Obfuscation: The Hidden Malware

Obfuscation camouflages telltale signs of malware, undermines antimalware software, and thwarts malware analysis. New antimalware approaches should focus on what malware is doing rather than how it's doing it.

Philip O'Kane, Sakir Sezer, and Kieran McLaughlin
Centre for Secure Information Technologies, Queen's University Belfast

In recent years, malware has seen massive growth, fueled by the evolution of the Internet, which is a "target-rich environment." That is, the Internet provides an increased number of victims in terms of network speed and number of accessible computers. A significant change has been the evolution from malware written by hobbyists, designed to cause havoc and demonstrate their ability, to malware written by criminals, designed to steal personal information for financial gain.[1]

According to the FBI Internet Crime Complaint Center (IC3), businesses and consumers lost US$559 million in 2009 (compared to $265 million in 2008) owing to online fraud.[2] The IC3 recorded 336,655 complaints. The top five categories were nondelivery or nonpayment of goods (19.9 percent), identity theft (14.1 percent), credit-card fraud (10.4 percent), auction fraud (10.3 percent), and vandalism of computers and property (7.9 percent).

Symantec reported that the number of computer viruses has reached 1.1 million.[3] This report stated that 711,912 new viruses were identified in 2008—a 468 percent increase from 2007. In 2004, Roger Grimes's honeypot experiments showed that an unprotected computer (Windows XP Service Pack 1) has a 50 percent probability of becoming infected within 32 minutes of being connected to the Internet.[4]

An effective tool for malware writers has been obfuscation. Such techniques obscure code into a form that's functionally identical to an original code, making it difficult to analyze. Legitimate software vendors obfuscate their software to conceal it from those who would try to steal their design or infringe on copyright licenses. Here, we discuss these techniques' development in the context of the evolving battle between malware writers and antimalware researchers.

Battling Malware

Antimalware software prevents malware's spread by detecting and neutralizing instances of malware. One popular approach is *signature detection*, which compares a file's contents to a database of known malware signatures. Because malware can be embedded in existing files, signature detection requires searching the entire file, not just as a whole but also as segments because the malware is often interleaved into the host file rather than just appended to the end. (See the sidebar for information on other antimalware methods.) However, signature detection can't protect against *0-day threats*—attacks that have unknown signatures or that target unknown vulnerabilities.

The Failure of Signature Detection

Until recently, signature detection has effectively detected malware (except 0-day threats). However, with the relentless deployment of new instances of malware, signature detection is becoming ineffective because it requires continuous analysis of new malware instances and updates into a database.

Signature-based detection, at best, can only attempt to keep pace with malware outbreaks because it relies on first capturing a malware instance and then analyzing it to determine a new signature for that in-

Malware-Detection Approaches

In addition to signature-based malware detection, there are several other popular approaches for detecting malware.

Semantics-based detection analyzes programs for threatening behavior, which makes this approach less vulnerable to code obfuscation.

Semantics analysis searches files for inappropriate resource use and analyzes programs for activities that use known security vulnerabilities.

File emulation runs the code in a secure virtual environment to determine if a program is trying to access privileged operations not normally associated with that type of program.

stance. This has resulted in a weapons race, in which attackers have employed obfuscation techniques to counter antimalware software.

The Weapons Race

Figure 1 shows a simplified view of this weapons race's four phases.

In the first phase, infections came from various sources, such as browser plug-ins (for instance, the Adobe Flash player) and other media such as instant messaging, email attachments, or downloaded applications with Trojan horses.

In the second phase, signature scanners detected malware as it downloaded. So, malware writers *packed* (that is, compressed) their malware to evade detection.

In the third phase, static analysis disassembled and deciphered the malware's assembly code before runtime. In response, polymorphic malware tried to evade detection during static analysis by generating new signatures each time the malware executed.

In the last phase, dynamic analysis examined code at runtime to detect signatures or nefarious activities. Such analysis overcame problems such as unknown packers and polymorphic code because the malware does the unpacking and deciphering. In response, metamorphic malware continually modified itself, generating new opcode (operation code) patterns each time it ran. This resulted in new signature definitions each time the code ran, thus evading signature detection by static and dynamic analysis because no signature would exist in the signature database.

