Behavior-Based Malicious Code Understanding and Detection

Proposal for the Ph.D. Preliminary Examination

Mihai Christodorescu
mihai@cs.wisc.edu

October 24, 2005

1 Introduction

Malicious-code detection identifies programs that have the potential to harm the machine on which they execute or the network over which they communicate. A *misuse malicious-code detector* (or, alternately, a *signature-based malicious-code detector*) uses a list of malicious signatures (traditionally known as a *signature database* [66]). For example, if part of a program matches a signature in the database, the program is labeled as malicious [23]. In this proposal I focus only on host-based malicious-code detectors. Henceforth all references to detection systems implicitly mean host-based malicious-code detection systems. The low false positive rate afforded by these systems and the ease of use has lead to widespread deployment and to a healthy commercial sector providing malicious-code detection tools.

Malware writers continuously test the limitations of malware detectors in an attempt to discover ways to evade detection. This leads to an ongoing game of one-upmanship [67], where malware writers find new ways to create undetected malware, and where anti-virus researchers design new signature-based techniques for detecting the malware. This coevolution is a result of the theoretical undecidability of malicious-code detection [1, 2]. This means that, in the currently accepted model of computation, no ideal malicious-code detector exists. The only achievable goal in this scenario is to design better detection techniques that jump ahead of attack techniques and make the malware writer's task harder.

1.1 The Malicious Code Threat Model

Attackers have resorted to two main approaches for creating new malicious programs: program obfuscation and program evolution. Program obfuscation transforms a program by inserting new code or modifying the existing code to make understanding and identification harder, at the same time preserving the malicious behavior (the effect on the machine and the network). Obfuscation transformations easily defeat detection mechanisms. If a malicious signature describes a certain sequence of instructions [23], then those instructions can be reordered or replaced with equivalent instructions [72, 73]. Such obfuscations are especially applicable on CISC architectures, such as the Intel IA-32 [97], where the instruction set is rich and many instructions have overlapping semantics. If a malicious signature describes a certain distribution of instructions in the program, insertion of junk code [58, 70] [73] that acts as a nop so as not to modify the program behavior can defeat frequency-based signatures. If a malicious signature identifies some of the read-only data of a program, packing or encryption with varying keys [53, 68] can effectively hide the relevant data.

The problem is illustrated in Listing 1 with the help of a code fragment in Intel IA-32 assembly from the mass-mailing e-mail worm Netsky.C [64]. The code retrieves the locations of the Microsoft Windows installation directory and then copies itself there under the name `winlogon.exe`. This code fragment corresponds (roughly) to the C source code snippet from Figure 2. Malware writers have at their disposal a large number of obfuscation tools that operate at the source-code level or at the binary level. These tools can easily create hundreds of variants from a single program and some of these tools are packaged as obfuscation libraries that, when integrated into a self-propagating malicious program, can generate different variants at each propagation step (e.g., Listings 2, 3, and 4 are all variants of Listing 1. *The first requirement of a robust malicious-code detection technique is to handle obfuscation transformations.*

```
        lea    eax, [ebp+Data]
        push   esi
        push   eax
        call   ds:GetWindowsDirectoryA
        lea    eax, [ebp+Data]
        push   eax
        call   _strlen
        cmp    [ebp+eax+var_425], 5Ch
        pop    ecx
        jz     short loc_402EFA
        lea    eax, [ebp+Data]
        push   offset asc_40ADE4
        push   eax
        call   _strcat
        pop    ecx
        pop    ecx
loc_402EFA:
        lea    eax, [ebp+Data]
        push   offset aWinlogon_exe
        push   eax
        call   _strcat
        pop    ecx
        lea    eax, [ebp+Data]
        pop    ecx
        push   ebx
        push   eax
        lea    eax, [ebp+ExistingFileName]
        push   eax
        call   ds:CopyFileA
```

Listing 1: Example of a malware code fragment.

Packing / Encryption

```
        mov    edi, offset decr_area
        mov    al, 99
loop_start:           ; decryption loop
        cld
        xor    byte ptr [edi], 1
        scasb
        loop   loop_start
        jmp    short decr_area
        ...
        ...
decr_area:            ; encrypted data
        db 8c 84 d9 ff
        db fe fe 57 51
        db fe 14 75 61
        db 41 01 8c 84
        db d9 ff fe fe
        db 51 e9 79 06
        db 01 01 81 bd
        db 04 d6 ff fe
        db fe 5d 58 75
        db 12 8c 84 d9
        db ff fe fe 69
        db 81 8c 41 01
        db 51 e9 7d 07
        db 01 01 58 58
        db 8c 84 d9 ff
        db fe fe 69 79
        db 8f 41 01 51
        ...
```

Listing 2: Result of encryption applied to malware code fragment from Listing 1.

Junk insertion

```
        mov    edi, offset decr_area
        mov    al, 99
loop_start:
        cld
        xor    byte ptr [edi], 1
        scasb
        push   edi
        add    [ebp], 1
        pop    edi
        dec    edi
        loop   loop_start
        jmp    short decr_area
        ...
decr_area:
        db 8c 84 d9 ff
        db fe fe 57 51
        db fe 14 75 61
        db 41 01 8c 84
        db d9 ff fe fe
        db 51 e9 79 06
        db 01 01 81 bd
        db 04 d6 ff fe
        db fe 5d 58 75
        db 12 8c 84 d9
        db ff fe fe 69
        db 81 8c 41 01
        db 51 e9 7d 07
        db 01 01 58 58
        ...
```

Listing 3: Junk code inserted into Listing 2.

Reordering

```
        mov    edi, offset decr_area
        jmp    insn2
insn9:
        loop   loop_start
        jmp    insn10
insn3:
        xor    byte ptr [edi], 1
        jmp    insn4
insn10:
        jmp    short decr_area
insn2:
        mov    al, 99
loop_start:
        cld
        jmp    insn3
insn6:
        add    [ebp], 1
        jmp    insn7
insn4:
        scasb
        push   edi
        jmp    insn6
insn8:
        dec    edi
        jmp    insn9
insn7:
        pop    edi
        jmp    insn8
        ...
```

Listing 4: One possible reordering of the code in Listing 3.

Figure 1: Example of obfuscation transformations. The original code from Listing 1 is first packed (Listing 2), then junk code is inserted in the unpacking routine (Listing 3). Finally the code is reordered to obtain Listing 4.

```
char data[len + 1];
GetWindowsDirectoryA( data, len );
if( data[strlen(data)-1] != '\\' )
{
   strcat( data, "\\" );
}
strcat( data, "winlogon.exe" );
CopyFileA( current_file_name, data, FALSE );
```

Figure 2: C source code corresponding to the assembly code in Listing 1.

Program evolution poses a significant problem to malicious-code detection. As new network protocols appear and become widespread, as operating system features become richer, as vulnerabilities are discovered, malware writers extend their programs to take advantage of the new infection vectors, the new, richer application programming interfaces, and the new network communication primitives. Program evolution is further enabled by two trends in the malware writing community. First, malware development applies code sharing and reuse to a large extent [56]. Attack vectors and code obfuscation techniques are readily shared among malware writers. This leads to modular designs of malware [51, 54, 55, 59–61], which in turn enable quick adaptation to new vulnerabilities. Second, much of the shared code appears online in source code form, allowing for high level manipulation and customization. Such "open source" approaches produce thousands of variants, each unique in its strengths and weaknesses and its malicious behavior. For example, the Agobot backdoor has hundreds of documented variants [63]. Thus, *the second requirement of a robust malicious detection technique is to handle program evolution.*

Recent results indicate that current commercial malware detectors do not meet these two crucial requirements. In previous work, I investigated the limitations of widely-used virus scanners in the presence of obfuscation [52]. My conclusion (based on the results in Figure 3) was that the resilience of these virus scanners to various obfuscations was very poor. A determined attacker can leverage these weaknesses to extract information about the signatures used by a virus scanner, thus simplifying the evasion task. The results of my obfuscation-based testing methodology are confirmed by empirical observations of the anti-malware industry. To detect seven different variants of the Netsky e-mail virus, the McAfee VirusScan detector requires six different signature instances [62], although all of the variants have similar characteristics: they spread using e-mail attachments, they embed their own SMTP engine for sending e-mail, and they harvest e-mail addresses from the infected machine. The prolific e-mail virus Beagle gained twelve new variants in one day (September 20, 2005), forcing F-Secure to release twelve corresponding updates to their signature database [57]. These events illustrate the power and the flexibility malware writers have in the process of generating new variants of existing malware, and highlight the limitations of current malware detection techniques.

Given the inability of malware detectors to cope with variants, the threat model of malicious code can be summed up as follows:

Threat: New malware instances evade detection through obfuscation and evolution.

To neutralize this threat, we need to detect all variants of a known malicious program. This formulation is imprecise in the use of the term "variant." One way to define a variant is to consider it an obfuscated version of the known program P *for a given set of obfuscations*. This definition enables the same fallacy as the one that underlies the virus–anti-virus coevolution. If a malware detector is restricted by design to a fixed set of obfuscation transformations, it is prone to evasion attacks that use any obfuscation outside the fixed set of obfuscations.

I present a definition of variant that does not have these limitations. Intuitively, a program P' is a variant of a program P is they both exhibit the *same observable behavior*. Observable behavior can be expressed as a sequence of relevant system calls (I formally define observable behavior in Section 7). Then, a program is a variant of another program if both programs exhibit the same sequence of relevant system calls during their execution. Since this definition does not refer to the actual syntactic form of either program (with the exception of the system calls which, as I argue later in this section, cannot be obfuscated), it describes a large number of variants of the same program, independent of the type or level of obfuscation and independent of the evolutionary path of each variant.

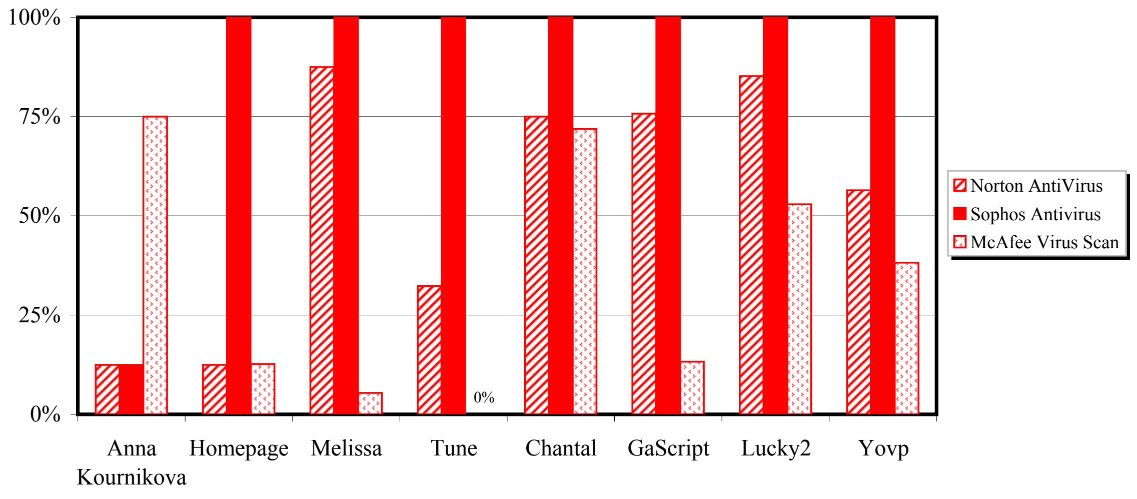


Figure 3: False negative rate for individual viruses, averaged over a set of common obfuscation transformations.

1.2 An Approach to Defeating Malicious Code

The end goal of any security system is to protect the data hosted on the system and to provide the level of service desired by the system's users. It does not matter how a malware detection system operates (statically or dynamically, host-based or network-based), only whether it stops the mass-mailing worm (i.e., the malicious behavior) from affecting the machine or the network it is part of. The focus must be on behavior, not on the particular form it takes (binary programs for Microsoft Windows, Java applets, or VBScript/Javascript web scripts). The core of my proposal is a language for specifying malicious behavior — this language has to be flexible enough to capture all undesirable behavior with the least number of false positives.

Any new program entering a system should be checked against a set of malicious behavior specifications. This check can be done statically. In the cases where static analysis cannot provide an assured answer (either "Yes, this program is definitely malicious" or "No, this program is definitely not malicious"), dynamic monitoring and analysis should be used to check for malicious behaviors. In both cases, the same set of malicious behavior specifications determines the boundaries of allowable behavior on a particular system.

The goal of my proposal is to research, design, and develop a system that counters the evolving threat of obfuscated malware.

Solution: Develop a behavior-based malware detector.

The hypothesis underlying the concept of a behavior-based malware detector is that there are fundamental malicious behaviors that do not change across malware instances. For example, most malware replicates through e-mail, peer-to-peer networks, or vulnerability exploitation. Many malicious programs carry payloads that open backdoors into victim systems, offering command and control facilities to the remote attacker. Observable behavior is by definition preserved in the case of obfuscation transformations, and multiple malware instances from the same family share a core set of malicious functions [56]. Thus malware detection needs to be behavior-based.

My proposal for a new malicious-code detection technique is based on three key observations. Each of these addresses part of the threat model, and together they form the building blocks of the behavior-based malware detector.

- **System calls cannot be obfuscated.** Malicious programs are not general Turing machines functioning in a vacuum. They crucially depend on their environment to "survive" and "replicate". Malware thrives on interconnected networks of common, widely deployed operating systems that provide networking functionality, file system operations, and interactive user interfaces. For example, to access the network on a UNIX-style operating system, a program eventually has to create, bind, and connect a socket. These operations are defined as operating system calls that exist at a well-defined, enforceable boundary between the application and the operating system kernel. Effectively, system calls represent an enforceable interface between malware and the trusted computing base (the operating system kernel).

