0.0097	arpl	0.0055
cmp	0.0260	test	0.0145	nop	0.0093	jo	0.0052

Table 4.1: The distribution of opcodes in normal Windows programs.

sampling from the actual binaries as the "Frankenstein" system does. This suggests that a useful avenue of future research would be investigating the efficacy of more complicated models, such as more general n-gram models.

4.4 Appearing Normal

The following experiment is intended to measure the effectiveness of this engine against detection techniques that rely on comparing the opcode histograms of suspected malware against opcode histograms of "normal" Windows programs.

Due to the potential ambiguity of what constitutes a normal Windows program, we chose a very conservative definition and built the opcode frequency distribution from executables contained in a vanilla install of the Windows operating system. In particular, the distribution was generated from each .dll and .exe in the subdirectories of the *C:/Program Files/* folder found in a fresh install of the 32-bit version of Microsoft Windows 7 Ultimate. The resulting distribution is built from a total of 248 files and the most frequent 32 opcodes are given in table 4.1.

The notation of classifying families of programs and files through frequency distributions has a strong precedent in literature [TS12, LS11, FJ07, LWSH05]. The basis for our evaluation is the classification system described in [TS12] because it presents a simple and concise metric that produces a real-valued score representing how likely a sample is to be malware as a opposed to a binary classification. The system uses Pearson's chi-squared test for goodness of fit to measure the likelihood that a file under consideration belongs to a particular family of programs.

Pearson's chi-squared test for goodness of fit tests the hypothesis that

	Xor	Factorial	Max
Uniform Random	100%	100%	100%
Sampled Random	100%	100%	100%
File Scanning	97%	52%	100%

Table 4.2: The percent of generated examples from each combination that were classified as benign.

an observed frequency distribution matches a given theoretical distribution. The theoretical distribution in the case of malware classification is the frequency distribution of opcodes in a specific family of executables. In [TS12] families of executables refers to various strains of metamorphic malware, in our case the family of programs we care about is the family of normal Windows executables.

The chi-squared test statistic, X^2, is defined by the following equation, where O_i is the observed frequency of opcode i in the sample, E_i is the expected frequency, and n is the total number of unique categories.

$$X^2 = \sum_{i=0}^{n} \frac{(O_i - E_i)^2}{E_i}$$

After the test statistic X^2 is calculated it is compared to the p-value of the chi-squared distribution parameterized with the appropriate degrees of freedom and false-positive error rate. If the calculated X^2 is less than the p-value then the hypothesis is accepted.

In general the degrees of freedom is equivalent to one minus the number of categories, in this case it is 559 which corresponds to one less than the number of unique opcodes in the $x86$ instruction space recognized by our disassembler. The false positive rate generally accepted as statistically significant and suggested by [TS12] is 0.05.

The corresponding p-value with this parameterization is 615.11. This means that any X^2 values less than 615.11 indicates that the sample program belongs to the family of normal Windows programs. Figure 4.2 presents the average X^2 values for each combination of library and sample program. Table 4.2 gives the percent of generated examples that are classified as benign by comparison to the p-value.

The proposed system outperforms the naive approach with both producing executables that consistently have chi-squared p-values below the threshold of statistical significance. This means that for each of the sample programs tested, the engine was able to produce executables that are
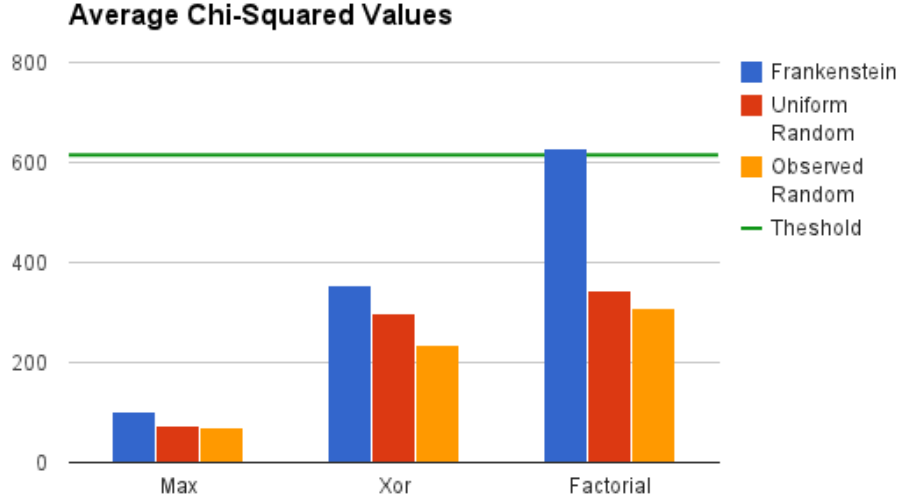
Figure 4.2: The average X^2 values of all generated instances for each combination of sample program and library. Lower numbers indicate a stronger similarity with the observed windows opcode distribution