Obfuscation Techniques

Here we examine packers, polymorphism, and metamorphism in more detail.

Packers

Packers were originally developed to minimize memory and bandwidth use during file storage and transfer. Storage and bandwidth have become more abundant, but many software companies still use packers to bundle executables with component files to deploy commercial software.

The downside is that a small change in any file causes a large change in the compressed file. This inherent encryption in a packer gives malware writers a powerful tool, which they've enhanced to increase the packers' effectiveness at evading detection.

Many packing tools are available from both corporate vendors and underground developers. For example, Jon Oberheide and his colleagues at the University of Michigan wrote PolyPack,[5] a Web-based application that supports 10 packers and 10 antimalware engines. After the user submits a file containing malware, each of the 10 packers automatically packs it, then each antimalware engine scans it. PolyPack generates a report of the engines' effectiveness on the different packed versions of the malware, so that users can pick the most effective packer. In testing with PolyPack, Themida, an application originally for protecting commercial software from software crackers, outperformed the other nine packers and evaded most antivirus scanners.[5]

The prevalence of packing. A survey of 1 million malware files on a public interface showed that 48.92 percent of files were packed.[6] A 2007 Avert Labs survey concurred that malware widely uses packers but observed more diversity in the packers used. This survey estimated that processing packers consumes 60 percent of antimalware software effort.[7] It advocated banning packers on the user end by totally blocking all packed software until an effective solution can be implemented. A 2006 survey by Black Hat suggested that 92 percent of malware programs are packed.[8]

Although surveys don't agree on the exact number of packers, packers are clearly a substantial part of the malware writers' arsenal.

Combating packing. Some antimalware tools use signature analysis to identify packers. One example is the freely available PEiD (Portable Executable Identification; www.peid.info); version 0.95 can identify 600 different packers. Signature-based packer detection's effectiveness is a function of the signature databases' completeness. As packers evolve and change, the number of signatures will grow, resulting in the same issues the original signature scanners experienced.

Several antimalware tools employ entropy analysis to identify packers. When a file is packed, encryption increases the randomness of that file's data, resulting in a higher entropy score. Entropy analysis can detect packed files with a high degree of confidence[9] but can't identify the specific packer.

PHAD (*PE Header Analysis*-Based Packed File *Detection*) uses entropy analysis to analyze eight characteristics of portable executable (PE) binary files. In tests, its detection rate was 93.59 percent.[9]

PE-Probe is another entropy-analysis–based packer detector. Its detection rates in tests were 99.6 percent for packed files and 99.4 percent for nonpacked files, with false-positive detection rates of 0.3 percent and 0.8 percent, respectively.[10]

Mandiant Red Curtain (MRC) employs entropy analysis to determine how suspicious malware files are. It performs structural analysis on a PE file to decide whether the file is using avoidance techniques (such as encryption) or packers.

Reminder (which stands for "response tool for malware indication") combines entropy analysis with analysis of packed files' write properties. The researchers who created it compared it to MRC and PEiD, using 400 files—200 unpacked Windows program files and 200 packed files captured in a honeypot.[11] MRC's detection rate was only 39.5 percent, whereas PEiD's was 88 percent and Reminder's was 97.5 percent.

Detecting packed files without identifying the packer algorithm is inadequate because you can't unpack and inspect the file contents. Legitimate software vendors use packers to protect and manage their software. So, without unpacking a file, it's difficult to determine whether it contains malware. Though we can reliably detect packed files, antimalware scanners can't always unpack them. According to Oberheide and his colleagues, approximately 40 percent of the 98,801 malware specimens they tested couldn't be unpacked.[5] In those cases, the files' contents will remain hidden from the scanners.

