Stealthy Software: Next-generation Cyber-attacks and Defenses

(Invited Paper)

Kevin W. Hamlen
Computer Science Department
The University of Texas at Dallas

Abstract—**Weaponized software is the latest development in a decades-old battle of virus-antivirus co-evolution. Reactively adaptive malware and automated binary transformation are two recently emerging offensive and defensive (respectively) technologies that may shape future cyberwarfare weapons. The former intelligently learns and adapts to antiviral defenses fully automatically in the wild, while the latter applies code mutation technology to defense, transforming potentially dangerous programs into safe programs. These technologies and their roles within the landscape of malware attack and defense are examined and discussed.**

"Prepare to hear of occurrences which are usually deemed marvellous. Were we among the tamer scenes of nature, I might fear to encounter your unbelief, perhaps your ridicule; but many things will appear possible in these wild and mysterious regions which would provoke the laughter of those unacquainted with the ever-varied powers of nature: . . . "
—Mary Shelley (Frankenstein, 1818)

I. Malware Evolution: Past and Present

The last few years have witnessed a fundamental paradigm shift in the objectives, power, and capabilities of malicious software (*malware*). Although malware technologies have historically undergone steady increases in sophistication, their history has been punctuated by periodic revolutionary leaps that have demanded heightened attention from the security community. Most recently, the emergence of advanced cyberwarfare weaponry and tactics is now shaping a new era of weaponized software attacks and defenses. This new theater of war demands new offensive and defensive software innovations from government, academia, and industry.

The first viruses of the early-to-mid 1980s were tiny programs consisting of just a few hundred bytes of machine code, often crafted by graduate students who were more interested in intellectual challenge than serious damage [1]. A major perceptual change occurred in 1988 when one of these intellectual exercises, the Morris worm, went wrong and infected a significant portion of the internet, denying service and costing an estimated millions of dollars in cleanup [2]. This brought malware to the attention of the wider public, giving birth in the ensuing years to an industry of antivirus defense.

Early malware detection systems relied upon simple byte scanning [3], which prompted malware authors to equip their creations with *polymorphism*. Polymorphic malware randomizes or mutates most or all of its constituent bytes on each propagation so that no two instances look alike. This frustrates syntax-based detectors, such as *signature-matchers*, which identify malware by scanning for malware-specific byte sequences (*signatures*). The most widely employed polymorphic strategy continues to be *packing*—compression and/or encryption of the malware payload with a randomly chosen dictionary and/or key. This randomizes all payload bytes each time the virus propagates, leaving few meaningful bytes visible to defenders.

The few remaining code bytes typically encode the decryptor, which must remain directly executable. This potential weak point is typically addressed through malware *metamorphism*—obfuscation algorithms that change the decryptor's binary implementation by randomly reordering instructions, adding junk code, or applying semantics-preserving transformations. The earliest dedicated polymorphic mutation engines appeared in 1991 (e.g., MtE), and packing-based, metamorphically-assisted mutation has since become a mainstay of virus stealth.

Over the last decade, two additional leaps in malware technology rose to prominence: zero-day exploits and botnets. In 2003 the Slammer worm infected an estimated 90% of infectable hosts within minutes of its release. Although the primary vulnerability it exploited was not a zero-day, the unprecedented speed with which it exploited a relatively new, widely unpatched vulnerability placarded the inadequacies of semi-manual responses to intrusions. Fully automated methods of reliably detecting and quarantining previously unseen malware, and patching newly discovered vulnerabilities, became a defensive imperative. Malware authors now prize and market zero-days as potent vehicles for rapid, large-scale attacks [4], [5]. Such vulnerabilities dominated the list of top cyber threats for the first quarter of 2013 [6].

The rise of numerous botnets in 2007, including the prolific Cutwail and Srizbi botnets, introduced a new wrinkle for defense: adversarial command and control (C&C). Through a C&C network, bot-masters can respond to defender gambits by uploading malware software updates and launching coordinated attacks against rival targets. This has transformed what was previously an isolate-and-destroy scenario for defenders into a chess match in which each side seeks to systematically capture and destroy the other's assets (cf., [7]).

