GPU-Assisted Malware

Giorgos Vasiliadis
FORTH-ICS, Greece
gvasil@ics.forth.gr

Michalis Polychronakis
Columbia University, USA
mikepo@cs.columbia.edu

Sotiris Ioannidis
FORTH-ICS, Greece
sotiris@ics.forth.gr

Abstract

Malware writers constantly seek new methods to obfuscate their code so as to evade detection by virus scanners. Two code-armoring techniques that pose significant challenges to existing malicious-code detection and analysis systems are unpacking *and* run-time polymorphism. *In this paper, we demonstrate how malware can increase its robustness against detection by taking advantage of the ubiquitous Graphics Processing Unit. We have designed and implemented unpacking and run-time polymorphism for a GPU, and tested them using existing graphics hardware. We also discuss how upcoming GPU features can be utilized to build even more robust, evasive, and functional malware.*

1 Introduction

Computer viruses, bot clients, rootkits, and other types of malicious software, collectively referred to as *malware*, abuse infected hosts to carry out their malicious activities. From the first viruses written directly in assembly language, to application-specific malicious code written in high-level languages like javascript, any action of the malware results in the execution of machine code on the compromised system's processor.

Besides the central processing unit, modern personal computers are equipped with another powerful computational device: the graphics processing unit (GPU). Historically, the GPU has been used for handling 2D and 3D graphics rendering, effectively offloading the CPU from these computationally-intensive operations.

Driven to a large extent by the ever-growing video game industry, graphics processors have been constantly evolving, increasing both in computational power and in the range of supported operations and functionality. The most recent development in the evolution chain is general-purpose computing on GPUs (GPGPU), which allows programmers to exploit the massive number of transistors in modern GPUs to perform computations that up till now were traditionally handled by the CPU. In fact, leading vendors like AMD and NVIDIA have released software development kits that allow programmers to use a C-like programming language to write general-purpose code for execution on the GPU [2, 13]. GPGPU has been used in a wide range of applications, while the increasing programmability and functionality of the latest GPU generations allows the code running on the GPU to fully cooperate with the host's CPU and memory.

Given the great potential of general-purpose computing on graphics processors, it is only natural to expect that malware authors will attempt to tap the powerful features of modern GPUs to their benefit [14]. Two key factors that affect the lifetime and potency of sophisticated malware are its ability to evade existing anti-malware defenses and the effort required by a malware analyst to analyze and uncover its functionality—often a prerequisite for implementing the corresponding detection and containment mechanisms. Packing and polymorphism are among the most widely used techniques for evading malware scanners [17]. Code obfuscation and anti-debugging tricks are commonly used to hinder reverse engineering and analysis of (malicious) code [6].

So far, these evasion and anti-debugging techniques take advantage of the intricacies of the most common code execution environments. Consequently, malware defense and analysis mechanisms, as well as security researchers' expertise, focus on IA-32, the most prevalent instruction set architecture (ISA). The ability to execute general purpose code on the GPU opens a whole new window of opportunity for malware authors to significantly raise the bar against existing defenses. The reason for this is that existing malicious code analysis systems primarily support IA-32 code, while the majority of security researchers are not familiar with the execution environment and ISA of graphics processors.

In this paper, we aim to raise awareness about the worrisome potential of GPU-assisted malware. To that end, we demonstrate the feasibility of implementing malware that utilizes the GPU to armor its code. Specifically, we present the design and implementation of GPU-based unpacking and runtime polymorphism, two techniques that pose significant challenges to existing malware detection and anal-

ysis systems. Furthermore, we discuss potential attacks and future threats that can be facilitated by next-generation GPGPU architectures.

We believe that a better understanding of the offensive capabilities of attackers can lead researchers to create more effective and resilient defenses.

2 GPGPU Programming

General-purpose computing on graphics processing units has drastically evolved in recent years. As graphics processors started becoming more powerful, programmers began exploring ways for enabling their applications to take advantage of the massively parallel architecture of modern GPUs.