- **Semantic information is necessary in capturing the malicious behavior.** Using just system calls as part of the malware detection technique is insufficient. System-call sequences do provide a higher-level abstraction in which to operate, when compared to sequences of instructions. This abstraction also has the side-effect of increasing the false positive rate of the detector. Many benign programs would be incorrectly flagged as malicious when looking for an e-mail virus, since they all use the `socket–conect–write` sequence of system calls (or the Microsoft Windows equivalents). Additional information is needed on how the arguments and return values of these system calls relate to each other.

  One option for capturing the relations between the arguments of system calls of interest is to express them as a sequence of program instructions from a malicious sample. Such syntactic signatures are limited to the form and structure of the binary encoding of those instructions, and to a malware writer's particular programming style. A better approach is to consider arguments to system calls as program state variables and to capture the relations betweens these variables as state transformers. Semantic descriptions of syntactic constructs (i.e., of program instructions) allow the signature to be expressed independently of any particular syntactic form.

- **Static and dynamic analyses can be combined to eliminate false positives.** The malware detection technique could be designed to use various decision procedures that build on static analysis results. The static analysis identifies, for example, program paths between consecutive system calls and provides information about the possible values of the arguments to system calls. This static analysis is conservative by design. When the information is ambiguous or imprecise, the static analysis produces a set of facts that overapproximate the behavior of the code sequence under analysis. From a security perspective, this is a safe approach in that all malicious programs with a particular observable behavior are detected. Although complete, a malware detector built on these techniques would suffer from false positives. For example, programs written in C++ make extensive use of indirection for runtime resolution of method calls. Lacking a precise pointer analysis (an undecidable problem [5, 8]), the static analysis could "discover" some invalid paths and raise alerts if they match the malicious behavior. Consider a more extreme example of a Perl interpreter that processes the input in a loop and generates corresponding system calls. A safe static analysis would raise alarms since any combination of system calls with any combination of data flows between them is possible. Nonetheless, there is a large set of Perl programs that are benign, so the Perl interpreter itself is not malicious. The solution is to use dynamic information to improve the detection decision wherever the static analysis loses precision through overapproximation.

  A dynamic analysis technique collects data during the execution of the program and derives information useful in the malware detection process. Furthermore, a dynamic analysis holds the promise of sound *and* complete information about a particular program execution, albeit with a (possibly significant) performance overhead. In contrast, the static analysis discovers information that describes all program executions, with some imprecision, but without impacting runtime performance. Combining static and dynamic analyses balances out the imprecision of static analysis and the cost of dynamic analysis. This hybrid analysis is the building block for a malware detector that can certify quickly and precisely that *during the current execution* a program has not exhibited any malicious behavior.

These key observations materialize into the core components of the behavior-based malicious code detection system I propose. In Figure 4, the semantic query engine is the base layer of the system. It provides semantic information about the instructions for a specific architecture and the system calls for a particular operating system. In my research I have focused on the Microsoft Windows operating system running on the Intel IA-32 (x86) processor architecture because together they are the dominant and the most attacked platform in the computing world. Using semantic information expressed as state transformers, the static analyses derive facts about the program (e.g., points-to facts, live ranges, paths). The dynamic analyses use a sandboxed runtime environment to monitor program execution and build traces for further analysis. Decision procedures provide a convenient mechanism to extract information about the program by answering queries about the facts derived by the static and dynamic analyses. For example, a decision procedure can confirm whether the value of a particular state variable created by a system call at program point A flows unmodified into another state variable used by a system call at program point B.

This semantic query engine is the building block for three components: a malware normalizer, a malware detector, and a generator of malicious-behavior specifications. The malware normalizer undoes obfuscations that could hinder the malware detection process. These obfuscations are identified with the help of the semantic query engine and then

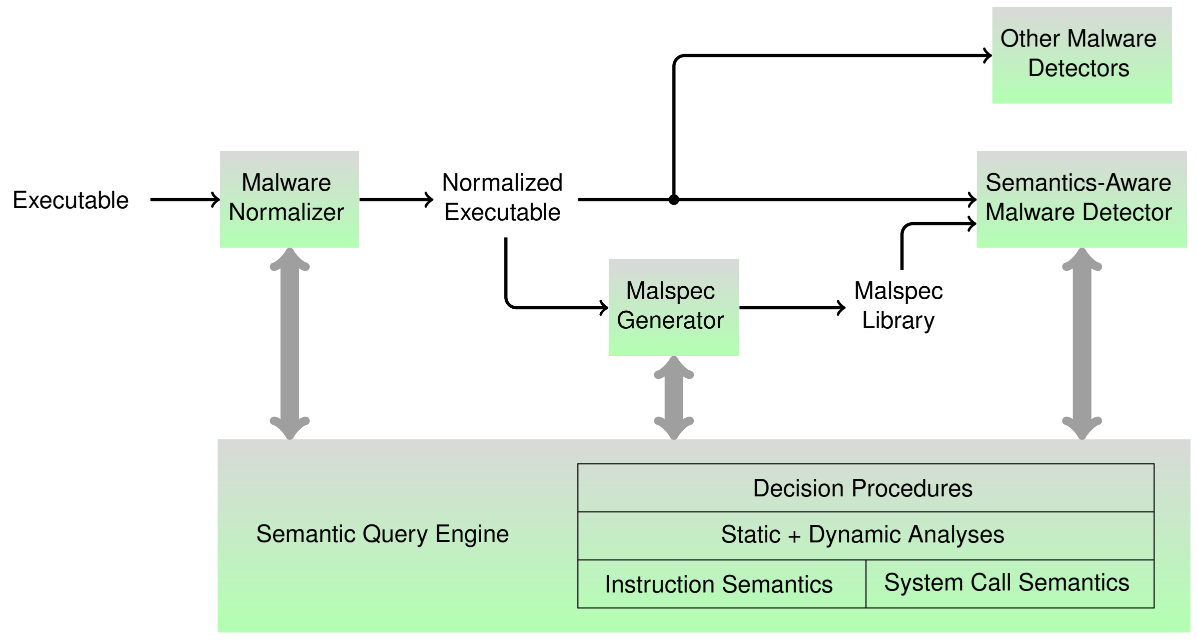


Figure 4: Architecture of the semantics-aware malicious-code analysis system. A *malspec* is a specification of malicious behavior.

excised from the program to obtain a normalized executable. The malware detector checks for the presence of malicious behavior in the program using a library of malicious-behavior specifications. A malicious-behavior specification (or *malspec*) is extracted automatically from known malicious programs by a malspec generator. The modular design benefits other malware detectors as well, as they can scan the normalized executable instead of the original executable. Note that the malware normalizer together with a traditional (non-behavior-based) malware detector cannot achieve the detection rate of the behavior-based malware detector.

1.3 Contributions

To the best of my knowledge, researchers and developers traditionally have not approached analysis and detection of malicious code in a formal way. Instead they count on incremental ad-hoc techniques to handle the threat of the day, effectively playing the coevolution game favored by malware writers. My initial results, summarized below, indicate that such a formal approach is needed and is practical. Furthermore, the application of formal methods could lead to different approaches to the malicious code problem, including the design of new operating system features to prevent malicious code.

Previous work. I have built prototypes for two components of the malicious-code analysis system of Figure 4. First, I implemented a prototype of the semantics-aware malware detection component to determine the importance of handling obfuscation in the detection process [14]. This malware detector prototype used a control-flow graph with uninterpreted variables as specification of malicious behavior. Several classes of obfuscations were manually modeled directly into the malspec. This implementation falls short of the malware detector described earlier (in Figure 4) in two aspects. First, the malspec contained arbitrary instructions (instead of system calls) with a predefined order (instead of an order constrained only by the required flow of data). Second, the obfuscations considered were explicitly encoded in the prototype, which is an unscalable approach as the number of obfuscation schemes increases. Nonetheless, the experimental results showed that this prototype detector, although limited, performed significantly better than commercial malware detectors when tested on obfuscated malware variants.

The second component that I built is a decision procedure to determine whether the values used by the instructions in a malspec are preserved across arbitrary program code sequences [15]. Combined with the previous prototype, it resulted in a malware detector that no longer needs to enumerate obfuscation types explicitly in order to handle obfuscated and evolved malware variants. One limitation, inherited from the previous prototype, is that the malspec contained arbitrary instructions (instead of system calls), thus unnecessarily narrowing the set of variants that can be correctly identified as malicious. A more significant limitation is the assumption that all relations between values in the malspec can be described through equality predicates. Even with these restrictions, the new semantics-aware malware detector could identify variants from the same malware family with only one malspec.

Future Work. In order to realize the behavior-based malicious-code detection system proposed here, the following tasks need to be completed. I believe each of them is a contribution to the computer security field.

- **A formal language for specifying malicious behavior**
  This language for creating malspecs has to be expressive enough to capture any malicious behavior. Its rules should be general enough to describe current and future malicious behavior. Its alphabet includes system calls and state transformers. This language should enable the specification of malicious behaviors as high-level abstractions, independent of syntactic representation. A malicious behavior can always be expressed as a sequence of instructions, but doing so limits the malware detection technique from the very beginning.

- **A behavior-based malware detector**
  My previous work has shown that a malware detection architecture that is semantics-aware is a significant improvement over the patchwork of incremental, signature-based detection schemes in existence. The malware detector has to answer the question whether, given a malspec in the formal language, a program contains any code that follows that malspec. The approach I propose is to treat the analysis engine (i.e., the decision procedures and the static and dynamic analyses) as a set of *oracles*. The detector only needs to know that the oracles can answer various questions. How the oracles answer these questions is not relevant to the malware detector. This allows for a clean interface between the detection algorithm and the decision procedures.

- **A malspec extractor**
  Extracting malicious behavior is currently done manually. For example, commercial anti-virus vendors have large teams of security analysts that pore over each new malicious program in order to develop a (syntactic) signature. This task has to result in a tool that, given several malware samples, can identify the common malicious behavior and express it in the formal language for malspecs. The main problem to solve is the generalization step. Ideally, the common behaviors are extracted as sequences of instructions and then automatically generalized to a malspec that is independent of any particular implementation of these behaviors. In practice, this task has to consider the contribution of a human expert that can guide the malspec extractor during generalization process.

- **A hybrid (static and dynamic) malware detector**
  The hybrid analysis engine at the core of the hybrid malware detector combines rich, yet imprecise, statically-derived information with precise, yet expensive, dynamically-acquired information. The end goal is to reduce false positives. I have identified three subtasks:

  - **Automatically determine where a static analysis loses precision**
    The program locations where a static analysis generates a safe approximation of the facts about the program are the best candidates for dynamic analysis.
  - **Convert precision-loss locations into event collection rules**
    The dynamic analysis aggregates program events (e.g., system calls, network traffic, stack operations). The more detailed and the more extensive the event capture, the higher the performance cost incurred. Static analysis can reduce the overhead of dynamic monitoring and analysis by focusing the need for information to particular points in the program. The events that are to be collected at runtime depend on the malspec.
  - **Extend the malware detector to make decisions at runtime**
    A static malware detector is safe because the underlying static analyses are overapproximating. Furthermore, a static analysis poses no threat to the host system. In contrast, a malware detector that has a dynamic component can endanger the security of the host by failing to detect the malware before the damage is done. My goal is to design a hybrid malware detector to ensure a level of safety (for a particular execution) equivalent to the safety of a static malware detector.

- **A theoretical measure of a malware detector's power**
  Beyond empirical evaluation against live malware families and synthetic variants, I plan to assess the theoretical

limitations of this behavior-based malicious-code detection system. A goal of this task is to find a theoretical formulation that connects the computational complexity of a decision procedure used by the malware detector with the computational complexity of applying an obfuscation used by an attacker. This type of result is particularly relevant in the context of self-propagating malware that modifies itself at each propagation in order to avoid detection. If the self-propagating malware has to perform expensive computations to create a new variant that evades detection, these computations become a rate-limiting mechanism that delay the spread of the malware.

I have listed below several other contributions that, although not part of my research focus, will necessarily result from the research work towards achieving my main goals.

- A collection of program obfuscation techniques that can be used for the comparative evaluation of malicious-code detection algorithms.
- A testing methodology for malware detectors, based on obfuscation transformations. Given a known malware instance, I generate several variants of it by applying various obfuscation transformations. These variants are then treated as test cases for a malware detector. In previous work, I developed an extraction algorithm for syntactic signatures which used a malware detector, such as a virus scanner, as a black box. Using the output of a malware detector on several obfuscations of a malware, I extracted the syntactic signature used by the detection algorithm.

  Experimental evaluation using these testing technique showed that the resilience of current virus scanners to obfuscation transformations was quite varied. Furthermore, using my signature-extraction algorithm, I was able to learn the signatures used by the virus scanners for various malware. From this experimental results I concluded that the state of the art for malware detectors is dismal, which further spurred my interest in developing better malicious-code detection techniques.

2 Related Work

There has been a tremendous amount of work in the field of malware analysis and detection [50, 69]. In this section I review work related to my research goals, focusing on theoretical models, malware detection techniques, malware description languages, and signature generation techniques.

2.1 Theoretical Aspects

The literature on the theoretical limits of malware detection focused predominantly on viruses, as in the beginning of the commercial Internet, those were the most wide spread threats. Fred Cohen defined a virus as a "computer program that can infect other programs by modifying them to include a possibly evolved copy of itself" [2]. I review the results on the theory of virus detection and highlight where these results have general applicability to the detection of any type of malicious program.