As such, it is no longer possible to win most prolonged cyber-battles with only a strong defense. Offensive capabilities are needed to take down rival C&C servers, perform reconnaissance, and infiltrate and sabotage enemy networks. Accordingly, the USAF recently acknowledged the existence of at least six cyber-weapons in its war arsenal [8], and DARPA is now actively soliciting cyberwarfare research that augments both national defensive and offensive capabilities [9].

While the sophistication of malware capabilities has thus risen steadily with the increasing profit available to cybercriminals, past malware pales in comparison to this emerging new generation of weaponized software. In 2010, the Stuxnet virus infected and destroyed nuclear centrifuges in the high-security uranium-enrichment plant in Natanz, Iran [10]. To accomplish this considerable feat, Stuxnet exploited an unprecedented four zero-day vulnerabilities, used stolen cryptographic credentials to authenticate itself, accessed C&C servers to self-update, and crossed architectural boundaries to infect centrifuge-controlling PLCs. The DuQu, Flame, and Gauss viruses have subsequently been identified as cyber-weapons of similar complexity that were likely authored by the same or collaborating nation-states [11].

Surprisingly, although Stuxnet and its variants boast superior capabilities in almost all aspects of malware development, they have little or no polymorphic functionality; each copy is roughly the same, making them relatively easy to detect once identified. Despite this apparent weakness, evidence indicates that Flame (which is a colossal 20MB in size) successfully operated in the wild, infecting large organizations and gathering intelligence, for over two years before it was finally discovered [12].

This embarrassing history testifies to a serious scalability problem in the modern antivirus industry. Malware analysts currently face a backlog of unclassified software samples so vast that complex monstrosities like Flame can hide in plain sight for years before analysts manage to fully vet them. Faster, more powerful automation is required to better prioritize unanalyzed software in these databases, and potentially detect previously unseen malicious behavior in the wild.

The deficiency also highlights an obvious avenue of potential future progress toward a strong, national cyber-offense. Hiding in plain sight is unlikely to be an effective stealth strategy for weaponized software in the long-term future. Advances in software stealth are therefore needed by nation-states wishing to maintain superiority in the cyber arms race.

II. Next-generation Attacks

A. Reactively Adaptive Malware

Reactively adaptive malware [13], [14] is one such offensive advance that we have been studying in the Software Security Lab at The University of Texas at Dallas (UTD). Such malware capitalizes on the observation that nearly all conventional malware obfuscation strategies have a common weakness: their mutations are *undirected*. For example, encryption-based polymorphic viruses randomly choose new encryption keys in the hope that the resulting random cyphertexts will contain few common, distinguishing features or patterns. Defenders exploit this weak assumption by unearthing invariant feature patterns, such as high entropy or unique decryptor logic, to craft signatures that match all variants of the malware. Once a signature is known, future mutations can be reliably identified (at least until a radical change to the malware is effected—usually with manual attacker assistance via a C&C server).

In contrast, reactively adaptive malware undergoes *directed* mutation. It intelligently learns a model of how signature-matchers identify malware, and then reverses it to discover obfuscations that defeat them. This allows it to quickly adapt and evade signature updates. Reactively adaptive malware thus instantiates true stealth rather than mere undirected diversity.

In recent work, we demonstrated the feasibility and effectiveness of a naturally reactively adaptive approach dubbed *Frankenstein* [14]–[16]. Rather than mutating purely randomly, Frankenstein re-implements itself entirely from code fragments that it harvests from (benign) programs already present on infected victim machines. Like the monster created by the scientist in Shelley's novel, each Frankenstein mutant is therefore the product of stitching together pilfered body parts from its unsuspecting victims.

Unlike conventional metamorphic malware, Frankenstein's corpus of mutations is therefore not limited to a finite set of code transformation rules; it can learn new implementations of itself from the ever expanding corpus of programs it encounters during its travels. Moreover, since each mutation is composed entirely of code from "normal" (usually benign) programs, they tend to exhibit statistical features typical of benign software.