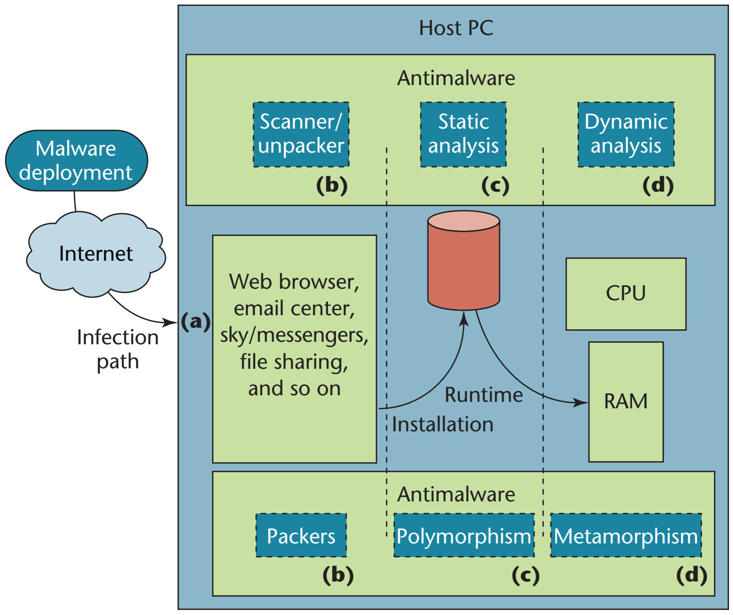

Figure 1. The antimalware-malware weapons race's four phases. (a) Systems were infected from various sources. (b) Signature scanners were countered by malware packing. (c) Static analysis was countered by polymorphic malware. (d) Dynamic analysis was countered by metamorphic malware.

Polymorphism

Polymorphism is an encryption method that mutates static binary code (rather than runtime code) to evade malware signature scanners. Malware that changes its contents can evade detection by signature scanners because no single signature sequence will match all the instances that the malware generates.

Figure 2 illustrates a simplified implementation of a polymorphic engine, which functions as follows:

1. *Gain entry.* The host application might start first, but at some point the polymorphic engine gains control of the CPU.
2. *Execute transfer function.* The polymorphic engine's transfer function deciphers the mutated malware code in the infected host into native opcode using the polymorphic key in the infected host file.
3. *Load executable.* The engine writes the native opcode into memory for execution.
4. *Run.* After the polymorphic engine has completed the decryption, it jumps to the start of the now deciphered malware code, which then runs.
5. *Execute nefarious code.* The malware carries out activities such as setting up a back door for a bot, setting up a keyboard logger, or stealing personal details.
6. *Generate new key.* For the polymorphic engine to generate a new variant of the mutated code, it must generate a new key, which it then stores in the new version of the malware.
7. *Execute inverse transfer function.* The inverse function transforms the opcodes (executable) back into mutated code. However, some information must be stored in the clear, such as the polymorphic engine's forward transfer function.

Each time the code runs, it mutates itself using a different encryption key, updating the polymorphic-key variable and writing it back to the infected host file. This results in each new copy of the code having a different signature.[12]

When the polymorphic engine deciphers and loads the malware into memory to run, the opcode is semantically the same for each instance. That is, the polymorphic engine doesn't significantly change the native opcode that runs in memory. So, it's possible to use signature detection when the malware is loaded into memory at runtime. Another detection technique involves using neural pattern recognition of opcode mnemonics distribution, which had a 99.9 percent detection rate on 500 sample files.[13]

