# Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code

Jesse C. Rabek Roger I. Khazan Scott M. Lewandowski Robert K. Cunningham

Massachusetts Institute of Technology Lincoln Laboratory 244 Wood Street Lexington, MA 02420-9108 {jesrab, rkh, scl, rkc}@ll.mit.edu

## ABSTRACT

This paper presents DOME, a host-based technique for detecting several general classes of malicious code in software executables. DOME uses static analysis to identify the locations (virtual addresses) of system calls within the software executables, and then monitors the executables at runtime to verify that every observed system call is made from a location identified using static analysis. The power of this technique is that it is simple, practical, applicable to real-world software, and highly effective against injected, dynamically generated, and obfuscated malicious code.

## **Categories and Subject Descriptors**

D.2.4 [**Software Engineering**]: Software/Program Verification – *Model checking*;

D.4.6 [**Operating Systems**]: Security and Protection – *Invasive* software (e.g., viruses, worms, *Trojan horses*), *Authentication*;

K.6.5 [Management Of Computing And Information Systems]: Security and Protection – Invasive software (e.g., viruses, worms, Trojan horses), Authentication.

### **General Terms**

Algorithms, Design, Security, Verification.

### Keywords

Malicious code detection. Intrusion detection. Anomaly detection. Code analysis. Static analysis. Dynamic analysis. System calls. Execution monitoring.

WORM'03, October 27, 2003, Washington, DC, USA.

### 1. INTRODUCTION

This paper presents DOME<sup>1</sup>, a powerful host-based detection technique for protecting software against the following challenging classes of executable malicious code (MC):

- Injected MC, such as worms that inject their code into running software processes using buffer overflow exploits;
- **Dynamically generated MC**, such as polymorphic viruses and trojans that store their code encrypted to impede their detection and analysis, and then decrypt and execute themselves at runtime;
- **Obfuscated MC**, such as viruses, trojans, and worms that disguise their code through data manipulations and obscure calculations to impede their detection and analysis.

DOME is not tied to any specific type of code injection, dynamic generation, or obfuscation. For example, it is capable of detecting both previously seen and novel MC (such as zeroday worms). Likewise, for injected worms, DOME works regardless of whether the worms are simple or complex, singleor multi-threaded, fast or slow, loud or stealthy, blind or targeted, monomorphic or polymorphic, etc.

While DOME can be applied to different operating systems, we focus on Microsoft Windows 2000 and above, and its standard executable format, the *Win32 Portable Executable File Format* (PE) [1]. We chose this OS family because it is the most widely deployed and is frequently targeted by MC.

The key idea of DOME is to preprocess software executables to identify the locations of Win32 API<sup>2</sup> calls in the software, and then to verify that every Win32 API call observed at runtime is made from a location identified during preprocessing. The elegance of this idea is that it is simple, practical, applicable to real-world software, and highly effective against injected, dynamically generated, and obfuscated MC.

According to our study, simple static analysis can be used to reliably identify the locations of Win32 API calls in typical compiler-generated software. This is, however, not the case for the three classes of MC that we are considering: For injected MC, its Win32 API calls will not be identified in the exploited

<sup>\*</sup> This research was sponsored by the Defense Advanced Research Project Agency (DARPA) under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are not necessarily endorsed by the US Government.

Copyright 2003 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

Copyright 2003 ACM 1-58113-785-0/03/0010...\$5.00.

<sup>&</sup>lt;sup>1</sup> DOME stands for Detection of Malicious Executables.

<sup>&</sup>lt;sup>2</sup> Win32 API functions are the standard library functions of Microsoft Windows operating systems (OS). We assume that MC interacts with the OS through the Win32 API.

software because such MC is injected into the software process at runtime, so it is absent from the software executable during preprocessing. For dynamically generated and obfuscated MC, their Win32 API calls will not be identified because the identification algorithm does not emulate runtime code generation, nor does it attempt to de-obfuscate intentionally obfuscated code.

Our technique is unique in being able to detect these generic and critical classes of MC in real-world software, with virtually no false-positives or false-negatives, and with low runtime overhead – approximately 5% slowdown per API call.

When deployed on host machines, DOME will monitor the execution of designated software executables and will detect the presence of MC at runtime. The detection will occur before the MC has a chance to interact with the OS, that is, MC will be detected before it has a chance to access OS protected resources, such as files or sockets. Since execution of the detected MC can be stopped before it does any damage, DOME can protect host machines against MC.