B. A Frankensteinian Example

Table I shows some assembly code from two Frankenstein-generated mutations of an oligomorphic obfuscator. The first column expresses the function of each code fragment as a goal predicate. Mutation #1 was generated from code fragments (*gadgets*) harvested from the Windows calculator application (`calc.exe`). Mutation #2's gadgets were harvested from Microsoft Paint (`mspaint.exe`). After mutation, we disassembled the victim applications to determine the sources of the gadgets Frankenstein chose. These origins are reported to the right of each code fragment.

The results show the natural diversity and even creativity that can arise from Frankenstein's mutation strategy. For example, while the first mutation of the second goal ($L_1' = L_1 + 1$) uses a simple `add` instruction to increment the `eax` register, the second mutation uses address arithmetic to increment `eax` by moving it into the `edi` register (via the `mov` instruction) and then loading the effective address of `edx+1` (via the `lea` instruction). Meanwhile, the ancillary `pop ebp` instruction from the first mutation's implementation is later used to achieve its next goal of locating a temporary variable v_1 at location `ebp-4`.

Despite the radically different functionalities of malicious payloads and benign victim programs, there is typically no shortage of gadgets available in the latter to implement the former. For example, to implement the first mutant's third gadget, Frankenstein repurposed the calculator's code for computing hyperbolic sines (asinhrat); whereas the corresponding gadget from the second mutant was lifted from a subroutine that draws with an opaque paint brush (BltReplace). The size and code diversity of most applications compiled from myriad optimizing Windows compilers, and the richness of the Intel CISC instruction architecture, provide Frankenstein an ample supply of diverse gadgets with which to implement its payloads.

For illustration purposes, the goal predicates in Table I are fairly simple and low-level, and the gadget implementations are relatively small and concise. However, with more abstract, higher-level goals, Frankenstein can find even an even more diverse array of implementations containing larger, more complex gadgets. For example, goals $L_1' = L_1 + 1$ and $v_1 = L_1$ could be coalesced into the single goal $v_1 = L_1' = L_1 + 1$, prompting Frankenstein to consider gadgetry that performs both assignments simultaneously, or in reverse order, or with instruction sequences that intertwine the two operations.

III. Next-generation Defenses

What detection methods are feasible for combating next-generation stealth technologies like Frankenstein? One avenue of clear promise is *semantics-based detection* [17], which statically infers models that approximate how untrusted software might behave when executed, irrespective of the syntax with which it implements that behavior. Thus, untrusted software is judged by what it will do, not what it looks like.

TABLE I. Two Frankenstein mutations of an oligomorphic obfuscator

Goal Predicate	Mutation from Windows Calculator		Mutation from Microsoft Paint	
	Mutant #1	**Gadget Origin**	**Mutant #2**	**Gadget Origin**
$L_1 = 0$	`push 0x19`; `pop edx`; `xor eax, eax`	CCalcEngine	`xor eax, eax`; `cmp [ebp+0x10], eax`	SetupPenBrush
$L'_1 = L_1 + 1$	`pop ebp`; `add eax, 1`	(misc)	`mov edi, eax`; `lea eax, [edi+1]`	GetBilinearFilteredSample
$v_1 = L'_1$	`push 0x80000000`; `push edi`; `mov [ebp-4], eax`	asinhrat	`mov [ebp-4], eax`; `push [esi+30]`; `mov eax, [ebp+8]`; `push [esi+0x2c]`; `push [ebp+0x10]`	BltReplace
$L_2 = arraySize$	`mov eax, [ebp+8]`; `mov ecx, [ebp+0x14]`	_Vector_iterator	`mov edi, [ebp+0x14]`; `and [0x108b40c], 0`; `and [0x108b408], 0`; `mov [0x108a2b0], ebx`; `mov [0x108a2ac], edi`; `mov [0x108b410], ebx`	PGSSkeletalStrokeHelper
$L_{12} = v_1$	`mov eax, [ebp-4]`; `mov edi, [ebp+8]`	scale	`mov ecx, [ebp-4]`; `mov edx, [ebp-8]`	AddRef
$L_6 = \&CipherText$	`push [ebp-4]`; `mov eax, [ebp+0xc]`	CreateDecoderFromResource	`mov eax[ebp+0xc]`; `mov esi, ecx`	CWIAMgr::Acquire
⋮	⋮		⋮	