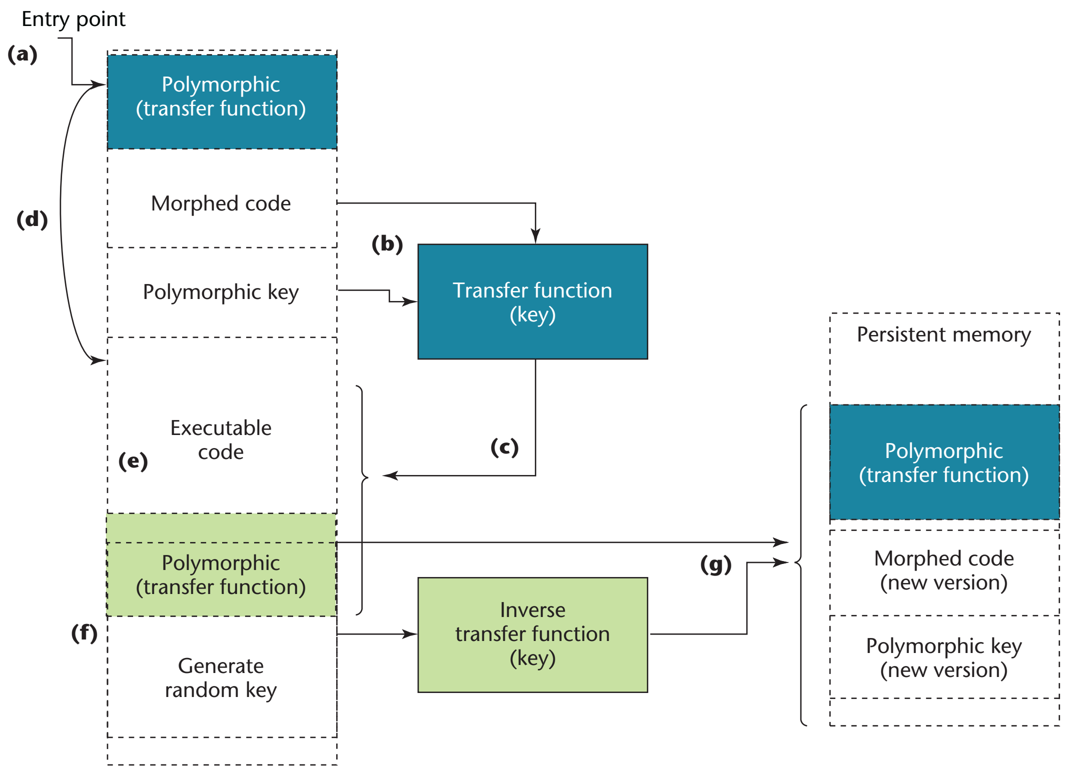
Entry point
(a)
Polymorphic (transfer function)
Morphed code
(d)
(b)
Transfer function (key)
Polymorphic key
Persistent memory
Executable code
(e)
(c)
Polymorphic (transfer function)
Polymorphic (transfer function)
Morphed code (new version)
(g)
Inverse transfer function (key)
(f)
Generate random key
Polymorphic key (new version)

Figure 2. A simplified polymorphic engine. (a) The malware gains entry. (b) The engine's transfer function deciphers the mutated malware code in the infected host into native opcode (operation code). (c) The engine writes that opcode into memory. (d) The malware runs. (e) The malware executes an activity such as stealing personal details. (f) The engine generates a new key. (g) The engine's inverse transfer function transforms the opcode back into mutated code.

Metamorphism

Each time metamorphic malware runs, it changes the opcode that's loaded into memory and then writes a new version of the malware back to the infected host file. The malware maintains its malicious behavior without ever having the same sequence of native opcodes in memory. So, conventional signature-based approaches would require scanning a program for millions of different signatures to detect one piece of malware.

Categories. Metamorphic malware falls into two categories, depending on whether the transformation uses a communication channel. *Open-world malware* can communicate with other sites on the Internet and download updates. In 2008, the Conficker worm appeared and rapidly spread, compromising many PCs using open-world communication (MS Windows RPC vulnerability). Conficker established a comprehensive command-and-control structure with 50,000 domains and 500 URLs.[14]

The second category, *closed-world malware*, doesn't rely on external communication to mutate. During evolution, the executable mutates the binary code itself (called the *binary transformer*) or uses a pseudocode (metalanguage) representation to generate the newly mutated code (see Figure 3). Win32/Apparition was the first virus to use these techniques and infected files on a host when a suitable compiler was present.

Hiding metamorphic malware. At the point of infection, malware writers can use different strategies to hide metamorphic malware, including providing each visitor to a malicious website with a different instance of obfuscated malware or self-mutating malware. Malware that propagates among infected users (rather than directly from an infected website) must support self-mutating code to evade detection. However, open-world malware can automatically update with new features, such as new code that supports mutation, and can therefore generate a new version of itself each time it runs. This is frequently called *dynamic code obfuscation*.