Standard graphics APIs, such as OpenGL and DirectX, do not expose much of the underlying computational capabilities that graphics hardware can provide. The task of using these APIs for general-purpose computation poses challenges when non-graphics applications are attempted to be ported to the GPU. Data and variables have to be mapped to graphics objects, while algorithms must be expressed as pixel or vertex shaders, pretending to perform graphics transformations. The lack of convenient data types, basic computational functionality, and a generic memory access model renders this environment far from attractive for developers accustomed to working in traditional programming environments.

The Compute Unified Device Architecture (CUDA) introduced by NVIDIA [13] is a significant advance, exposing several hardware features that are not available via the graphics API.[1] CUDA consists of a minimal set of extensions to the C language and a runtime library that provides functions to control the GPU from the host, as well as device-specific functions and data types.

At the top level, an application written for CUDA consists of a serial program running on the CPU, and a parallel part, called a *kernel*, that runs on the GPU. A kernel, however, can only be invoked by a parent process running on the CPU. As a consequence, a kernel cannot be initiated as a stand-alone application, and it strongly depends on the process that invokes it.

Each kernel is executed on the device as many different *threads* organized in thread *blocks*. The thread blocks are executed by the *multiprocessors* of the GPU in parallel. Each multiprocessor consists of eight *stream processors*, operating on a SIMD fashion. In order to maximize the use of the multiprocessors' computational resources, a thread scheduler periodically switches from one thread block to another.

In addition to program execution, CUDA also provides appropriate functions for data exchange between the host and the device. All I/O transactions are performed over the PCI Express bus. Furthermore, memory operations can be performed through DMA in order to facilitate concurrent execution between the CPU and the GPU. A block of page-locked host memory can also be mapped into the address space of the GPU, enabling the program running on the CPU and the kernel executing on the GPU to directly access the same data.

[1] AMD offers a similar SDK for its ATI line of GPUs [2].

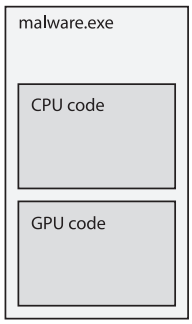

Figure 1. A GPU-assisted malware binary.

From the perspective of the malware author, a GPU-assisted malware binary contains code destined to run on different processors, as shown in Figure 1. Upon execution, the malware loads the device-specific code on the GPU, allocates a memory area accessible by both the CPU and the GPU and initializes it with any shared data, and schedules the execution of the GPU code. Depending on the design, the flow of control can either switch back and forth between the CPU and the GPU, or separate tasks can run in parallel on both processors.

A major advantage for the malware author is that the malware can be statically linked with the CUDA library into a single stand-alone executable. Thus, the malware becomes completely self-contained, without the need to install any files on the infected system. Furthermore, the execution of GPU code, as well as data transfers between the host and the device do not require any administrator privileges. In other words, the malware will run successfully even under user privileges, making it more robust and deployable.

3 Proof-of-Concept Implementation

Malware can exploit the rich functionality of modern GPUs in an plethora of ways. In this section, we describe the design and implementation of two code-armoring techniques based on GPU code. These prototypes not only demonstrate the feasibility of GPU-assisted malware, but also already pose significant challenges to existing malware analysis and detection systems.