Cohen was first to point out that virus detection is undecidable, using a standard diagonal argument [2]. If we had a perfect detection algorithm D (with no false positives and no false negatives), then it is possible to create a virus P of the form shown below that cannot possibly be detected by the algorithm D.

```
if( D( P ) == true )
{
   // do nothing
}
else
{
  spread;
  perform malicious behavior;
}
```

Cohen formalized this intuition to show that determining whether a particular input tape sequence is a virus for a given Turing Machine is undecidable [3]. Later work by Chess and White showed that there exists a virus for which it is impossible to build a perfect detector [1]. A related result, also by Chess and White, states that there is no detection algorithm with no false positives [1]. Note that, while these results consider a detection model where a program is a virus if it *always* acts maliciously, the same undecidability limitations hold for program that are malicious only in some executions (e.g., a trojan horse [65]). Furthermore, these results do not make particular use of the definition of a virus as a program that spreads copies of itself, and are thus applicable to any type of malicious behavior.

Cohen's work manipulated standard Turing Machines, which are not accurate models of modern computers with limited resources. Many other theoretical models have been proposed to describe more accurately modern computers (Random Access Stored Program Machine with Attached Background Storage [6]), to evaluate the infection process that characterizes viruses (Universal Turing Machines [4, 7]), and to dissociate the description of viruses from their physical representation on disk (recursively enumerable sets [9, 10]). In all such models, detection of malicious code is undecidable.

2.2 Malware Detection

I review here malware detection techniques. A malware detection technique is fundamentally characterized by the language it uses to express malicious behavior. A malicious behavior is expressed as a logical formula over elements of the executable file containing the program. The malware detector checks for the validity of the logical formula for a suspicious program. This model of detection can be described using three basic components.

- **Detection time.** Detection can be done statically, in which case the detection reaches a decision using information extracted from the executable file on disk. Detection can be performed dynamically, through monitoring of program execution, possibly in a controlled environment.
- **Detection alphabet.** The basic items that are used to reach a decision during the detection process form the detection alphabet. For example, in the case of sequences of bytes, the alphabet is the set of byte values 0–255. In the case of detection using system calls, the alphabet is the set of all system calls in a particular operating system. The detection alphabet determines the abstraction level at which the detection technique operates.
- **Detection language.** Malicious behavior is expressed as a formula over the detection alphabet. A language defines the predicates, the logical connectives, and the algebra for constructing well-formed formulas of malicious behavior.

Depending on the actual choices for these three components, detection techniques have significantly different characteristics. Below I summarize the choices for the detection alphabet and language, with representative examples of detection techniques. Based on the detection time I classify these techniques into one of two broad classes, static detection and dynamic detection.

2.2.1 Static Malware Detection

I consider static analysis to be any technique that uses information about the binary file representation of a program. This includes information about the file format (e.g., Microsoft Windows Portable Executable [100]), about dependencies on the environment declared in the binary file (e.g., dynamic linking import tables), and about the order and format of instructions in the binary file.

Detection Methods Using Sequences of Bytes and Instructions

- *Scan Strings*

  The first generation virus scanners used the byte representation of some sequence of instructions and data as signature for which to scan [23]. This byte sequence (known as a *scan string*) is extracted from a virus sample such that it is typical of the virus but not likely to be found in clean programs. If a file shares a byte sequence with a known malware instance, it is probably infected with the malware or it is a copy of the malware. Scanning for viruses becomes a simple search for particular byte sequences in the suspicious file. Algorithms for such substring searches are available and allow for efficient virus scanner implementations [86, 105]. Byte scan strings are widely used in practice today, because of their low false positive rate (as they exactly match the corresponding virus sample). This high specificity is also the downfall of this scanning technique.
- *Scan Strings with Regular Expressions*

  As malware writers added register renaming and `nop`-insertion [58, 70] to their collection of evasion techniques, scan strings were extended to regular expressions over bytes [23], as well as various particular subclasses of regular languages. Scan strings with wildcards [23] allow the arguments of an instruction to be specified using a regular expression, thus defeating the register renaming obfuscation. Scan strings with mismatches [23] can account for some evolutionary transformations in the virus code, as they allow for any number of bytes in the matched string to take any value. Smart scanning enhance wildcards in scan strings to use regular expressions for whole instructions [23]. This technique is targeted at detecting junk instructions such as the `nop` instruction. All of these techniques match directly against the bytes in the binary file and are prone to more false positives since actual instruction boundaries are not known or used. Regular expressions over disassembled code attempt to alleviate this problem by first disassembling the program, identifying instructions, and then matching the instruction stream (not the byte stream) against the regular expression [70].

- *Computational Learning Techniques*

  A slightly different approach is the use of byte content distributions (and, in general, n-gram distributions) to distinguish between benign and malicious files [18, 22].

  Computational learning methods (e.g., neural networks, Bayes) have been applied to byte trigrams that appear in infected boot sectors [24] with good results. The same technique, when applied to Microsoft Windows viruses (specifically using the Win32 API), failed to produce satisfactory results (i.e., high detection rate with low false positives) due to the high variability of the bytes in the PE file format [11].

  Schultz, Eskin, Zadok, and Stolfo improved on this technique by adding to the feature set (beyond byte sequences) the list of dynamically linked libraries (DLLs), the list of DLL functions imported by the binary program, the number of functions imported from each DLL, and the set of printable-ASCII strings [21]. Multiple learning algorithms (inductive rule learning, naïve Bayes, multi-naïve Bayes) were used to generate rules over the set of features. The results show an improvement over previous techniques, but the threat is still present, since the attacker only has to change the form of the malware instance, not the behavior.

All of these techniques are fundamentally limited in their detection capabilities, since they count on the presence of certain bytes in the virus body. Most program transformations performed by the malware writer at the source code level, together with the subsequent automatic transformations (compilation, linking, packing), result in distinct binary programs that evade detection. I plan to address such challenges by basing my detection techniques on program structures at higher semantic levels, including control flow graphs and data dependence graphs. Since these structures provide information that is independent of the actual byte representation, my detection algorithm is immune to such evasion attacks.

Detection Methods Using Sequences of System Calls

A better choice of tokens on which to base detection is the set of system calls. Mukkamala, Sung, Xu, and Chavez [20, 25] introduced a system for malware detection that used similarity measures between the system call sequence of a program and a malicious system call sequence[1]. A sequence contains system calls in the order in which they appear inside the code section of the binary file. This approach cannot both identify variants and maintain a low false positive rate, because obfuscations such as code reordering and junk insertion can affect the static order and the number of system calls in a binary. The lack of control flow or data dependence information renders this technique of little use in all but a small set of cases. My proposed approach to detection uses control flow information to determine the correct sequencing of system calls in a program, thus identifying variants with a low false positive rate.

Detection Methods Using Semantic Structures

Semantic structures include the control-flow graphs (CFGs), the data-dependence and control-dependence graph (DDG and CDG) for each program function, the call graph, as well as the system-dependence graph (SDG) (for reference on each of these structure, see [102]). The Malicious Code Filter (MCF) project [19] introduced the use of program slicing [82] to identify the presence of *tell-tale signs* that are often present in malware. The MCF tool extracts

[1]In their work, Mukkamala, Sung, Xu, and Chavez used calls to the Microsoft Windows library APIs instead of the Microsoft Windows system calls. This provides a weaker security guarantee, since library API calls can be circumvented by directly invoking the system calls. Nonetheless, the core idea of the system is the same for both library API calls and system calls.

the slices corresponding to tell-tale signs (e.g., operations on files, accesses to the network, change of privileges, time-dependent computation) and presents it to an analyst to help her decide whether the program is malicious or not. Although a useful forensic tool. it is unclear how MCF can be used as an automated tool for malware detection.

Crocker and Pozzo proposed the concept of a virus filtering system that uses program verification to check that the program behaves within the bounds of the host system's anti-virus policy [16]. This proposal is analogous to George Necula's work on certified compilation [104] and proof-carrying code (PCC) [103], where programs are accompanied by proofs (generated by the program producer) that prove adherence to a set of safety rules (set by the program user). In PCC, if the proof verified successfully, then the program was safe to execute. The proposal by Crocker and Pozzo checked the specification against the program, implicitly producing and verifying a proof of correctness. In contrast with PCC, Crocker and Pozzo envisioned that the specification was also checked against the user's security policy.

Bergeron, Debbabi, Erhioui, and Ktari improved the slicing algorithms for binary programs by making use of *idioms* which are "sequence[s] of instructions that [have] a logical meaning which cannot be derived from individual instructions" [13]. These idioms are used to undo some of the transformations applied by the compiler and recover high-level constructs such as function calls with arguments, and complex expressions with no stack operations. An extension to this work applied these static analysis idioms to construct a simplified control-flow graph of security-relevant library (API) calls (ignoring data flow) in the program [12]. The authors planned to check this API flow graph against a security automaton [106], but no automatic technique to do so was presented.

Kinder, Katzenbeisser, Schallhart, and Veith [17] cast the problem of identifying malicious behavior in a program as a model-checking question. A specification language called Computation Tree Predicate Logic (CTPL) extends the well-known CTL by adding quantifiers over free variables to describe the malicious behavior in a more concise way. A CTPL specification describe sequences of assembly-level instructions and allows for limited code sequences between these instructions. The goal is to detect a larger number of malware variants with a single specification. I propose to adopt features of CTPL in my work on malware detection, as it cleanly describes the data flow properties relevant to malicious behavior.

Anomaly-Based Detection Methods

Anomaly-based detection compares information about the suspicious program with information about a baseline benign program. This is in contrast to misuse-based techniques that compare the suspicious program with a known malicious program.

- *File-Structure Heuristics*

  The format for storing a program on disk has a specific set of rules that most programs follow. The runtime loader interprets this file format and prepares the program for execution by loading it into memory. In many cases, the executable files infected by viruses are incorrect according to the file format specification, but are still accepted by the runtime loader [33]. Malware detectors use these structural differences as indicators of a possible infection: suspicious or missing section characteristics, incorrect virtual section sizes, gaps between sections, suspicious header information, suspicious imports, *etc*. These traits are not necessarily indicative of malicious behavior and can become less useful as structurally valid, standalone programs (e.g., bots, trojans, and spyware) become more prevalent. Due to focus on code behavior, the structure of the file does not affect my proposed malicious detection technique.

- *Computational Learning Techniques*

  Cai, Theiler, and Gokhale explored investigated the use of multivariate Gaussian likelihood models and one-class Support Vector Machines models over byte sequence frequencies [27]. Their goal was to profile only benign data in order to identify malicious executables as outliers or anomalies that significantly deviate from the normal profile. They show that the multivariate Gaussian likelihood models perform better, but in both cases the false positive rate was relatively high (at least 5–70%), compared to the very low false positives rates of current commercial virus scanners (usually 0%).

- *Integrity Checking*

  Integrity checking compares the contents of the binary program file against an earlier copy of itself. For efficiency reasons, the comparison is usually performed over checksum values of the current and previous copy of the file. In the case of viruses that spread by infecting files, on-demand integrity checkers provide the best generic method for malware detection [29]. The checksum computation can be done efficiently [32], but integrity checking suffers from high false positive rates due to changes in applications (e.g., encryption or packing), application updates, and new application installations [23]. Integrity checking also requires a clean, uninfected baseline and support for partial-file checksums (e.g., for Microsoft Word documents, only the macro code section needs to have its integrity checked). Some of these problems have been addressed [26], while others (such as the clean baseline and the integrity of the checker itself) require a secure bootstrap method [28, 34].

- *Authenticated Code*

  If software producers were to sign their code in a verifiable fashion, then users could easily protect themselves from malicious code by running only programs from trusted sources. This approach shifts the decision focus from the program to the certificate chain that establishes trust. One straightforward way to simplify the trust relationship is to use identity-based cryptography [107]. E. Okamoto and H. Masumoto proposed such a system where software producers register with a central authority and then sign their software using their identity as key [31]. Users can then verify this (human-readable) identity by checking the signature on the program before executing it.

  This approach has the central authority that both the software producers and the software users have to trust as a single point of failure. An attack against this authority could invalidate the secret keys underlying the certification process. Harn, Lin, and Yang proposed the use of a multiple certification authorities that each assures the integrity of some part of a software producer's signature [30]. The widespread implementation of signed code using Microsoft's Authenticode [101] showed that end-users tend to ignore the signature on the program because they have no means of verifying this signature in the absence of a trustworthy PKI infrastructure or some other trust-establishment side channel.

  Both integrity checking and authenticated code are useful in securing a system, as long as they operate from a trusted code base such that the enforcement mechanisms cannot be subverted. I see these techniques as complementary to my proposed malware detector.

2.2.2 Dynamic Malware Detection

Dynamic analysis collects information about the program while it is running and enables the detection of malicious behavior evident in the current execution of the program. Dynamic analysis provides results about a particular execution for a particular set of inputs, while static analysis discovers information about multiple program executions without having specific knowledge about the program inputs.

Defeating Encrypted and Compressed Malware

Many malicious programs, even non-polymorphic ones, have their main body of code encrypted or compressed. During execution, the malicious program decrypts or unpacks the rest of the program and continues execution with the original body of code. Any technique that relies only on the static form of the program code in order to detection the malicious code fails in such cases. Most techniques to extract the main body of code automatically share the assumption that the decryption and decompression routines are self-contained (i.e., both the encrypted or compressed code and the encryption key are available). X-raying [39], sandboxing [37, 38], and emulation are all techniques that execute the program in a contained environment and stop execution after the program completes the decryption (or decompression) routine and before transferring control to the decrypted body of code. All of these techniques solve the problem of decrypting/unpacking the malicious payload quite well, and I use emulation as part of my proposed malware detection system.

In-Memory Scanning

Once the main body of the program is decrypted or decompressed in memory, any of the static analysis techniques can be applied to the memory region containing the program body. Commercial tools apply scan strings directly to the memory regions located during the decryption/decompression step [23]. The strengths and limitations of each of these static analyses do not change when applied to a memory snapshot of the executing program.

Advanced Dynamic Analysis

Runtime monitoring in an emulator enables the use of signatures that refer to exact program state. Signatures refer to values at particular memory locations and in registers that were observed to be constant across variants from the same malware family [23]. A runtime monitoring approach also allows the use of heuristics that compare the system state (at a high level of abstraction, e.g., the filesystem) before and after the execution of malware [23].