On the basis of operations performed, obfuscation methods fall into five categories. *Garbage insertion* inserts benign instructions such as `nop` and `clc` (clear an unused register). *Code substitution* switches opcodes to equivalent opcodes that accomplish the

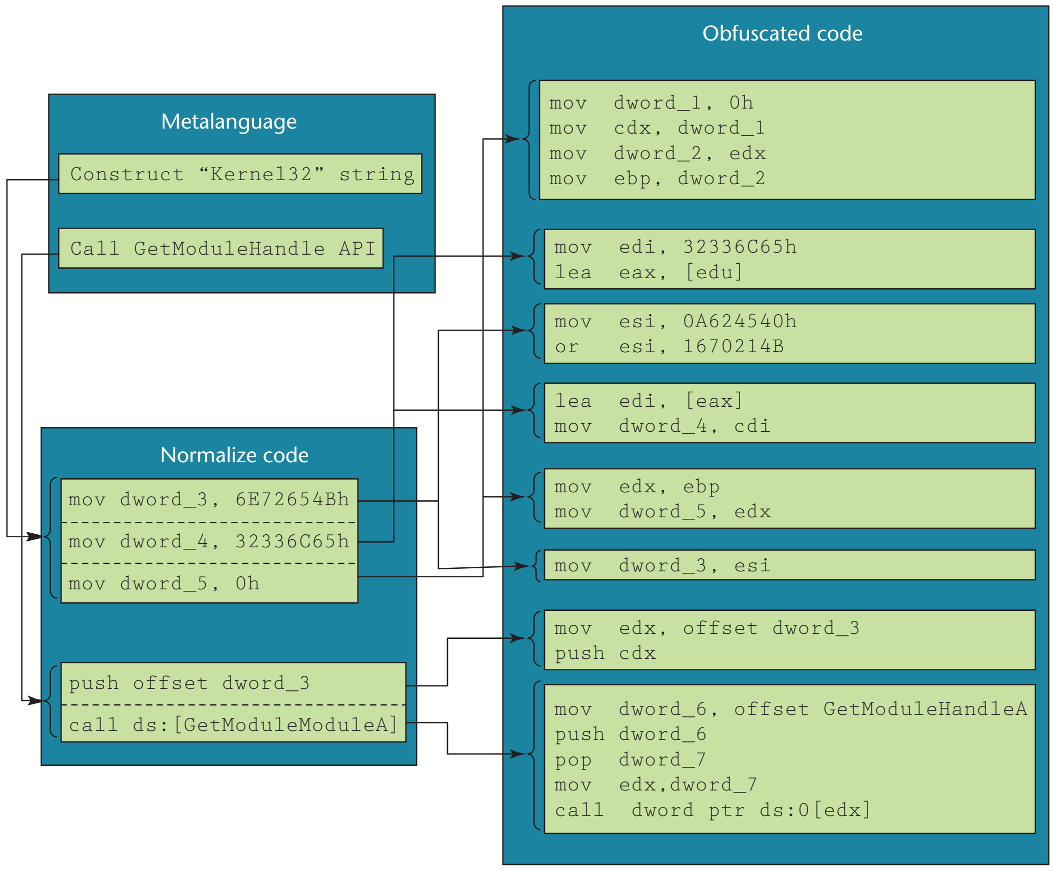

Figure 3. Metamorphic obfuscation code. This illustrates the permutation and expansion of the runtime opcode introduced by the metamorphic engine.

same thing. *Code insertion* interlaces the malware with its host's binary code. *Register swapping* substitutes one register for another in the malware's code. Finally, *changing flow control* jumps and reorders the call sequence by adding subroutines.

The five steps. Metamorphic malware is more complex than other malware and involves up to five steps.