Notice that DOME does not just detect MC, it actually pinpoints the parts of MC that result in Win32 API calls. This information can be used as a starting point for further MC analysis and can help understand and respond to MC attacks.

For MC embedded in software executables, DOME relies on the MC's attempts to avoid detection and analysis to detect it. As such, DOME is not designed to detect unobfuscated viruses and trojans whose code is embedded within a software executable, prior to the executable being preprocessed by DOME. Furthermore, DOME is limited to executable MC that uses Win32 functions, and will therefore miss MC that causes harm by corrupting, crashing, or hanging infected software. Also, DOME does not work for worms that spread using techniques other than code injection, such as script-based worms or worms that infiltrate by social engineering and spread through drive sharing. In order to ensure full protection from the MC threat, a system based on DOME should be deployed in conjunction with other detection-response systems designed to address the MC threats not covered by DOME.

The rest of the paper is organized as follows: Section 2 defines the MC space covered by DOME. Section 3 describes the DOME technique. Section 4 reports on a proof-of-concept study that we carried out to assess the feasibility of implementing DOME and its ability to detect MC. Section 5 considers different settings in which DOME can be applied. Section 6 discusses related work, and Section 7 concludes.

### 2. AREA OF COVERAGE

DOME is designed to detect the following three general classes of MC:

- 1. **Injected code** code that is introduced into a process' address space at runtime.
- 2. **Dynamically generated code** code that is created by a process at runtime.

3. **Obfuscated code** – code that is present in the process' original code but whose true intentions are hidden with obscure calculations and data manipulation.<sup>3</sup>

Most of the worms that use exploits such as buffer overflows to inject themselves into software processes fall into class 1. Polymorphic viruses, which encrypt and embed themselves inside software executables on disk, are examples of class 2. Like dynamic code generation, code obfuscation is traditionally used by viruses and trojans, not worms; however, it is likely that next-generation worms will use these sophisticated techniques to hinder their detection and subsequent analysis.

The area of coverage is further characterized by the following assumptions:

**Assumption 1:** Any injected, dynamically generated, or obfuscated code is assumed to be malicious.

This assumption is reasonable because these types of code do not typically occur in non-malicious software. This is especially true for injected code. Obfuscated code is sometimes used in software executables to protect proprietary algorithms or to prevent software from being reverse engineered. Dynamically generated code can also sometimes be found in software executables. Examples of this type of code include: stack trampolines, which facilitate the use of nested functions; just-intime compilers, which create native machine code from bytecode; and executable decompressors, which at runtime decompress previously compressed executable code loaded from disk. In our future work, we will investigate how these special cases can be addressed by DOME.

Assumption 2: MC interacts with the OS.

Most types of malicious activities, such as accessing network or file services, involve interactions with the OS; others, e.g., [2, 3], have made a similar observation. However, some malicious activities, such as denying service or corrupting data, can be done without interactions with the OS; MC that limits itself to such activities will not be detected by DOME.

**Assumption 3:** In interacting with the OS, MC uses the Win32 APIs.

Instead of using the Win32 APIs, it is possible to interact with the OS through the Windows NT native API functions. DOME can be extended to cover this type of interaction. One possible solution is to consider as malicious all Windows NT native API calls made by user-mode executables.

**Assumption 4:** When MC hides itself from detection and analysis by using dynamic code generation and obfuscation, its Win32 API usage is hidden as well.

Since the Win32 API calls made by MC embody the essence of what the MC does and how it works, if the MC's goal is to hinder detection and analysis, it makes sense for MC to hide its Win32 API usage. This is typically done either with dynamic code generation or with obfuscation. One common obfuscation technique is to use complicated calculations or in-memory code scanning to determine the address or string name of an API

<sup>&</sup>lt;sup>3</sup> This definition is not as precise as the previous two. Assumption 4 clarifies what we mean by obfuscation.

function. Another technique is to use a dynamic binding function (e.g., GetProcAddress) in non-standard ways.

**Assumption 5:** Software executables that are to be protected can be successfully disassembled, and the Win32 APIs used by these executables can be effectively monitored at runtime.

We expect that most compiler-generated software satisfies this assumption.