We have chosen to implement our prototypes using NVIDIA CUDA [13], the most widely used GPGPU frame-

```
.entry _Z8unpckrPhii (
  .param .u32 __cudaparm__Z8unpckrPhii_a,
  .param .s32 __cudaparm__Z8unpckrPhii_N,
  .param .s32 __cudaparm__Z8unpckrPhii_key)
{
  .reg .u32 %r<12>;
  .reg .pred %p<4>;
  .loc    28      31      0
$LBB1__Z8unpckrPhii:
  ld.param.s32    %r1, [__cudaparm__Z8unpckrPhii_N];
  mov.u32         %r2, 0;
  setp.le.s32     %p1, %r1, %r2;
  @%p1 bra        $Lt_0_1282;
  ld.param.s32    %r1, [__cudaparm__Z8unpckrPhii_N];
  mov.s32         %r3, %r1;
  ld.param.u32    %r4, [__cudaparm__Z8unpckrPhii_a];
  mov.s32         %r5, %r4;
  add.u32         %r6, %r1, %r4;
  ld.param.s32    %r7, [__cudaparm__Z8unpckrPhii_key];
  mov.s32         %r8, %r3;
$Lt_0_1794:
//<loop> Loop body line 31, nesting depth: 1,
//estimated iterations: unknown
  .loc    28      35      0
  ld.global.u8    %r9, [%r5+0];
  .loc    28      31      0
  ld.param.s32    %r7, [__cudaparm__Z8unpckrPhii_key];
  .loc    28      35      0
  xor.b32         %r10, %r7, %r9;
  st.global.u8    [%r5+0], %r10;
  add.u32         %r5, %r5, 1;
  setp.ne.s32     %p2, %r5, %r6;
  @%p2 bra        $Lt_0_1794;
$Lt_0_1282:
  .loc    28      37      0
  exit;
$LDWend__Z8unpckrPhii:
} // _Z8unpckrPhii
```

Figure 2. The intermediate PTX code of a simple XOR-based unpacking function for NVIDIA graphics cards.

work today. An attacker can easily include multiple versions of the GPU-specific code in the same executable to keep the malware functional across different GPU architectures. In fact, supporting just the two major vendors would allow for almost complete coverage. The wide adoption of OpenCL [10], a cross-platform GPGPU framework that aims to unify vendor-specific APIs into a single one, will obviate the need for embedding different versions of the same code.

3.1 Self-unpacking Malware

Code packing is one of the most common approaches malware writers employ to protect their code and evade detection [7]. Using this technique, the code of the malware is converted to data using compression, encryption or any other data transformation techniques. At runtime, an embedded decryption routine first *unpacks* the concealed code and then transfers control to the actual malicious code that has been revealed on the host's memory. Using variations in the transformation method and the code of the decryption routine, attackers are able to easily produce new variants of the same malware that can effectively evade existing detectors [15].

Implementing the self-unpacking functionality of a malware binary using GPU code can pose significant obstacles to current malware detection and analysis systems. A malware author can take advantage of the computational power of modern graphics processors and pack the malware with extremely complex encryption schemes that though can be efficiently computed due to the massively parallel architecture of GPUs. By relying heavily on GPU-friendly transformations—quite costly when implemented with CPU code—the same unpacking algorithm would take a prohibitively long time to complete when running solely on a CPU. This can severely affect existing malware scanners, which typically employ specific unpacking routines for different known packers to recover the original malware binary [11].

Furthermore, many systems for the automated extraction of packed executables inherently cannot handle GPU-based self-unpacking malware. For instance, PolyUnpack [15] relies on single-step execution and dynamic disassembly during the unpacking process, However, in contrast to x86 code, static and dynamic analysis of GPU machine code is at a nascent stage, and it is currently not supported by existing malware analysis systems.

Other unpacking systems like Renovo [9] monitor the execution of malware samples using a virtual machine. Unfortunately, existing virtual machine monitors provide only simulated graphics devices which currently do not support any GPGPU functionality. Thus, any malware sample that employs a GPU-based unpacking routine *will not run at all* on a VM—a severe consequence for a multitude of existing dynamic malware analysis systems built on top of VMs and system emulators [12, 16]. Although automated unpacking is still possible by systems that monitor for execute-after-write memory operations using only OS modifications [11], in practice, even these systems are usually deployed in combination with a virtual machine, for expediting system restore to a clean state and safeguarding malware execution.

For our proof-of-concept implementation of GPU-based unpacking we chose a simple XOR-based encryption scheme using a random key. The embedded unpacking function is compiled to device code of the underlying GPU instruction set. The intermediate code of the unpacking routine for CUDA, called PTX, is shown in Figure 2. Both the GPU unpacking function and the malware code are embedded into the same executable.