classifiably benign under this metric.

The "Frankenstein" system surprisingly performs worst under this metric, with only roughly half of the executables generated for the factorial sample program classified as benign.

Examination of the generated executables suggests that the high density of cooperating common instructions in normal executables that contributed to the "Frankenstein" system's success in the previous test causes it to fail in this scenario. Note that in table 4.1, the table of most frequent opcodes, the first 3 instructions occur with significantly more frequency than the following most common instructions. By promoting more junk instructions to be included in gadgets, the "Frankenstein" system increases the likelihood of less common instructions appearing in gadgets that correspond to more likely instructions. This could lead to inflation in the counts for the less frequent instruction which would increase the chi-squared score signalling that the executable is less likely to be benign.

Chapter 5

Conclusion

Commercial antivirus software has historically relied on static techniques to detect instances of malware in the wild. Typical implementations rely on byte signatures to detect malicious programs. An active area of research is creating robust detection techniques capable of coping with metamorphic malware which is able to obfuscate its byte signature. An emerging static technique that has shown academic success uses knowledge of the structure of benign programs to classify malware. A prominent example of this is opcode frequency analysis [TS12].

The metamorphic system presented in this paper is shown to be capable of producing programs that exhibit a high degree of variation in their byte signatures while at the same time maintaining an opcode frequency distribution similar to those found in benign Windows programs. It performs well compared to state of the art gadget-based metamorphic engines in terms of byte signature variance and out performs the state of the art in respect to conforming to benign opcode distributions.

Further, this paper has also provided empirical support for the claim made in [MH12] that suggests that when nondeterministically stitching together a program out of gadgets, the resulting program's overall opcode distribution can be controlled by altering the distribution of bytes that the gadgets are sampled from.

By achieving high levels byte signature variance and opcode frequency conformity the described engine demonstrates that it is currently feasible for malware authors to evade both the current algorithms and emerging static detection techniques.

5.1 Future Work

The system described in this paper is a proof of concept and as such contains many areas that would benefit from further investigation.

5.1.1 Improvements In Modelling Benign Programs

The byte source proposed by this paper uses a unigram byte model to describe byte distributions and is a trivial improvement over the naive approach of sampling gadgets from a uniform distribution. It would be beneficial to examine the performance of the engine under more complicated models such as general n-gram models, and models based on opcodes frequencies rather than byte frequencies.

There is also the potential of incorporating the model into the code generation algorithm by choosing gadgets based on their probability instead of the current approach of uniformly randomly choosing them.

5.1.2 Code Generation

There are several pruning techniques used in the engine that limit the variation in the programs that can be generated. Some of these can be accounted for by incorporating existing techniques, such as intelligent register allocation [CAC+81].

Others require more specialized algorithms. This paper presents a novel approach for generating constants not found in the gadget library that achieves reasonable performance at the cost of variability. A more general approach that does not sacrifice variability would be desirable in a mature system.

5.1.3 Abstract Evaluator

Building a large library of gadgets that covers a wide range of operations is necessary to create complex programs. The current implementation handles only 19 of the over 500 unique $x86$ opcodes. It would be interesting to attempt to correlate the performance of the system to the number of supported opcodes.

Similarly, in order to take full advantage of the supported opcodes, the evaluator would need to handle semantically equivalent by syntacticly distinct expressions, e.g. $(1 + 1) * eax = 2 * eax = (eax + eax)$.