# **3. DETECTION TECHNIQUE**

At its core, DOME involves two steps that are applied to software executables being protected against exploitation by MC:

- 1. Preprocess each software executable to identify the instructions that call into Win32 APIs and save their virtual addresses and the API names as a model of the Win32 API calls that the executable makes.
- 2. Monitor Win32 API calls made by software executables at runtime. When a Win32 API is called, identify the instruction that produced the call and its address within the executable. Then, validate the instruction address and the API name against the model generated during the preprocessing step. If a mismatch occurs, signal detection.<sup>4</sup> At this point, a response system can protect the host by blocking the API call.

We are assuming that the software executables do not change after the preprocessing step. If software is updated, the preprocessing step must be repeated. Modifications of software executables due to a viral infection that occurs after the preprocessing step can be easily detected with an integrity checking approach based on an MD5 or SHA-1 file hash.

The introduction explains why DOME is successful at detecting injected, dynamically-generated, and obfuscated MC. We now describe the two steps in detail, and then consider how DOME must be extended to handle MC that uses the knowledge of how DOME works to bypass it.

### 3.1 Preprocessing

In the preprocessing step, software executables are disassembled and analyzed to identify the instructions that call into Win32 APIs. The virtual addresses of these instructions and the API names are then recorded. For reasons that will become clear in the next subsection, we also record the addresses of the instructions that occur immediately after the identified Win32 API calls – these are the return addresses for the Win32 API calls, and they should appear on the top of the runtime stack when the calls are made.

The identification mechanism draws a line between which Win32 API calls will be treated as normal and which as malicious at runtime. The identification mechanism should be designed so that it can see all of the Win32 API calls made by normal compiler-generated code, but none of the Win32 API calls that are intentionally hidden. Luckily, designing an identification mechanism with such a property is straightforward because the way Win32 API calls are made by "normal" code is significantly different from the way these calls appear in intentionally obfuscated code.

In compiler-generated code, Win32 APIs are typically called by referencing the appropriate entries in the import address table (IAT). The calls are either direct references to the IAT, as in "call [IAT Entry 4]," or they are indirect references that can be identified with simple static analysis.

For example, a common way for optimized code to make a Win32 API call is to load the address of the API's IAT entry into a CPU register and then issue a call instruction referencing the register. Simple backward slicing on the register from the point of the call instruction can be used to identify that this call instruction is meant to invoke a specific API.

Static analysis can also be used to identify calls to late-bound Win32 APIs, which are APIs whose addresses are determined at runtime using GetProcAddress. Upon encountering a call to GetProcAddress, the Win32 API name can be associated with the registers or memory locations that are to be bound to the Win32 API addresses.

To accommodate real-world software, the preprocessing step should be able to handle software comprised of multiple executable components, such as custom DLLs.

## **3.2 Monitoring and detection**

This step monitors the Win32 API calls made by software processes and verifies that the instruction addresses from which the calls were made and the names of the corresponding Win32 APIs were identified during the preprocessing step. There are two logical parts to this step: monitoring Win32 API calls, and validating the calls against the information recorded during the preprocessing step.

**Monitoring Win32 API calls:** A number of methods can be used to monitor the Win32 API calls made by processes [4]. In our proof-of-concept study, we chose to use the direct patching method implemented by the Detours package [5], which instruments the DLLs containing the Win32 APIs at load time. By directly patching the entry point of each Win32 API, all Win32 API calls can be monitored. Patching DLLs at load time allows software executables to be monitored selectively.

Figure 1 depicts how a call to a Win32 API occurs from a software process when the API is patched with Detours. The process makes a call into the API function (1), the first instruction of which is an unconditional jump to the Detours wrapper (2). The wrapper may execute pre-stub code before returning control to the Win32 API body (3 and 4). After the Win32 API body finishes executing, control is returned back to Detours (5), which may execute post-stub code before returning control to the caller (6). The pre-stub code is where DOME validates the Win32 API call against the information identified during the preprocessing step.

Similarly to the preprocessing step, the monitoring step should be able to handle software comprised of multiple executable components.

<sup>&</sup>lt;sup>4</sup> A version of DOME can be implemented without the preprocessing step: it can monitor Win32 API calls and determine at runtime if the calls are identifiable at the right locations in the disk copy of the software executable. This version has a higher runtime overhead and may be less accurate.