At start-up, the GPU code, usually referred to as the *kernel*, is loaded on the device and the CPU code starts executing, as shown in Figure 3. During the bootstrapping phase, the malware allocates a memory-mapped buffer that is used to store the packed binary data. As of CUDA v2.2 and later,

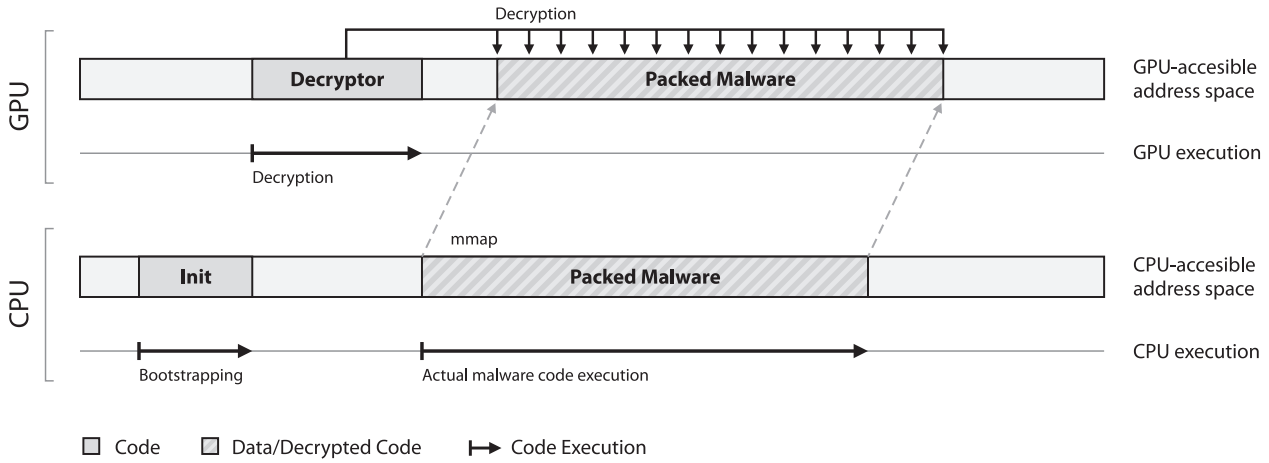

Figure 3. Schematic representation of the execution of GPU-based self-unpacking malware.

it is possible to allocate and map an area of host memory that will be accessible from the device. Therefore, a kernel running on the GPU can access host memory directly, allowing the CPU and the GPU to share the same data. The flow of control is then transferred to the GPU, where the decryption routine unpacks the binary by modifying directly the mapped buffer. Upon decryption, control is transferred back to the CPU which executes the unpacked code.

The only CPU code that is exposed in the original malware image consists of the few instructions that copy the packed data to the newly allocated buffer and bootstrap the execution of the unpacking routine on the GPU. This minimal x86 code footprint does not leave much to existing static and dynamic malicious code analysis systems to actually analyze.

3.2 Run-time Polymorphism

No matter how complex the encryption scheme in a packed malware is, upon the end of the unpacking process the code of the original malware will be restored on the host's memory. At that point, a malware analyst can take a snapshot of the process' address space and analyze the exposed malicious code. Similarly, runtime malware scanners that inspect the address space of all running processes will be able to detect the original malicious code.

A well known technique to hinder the extraction of a complete process image is to decrypt only the parts of code needed at any given point in a just-in-time fashion [4]. Before decrypting a new part of code, any previously decrypted code that is no longer needed is re-encrypted. The finer the granularity of the on-demand decrypted parts, the smaller the code area that remains exposed on the host's memory.

For our proof-of-concept implementation of on-demand decryption based on GPU code we chose *functions* as the unit of decryption. The machine code corresponding to each function in the original source code is encrypted separately using a different key. At run-time, the code of each function is decrypted on demand before its executed, and is re-encrypted just before returning to the caller.

Figure 4 shows how the GPU can be utilized for on-demand code decryption. All code for decrypting and re-encrypting each function resides entirely on the GPU, and thus the CPU is responsible for transferring control to the dispatching code running on the GPU right before and after function execution. Thus, during execution, the flow of control is constantly switching between the CPU and the GPU.