Appendix A

Test Suite

A.1   Max

```
import Igor

main = defineMethod "max" $ do
    [a, b]      <-  makeInputs 2
    [aLessB]    <-  makeLabels 1

    jump    aLessB  (a -<- b)
    ret     a
    label   aLessB
    ret     b
```

A.2 Xor

```
import Igor

main = defineMethod "xorEnc" $ do
    [array, length, key]        <- makeInputs 3
    [tmp, current, arrayEnd]    <- makeLocals 3
    [loop]                      <- makeLabels 1

    mul     arrayEnd            length               (4 :: Integer)
    add     arrayEnd            arrayEnd             array
    move    current            array

    label   loop
    xor     (W,current,0)       key                  (R,current,0)
    add     current            current              (4 :: Integer)
    jump    loop                ( current -!=- arrayEnd )
```

A.3 Factorial

```
import Igor

main = defineMethod "factorial" $ do
    [n]                     <- makeInputs 1
    [tmp, total]            <- makeLocals 2
    [begin]                 <- makeLabels 1

    move    total   tmp

    move    total   (1 :: Integer)
    move    tmp     n

    label   begin
    mul     total   total   tmp
    sub     tmp     tmp     (1 :: Integer)
    jump    begin   ((1 :: Integer) -<- tmp)

    ret     total
```

Bibliography

[Ale96]   Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[Bea07]   Philippe Beaucamps. Advanced polymorphic techniques. In *World Academy of Science, Engineering And Technology*, volume 25, November 2007.

[CAC$^+$81]   Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[FJ07]   Eric Filiol and Sébastien Josse. A statistical model for undecidable viral detection. *Journal in Computer Virology*, 3(2):65–74, 2007.

[GA96]   Lal George and Andrew W Appel. Iterated register coalescing. *ACM Transactions on Pro (TOPLAS)*, 18(3):300–324, 1996.

[KPY07]   Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pages 46–53, New York, NY, USA, 2007. ACM.

[LS11]   Da Lin and Mark Stamp. Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, 7:201–214, 2011.

[LWSH05]   Wei-Jen Li, Ke Wang, Salvatore J Stolfo, and Benjamin Herzog. Fileprints: Identifying file types by n-gram analysis. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 64–71. IEEE, 2005.

[MCJ07]   Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware.

In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441. IEEE, 2007.

[MDL+12]   Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8:1–13, 2012.

[MH12]   V. Mohan and K.W. Hamlen. Frankenstein: Stitching malware from benign binaries. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 8–8. USENIX Association, 2012.

[RHD+06]   P. Royal, M. Halpin, D. Dagon, R. 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, dec. 2006.

[RMI12]   Babak Bashari Rad, Maslin Masrom, and Suahimi Ibrahim. Opcodes histogram for classifying metamorphic portable executables malware. In *e-Learning and e-Technologies in Education (ICEEE), 2012 International Conference on*, pages 209 –213, sept. 2012.

[RMS12]   et al. Richard M. Stallman. Using the gnu compiler collection, 2012.

[Sha07]   H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[Spi03]   D. Spinellis. Reliable identification of bounded-length viruses is np-complete. *Information Theory, IEEE Transactions on*, 49(1):280 – 284, jan 2003.

[SW06]   Mark Stamp and Wing Wong. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, December 2006.

[SYR+10]   Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware detection based

on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1020–1025, New York, NY, USA, 2010. ACM.

[TS12]   Annie Toderici and Mark Stamp. Chi-squared distance and metamorphic virus detection. *Journal in Computer Virology*, pages 1–14, 2012.

[VLG⁺09]   P Vinod, Vijay Laxmi, Manoj Singh Gaur, GVSS Kumar, and Yadvendra S Chundawat. Static cfg analyzer for metamorphic malware code. In *Proceedings of the 2nd international conference on Security of information and networks*, pages 225–228. ACM, 2009.

[WGXW10]   Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. Mimimorphism: a new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 536–546, New York, NY, USA, 2010. ACM.

[WHKD01]   Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of software-based survivability mechanisms. In *DSN*, pages 193–202. IEEE Computer Society, 2001.