Figure 1: Detoured API Call

Validation of Win32 API calls: As was mentioned above, the validation of Win32 API calls is done by the pre-stub code of the API's wrapper. When a Win32 API call is made and the pre-stub code gets control, the top of the runtime stack is supposed to contain the return address for the call. To validate the call, DOME checks whether the return address and the API name were recorded during the preprocessing step. If they were, the wrapper control transfers to the Win32 API body. Otherwise, detection is signaled.

To handle DLL relocation (rebasing), which may occur when two or more DLLs want to be loaded into conflicting address ranges, DOME should use instruction addresses relative to the DLLs' base addresses.

### 3.3 Handling bypassability

The basic version of DOME described so far is simple, and yet is highly effective at detecting most of MC within its area of coverage. The notable exception is MC that intentionally avoids DOME and/or the underlying monitoring technique [6]. To handle such MC, DOME needs to be extended. We now outline how this can be done; we intend to make these extensions a part of our future work.

**DOME bypassability:** One way that MC may attempt to circumvent DOME is to forge the return address on the top of the runtime stack, making it appear that the call originated from one of the statically identified locations. Another way is for MC

to use the software's own instructions that call Win32 APIs, while possibly supplying its own malicious arguments. There is a number of measures that can be implemented to counter such attacks. Two promising techniques are identifying and recording static Win32 API arguments during preprocessing and then validating them at runtime, and performing runtime stack verification.

Wrappers bypassability: Any API wrapper system implemented in user-mode can be bypassed. In particular, if MC is designed with the knowledge that the detection system uses Detours, it can manipulate memory and disable the wrappers prior to calling any APIs. In addition, MC can call directly into the kernel, thus avoiding the Win32 API calls and their wrappers. On IA32 systems, calls into the kernel typically rely on a privilege change triggered by an interrupt or the sysenter instruction. One way to prevent wrappers from being bypassed is to add a kernel-level authentication mechanism that verifies that the APIs are reached only after the execution has passed through the unmodified wrappers.

### 4. PROOF-OF-CONCEPT STUDY

In order to assess the feasibility of implementing DOME and its ability to detect MC, we performed a proof-of-concept study. The specific goals of this study were to verify the following three assertions:

- 1. It is possible to identify API calls in real-world software using static analysis.
- 2. It is possible to monitor API calls at runtime and to identify the instructions responsible for the observed API calls.
- 3. Provided the above two assertions are true, DOME is able to accurately distinguish between normal code and code that is injected, dynamically generated, or obfuscated.

004013AC	ExitProcess
004013BD	GetModuleHandleA
004013FF	GetVersionExA
00401434	GetEnvironmentVariableA
00401494	GetModuleFileNameA
00401539	HeapCreate
00401578	HeapDestroy

# Figure 2: Sample output produced by the preprocessing and the monitoring steps

The preprocessing step was done using the IDA Pro disassembler [7], which, in addition to disassembling executables, also identifies and annotates instructions that make Win32 API calls. The monitoring step was implemented using the Detours wrapper package [5]. Each step produced an output file consisting of API calls and their locations, as depicted in Figure 2. The output files were then compared to identify the API calls that occurred at runtime from the locations that were not identified during the preprocessing step.

We evaluated DOME's performance on a number of benign executables, benign executables that had malicious code embedded in them, and benign executables that had malicious code injected into them at runtime.

## 4.1 Benign executables

Table 1 lists the benign samples that we used. The samples include software applications that were created using different compilers and that involve different types of resources (e.g., network and file system).

Application	Vendor	Compiler	Key Resources
Ipconfig	Microsoft	VC++	Network
Front page	Microsoft	MS Internal	File, Network, COM Interfaces
WinVNC	AT&T	Borland	Network
Acrobat	Adobe	VC++	File, DLL plugins
Mozilla	Mozilla	Gcc	Network
Notepad*	Microsoft	VC++	File
Perfmon*	Microsoft	MS Internal	Registry
Chlinst*	Microsoft	MS Internal	Registry

**Table 1. Selected Benign Executables** 

The tests were all successful: we did not observe any unexpected API calls. The only false positives we observed were due to dynamic binding of APIs, which we expected because the static analysis performed by IDA Pro does not handle this case.