Daniel Guinier proposed the use of neural networks to identify the high level operations that differentiate malicious programs from benign programs [36]. The input to the technique is the set of observed actions of the program (e.g., console output, files written or read, network accesses) and the neural network is trained on these observed actions. This approach requires a user (most likely a security expert) to describe the operations performed by the program, as no automatic system for deriving them is provided.

Salois and Charpentier described a system for dynamic detection that checks each instruction executed by the program against a given security policy [40]. This dynamic analysis has to track all data flows through the program, leading to a significant performance reduction. A later extension on this work focused only on security-critical API calls and resources [35]. Calls are monitored and a security automaton [106] is operated over these calls.

All of these dynamic analysis approaches are either low-level and very accurate or high-level and with too many false alarms. This set of extremes illustrates the trade-off that dynamic analysis has to make between performance (where few runtime events are collected, but the false positive rates are high) and precision (where detailed events are analyzed, but the cost of monitoring is prohibitive). In this proposal, I introduce a technique that combines static and dynamic analyses to balance these two opposing requirements.

2.2.3 Hybrid (Static–Dynamic) Detection

There is little to no research on hybrid (combined static and dynamic) malicious code detection techniques. In many cases, static analysis is used as a filter to preselect programs that then get checked using an expensive dynamic technique. Such a combination is not a true hybrid detection, since no information passes from the static analysis step to the dynamic analysis step (other than the fact that the problem is suspicious and requires additional checking).

In other fields, static–dynamic analysis has been used to great success. For example, Colby showed how to check guards loop expressions statically and, for those guards that cannot be proven right statically, insert dynamic checks that perform the checks at runtime [78]. Similarly, the CCured project used dynamic checks to determine correct pointer accesses [81]. These checks were inserted at locations where static analysis could no prove the type-safety of a pointer access. A combination of static and dynamic analysis for malicious code detection can lead to lower false positive rates as well as improvements in performance.

2.3 Malware Description Languages

Languages for expressing malicious behaviors are few and far between. The commercial malware detectors do not document the internal language used to represent their signatures, as they consider their signatures intellectual property. The few examples of malware description languages that are published (Virtran [43], VERV [41], CVDL [44], ClamAV's language [42], TDL/V [46]) are limited extensions of regular languages over byte sequences and are designed specifically for improved scanning performance and for ease of updating.

Two languages merit further attention. The MetaMS language [45] describes malware written in a scripting language (e.g., e-mail viruses written in VBA). It uses a hierarchical structure to capture function bodies, loops, and branches that contain security-relevant operations such as replication or sensitive-file access. While MetaMS represents these operations in a format that is independent of variable names, specific constant values, and comments. It is still limited by the presence of syntactic information (e.g., function boundaries).

CTPL is a program logic (with a corresponding language) [17] that models sequences of instructions with quantified variables that represent registers and memory locations. A special predicate `#loc(`L`)` expresses data dependencies between arguments to different instructions. CTPL handles changes to variable values between consecutive instructions only if the code causing the change is explicitly defined in the CTPL formula.

2.4 Signature Generation

There is a dearth of techniques to generate signatures automatically. The fundamental problem is that of identifying a signature that balances generality (that helps the detection rate) and accuracy (that reduces false alarms).

Kephart and Arnold proposed a technique for generating scan strings composed of 12–36 bytes [48, 49]. These byte sequences are extracted from the code sections of known malicious binaries and then the byte sequence with no false positives is chosen as the signature. This approach uses frequencies of trigrams in benign programs to estimate the probability for a byte sequence to match a randomly-chosen block of bytes in machine code, while ignoring the semantics of the underlying code.

Deng, Wang, Shieh, Yen, and Tung presented an improved signature generation technique that allows for variable-length signatures, generates multiple signatures from a single malware sample, and applies better classifiers for reducing the false positive rate [47].

Nonetheless, these techniques generate only syntactic (byte-based) signatures. There are no known techniques for extracting signatures expressed in more abstract virus description languages, such as CTPL [17]. My proposed research plan includes tasks to address the problem of automatically extracting behavioral signatures.

3 A Language for Specifying Malicious Behavior

A language for malicious behavior needs to be expressive enough to capture all types and nuances of malicious functionality. It cannot depend on features that can be obfuscated. It must have only limited dependencies, if any, on the operating systems and the associated runtime environment, since both of them are frequently and unpredictably updated. Last but not least, a language for malicious behavior must deal in high-level concepts so that security policies can be easily translated into expressions of undesirable behavior.

A malicious behavior is specified using an graph that describes all possible programs that exhibit the malicious observable behavior. Intuitively, this malicious specification, henceforth referred to as a *malspec*, describes sequences of system calls together with relations between their arguments and their return values.

Definition 1 [MALSPEC]
A malicious-code specification (malspec) $\mathcal{M}$ is a 4-tuple $(\Sigma, V, \mathcal{A}, G)$, where

- *Σ is a set of system calls, $\Sigma = \{\ldots, \sigma_k(\alpha_1, \ldots, \alpha_{n_k}), \ldots\}$, where σ_k represents the k-th system call with its n_k typed arguments $\alpha_1 : \tau_1, \ldots, \alpha_{n_k} : \tau_{n_k}$,*
- *V is a set of uninterpreted, typed variables, $V = \{v_1 : \tau_1, v_2 : \tau_2, \ldots\}$,*
- *$\mathcal{A}$ is an algebra for formulas over uninterpreted variables and constant values,*
- *G is a directed graph, $G = (N, E)$, where*
  - *each node $n \in N$ is labeled with a system call σ_k instantiated with uninterpreted variables,*
  - *each edge $e \in E$ is labeled with a predicate expressed as an $\mathcal{A}$-formula.*

The set Σ contains all of the system calls available for a particular platform. In this respect, Σ encapsulates any platform-specific details. A system call is, symbolically, represented by $\sigma_k(\alpha_1, \ldots, \alpha_{n_k})$, where each α_i is a typed parameter. For example, for the UNIX system call `creat`, the arguments are α_1=`pathname` of type `char *` and α_2=`mode` of type `mode_t`.

Each node in the graph is labeled with a system call instantiated with uninterpreted variables:

$$n : \boxed{v_i \leftarrow \sigma(v_{j_1}, v_{j_2}, \ldots, v_{j_k})}$$

This notation reads as follows: the system call σ takes for its k arguments the variables $v_{j_1}, \ldots, v_{j_k}$ and returns a value that is then assigned to variable v_i. All the arguments are passed using reference semantics so that, depending on the particular implementation of each system call, the input variables $v_{j_1}, \ldots, v_{j_k}$ might receive new values after the system call returns. For simplicity, the case of input variables that are not modified by the system call is interpreted to assign new values that are equal to the previous values (i.e., for such variables the system call acts as the identity function).

The algebra $\mathcal{A}$ defines the structure of boolean formulas over uninterpreted variables and constants. Implicitly, this algebra defines the logic in which to evaluate the boolean formulas. The graph structure of the malspec restricts the uninterpreted variables appearing in a formula. A formula ϕ that labels the edge from a node n_1 to a node n_2 can refer to:

- the values of uninterpreted variables *used* by the node n_2, and
- the values of uninterpreted values *defined* by any predecessor of the node n_2.

The type constructors build upon simple integer types (listed below as the *ground* class of types), and allow for array types (with two variations: the pointer-to-start-of-array type and the pointer-to-middle-of-array type), structures and unions, pointers, and functions. Two special types $\perp(\mathrm{n})$ and $\top(\mathrm{n})$ complete the type system lattice. $\perp(\mathrm{n})$ and $\top(\mathrm{n})$ represent types that are stored on n bits, with $\perp(\mathrm{n})$ being the least specific ("any") type and $\top(\mathrm{n})$ being the most specific type. Table 1 describes the constructors allowed in the type system.

τ	::	ground	*Ground types*
	|	τ`[n]`	*Pointer to the base of an array of type τ and of size* n
	|	τ`(n]`	*Pointer into the middle of an array of type τ and of size* n
	|	τ `ptr`	*Pointer to τ*
	|	$\mathrm{s}\{\mu_1, \ldots, \mu_\mathrm{k}\}$	*Structure (product of types of μ_i)*
	|	$\mathrm{u}\{\mu_1, \ldots, \mu_\mathrm{k}\}$	*Union*
	|	$\tau_1 \times \cdots \times \tau_\mathrm{k} \rightarrow \tau$	*Function*
	|	$\top$`(n)`	*Top type of* n *bits*
	|	$\perp$`(n)`	*Bottom type of* n *bits (type "any" of* n *bits)*
μ	::	(l, τ, i)	*Member labeled l of type τ at offset i*
ground	::	`int`$(g$`:`s`:`$v)$ | `uint`$(g$`:`s`:`$v)$ | ...	

Table 1: A simple type system.

The type $\mu\,(l, \tau, i)$ represents the type of a field member of a structure. The field has a type τ (independent of the types of all other fields in the same structure), an offset i that uniquely determines the location of the field within the structure, and a label l that identifies the field within the structure (in some cases this label might be undefined).

Physical subtyping takes into account the layout of values in memory [75, 83]. If a type τ is a *physical subtype* of τ' (denoted it by $\tau \leq \tau'$), then the memory layout of a value of type τ' is a prefix of the memory layout of a value of type τ. We will not describe the rules of physical subtyping here as we refer the reader to Xu's thesis [83] for a detailed account of the typestate system (including subtyping rules).

The type `int`$(g$`:`s`:`$v)$ represents a signed integer, and it covers a wide variety of values within storage locations. It is parametrized using three parameters as follows: g represents the number of highest bits that are ignored, s is the number of middle bits that represent the sign, and v is the number of lowest bits that represent the value. Thus the type `int`$(g$`:`s`:`$v)$ uses a total of $g + s + v$ bits.

$$\underbrace{d_{g+s+v} \ldots d_{s+v+1}}_{\text{ignored}} \underbrace{d_{s+v} \ldots d_{v+1}}_{\text{sign}} \underbrace{d_v \ldots d_1}_{\text{value}}$$

The type `uint`$(g$`:`s`:`$v)$ represents an unsigned integer, and it is just a variation of `int`$(g$`:`s`:`$v)$, with the middle s sign bits always set to zero.

The notation `int`$(g$`:`s`:`$v)$ allows for the separation of the data and storage location type. In most assembly languages, it is possible to use a storage location larger than that required by the data type stored in it. For example, if a byte is stored right-aligned in a (32-bit) word, its associated type is `int(24:1:7)`. This means that an instruction such as *xor on least significant byte within 32-bit word* will preserve the leftmost 24 bits of the 32-bit word, even though the instruction addresses the memory on 32-bit word boundary.

This separation between data and storage location raises the issue of alignment information, i.e., most computer systems require or prefer data to be at a memory address aligned to the data size. For example, 32-bit integers should be aligned on 4-byte boundaries, with the drawback that accessing an unaligned 32-bit integer leads to either a slowdown (due to several aligned memory accesses) or an exception that requires handling in software. Presently, we do not use alignment information as it does not seem to provide a significant covert way of changing the program flow.

Table 2 illustrates the type system for Intel IA-32 architecture. There are other IA-32 data types that are not covered in Table 2, including bit strings, byte strings, 64- and 128-bit packed SIMD types, and BCD and packed BCD formats. The IA-32 logical address is a combination of a 16-bit segment selector and a 32-bit segment offset, thus its type is the cross product of a 16-bit unsigned integer and a 32-bit pointer.

IA-32 Datatype	*Type Expression*
`byte unsigned int`	`uint` (0:0:8)
`word unsigned int`	`uint` (0:0:16)
`doubleword unsigned int`	`uint` (0:0:32)
`quadword unsigned int`	`uint` (0:0:64)
`double quadword unsigned int`	`uint` (0:0:128)
`byte signed int`	`int` (0:1:7)
`word signed int`	`int` (0:1:15)
`doubleword signed int`	`int` (0:1:31)
`quadword signed int`	`int` (0:1:63)
`double quadword signed int`	`int` (0:1:127)
`single precision float`	`float` (0:1:31)
`double precision float`	`float` (0:1:63)
`double extended precision float`	`float` (0:1:79)
`near pointer`	$\perp$ (32)
`far pointer (logical address)`	`uint` (0:0:16) $\times$ `uint` (0:0:32)
`eax, ebx, ecx, edx`	$\perp$ (32)
`esi, edi, ebp, esp`	$\perp$ (32)
`eip`	`int` (0:1:31)
`cs, ds, ss, es, fs, gs`	$\perp$ (16)
`ax, bx, cx, dx`	$\perp$ (16)
`al, bl, cl, dl`	$\perp$ (8)
`ah, bh, ch, dh`	$\perp$ (8)

Table 2: IA-32 datatypes and their corresponding expression in the type system from Table 1.

3.1 Discussion

Our language for specifying malicious behavior benefits the task of malware detection because of its structure and higher level semantics. A malspec does not specify syntactic constructs such as functions, basic blocks, or instruction sequences. Thus a malspec is transparent to obfuscations that affect the instructions in the program and to the obfuscations that preserve the behavior of a program.

A malspec encapsulates the minimum amount of information required to express a behavior. Since the malspec employs formal predicates to describe data flow between system calls, instead of instruction sequences, it is independent of any representation format on disk. Furthermore, a malspec can be considered as a systemwide specification, since it does not require the behavior to appear all in one program or one process. This becomes important in the context of multi-agent malware, where multiple programs collaborate (usually on the same machine) to achieve some malicious end [71].

Finally, the malspec language is not affected by program evolution. Any transformation that adds or restructures functionality is orthogonal to the existing functionality as embodied in a malspec. Thus, this language is expressive enough to describe malicious behavior, even in the presence of obfuscation or evolutionary program transformations.