The encrypted code of each function is stored in memory segments that are accessible from both the CPU and the GPU. In contrast, the decryption keys are stored in private device memory that is *not* accessible from the CPU. This effectively hinders existing analysis methods that extract the keys and decrypt all encrypted code blocks using runtime instrumentation [5]. Moving a step further, after execution each function is re-encrypted using a different randomly generated key, causing the malware to constantly mutate in unpredictable ways in the host's memory.

Although complete extraction of the original code is still possible by a determined malware analyst, when combined with existing anti-debugging techniques [5, 6], this form of GPU-assisted runtime polymorphism makes the whole reverse engineering process a challenging and time-consuming task. For instance, the GPU is a perfect fit for the implementation of runtime code checksumming, a quite effective anti-debugging technique [3]. In contrast to existing CPU-only implementations that use periodic checks, the GPU can constantly compute checksums of different code parts in a truly parallel fashion.

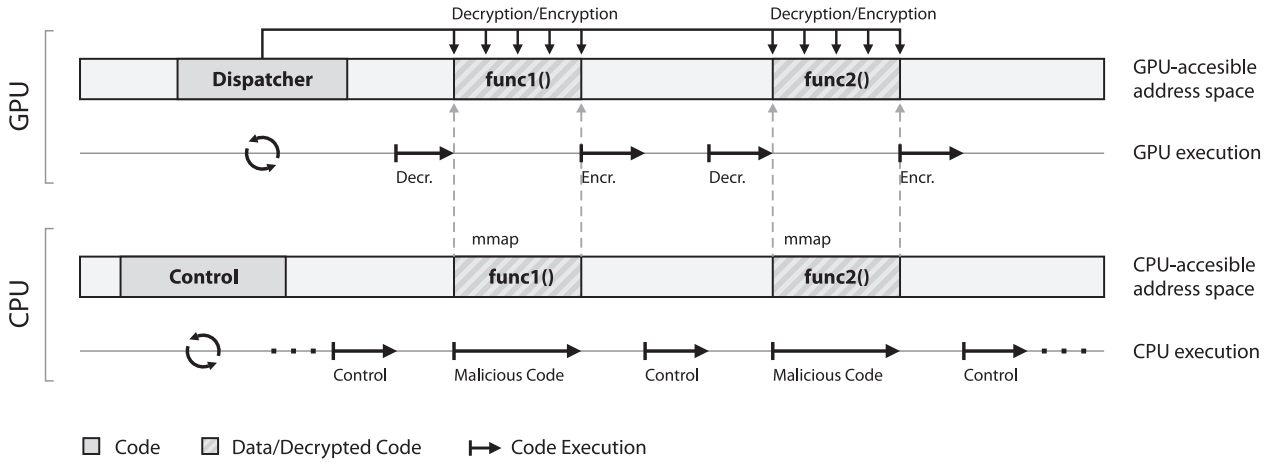

Figure 4. Schematic representation of the execution of GPU-assisted malware that employs runtime polymorphism.

4 Future Attacks

In the previous section we showed how malware can split its execution flow between the CPU and the GPU and thus evade traditional anti-malware systems. While already powerful, the techniques described here utilize only a fraction of the functionality provided by modern GPUs. We expect that malware writers will soon start taking extensive advantage of both graphical and computational capabilities of graphics processors.

GPUs offer massive parellelism, which can be used to speed up CPU-intensive operations. For example, a botnet can be set up for large-scale password cracking—a task that GPGPUs excel in [1, 8]. Bots can easily be extended with GPGPU support and then use the GPUs of infected hosts to offload password cracking.

This would not only result in a significant increase in overall password cracking performance, but would also hide the ongoing malicious activity. Since the GPU workload cannot be monitored in real time to identify what code is running, it is quite difficult to determine the presence of the password cracking code on the GPU. Additionally, the CPU will not be occupied at all during this computationally-intensive process, and thus CPU-load monitors will not be helpful in detecting the malicious activity.