 Table 2. Selected Malicious Executables

Malicious	Host	Туре	Class
Code	Application		
W32-Crypto	Notepad	Virus	Dyn. Code Gen.
W32-Simile	Perfmon	Virus/	Obfuscation
		Worm	
W32-Magistr	Chlinst	Virus/	Dyn. Code Gen.
		Worm	
W32-CTX	Eclabm13	Virus	Dyn. Code Gen.
W32-Roach	Cookie	Worm	Dyn. Code Gen.
W32-Sapphire	MS SQL server	Worm	Code Injection

### 4.2 Malicious code samples

Table 2 lists the MC samples. These consist of the viruses and worms that use dynamic code generation (polymorphism), obfuscation, and code injection. For each of the samples, the proof-of-concept implementation successfully detected API calls made by the MC.

The proof-of-concept implementation produces a trace of the Win32 API calls that were observed at runtime but that were not identified during preprocessing. This trace, in a way, "tells a story" of how the MC works, which can be used to analyze the MC further and to produce human-readable descriptions of what the MC does. As an illustration, Table 3 shows a sample of the

trace for an application infected with the W32-Simile virus and compares it with the analysis of W32-Simile presented in Virus Bulletin [8], which states that

W32-Simile is highly obfuscated and challenging to understand. The virus attacks disassembling, debugging and emulation techniques, as well as standard evaluationbased techniques for virus analysis. In common with many other complex viruses, Simile uses [entry-point obfuscation] EPO techniques.

As can be seen from Table 3 the output produced by DOME matches the human-written description of W32-Simile, yet this output was generated without human guidance.

#### **4.3** Performance overhead

In our experience, IDA Pro can statically analyze PE executables at around 5KB/s on a 600MHz Pentium machine. The Detours wrappers add around a 5% runtime overhead to each API call, which is consistent with the figures cited by [5].

### 5. DEPLOYMENT OPTIONS

DOME has been primarily designed as a host-based, online detection technique capable of monitoring and protecting realworld software. However, the technique can also be implemented in offline scanners and MC analysis tools to detect dynamically generated and obfuscated code in software executables.

### 5.1 Online detection and blocking

In this instantiation, DOME can be used to preprocess and monitor designated software executables, and can detect and stop worms injected into these executables at runtime, as well as dynamically generated and obfuscated MC embedded in these executables prior to the preprocessing step. Note that DOME is also capable of detecting both simple and complex viruses that infect software executables after they are preprocessed; however, such alterations to software executables can be detected via simpler means, such as comparing the executables' current and original hashes.

In a real-world deployment scenario, there are a number of alternative approaches to preprocessing and monitoring. Preprocessing can be done for all or selected executables, and for each installed copy separately or once for a set of installations either by a site administrator, software manufacturer, or trusted third-party. Monitoring of Win32 API calls can be done per executable, or system-wide by rewriting DLLs.

As was mentioned earlier, some software applications use obfuscation to protect proprietary algorithms or to prevent software from being reverse engineered. If such software needs to be protected by DOME, an administrator, at the time of system deployment and/or tuning, could mark detected API calls as legitimate. Also, in a military or government environment, it is reasonable to require obfuscated software to come equipped with some sort of guarantees of its behavior, which could include the list of API calls that the software makes along with their locations.

<sup>\*</sup> Our malicious code samples were embedded in these applications, so we felt that we should also analyze the original executables to verify that our system identified only the API calls made by the MC.

Table 3. Comparison for W32 Simile

Human Analysis (Virus Bulletin)	Malicious Win32 API call trace detected by DOME		
"On initial execution, the virus body will retrieve the addresses of 20 APIs that it requires for replication and for displaying the payload."	<ol> <li>013FDF09 GetProcAddress (CreateFileA) KERNEL32</li> <li>013FDF09 GetProcAddress (CreateFileMappingA) KERNEL32</li> <li>013FDF09 GetProcAddress (MapViewOfFile) KERNEL32</li> <li>013FDF09 GetProcAddress (UnmapViewOfFile) KERNEL32</li> <li>013FDF09 GetProcAddress (GetSystemTime) KERNEL32</li> <li>013FDF09 GetProcAddress (MessageBoxA) USER32</li> </ol>		
"Next the replication phase begins. It starts by searching for *.exe in the current directory, then on all fixed and mapped network drives."	0140A544 FindFirstFileA 013F7616 FindNextFileA 013F7616 FindNextFileA  013FC5B5 GetFileAttributesA (API calls infecting the file) 0140B0B4 SetFileAttributesA 013F7616 FindNextFileA 	0140ACA9 SetCurrentDirectoryA 0140A544 FindFirstFileA 013F7616 FindNextFileA  01408550 GetLogicalDriveStringsA 013F7485 GetDriveTypeA 0140ACA9 SetCurrentDirectoryA 0140A544 FindFirstFileA 013F7616 FindNextFileA 	