3.2 Independence of Interface

The definition of a malspec hinges on the alphabet Σ of system calls. This set defines the interface between the malware (or, generally, the untrusted program) and the operating system (i.e., the trusted computing base). I chose the system-call interface for this proposal since it is a widespread and convenient boundary in modern operating systems. There is nothing in the definition of a malspec that limits the interface to system calls. Depending on the definition of the trusted computing base (TCB), the corresponding interface could be plugged into the definition of a malspec. For example, if the Microsoft Outlook is part of the TCB together with the operation system, then the interface over which to define a malspec is the Visual Basic for Applications (VBA) embedded language that Outlook supports.

4 Malicious Behavior Detection

The problem that a behavior-based malware detector solves can be expressed as follows:

> Given a malspec $\mathcal{M}$ (defined in the language of Section 3) and a program P, determine whether any execution trace of P contains a sequence of system calls in the relation described by the malspec $\mathcal{M}$.

This section describes a static analysis solution. For the static analysis to perform correctly, the program needs to be free of any obfuscations that hide the system calls (e.g., encryption, compression). A malware normalizer deobfuscates a program to expose the system calls used.

Malware Normalization. A *malware normalizer* is a system that takes an obfuscated executable, undoes the obfuscations, and outputs a normalized executable. At an abstract level, a malware normalizer thus performs a functionality at the host that is analogous to the functionality provided by a traffic normalizer at the network level [96]. The malware normalizer automatically unpacks and decrypts a packed executable using emulation of the program and its runtime environment (operating system, associated libraries, etc.). A runtime monitor processes the stream of executed instructions and determines the location of the encrypted code and the moment when the decrypted code is present in memory. With this information a decrypted executable is reconstructed. An iterative use of the normalizer can handle malware that decrypts itself in several stages.

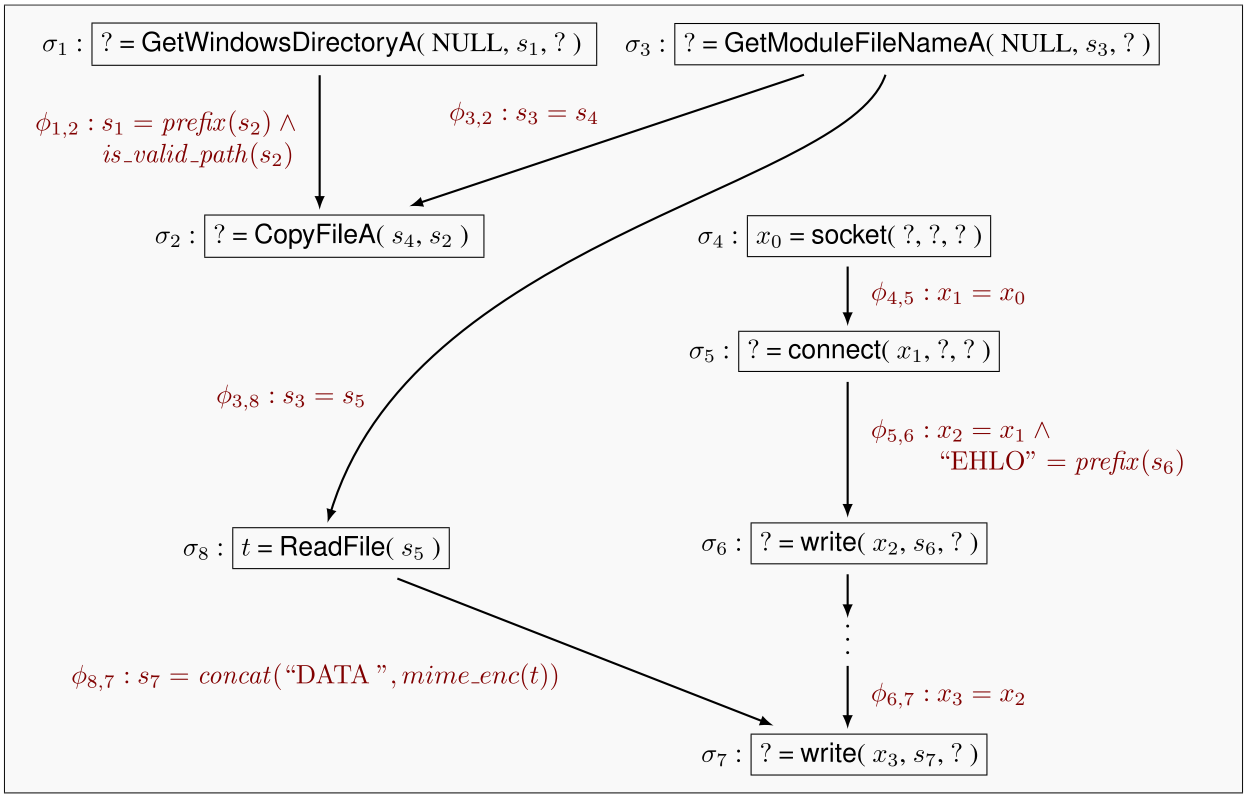
A malware normalizer can be extended to perform additional deobfuscation tasks that simplify the executable file. These deobfuscation transformations can be applied to an executable *independent of any malspec* that is checked by the malware detector. For example, code reordering and junk insertion can be fully reversed using static analysis techniques. An executable that has been processed by a malware normalizer does not have these obfuscations.

Malware Detection. Suppose the malspec $\mathcal{M}$ is defined as in Figure 5, for some algebra $\mathcal{A}$ and the set of system calls $\Sigma = \{$ GetWindowsDirectoryA, GetModuleFileNameA, CopyFileA, socket, connect, write, ReadFile $\}$[2]. The question marks represent arguments and return values that are unconstrained. An instruction-level execution trace of the program P that matches the malspec $\mathcal{M}$ could have the form shown in Figure 6 (where some of the system call names are shortened for layout purposes). The instruction sequences I_1 through I_7 have to fulfill the seven predicates in the malspec $\mathcal{M}$. In Table 3, $I_1 \cdot I_2$ denotes the instruction sequence obtained by concatenation of instruction sequences I_1 and I_2.

The goal of the malware detector is to determine whether such traces could be generated by a program P. I present here an algorithm based on static analysis to solve this problem. The benefit of using static analysis is two-fold. First, static analysis produces results about all program paths, and, thus, all execution traces. Second, static analysis overapproximates its results, leading to a safe solution from a security perspective. This static analysis algorithm answers the question whether there *might* exist a path through the program that generates a desired execution trace.

The algorithm operates over a *system control flow graph* obtained by connecting all of the control flow graphs (CFGs) of the program using the procedure call and return edges. One possible implementation is illustrated by Algorithm 1. This implementation only highlights the conceptual steps that the malware detection algorithm takes and is not optimized for practical use. The algorithm constructs a correspondence (denoted by the variable $Matches$) between nodes in the malspec and locations in the program. A location in the program matches a node in the malspec if they both contain the same system call (line Unif in Algorithm 1), if all of the predecessors of the node are already matched (line Prev in Algorithm 1), and if, for each predecessor of the malspec node, there is a program path from one of the predecessor's program locations to the current program location that models the predicate on the corresponding

[2]The hypothetical system call ReadFile that reads the full contents of a file into a buffer is used here for brevity.



σ_1 : ? = GetWindowsDirectoryA(NULL, s_1, ?)
σ_3 : ? = GetModuleFileNameA(NULL, s_3, ?)
$\phi_{1,2} : s_1 = \mathit{prefix}(s_2) \wedge \mathit{is\_valid\_path}(s_2)$
$\phi_{3,2} : s_3 = s_4$
σ_2 : ? = CopyFileA(s_4, s_2)
σ_4 : x_0 = socket(?, ?, ?)
$\phi_{4,5} : x_1 = x_0$
σ_5 : ? = connect(x_1, ?, ?)
$\phi_{3,8} : s_3 = s_5$
$\phi_{5,6} : x_2 = x_1 \wedge$ "EHLO" $= \mathit{prefix}(s_6)$
σ_8 : t = ReadFile(s_5)
σ_6 : ? = write(x_2, s_6, ?)
$\phi_{8,7} : s_7 = \mathit{concat}($"DATA "$, \mathit{mime\_enc}(t))$
$\phi_{6,7} : x_3 = x_2$
σ_7 : ? = write(x_3, s_7, ?)

Figure 5: Example of a malspec.

$$\langle \ldots \mathsf{GMFNA} \underbrace{\ldots\ldots}_{I_1} \mathsf{socket} \underbrace{\ldots\ldots}_{I_2} \mathsf{connect} \underbrace{\ldots\ldots}_{I_3} \mathsf{write} \underbrace{\ldots\ldots}_{I_4} \mathsf{ReadFile} \underbrace{\ldots\ldots}_{I_5} \mathsf{write} \underbrace{\ldots\ldots}_{I_6} \mathsf{GWDA} \underbrace{\ldots\ldots}_{I_7} \mathsf{CFA} \ldots \rangle$$

Figure 6: An instruction-level execution trace.

Instruction sequence...	*satisfies predicate...*
$I_1 \cdot \langle\mathsf{socket}\rangle \cdot \ \ldots \ \cdot I_7$	$\phi_{3,2}$
$I_1 \cdot \langle\mathsf{socket}\rangle \cdot \ \ldots \ \cdot I_4$	$\phi_{3,8}$
$I_4 \cdot \langle\mathsf{ReadFile}\rangle I_5$	$\phi_{6,7}$

Instruction sequence...	*satisfies predicate...*
I_7	$\phi_{1,2}$
I_2	$\phi_{4,5}$
I_3	$\phi_{5,6}$
I_5	$\phi_{8,7}$

Table 3: Conditions for the instruction-level execution trace from Figure 6 to match the malspec from Figure 5.

malspec edge (line Test in Algorithm 1). The algorithm is then simply a fix-point computation over this set of constraints. If the fix-point solution (i.e., the variable $Matches$) maps all of the nodes in the malspec to program locations, then the program exhibits the malicious behavior captured by the malspec.

Algorithm 1: The core of the behavior-based malware detector determines whether a program P exhibits a malicious behavior specified by malspec $\mathcal{M}$.

```
Input: A malspec $\mathcal{M} = (\Sigma, V, \mathcal{A}, G)$, where $G = (N, E)$, and a program $P$.
Output: $true$ if the program $P$ exhibits the malicious behavior specified by $\mathcal{M}$.

MaliciousCodeDetector( $\mathcal{M}$, $P$ )
begin
    /* The variable $Matches$ is a map from the nodes in the malspec to
       corresponding locations in the program.    */
    foreach $n \in N$ do $Matches(n) \leftarrow \emptyset$

    repeat
        foreach $n \in N$ do
(Prev)      if $\exists m \in Predecessors(n) . Matches(m) = \emptyset$ then continue

(Unif)      foreach program point $l \in P$ such that the instruction at $l$ matches the system call at $n$ do
(Test)          if $\exists m \in Predecessors(n) . \exists k \in Matches(m) . \exists p \in Paths(k, l) . p \models \phi_{m,n}$ then

                endif
                add $l$ to $Matches(n)$
    until no changes to $Matches$

    if $\forall n \in N . Matches(n) \neq \emptyset$ then return $true$
    else return $false$
end
```

The algorithm described above relies on additional information sources that can be modeled as queries over facts gathered by other static analysis. In particular, the line labeled Test in Algorithm 1 requires precise information about program control flow (e.g., What are the successors of a program location? What are the valid paths between two program locations?) and also requires a way to determine whether a malspec predicate is validated by the instructions on a given program path. The line Unif makes implicit use of unification to instantiate the uninterpreted instructions when a match is considered between a malspec node and a program instructions. I refer to these static analyses as *oracles* because from a security perspective I am interested only in their answers, not in their inner workings. This encapsulation of program facts and queries over these facts behind well-defined interfaces has several benefits. First, it separates the core of the malicious-behavior detection from any other analyses it depends on. This means that the computational complexity of the detector can be judged more easily, regardless of the particular choices for the underlying static analyses. Second, from a software engineering perspective, this separation between the core algorithm and the static analyses leads to a simpler implementation that can be easily updated with improved static analyses as they become available. Third, security guarantees (in this case, guarantees about the classes of malware variants detected) can be expressed in terms of oracles.

Control-flow oracle. The first oracle of interest answers questions about the program control flow. The queries take the form: given a valid partial path through the program, which are the possible control-flow transitions that can extend the path while preserving its validity? If the given partial path is empty, then the query is simply about the possible successors of a program node. Such an query is not trivial, since almost all programs make use (especially when in binary form) of indirection in their control flow (e.g., function pointers, computed indirect jumps, jump tables). There has been a lot of work on recovering control flow [74, 76, 77, 80] and I plan to use these established techniques to implement the control-flow oracle.

Unification. The unification step is required when matching program instructions with malspec nodes with uninterpreted instructions. Definitions and algorithms related to unification are standard and can be found in [95]. I use one-way matching which is simpler than full unification, made possible by the fact that each program instruction is a ground term. The variables in the malspec get instantiated so that they match the corresponding terms in the instruction. The result of the unification step is a binding map, defined as a set of pairs [variable v, value x]. Formally, a binding $\mathcal{B}$ is defined as $\{\ [x, v] \mid x \in V,\ x : \tau,\ v : \tau',\ \tau \leq \tau'\ \}$. If a pair $[x, v]$ occurs in a binding $\mathcal{B}$, then we write $\mathcal{B}(x) = v$. Two bindings $\mathcal{B}_1$ and $\mathcal{B}_2$ are said to be *compatible* if they do not bind the same variable to different values:

$$Compatible(\mathcal{B}_1, \mathcal{B}_2) \overset{def}{=} \forall x \in V\ .\ ([x, y_1] \in \mathcal{B}_1 \wedge [x, y_2] \in \mathcal{B}_2) \Rightarrow (y_1 = y_2)$$

The *union of two compatible bindings* $\mathcal{B}_1$ and $\mathcal{B}_2$ includes all the pairs from both bindings. For incompatible bindings, the union operation returns an empty binding.

$$\mathcal{B}_1 \cup \mathcal{B}_2 \overset{def}{=} \begin{cases} \{[x, v_x] : [x, v_x] \in \mathcal{B}_1 \vee [x, v_x] \in \mathcal{B}_2\} & \text{if } Compatible(\mathcal{B}_1, \mathcal{B}_2) \\ \emptyset & \text{if } \neg Compatible(\mathcal{B}_1, \mathcal{B}_2) \end{cases}$$

The binding maps resulting from the unification step participate in the verification step, where the validity of malspec predicates is checked on selected program paths. The binding maps provide the values necessary to instantiate the uninterpreted variables appearing in the malspec predicates. Thus, the check on line Unif of Algorithm 1 is more appropriately expressed as $p \models \phi_{m,n}|_{\mathcal{B}}$, where $\phi_{m,n}|_{\mathcal{B}}$ represents the predicate $\phi_{m,n}$ with its uninterpreted variables instantiated with values from the binding map $\mathcal{B}$.