Now, recall that the *framebuffer*, which is part of the device memory of the VPU, contains what is displayed on the monitor at any given time. Having unrestricted access to the framebuffer opens the way for a wide range of possible attacks. For instance, malicious code running on the GPU could access the screen buffer periodically and harvest private data displayed on the user screen, and do so in a more stealthy way than existing screen capture methods. Alternatively, more sophisticated malware could attempt to trick users by displaying false, benign-looking information when visiting rogue web sites (e.g., overwriting suspicious URLs with benign-looking ones in the browser's address bar).

We are currently exploring the feasibility of such attacks using existing GPUs. Unfortunately, in the current GPGPU architectures the framebuffer memory is protected from read and write operations. However, as vendors constantly try to improve the graphics interoperability between GPGPU SDKs and graphics APIs such as OpenGL and DirectX, it is quite possible that in future releases a kernel will have full access to the screen framebuffer. Accessing the screen pixels directly will increase the performance of many graphics operations, such as 3D transformations and video compression and decompression, by reducing data transfers between the CPU and the GPU. So this is a feature that will inevitably be part of future hardware releases.

Going one step further, future GPGPU architectures could enable the implementation of GPU-hosted malware, i.e., malware that runs solely on the GPU, without any association with a process running on the CPU. However, a major restriction of current graphics hardware architectures is that they do not support multitasking; only one task can utilize the GPU at any time. Hence, if the malware is launched directly on the GPU, without the intervention of the CPU, it may parasitically consume all GPU cycles without being context-switched. Consequently, the task responsible for rendering the monitor would not run, and the display would freeze. Although this scenario seems quite unrealistic right now, as many technical hurdles need to be overcome, it is possible that future graphics hardware will have the necessary functionality for next-generation malicious code that fully utilizes the GPU.

5 Conclusion

The rapid evolution of general-purpose computing on graphics processors enables malware authors to take advantage of the GPU present in modern personal computers and increase the robustness of their code against existing defenses. The code armoring techniques presented in this paper—GPU-based unpacking and runtime polymorphism—not only demonstrate the feasibility of GPU-assisted malware, but also show the great potential that general-purpose computing on GPUs has in enhancing the evasiveness and functionality of malicious code. Both techniques have been implemented and tested using existing graphics hardware, and pose significant challenges to existing malware analysis and detection systems, which mostly handle only IA-32 code.

Taking a step further, we describe potential attacks that malware can employ using upcoming features of next-generation GPUs. The constantly enhanced capabilities of graphics processors, for both graphics and general-purpose computations, can make graphics cards a promising environment for future malware.

Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS, by the Marie Curie FP7-PEOPLE-2009-IOF project MALCODE, and by the project i-Code funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission – Directorate-General for Home Affairs. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein. Giorgos Vasiliadis and Sotiris Ioannidis are also with the University of Crete.

References

[1] Russian crackers throw GPU power at passwords. `http://arstechnica.com/business/news/2007/10/russian-crackers-throw-gpu-power-at-passwords.ars`.

[2] AMD. ATI Stream Software Development Kit (SDK) v2.1. `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`.

[3] P. Biondi and F. Desclaux. Silver Needle in the Skype. BlackHat Europe 2008.

[4] J. Cappaert, B. Preneel, B. Anckaert, M. Madou, and K. D. Bosschere. Towards tamper resistant code encryption: Practice and experience. In *Proceedings of the 4th Information Security Practice and Experience Conference (ISPEC)*, 2008.

[5] C. Eagle. Strike/counter-strike: Reverse engineering shiva. BlackHat Federal 2003.

[6] P. Ferrie. Anti-Unpacker Tricks. In *Proceedings of the 2nd International CARO Workshop*, 2008.

[7] grugq and scut. Armouring the ELF: Binary encryption on the UNIX platform. *Phrack*, 11(58), Dec. 2001.

[8] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[9] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring Malcode (WORM)*, 2007.

[10] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[11] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[12] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, 2007.

[13] NVIDIA. Compute Unified Device Architecture Programming Guide, version 3.0. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`.

[14] D. Reynaud. GPU Powered Malware. Ruxcon 2008.

[15] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[16] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[17] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.