When a new software executable is installed and run by a user before the preprocessing step is done, an alternative version of the system can be employed: when the executable calls an API, the system can read the corresponding code on disk and, using local static analysis, determine if the API call can be identified at that location (the analysis results can be cached). If not, the system could signal detection and block the API call. The local code analysis performed at runtime will impose additional overhead and may be less accurate than full analysis performed as the preprocessing step.

The most effective way to protect a network against fast-spreading worms is to deploy DOME systems on every machine. This will protect the network from a distributed, targeted attack capable of compromising the network in only one to three generations of worm propagation.

In order to ensure full protection from the MC threat, a system based on DOME should be deployed in conjunction with other detection-response systems designed to address MC not covered by our technique, such as scripts and social-engineering worms. DOME systems can also be deployed on honeypots [9] to monitor their network services and facilitate early detection and analysis of worm-based attacks.

### 5.2 Offline software scanning

The DOME technique can also be used in an antivirus-like scanner. Such a scanner could preprocess designated software executables and then launch the executables to see if they produce any Win32 API calls from locations that were not identified.

For thorough scanning, the executables need to be driven through all possible execution paths; however, the problem of application driving is an active area of research that currently does not have practical application-independent solutions. A practical approach is to simply launch the executables and then terminate them after some small amount of time. This approach would detect MC that executes at least one Win32 API call every time its host executable is launched, which is typical of existing MC and is consistent with what we observed during our proof-of-concept study. For example, MC that uses a temporal trigger to control when its malicious body is run will typically call a time API to check the trigger conditions every time the host executable is launched.

To ensure the host system is not affected by MC during scanning, the Win32 API calls that are identified as malicious need to be blocked.

# 5.3 Online and offline analysis

DOME does not just detect MC, it actually pinpoints the instructions belonging to MC. This information can be used by an online or offline analysis tool to isolate and analyze MC. Possible goals of such analysis might be to generate detection signatures and firewall rules, to analyze the payload and trigger mechanisms, to predict propagation vectors, or to identify code lineage and perform attribution. DOME can also serve as the foundation for a tool that generates human-readable descriptions of how MC works.

# 6. RELATED WORK

Methods of detecting MC can generally be classified into one of the following two categories: *misuse detection* and *anomaly detection*. Misuse detection schemes focus on "maliciousness". They attempt to identify code characteristics and/or runtime behaviors that are defined to be malicious. Unlike misuse detection schemes, anomaly detection schemes focus on "normalcy". They attempt to identify code characteristics and/or runtime behaviors that deviate from those that are defined to be normal, i.e., non-malicious.

DOME is an anomaly detection technique. Normal runtime behavior consists of the Win32 API calls that occur from the locations that have been identified by DOME during the preprocessing step.

Many existing anomaly detection techniques, such as [3, 10-14], create models of normal behavior based on sequences of system

calls; Feng et al. [10] provide a comprehensive review of these techniques. In contrast, DOME does not use system call traces. It is unique in using the addresses of the system call instructions as the basis for its model of normal behavior. The advantages of this model include simplicity, practicality, and effectiveness.

Most anomaly detection techniques, especially those that use system calls [10, 11, 13, 14], create models of normal behavior by monitoring software at runtime. Then, when the learning phase is completed, they switch to an anomaly detection phase, during which they continue to monitor the software's execution looking for deviations from the behavior that was learned.

A limitation of these techniques is that their models include only the behavior observed during the learning phase, which is likely to be only a fraction of all of the behaviors that the software can exhibit. Unlearned behavior observed during the anomaly detection phase results in false positives [15]. DOME does not have this limitation since its models include all the non-malicious system calls that the executables can make at runtime; as a result, DOME generates virtually no false-positives.