Predicate-verification oracle. This oracle checks the truth value of the formula $p \models \phi_{m,n}|_{\mathcal{B}}$. If the program path p contains no backedges (i.e., no loops), then p can be translated to a state transformer formula ϕ_p constructed using the semantic information for instructions and system calls. Then a decision procedure for the logic of the formulas ϕ_p and $\phi_{m,n}|_{\mathcal{B}}$ is used to determine whether $\phi_p \Rightarrow \phi_{m,n}|_{\mathcal{B}}$. I present here two decision procedures that implement this oracle with different degrees of success.

4.1 The Explicit Deobfuscation Oracle

The deobfuscation oracle implements a series of simple checks to determine whether a sequence of instructions on a program path (also known as a *program fragment*) does not affect the state of the program. This oracle is specialized along two axes. First, it checks only against well-known obfuscation techniques. Second, the only predicates that it can prove valid are the one that preserve program state (i.e., all of the state values before the program fragment are identical to the state values after the program fragment). The oracle implementation is based on simple pattern matching against two types of junk code:

- Sequences of `nop` instructions, and
- Sequences of instructions that save and restore the program state, including for example, sequences of paired-up instructions `push` A—`pop` A, for any register A.

4.2 The Value-Preservation Oracle

A value-preservation oracle is a decision procedure that determines whether a given program fragment is a semantic nop with respect to certain program variables. Formally, given a program fragment P and program expressions e_1, e_2, a decision procedure $\mathcal{D}$ determines whether the *value predicate* $\phi(P, e_1, e_2) \equiv \forall$ paths $p \in P\ .\ val_{\text{pre}\langle p\rangle}(e_1) = val_{\text{post}\langle p\rangle}(e_2)$ holds, where $val_{\text{pre}\langle p\rangle}(e)$ and $val_{\text{pre}\langle p\rangle}(e)$ represent, respectively the value of the expression e before and after the execution of the program path p, for all paths p in the program fragment P.

$$\mathcal{D}(P, e_1, e_2) = \begin{cases} \text{true} & \text{if } P \text{ is a semantic nop, i.e., } \phi(P, e_1, e_2) \text{ holds} \\ \bot & \text{otherwise} \end{cases}$$

Similarly, I can define decision procedures that determine whether $\neg\phi(P, e_1, e_2)$ holds (in this case, the result of $\mathcal{D}(P, e_1, e_2)$ is "false" or $\bot$). Denote by $\mathcal{D}^+$ a decision procedure for $\phi(P, e_1, e_2)$, and by $\mathcal{D}^-$ a decision procedure for $\neg\phi(P, e_1, e_2)$.

As the value preservation queries are frequent in our algorithm (possibly at every step during node matching), the prototype use a collections of decision procedures ordered by their precision and cost. Intuitively, the most naïve decision procedures are the least precise, but the fastest to execute. If a $\mathcal{D}^+$-style decision procedure reports "true" on some input, all $\mathcal{D}^+$-style decision procedures following it in the ordered collection will also report "true" on the same input. Similarly, if a $\mathcal{D}^-$-style decision procedure reports "false on some input, all $\mathcal{D}^-$-style decision procedures following it in the ordered collection will also report "false" on the same input. As both $\mathcal{D}^+$- and $\mathcal{D}^-$-style decision procedures are sound, I can define the order between $\mathcal{D}^+$ and $\mathcal{D}^-$ decision procedures based only on performance.

This collection of decision procedures provides us with an efficient algorithm for testing whether a program fragment P preserves expression values: iterate through the ordered collection of decision procedures, querying each $\mathcal{D}_i$, and stop when one of them returns "true" ("false" for $\mathcal{D}^-$-style decision procedures). This algorithm provides for incrementally expensive and powerful checking of the program fragment, in step with its complexity: program fragments that are simple semantic nops will be detected early by decision procedures in the ordered collection. Complex value preserving fragments will require passes through multiple decision procedures. I present, in order of increasing complexity, four decision procedures.

Nop Library $\mathcal{D}^+_{\textbf{NOP}}$. This decision procedure identifies sequences of actual `nop` instructions, which are processor-specific instructions similar to the *skip* statement in some high-level programming languages, as well as predefined instruction sequences known to be semantic nops. Based on simple pattern matching, the decision procedure annotates basic blocks as nop sequences where applicable. If the whole program fragment under analysis is annotated, then it is a semantic nop. In this respect, the nop library decision procedure builds on the explicit deobfuscation oracle from Section 4.1. Furthermore, the nop library acts as a cache for queries already resolved by other decision procedures.

Randomized Symbolic Execution $\mathcal{D}^-_{\textbf{RE}}$. This oracle is based on a $\mathcal{D}^-$-style decision procedure using randomized execution. The program fragment is executed using a random initial state (i.e. the values in registers and memory are chosen to be random). At completion time, I check whether it is true that $\neg\phi(P, e_1, e_2)$: if true, at least one path in the program fragment is not a semantic nop, and thus the whole program fragment is not a semantic nop.

Theorem Proving $\mathcal{D}^+_{\textbf{ThSimplify}}$. The value preservation problem can be formulated as a theorem to be proved, given that the program fragment has no loops. I use the Simplify theorem prover [94] to implement this oracle: the program fragment is represented as a state transformer δ, using each program register and memory expression converted to SSA form. I then use Simplify to prove the formula $\delta \Rightarrow \phi(P, e_1, e_2)$, in order to show that all paths through the program fragment are semantic nops under $\phi(P, e_1, e_2)$. If Simplify confirms that the formula is valid, the program fragment is a semantic nop. One limitation of the Simplify theorem prover is that it does not support the theory of bit vectors, which binary programs are based on. Thus, I can query Simplify only on programs that do not use bit-vector operations.

Theorem Proving $\mathcal{D}^+_{\textbf{ThUCLID}}$. A second theorem proving oracle is based on the UCLID infinite-state bounded model checker [99]. For our purposes, the logic supported by UCLID is a superset of that supported by Simplify. In particular, UCLID precisely models integer and bit-vector arithmetic. I model the program fragment instructions as state transformers for each register and for memory (represented as an uninterpreted function). UCLID then simulates the program fragment for a given number of steps and determines whether $\phi(P, e_1, e_2)$ holds at the end of the simulation.

For illustration, consider Table 4: the value preservation problem consists of a program fragment, created from program locations 2, 3, and 4, and the value predicate $\phi \equiv \forall I \in R \,.\, val_{\text{pre}\langle I\rangle}(eax) = val_{\text{post}\langle I\rangle}(ecx - 1)$. To use the Simplify theorem proving oracle, the formula shown in Table 4 is generated from program fragment R.

Instruction sequence:

```
2: ebx   =   0x400000
3: nop
4: ecx   =   eax + 1
```

Value predicate:
$\forall I \in R \,.\, val_{\text{pre}\langle I\rangle}(eax) = val_{\text{post}\langle I\rangle}(ecx - 1)$

Simplify formula:

```
(IMPLIES (AND (EQ ebx_1 4194304)
              (EQ ecx_4 (+ eax_pre 1) )
              (EQ ecx_post ecx_4)
         )
         (EQ eax_pre (- ecx_post 1) )
)
```

Table 4: Example of Simplify query corresponding to program fragment R and value predicate ϕ.

5 Malicious Behavior Extraction

I turn now to the problem of automatically constructing malspecs. The fundamental problem is to derive a *representative malspec* from one or more programs that are known to be malicious. A representative malspec is a malspec that matches (using the algorithm from Section 4) against the program from which it was derived as well as against all its variants. An additional requirement for the representative malspec is to generate few or no false positives. This requires the extraction tool to analyze the malicious sample(s) from which to create a malspec that is generic enough to detect variants, yet specific enough to match few or no benign programs.

A malspec is defined by two components, the nodes labeled with system calls and the edges labeled with state transformers describing relations between the system calls. The task of the malspec extractor can be divided into two corresponding subtasks:

1. **Determine a part of the program that contains specific malicious behavior.** This step identifies the sequences of system calls that perform a malicious behavior during the execution of the program. Then the program fragment containing all of the program paths generating the malicious sequences of system calls is isolated for the second analysis step.

2. **Extract the most general set of state transformers.** Given the program fragment identified in the previous step, the goal is to extract a malspec that captures the system calls in the program fragment and the data flow dependencies between them.

Step 1: Isolating the malicious behavior. I describe here two approaches to identifying the part of the program that exhibits the malicious behavior of interest. If multiple malicious samples are available, the malicious behavior can be defined as the behavior common to all of the samples under analysis. The goal is to locate the sequences of system calls that are common to all of the malicious samples. This approach works particularly well if the malicious samples are variants of each other, for example obtained by applying obfuscation transformations or by evolving from a common core through addition of new functionality. In such cases, the extractor can accurately identify the malicious behavior that was preserved across variants.

Consider, for example, the case of the Beagle e-mail virus [56]. To propagate via e-mail, Beagle uses its own SMTP implementation to send e-mail messages. Early versions had a very simple implementation consisting of a sequence of `send` commands (one for each step of the SMTP protocol), repeated for each e-mail address. Later versions added error checking and then separated the networking code from the message composition code by placing them in distinct functions. By comparing the Beagle variants, the malspec extractor can identify the sequence of `send` commands (together with their arguments) that is common among the variants. Furthermore the malspec extractor should locate the code for constructing the list of e-mail addresses, since this list controls the execution of the SMTP-related commands.

The problem of extraction a malspec from multiple different malicious samples is reducible to the problem of subgraph isomorphism. The system control flow graphs of the malicious samples are reduced to graphs containing only system calls by replacing all other nodes (and their incoming and outgoing edges) with ϵ-edges from their predecessor nodes to their successor nodes. If necessary, ϵ-reduction is performed to simplify the resulting graphs. Then the problem of extracting a malspec becomes the problem of finding the common subgraph across the simplified control flow graphs that contain only system calls. Although subgraph isomorphism is known to be NP-complete [84], there are a number of heuristics that can be used to obtain adequate solutions [88–90].

The second approach addresses the case when only one malicious sample is available (or, equivalently, when there are multiple, unrelated malicious samples whose joint analysis would lead to malspecs with high false positive rates). The goal is, again, to identify the system call subgraph containing some malicious behavior of interest. In this case I use input from a human expert to guide the extraction algorithm toward a certain class of system calls. This approach uses the fact that, in most modern operating systems, system calls can be placed into disjoint classes based on the resources they control. For example, a simple categorization would divided system calls into those that operate on files, those that operate on network connections, and those that operate on processes and other non-persistent objects. Each such class of system calls is characterized by rules that constrain the valid usage of the system calls in the class. Often these rules can be expressed using a temporal logic derived from semantic information about the manipulated objects (e.g., you have to `open` a file before you can `read` from it) or derived from the protocol semantics of the data communications that these system calls facilitate (e.g., the Simple Mail Transport Protocol [98] requires a `HELO` command to be sent before any other commands). Using the temporal constraints over system calls from a particular class of interest, one can extract the particular system call sequences that are used in the malicious sample. Conceptually, the extraction problem can be modeled as a language intersection problem, where the language of all system call sequences generated by the malicious sample is intersected with the language of all system call sequences that fulfill the constraints of the given class.

Step 2: Obtaining a malspec from a system call subgraph. With the system call subgraph obtained in the previous step, one can isolate the program fragment that contains all the instructions relevant to the malicious behavior exhibited in this system call subgraph. This program fragment is the union of all of the backward slices from the system calls in the subgraph. The resulting program fragment contains only the program code that affects any of the arguments of the system calls of interest.

Assume that the program fragment contains the minimum number of instructions necessary to encode the data flow between the system calls. Then a set of predicates for the malspec is computed by summarizing this minimal program fragment. Symbolic execution [79] provides the conceptual framework for deriving these predicates. The arguments to each system call in the program fragment, as well as the values of those arguments after the system call, are uniquely associated with a symbolic variable α_i, for some i. Then the predicates are the formulas obtained by symbolically executing the program fragment using symbolic variables as input values and then capturing the symbolic program state before each system call.

If the program fragment is not minimal, this means there are instructions that are redundant in the current program fragment. The problem is similar to the one solved by optimizing compilers that attempt to find the most concise (or otherwise optimal under some target metric) code sequence. A supercompiler can provide the answer in this case [109].

6 A Hybrid Static–Dynamic Approach to Malicious Behavior Detection

Static analysis is particularly well suited as the building block of a security system, since it provides information that is an overapproximation of the truth. Thus, it is feasible to build a malicious behavior detector using static techniques since it will generate alerts when a program *might* contain malicious functionality. This means that such a malware detector catches all of the programs actually containing the malicious functionality, while it also raises false alarms about benign programs. These false alarms are undesirable as they can detract the user from the real threats, leading the user to distrust the malware detector in the long term. The goal of the hybrid approach is to balance the safety of static techniques with the precision of dynamic techniques.