First, a disassembler decodes the opcodes. Windows is an IA-32 (Intel Architecture, 32-bit) architecture, which is considered a CISC (complex-instruction-set computing) processor. That is, it has a rich instruction set, and many of the opcodes can perform identical tasks when combining multiple operations and operands. In addition, IA-32 supports variable-length opcodes and opcode extensions. This results in a complex task of parsing and decoding opcodes and their operands as well as calculating `jmp` and `call` instructions.

Second, a shrinker compresses the disassembled code, which prevents the continuous growth of code with each new generation. Left uncompressed, the malware, and hence the host file, would grow and become bloated and oversized. (Because not all metamorphic malware implements an expander, not all of it uses a shrinker.)

Third, a permutator reorders instructions by placing them in random order, connects them with `jmp` opcodes, and implements register and opcode substitution.

Fourth, an expander recodes a single opcode instruction into several instructions and inserts benign opcode such as `nop` opcodes (see Figure 3). This happens each time the malware runs to generate a new instance of the malware.

Finally, an assembler recodes what the expander produced, calculates the new `jmp` and `call` vectors, and recalculates (repairs) the host file's `jmp` and `call` addresses owing to the malware being injected (interleaved). For malware represented as a metalanguage or pseudocode, the assembler will reassemble this into the target opcode as well.

Virus code performs the tasks of opening back doors, stealing user details, and so on.

The challenges of writing metamorphic malware. Metamorphic malware might appear to be a panacea for malware writers, but they must deal with several challenges.

One challenge is size. Metamorphic engines are larger than polymorphic engines. Polymorphic engines can mutate an 8-Kbyte malware program with a polymorphic engine with a 200-byte code length, whereas metamorphic engines take up approximately 80 to 90 percent of the malware. For example, the Win32/Simile virus consisted of approximately 14,000 lines of assembly code, with the metamorphic engine occupying approximately 90 percent of the 32-Kbyte file.[15]

Small malware programs can be injected (interleaved into the unused space) without significantly increasing the file size. Large malware is harder to hide and inject into the gaps of other files, so it requires additional stealth techniques such as a rootkit to conceal the large files.

However, malware writers can reduce the malware's footprint on the target PC by using an open-world approach. The metamorphic engine resides on a server, and the malware on the target PC is changed through a back door.

Another challenge is CPU workload. The metamorphic engine produces more obfuscated opcode at runtime. So, highly obfuscated opcode from a metamorphic engine will reduce the idle time—the increased amount of opcode increases the CPU workload. In contrast, smaller, more efficient malware such as a polymorphic engine might not significantly increase the CPU workload.

To exploit this problem, some antimalware software monitors the CPU idle time relative to known applications running and setting a benchmark. When a malware program runs with a high CPU workload, the antimalware software monitors the CPU and so might detect the malware's CPU workload signature.

Such problems introduce leakage through side channels, opening up opportunities for developing effective dynamic-analysis methods. For example, normalization of obfuscated code could produce a signature that can identify malware.[16]

It's widely believed that malware is a multibillion-dollar industry. With such revenues, malware writers are unlikely to fade away. On the contrary, they are well funded and therefore will perpetuate the weapons race by targeting new technologies. As malware migrates to these new technologies, antimalware researchers will face new challenges.

Malware code has become harder to detect, and the days of relying on simple signature string detection are over, along with any hope that a single approach (silver bullet) will stop malware in its tracks. Although the way forward is uncertain, one certainty exists: we must find new approaches that rely on incorporating behavioral information with anomaly analysis. That is, we must focus on what the suspected malware is doing rather than how it's doing it. This strategy might yield optimal detection because it minimizes reliance on the underlying technology, which is constantly evolving.