Moreover, in comparison with observation-based anomaly detection, DOME provides a wider area of coverage. Observationbased systems are designed to detect MC intrusions that occur during the anomaly detection phase, after the models of normal behavior are learned. DOME is able to detect not only MC injected at runtime but also sophisticated MC embedded in the software executables prior to the preprocessing step.

Like DOME, the techniques of Wagner et al. [3] and Giffin et al. [12] use static analysis of software to construct models of the software's normal behavior.

The technique of Wagner et al [3] operates on source code, and makes a number of simplifying assumptions regarding the complexity of the source code. The technique constructs a global control-flow graph of the software and then converts the graph into a nondeterministic finite state automaton (NFA) or a nondeterministic pushdown automaton (NPDA) to model the sequences of system calls that the software can make. These models are complex and it is unclear whether they can be constructed for real-world software; moreover, the monitoring overhead is substantial because of the nondeterminism of the automata. The NFA model has an inherent imprecision problem: it includes system call traces that are not present in the software; if such traces are produced by MC they will not be detected. The NPDA model addresses this imprecision problem by including an abstract version of the runtime stack in the model, but this extension makes the model even more complex and results in impractical runtime overhead.

The technique of Giffin et al. was developed for securing mobile code, such as remote procedure calls [12]. It operates on executable code and creates models that are similar to the NFA model of Wagner et al. The authors suggest several program transformation techniques to reduce the amount of nondeterminism and make the model more precise; such transformations may be appropriate for mobile code, but are unlikely to be appropriate for traditional host-based software because of legal and interoperability issues.

#### 7. SUMMARY

We presented DOME, a technique for detecting injected, dynamically generated, and obfuscated MC in software executables. The results of our proof-of-concept study suggest that DOME is effective at detecting MC. The main idea of DOME is to use static analysis to identify the locations of Win32 API calls within software executables and to use these locations as a model of which Win32 API calls are allowed to occur at runtime. This basic model can be extended in a number of ways to counter MC's attempts to bypass DOME; one promising idea is to include in the model information about the Win32 API call arguments. We will pursue this and other extensions in our future work, when we implement and evaluate an online detection system based on DOME.

### 8. REFERENCES

- [1] Pietrek, M. Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format (Part I). In www.msdn.microsoft.com. 2002.
- [2] Bergeron, J., M. Debbabi, M.M. Erhioui, and B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviours. In WET ICE 99. 1999.
- [3] Wagner, and Dean. Intrusion Detection via Static Analysis. In IEEE Symposium on Research in Security and Privacy. 2001. Oakland, CA.
- [4] Kaplan, Y. *API Spying Techniques for Windows 9x, NT and 2000.* http://www.internals.com/articles/apispy/apispy.htm
- [5] G. Hunt, D.B., *Detours: Binary Interception of Win32 Functions*. 1999, Microsoft Research.
- [6] Wagner, D., and P. Soto. Mimicry Attacks on Host Based Intrusion Detection Systems. In 9th ACM Conference on Computer and Communications Security. 2002. Washington, DC, USA.
- [7] Data Rescue. *IDA Pro Disassembler*. http://www.datarescue.com/idabase/
- [8] Frédéric Perriot, P.F., Péter Ször, *Striking Similarities*, in *Virus Bulletin*. 2002. p. 4-6.
- [9] Tünnissen, J. Intrusion Detection, Honeypots & Incident Response resources. http://www.honeypots.net/
- [10] Feng, H., O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In IEEE Security and Privacy. 2003. Oakland, CA.
- [11] Sekar, R., A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-Based Anomaly Detection: A New Approach for Detecting Network Intrusions. 2002. Washington, DC, USA.
- [12] Giffin, J.T., S. Jha, and B.P. Miller. Detecting Manipulated Remote Call Streams. In 11th USENIX Security Symposium. 2002.
- [13] Ghosh, A.K., A. Schwartzbard, and M. Schatz. Learning Program Behavious Profiles for Intrusion Detection. In Usenix Workshop on Intrusion Detection and Network Monitoring. 1999. Santa Clara, CA.
- [14] Warrender, C., S. Forrest, and B. Pearlmutter. *Detecting Intrusions Using System Calls: Alternative Data Models*. In *IEEE Symposium on Security and Privacy*. 1999.
- [15] Axelsson, S. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In ACM Conference on Computer and Communications Security. 1999.