Dynamic analysis monitors a program's execution by receiving events about the program state. The benefit of dynamic analysis is that the events provide true information about the current execution. These events can be made as detailed as possible to eliminate any need for approximation. The downside is that dynamic analysis is necessarily an incomplete technique, since it analyzes one execution at a time. The immediate implication is that *every* execution of the program has to be monitored in order to have a safe solution from a security perspective. Since runtime monitoring and dynamic analysis are not cost-free, a dynamic-only approach incurs a continuous overhead that is unacceptable in most scenarios.

The key insight to a hybrid, static–dynamic approach is that the precision of dynamic analysis is required only in the program locations where static analysis fails (i.e., where it overapproximates). Conversely, the continuous overhead of dynamic analysis can be restricted to the program locations where static analysis is imprecise. In the case of the malicious behavior detection algorithm from Section 4, imprecision at static analysis time can arise either from the control-flow oracle (e.g., invalid paths) or from the predicate-verification oracle (e.g., behavior overapproximation). The solution for such cases when imprecision is detected is to configure the runtime monitor to stop the program's execution at the location where static analysis lost precision, determine whether the current execution fulfills or not the static analysis predicate, and then take appropriate action. If the current execution indicates that the program state (e.g., a function pointer) fulfills the static analysis predicate, then the program does contain a malicious behavior and its execution should be stopped. If the current program state does not fulfill the predicate, then the runtime monitor can only say that this particular execution does not exhibit the malicious behavior. These runtime checks could also be performed via instrumentation similar to Inline Reference Monitors (IRM) [106].

I illustrate this approach using the malspec from Figure 5. Assume that the malware detector based on static analysis finds a complete match between the malspec and some program P, with the caveat that the predicate $\phi_{1,2}$ could not be precisely validated against a corresponding program path. This means that there exists some program path p from location l_1 to l_2, where l_1 matches system call σ_1 and l_2 matches system call σ_2, such that $p \models \phi_{1,2}|_{\mathcal{B}}$ with a loss of precision at some program location l on the path. The path can be split into three components $p_1 \cdot \langle l \rangle \cdot p_2$. Similarly, the predicate $\phi_{1,2}|_{\mathcal{B}}$ can be divided into three parts, $\phi_{1,2}|_{\mathcal{B}} \equiv \gamma_1 \wedge \rho \wedge \gamma_2$, so that $p_1 \models \gamma_1$, $\langle l \rangle \models \rho$, and $p_2 \models \gamma_2$. Since l is the only point of precision loss, the validity of the formula $\langle l \rangle \models \rho$ has to be checked at runtime. The dynamic analyzer is then given the following rule for runtime enforcement:

If execution reaches program point l and ρ holds, then stop execution.

The hybrid static–dynamic malicious behavior detection operates as follows:

1. Apply Algorithm 1 for static detection.
2. Determine the set L of program locations where static analysis loses precision.
3. For each program location $l \in L$, compute:
   - A predicate α_l describing the prefix of the path on which precision is lost.
   - A predicate ρ_l describing the condition under which the static analysis result on that path is valid.
4. For each program location $l \in L$, convert the condition $(pc = l) \wedge \alpha_l$ into an event E_l for the runtime monitor.
5. Execute the program under monitoring. If an event E_l fires, and if ρ_l holds true for the current program state, alert that the program might be malicious.

7 Theoretical Limits of Behavior-Based Malware Detection

There are two theoretical questions I want to address in regard to the language of malspecs and the associated detection algorithm. First, there is the question of soundness and completeness with respect to the set of all the variants of a program. Second, given this language and the detection algorithm of Section 4, I would like to assess their resilience to more complex obfuscation attacks.

The detection algorithm is sound if any two programs P and P' that match the same malspec are variants of each other. The detection algorithm is complete if any two programs P and P' that are variants of each other match the same malspec. To answer both of these question, a formal definition of *variant* is necessary. In Section 1, variants were defined in terms of observable behavior. I offer two possible definitions of observable behavior. For both of these definitions I assume that the program is type-safe and does not have vulnerabilities (such as buffer overflows) that could lead to execution of arbitrary code.

Definition 2 [OBSERVABLE BEHAVIOR]
The observable behavior of a process is the set of all system call traces it can generate during execution.

Definition 3 [SECURITY-OBSERVABLE BEHAVIOR]
The security-observable behavior of a process is the set of all of the system call traces it can generate during execution such that they affect security-relevant system state.

To understand the difference between these two definitions, consider the C code fragment from Figure 2. It could generate the system call trace in Listing 5. Another program could generate the system call trace shown in Listing 6. In both cases, one of the end results is the copying of the file named `.\foo.exe` to the `C:\Windows` directory under the new name of `winlogon.exe`. In the second trace, the program uses an intermediate file, thus performing the copy in two steps. Under the observable behavior definition, the programs are not variants, since they differ in one system call. Under the security-observable behavior definition, if the directory `C:\Windows` is considered security-sensitive and the directory `C:\Temp` is not, then the two programs are variants of each other.

The second theoretical goal is to determine the power of the detection algorithm, possibly expressed in computational complexity terms. The power of the detection algorithm directly determines the ease with which a malware writer can evade detection. The current state of measuring the strength of program understanding for malware detection is similar to the classic cryptographic view that mandates that it is impossible to design a crypto code that cannot be broken [108]. The cryptography world has since moved away from a pure entropy-based measure of cryptographic

```
GetWindowsDirectory( ... )
CopyFile( ".\foo.exe", "C:\Windows\winlogon.exe" )
```

Listing 5: Simple system call trace for the example from Figure 2.

```
GetWindowsDirectory( ... )
CopyFile( ".\foo.exe", "C:\Temp\bar.exe" )
CopyFile( "C:\Temp\bar.exe", "C:\Windows\winlogon.exe" )
```

Listing 6: Another system call trace that is equivalent to Listing 5 only under the definition of security-observable behavior.

strength to complexity theoretical measures that quantify cryptographic strength in relation to the computational power of the adversary [87].

I would like to explore a similar approach to malicious code detection, where the adversary (the malware writer) is constrained by some computational bounds. One possible approach, suggested by recent results [91, 92], is to tie the complexity of any transformation performed by the attacker to the complexity of deobfuscation expressed as abstract interpretation over appropriately chosen abstract domains.

8 Current Status

I list here the work that I have performed up to now towards a complete behavior-based malware detection system.

A core detection algorithm. I implemented Algorithm 1 using the disassembler IDAPro [93] as control-flow oracle. The current detector is interprocedural to handle outlining transformations, but it is context-insensitive which can some times lead to an increased number of false positives.

An explicit deobfuscation oracle. A set of simple checks for `nop` and pairs of save–restore instructions are the basis for this oracle.

A library of state transformers for the Intel IA-32 (x86) ISA. The Intel instruction set contains more than 300 instructions. I wrote state transformers for the top-100 most frequently used instructions. This subset provided enough coverage to handle a large number of programs, both malicious and benign.

A value-preservation oracle based on random execution, the theorem prover Simplify, and the bounded model checker UCLID. This implementation provides me with the tools to pose arbitrary queries about instruction sequences. For example, to query the theorem prover Simplify about a code fragment, the code is converted (using the state transformers) into a Simplify formula that is then integrated into the given query.

A processor emulator as an instruction-level runtime monitor. I adapted the Qemu [85] Intel-compatible processor emulator for use in monitoring processes running under an emulated Microsoft Windows environment. This tool allows tracing at the instruction granularity, at memory-access granularity, and at system call granularity.

A malware normalizer. Using the trace-enabled emulator, I implemented a malware normalizer that can handle packed binaries. Packed (or compressed) binaries are a significant threat to static analysis tools, since this type of obfuscation effectively hides the program code. Using this normalizer, any packed binary can be automatically unpacked. When used in combination with existing anti-virus tools, the malware normalizer leads to increased detection rates.

The implementation of the core detection algorithm coupled with the explicit deobfuscation oracle was used as a first prototype of the behavior-based malware detector with great success [14]. When compared against commercial anti-virus tools, this prototype proved to be significantly more resilient to common obfuscation transformations.

The value-preservation oracle has been integrated into the core detection engine [15]. The resulting behavior-based malware detector is powerful enough to detect multiple variants from the same malware family with only one malspec, in contrast with existing anti-virus tools that require one signature per malware variant. Moreover, this semantics-aware malware detection algorithm is resilient to common obfuscations used by hackers.

9 Planned Work

This section describes the major milestones that are necessary to achieve the goals I proposed in Section 1. I designed each milestone to be as close as possible to the amount of novelty appropriate for an academic paper. I grouped the milestones into short-term, medium-term, and long-term goals, where short-term means within 6 months, medium-term within 1–1.5 year, and long-term represents anything beyond 1.5 years. Long-term goals might not be achievable in the time frame between the preliminary exam and the Ph.D. defense, and I hence consider them optional.

Short-Term Goals

- *Malspec extraction from multiple malicious samples.* This task addresses the problem of deriving malspecs from malicious variants from the same family. This will require the completion of two subtasks:
  - A system to find common subgraphs containing malicious sequences of system calls. This problem can be solved using a subgraph isomorphism heuristic or a learning algorithm.
  - A system to derive predicates with uninterpreted variables from sequences of instructions on paths between system calls. A symbolic execution engine for the Intel IA-32 (x86) architecture is the necessary component in this case.
- *Malspec extraction from a single malicious sample.* This task builds on the infrastructure from the previous step to allow the extraction of malicious behavior from a single malicious sample. In addition to the use of the symbolic execution engine of the previous task, three subtasks are part of this goal:
  - The creation of a complete set of operating system state transformers for system calls. In particular, these transformers are necessary for the Microsoft Windows platform.
  - The development of a set of rules describing valid sequence of system calls, as well as rules describing network protocol communications in terms of sequences of system calls.
  - A tool to perform intersection between the language of valid system call sequences and the language of system calls generated by a program.

Medium-Term Goals

- *Low-cost on-demand runtime monitoring.* The most important part in the development of a runtime monitor is the overhead incurred during monitoring. The hybrid analysis holds promise to reduce the monitoring overhead, but it requires a different monitoring infrastructure. Specifically, the monitor has to be able to record and interrupt the execution of the program at any point, without a heavyweight event infrastructure.
- *Dynamic detection technique.* I plan to build a runtime monitor that checks the program execution against a malspec. This purely dynamic scenario corresponds to the case when static analysis cannot recover any information from the program. This dynamic malware detector will establish a baseline for the performance assessment of the hybrid detector.
- *Hybrid detection technique and implementation.* The hybrid detection technique uses the lightweight runtime monitor from the previous stage. It builds on two subcomponents:
  - A tool to determine when static analysis loses precision.
  - A system to convert static analysis queries into dynamic event-collection rules at the points where precision is lost.

Long-Term Goals (OPTIONAL)

- *Theoretical results.* This task includes the proofs of completeness and soundness of the behavior-based malware detection system, as well as computation complexity measures for the detector and for the malicious code.
- *Further exploration of the uses of malspecs.*
  - *Malicious behavior elimination.* Malicious behavior elimination refers to the automatic editing of a program to excise a malicious behavior, without affecting the remainder of a program's functionality.
  - *Automatic recovery from malicious behavior.* Any dynamic technique has the downside of possibly allowing the host system to be maliciously affected by the monitored program. The goal is to design techniques to recover from malicious activities automatically either through checkpointing and rollback or through "undo" operations that reverse the state changes introduced by the malicious code.

9.1 Timeline

Nov. 2005 – Feb. 2006	Malspec extraction from multiple malicious samples, and preparatory work for malspec extraction from a single sample.
Mar. 2006 – May 2006	Malspec extraction from a single malicious sample, and initial work on the low-cost runtime monitor.
June 2006 – July 2006	Implementation of the runtime monitor and event collection system.
Aug. 2006 – Oct. 2006	Development of the hybrid detection technique.
Nov. 2006 – Feb. 2007	Theoretical results.
Feb. 2007 – May 2007	Thesis writing, interview season.
May 2007	Thesis defense, graduation!

References on the Theory of Malware Detection

[1] D.M. Chess and S.R. White. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*, 2000.

[2] Frederick B. Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6:22–35, 1987.

[3] Frederick B. Cohen. Computational aspects of computer viruses. *Computers and Security*, 8:325–344, 1989.

[4] Kimmo Kauranen and Erkki Mäkinen. A note on Cohen's formal model for computer viruses. Technical Report A-1990-3, University of Tampere, 1990.

[5] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323 – 337, December 1992.

[6] Ferenc Leitold. Mathematical model of computer virus. In *Proceedings of the 6th International Virus Bulletin Conference (VB1996)*, pages 133–148, Brighton, UK, September 1996.

[7] Erkki Mäkinen. Comment on 'A framework for modelling trojans and computer virus infection'. *The Computer Journal*, 44(4):321–323, November 2001.

[8] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[9] Harold Thimbleby, Stuart Anderson, and Paul Cairns. A framework for modelling trojans and computer virus infection. *The Computer Journal*, 41(7):444–458, 1999.

[10] Harold Thimbleby, Stuart Anderson, and Paul Cairns. Reply to "Comment on 'A framework for modelling trojans and computer virus infection' " by E. Mäkinen. *The Computer Journal*, 44(4):324–325, March 2001.

References on Static Methods of Malware Detection

[11] William Arnold and Gerald Tesauro. Automatically generated Win32 heuristic virus detection. In *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*, pages 51–60, September 2000.

[12] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS'01)*, Indianapolis, IN, USA, March 2001.

[13] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the IEEE 4th International Workshop on Enterprise Security (WETICE'99)*, Stanford University, CA, USA, June 1999. IEEE Press.

[14] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, August 2003.

[15] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, pages 32–46, Oakland, CA, USA, May 2005. IEEE Computer Society.