Security policies and restrictions for the Internet and its users would also help reduce the risk of malware infection. These policies would encompass profiling users, restricting and enforcing network protocols, and policing the various components of the Internet through white-list certification. However, the Internet has long been considered a medium that supports access and free speech, which might raise ethical issues about any attempts to police it. □

References

1. J.M. Bauer, M.J.G. van Eeten, and Y. Wu, *ITU Study on the Financial Aspects of Network Security: Malware and Spam,* tech. report, ICT Applications and Cybersecurity Division, Int'l Telecommunication Union, 2008; www.itu.int/ITUD/cyb/cybersecurity/docs/itu-study-financial-aspects-of-malware-and-spam.pdf.
2. I. Thomson, "FBI Reports Online Crime Losses Double in 2009," V3.co.uk, 13 Mar. 2010; www.v3.co.uk/v3/news/2259467/fbi-reports-online-crime-losses.
3. "Symantec Internet Security Threat Report: Trends for July–December 07," white paper, Symantec, Apr. 2008; http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf.
4. R.A. Grimes, *Honeypots for Windows*, A-Press, 2004.
5. J. Oberheide, M. Bailey, and F. Jahanian, "PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion," *Proc. 3rd Usenix Conf. Offensive Technologies* (WOOT 09), Usenix Assoc., 2009, p. 9; www.usenix.org/event/woot09/tech/full_papers/oberheide.pdf.
6. U. Bayer et al., "A View on Current Malware Behaviors," *Proc. 2nd Usenix Conf. Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (LEET 09), Usenix Assoc., 2009, p. 8.
7. G. Taha, "Counterattacking the Packers," white paper, McAfee Avert Labs, 2007.
8. T. Brosch and M. Morgenstern, "Runtime Packers: The Hidden Problem?" PowerPoint presentation at Black Hat USA, 2006; www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.
9. Y.-S. Choi et al., "PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)," *Proc. Int'l Conf. Computer Science and Its Applications* (CSA 08), IEEE CS Press, 2008, pp. 28–31.
10. M.Z. Shafiq, S.M. Tabish, and M. Farooq, "PE-Probe:

Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables," *Proc. 18th Virus Bulletin Conf.* (VB 09), Virus Bulletin, 2009, pp. 29–33.
11. S. Han, K. Lee, and S. Lee, "Packed PE File Detection for Malware Forensics," *Proc. 2nd Int'l Conf. Computer Science and Its Applications* (CSA 09), IEEE CS Press, 2009, pp. 1–7.
12. W. Wong and M. Stamp, "Hunting for Metamorphic Engines," *J. Computer Virology*, vol. 2, no. 3, pp. 211–229.
13. R. Santamarta, "Generic Detection and Classification of Polymorphic Malware Using Neural Pattern Recognition," white paper, ReverseMode, June 2006.
14. R. Livintz, "Conficker—One Year After (Part One)," blog, 17 Nov. 2009; www.malwarecity.com/blog/conficker-one-year-after-part-one-672.html.
15. F. Perriot, P. Ször, and P. Ferrie, "Striking Similarities: Win32/Simile and Metamorphic Virus Code," white paper, Symantec Security Response, 2003; www.symantec.com/avcenter/reference/striking.similarities.pdf.
16. A. Walenstein et al., "Normalizing Metamorphic Malware Using Term Rewriting," *Proc. 6th IEEE Int'l Workshop Secure Code Analysis and Manipulation* (SCAM 06), IEEE CS Press, 2006, pp. 75–84.

Philip O'Kane *is a lead architect and senior software engineer at the Centre for Secure Information Technologies at Queen's University Belfast. His research interests include low-level analysis of attributes that can help verify the presence of malware. O'Kane has an MSc in informatics from the University of Ulster. Contact him at pokane17@qub.ac.uk.*

Sakir Sezer *is a professor and head of network and cybersecurity research at the Centre for Secure Information Technologies at Queen's University Belfast. His research is leading major advances in the field of high-performance content and security processing. Sezer has a PhD in electronic engineering from Queen's University Belfast. He's cofounder and CTO of Titan IC systems and a member of various research and executive committees, including the IEEE International System-on-Chip Conference executive committee. Contact him at sezer@qub.ac.uk.*

Kieran McLaughlin *is a senior engineer at the Centre for Secure Information Technologies at Queen's University Belfast and the lead architect for various technologies for cybersecurity, smart grid security, and high-throughput network security processing. His research interests include high-throughput technologies for network security processing. McLaughlin has a PhD in electrical and electronic engineering from Queen's University Belfast. Contact him at kieran.mclaughlin@ee.qub.ac.uk.*