[16] Stever Crocker and Maria M. Pozzo. A proposal for a verification-based virus filter. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy (Oakland'89)*, pages 319–324, Oakland, CA, USA, May 1989.

[17] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In Klaus Julisch and Christopher Krügel, editors, *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187, Vienna, Austria, July 2005. Springer-Verlag.

[18] Wei-Jen Li, Ke Wang, and Salvotore J. Stolfo. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Systems, Man, and Cybernetics (SMC) Workshop on Information Assurance*, pages 64–71, West Point, NY, June 2005. United States Military Academy.

[19] R.W. Lo, K.N. Levitt, and R.A. Olsson. MCF: A malicious code filter. *Computers & Society*, 14(6):541–566, 1995.

[20] Srinivas Mukkamala, Andrew Sung, Dennis Xu, and Patrick Chavez. Static analyzer for vicious executables (SAVE). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 326–334, Tucson, AZ, USA, December 2004.

[21] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (Oakland'01)*, pages 38–49, May 2001.

[22] Salvotore J. Stolfo, Wei-Jen Li, and Ke Wang. Fileprint analysis for malware detection. Published online at http://worminator.cs.columbia.edu/papers/2005/WormPaper-Final.pdf. Last accessed: 12 Oct. 2005., June 2005.

[23] Peter Szor. *The Art of Computer Virus Research and Defense*, chapter 11, pages 425–494. Addison-Wesley, 2005.

[24] G. J. Tesauro, J. O. Kephart, and G. B. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5–6, August 1996.

[25] Jianyun Xu, Andrew H. Sung, Patrick Chavez, and Srinivas Mukkamala. Polymorphic malicious executable scanner by API sequence analysis. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems (HIS 2004)*, pages 378–383, Kitakyushu, Japan, December 2004. IEEE Computer Society.

References on Anomaly-Based Methods of Malware Detection

[26] Vesselin Bontchev. Possible virus attacks against integrity programs and how to prevent them. In *Proceedings of the 1992 Virus Bulletin Conference (VB1992)*, pages 131–141, 1992.

[27] D. Michael Cai, James Theiler, and Maya Gokhale. Detecting a malicious executable without prior knowledge of its patterns. In B. V. Dasarathy, editor, *Proceedings of the Defense and Security Symposium 2005, Conference on Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security*, volume 5812, pages 1–12. The International Society for Optical Engineering (SPIE), March 2005.

[28] Federick B. Cohen. A note on high-integrity PC bootstrapping. *Computers & Security*, 10(6):535–539, 1991.

[29] Frederick B. Cohen. *A Short Course on Computer Viruses*. Wiley, 2nd edition, 1994.

[30] Lein Harn, Hung-Yu Lin, and Shoubao Yang. A software authentication system for the prevention of computer viruses. In *Proceedings of the ACM Annual Conference on Computer Science*, pages 447–450. ACM Press, March 1992.

[31] E. Okamoto and H. Masumoto. ID-based authentication system for computer virus detection. *Electronic Letters*, 26(15):1169–1170, July 1990.

[32] Yisrael Radai. Checksumming techniques for anti-viral purposes. In *Proceedings of the 1991 Virus Bulletin Conference (VB1991)*, pages 39–68, 1991.

[33] Peter Szor. Attacks on Win32. In *Proceedings of the 1998 Virus Bulletin Conference (VB1998)*, pages 57–84, Munich, Germany, October 1998.

[34] Trusted Computing Group. *TCG Specification Architecture Overview Revision 1.2*, April 2004.

References on Dynamic Methods of Malware Detection

[35] M. Debbabi, M. Girard, L. Poulin, M. Salois, and N. Tawbi. Dynamic monitoring of malicious activity in software systems. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS'01)*, pages 1–10, Indianapolis, IN, USA, March 2001.

[36] Daniel Guinier. Computer ”virus” identification by neural networks: An artificial intelligence connectionist implementation naturally made to work with fuzzy information. *ACM SIGSAC Review*, 9(4):49–59, Fall 1991.

[37] Kurt Natvig. Sandbox technology inside AV scanners. In *Proceedings of the 2001 Virus Bulletin Conference*, pages 475–487. Virus Bulletin, September 2001.

[38] Kurt Natvig. Sandbox II: Internet. In *Proceedings of the 2002 Virus Bulletin Conference*, pages 1–18. Virus Bulletin, 2002.

[39] Frederic Periot and Peter Ferrie. Principles and practice of X-raying. In *Proceedings of the 2004 Virus Bulletin Conference (VB2004)*, pages 51–65, September 2004.

[40] Martin Salois and Robert Charpentier. Dynamic detection of malicious code in COTS software. In *Proceedings of the Information Systems Technology Panel (IST) Symposium on Commercial Off-the-Shelf Products in Defence Applications ”The Ruthless Pursuit of COTS”*, pages 16–1—16–13, Brussels, Belgium, April 2000. NATO Research and Technology Organization.

References on Languages for Malware Detection

[41] David M. Chess. Virus verification and removal tools and techniques. In *Proceedings of the Fifth International Computer Virus and Security Conference*, page 755, 1992.

[42] ClamAV. Creating signatures for ClamAV. Published online at http://www.clamav.net/doc/latest/signatures.pdf. Last accessed on 16 Oct. 2005.

[43] Jed McNeil and Lisa Milburn. Advanced virus detection technology for the next millenium. *DataBus*, 40(2):14–20, February–March 2000.

[44] Peter V. Radatti. The CyberSoft virus description language. Published online at http://www.cybersoft.com/whitepapers/papers/cvdl.shtml. Last accessed on 16 Oct. 2005., August 1995.

[45] Markus Schmall. *Classification and identification of malicious code based on heuristic techniques utilizing Meta languages*. PhD thesis, University of Hamburg, Hamburg, Germany, 2003.

[46] Morton G. Swimmer. Response to the proposal for a C-virus database. *SIGSAC Review*, 8(1):1–5, April 1990.

References on the Generation of Signatures for Malware Detection

[47] Peter Shaohua Deng, Jau-Hwang Wang, Wen-Gong Shieh, Chin-Pin Yen, and Cheng-Tan Tung. Intelligent automatic malicious code signatures extraction. In *Proceedings of the IEEE 37th Annual International Carnahan Conference on Security Technology*, pages 600–603, October 2003.

[48] Jeffrey O. Kephart. Method and apparatus for evaluating and extracting signatures of computer viruses and other undesirable software entities. U.S. Patent 5,452,442, September 1995.

[49] Jeffrey O. Kephart and William C. Arnold. Automatic extraction of computer virus signatures. In Richard Ford, editor, *Proceedings of the Fourth Virus Bulletin International Conference*, pages 178–184. Virus Bulletin Ltd., 1994.

Malware References

[50] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salois, and N. Tawbi. Detection of malicious code in COTS software: A short survey. In *Proceedings of the 1st International Software Assurance Certification Conference (ISACC'99)*, Washington D.C., USA, March 1999. IEEE Press.

[51] John Canavan. Reduce, reuse, recycle: W32/Orpheus. *Virus Bulletin*, pages 6–8, January 2005.

[52] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04)*, pages 34–44, Boston, MA, USA, July 2004. ACM SIGSOFT, ACM Press.

[53] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus von Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61):published online at http://www.phrack.org. Last accessed: 16 Jan. 2004, August 2003.

[54] Peter Ferrie. How Dumaru? *Virus Bulletin*, pages 4–9, March 2004.

[55] Peter Ferrie and Heather Shannon. It's Zell(d)ome the one you expect. *Virus Bulletin*, pages 7–11, May 2005.

[56] Jason Gordon. Lessons from virus developers: The Beagle worm history through April 24, 2004. In *SecurityFocus Guest Feature Forum*. SecurityFocus, May 2004. Published online at http://www.securityfocus.com/guest/24228. Last accessed: 9 Sep. 2004.

[57] Mikko Hypponen. It's a record! Published online at http://www.f-secure.com/weblog/archives/archive-092005.html#00000658 as part of the *F-Secure: News from the Lab* weblog. Last accessed: 4 Oct. 2005.

[58] Myles Jordan. Dealing with metamorphism. *Virus Bulletin*, pages 4–6, October 2002.

[59] LURHQ Threat Intelligence Group. Sobig.a and the spam you received today. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig.html. Last accessed on 16 Jan. 2004.

[60] LURHQ Threat Intelligence Group. Sobig.e - Evolution of the worm. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig-e.html. Last accessed on 16 Jan. 2004.

[61] LURHQ Threat Intelligence Group. Sobig.f examined. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig-f.html. Last accessed on 16 Jan. 2004.

[62] McAfee AVERT. Virus information library. Published online at http://us.mcafee.com/virusInfo/default.asp. Last accessed: 16 Jan. 2004.

[63] McAfee AVERT. W32/Gaobot.worm.ali. In *Virus Information Library*. McAfee, May 2004. Published online at http://vil.nai.com/vil/content/v_125006.htm. Last accessed: 4 Oct. 2005.

[64] McAfee AVERT. W32/Netsky.c@MM. In *Virus Information Library*. McAfee, February 2004. Published online at http://vil.nai.com/vil/content/v_101048.htm. Last accessed: 27 Sep. 2005.

[65] G. McGraw and G. Morrisett. Attacking malicious code: Report to the Infosec research council. *IEEE Software*, 17(5):33 – 41, Sept./Oct. 2000.

[66] Peter Morley. Processing virus collections. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*, pages 129–134, September 2001.

[67] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, January 1997.

[68] Rajaat. Polymorphism. *29A Magazine*, 1(3), 1999.

[69] P.K. Singh and A. Lakhotia. Analysis and detection of computer viruses and worms: An annotated bibliography. *ACM SIGPLAN Notices*, 37(2):29–35, February 2002.

[70] Peter Ször and Peter Ferrie. Hunting for metamorphic. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*, pages 123 – 144, Prague, Czech Republic, September 2001. Virus Bulletin.

[71] Franklin Veaux. Analysis of a VX2 adware infection. Published online at http://www.xeromag.com/vx2.html. Last accessed on 22 Oct. 2005.

[72] z0mbie. Automated reverse engineering: Mistfall engine. Published online at http://z0mbie.host.sk/autorev.txt. Last accessed: 16 Jan. 2004.

[73] z0mbie. RPME mutation engine. Published online at http://z0mbie.host.sk/rpme.zip. Last accessed: 16 Jan. 2004.

Static Analysis References

[74] Gogul Balakrishnan and Tom Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC'04)*, pages 5–23, 2004.

[75] S. Chandra and T.W. Reps. Physical type checking for C. In *ACM SIGPLAN - SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 66 – 75. ACM Press, September 1999.

[76] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237, 1998.

[77] Christina Cifuentes and A. Fraboulet. Interprocedural dataflow recovery of high-level language code from assembly. Technical report, University of Queensland, 1997.

[78] Christopher Paul Colby. *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations*. PhD thesis, Carnegie Mellon University, August 1996.

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

[80] Arun Lakhotia and Eric Uday Kumar. Abstract stack graph to detect obfuscated calls in binaries. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 17–26, Chicago, IL, USA, September 2004. IEEE Computer Society.

[81] George Ciprian Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

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

[83] Z. Xu. *Safety-Checking of Machine Code*. PhD thesis, University of Wisconsin, Madison, 2000.

Other References

[84] Mikhail J. Atallah. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 1999.

[85] Fabrice Bellard. Qemu. Published online at http://fabrice.bellard.free.fr/qemu/. Last accessed on 16 Aug. 2005.

[86] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[87] G. Brassard. A note on the complexity of cryptography. *IEEE Transactions on Information Theory*, 25:232–233, 1979.

[88] J. K. Cheng and T. S. Huang. A subgraph isomorphism algorithm using resolution. *Pattern Recognition*, 13:371–379, 1981.

[89] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17:51–64, 1970.

[90] Jordi Cortadella and Gabriel Valiente. A relational view of subgraph isomorphism. In *Proceedings of the 5th International Seminar on Relational Methods in Computer Science*, pages 45–54, 2000.

[91] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Proceedings of the 3rd IEEE International Conference on Software Engineeering and Formal Methods* (*SEFM '05*), pages 301–310, 2005.

[92] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming* (*ICALP '05*), volume 3580 of *Lecture Notes in Computer Science*, pages 1325–1336. Springer-Verlag, 2005.

[93] DataRescue sa/nv. IDA Pro – interactive disassembler. Published online at http://www.datarescue.com/idabase/. Last accessed on 3 Feb. 2003.

[94] D. Detlefs, G. Nelson, and J. Saxe. The Simplify theorem prover. Published online at http://research.compaq.com/SRC/esc/Simplify.html. Last accessed on 10 Nov. 2004.

[95] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.

[96] Mark Handley, Christian Kreibich, and Vern Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium (Security'01)*, pages 115–131, Washington, D.C., USA, August 2001. USENIX.

[97] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*.

[98] J. Klensin. *Simple Mail Transfer Protocol*.

[99] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478, Boston, MA, USA, July 2004. Springer-Verlag.

[100] Microsoft. Portable executable and common object file format specification. Published online at http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx. Last accessed 14 Oct. 2005.

[101] Microsoft. Signing and checking code with authenticode. Published online at http://msdn.microsoft.com/workshop/security/authcode/authenticode_ovw_entry.asp. Last accessed on 14 Oct. 2005.

[102] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[103] George Ciprian Necula. Proof-carrying code. In *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI'96)*, pages 229–243, Seattle, WA, USA 1996.

[104] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998.

[105] Roger Riordan. Polysearch: An extremely fast parallel search algorithm. *Proceedings of the 1992 Computer Virus and Security Conference*, pages 631–640, 1992.

[106] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30 – 50, February 2002.

[107] Adi Shamir. Identity-based cryptosystems and signature schemess. In *Proceedings of the CRYPTO 1984 on Advances in Cryptology*, pages 47–53, Santa Barbara, CA, USA, 1985. Springer-Verlag.

[108] C. E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[109] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.