Unfortunately, semantic-based detectors typically require a model of malicious behavior to which untrusted software can be compared. By definition, good models rarely exist for zero-day exploits. Moreover, fundamental limits of computability dictate that there will always be some software implementations too opaque for such detectors to learn [18], and that will therefore either be conservatively rejected (potentially harming mission-critical programs) or unsafely permitted (inviting attacks).

Software monitoring (e.g., via virtualization) avoids some of these limitations by catching malicious behavior as it occurs instead of trying to predict it statically. However, significant classes of mission-critical software cannot be virtualized (e.g., the PLCs infected by Stuxnet, which interface with hardware), and the monitors themselves are often susceptible to compromise. For example, the Java VM, despite its array of sandboxing, type-checking, and object encapsulation safety measures, has been plagued with about 20 highest-severity zero-day vulnerabilities during the first quarter of 2013 alone [19].

Our ongoing work at UTD has been exploring an unusual alternative: automated, preemptive, binary *transformation* of untrusted code [20]–[23]. Rather than merely inspecting untrusted programs for malicious programming, or executing them as-is in a heavily monitored environment, our work preemptively modifies untrusted binary programs before they execute to make them incapable of violating system- or user-specified safety policies. The code transformations are carefully designed so that non-malicious code suffers no ill effects; its (safe) behaviors are preserved. However, malicious (possibly concealed) programming is rendered inoperable.

An intriguing advantage is that the transformation algorithm need not actually detect all malicious code in order to successfully deactivate it. For example, to secure an operation $f(x)$ whose safety depends on the runtime value of argument x (which cannot be statically predicted in general), it can conservatively replace $f(x)$ with alternative code that proceeds at runtime only when x is safe. In this way the need to statically predict whether x is potentially unsafe is avoided, bypassing the historic computability limitations of static analyses.

Moreover, unlike traditional runtime monitors or virtual machines, code transformation results in a relatively small, self-contained, self-monitoring, binary program that is amenable to automated, formal security analysis. Our prior work has developed fully automated verification systems that can independently prove that each transformed program has been rendered incapable of violating the security policy [20], [22]. This eliminates the need to trust the transformation algorithm or any of the (possibly complex) infrastructure that was needed to implement it, providing exceptional levels of security assurance.

As an example, Fig. 1 illustrates the system architecture of REINS [20], which automatically transforms Windows native code applications by instrumenting them with security checks that enforce a specified safety policy. REINS takes only the raw application binary and the policy as input. It requires no application source code or debugging information, and its transformations are agnostic to the source language, compiler, and tool chain used to generate the original program. Formal verification of the transformed code's safety is quick, taking about 0.4 s/MB on a standard laptop, and the transformations introduce an average of just 2.4% runtime overhead. In general, REINS rarely knows whether the programs it transforms were malicious. It blindly transforms them such that any policy-violating functionality that may have been lurking within them is rendered unreachable or inert.

Of course REINS is not a silver bullet. It still has the limitation of requiring a security policy specification to enforce. If the correct policy is unknown (e.g., the threat is zero-day), an adequate specification might not be available. However, in some cases even this requirement can be relaxed.

For example, the STIR system [21] protects binary native code applications from *return-oriented programming* (ROP) attacks [24] by imbuing them with the power to randomize their own basic block layouts each time they are loaded. In order for ROP attacks to be effective, attackers must typically predict the address locations of abusable gadgets in victim process address spaces. By frequently re-randomizing all such locations, successful prediction through rote probing becomes vanishingly improbable. Thus, ROP attacks are stymied by the transformation without the need for a precise and comprehensive policy that identifies all exploitable software vulnerabilities.

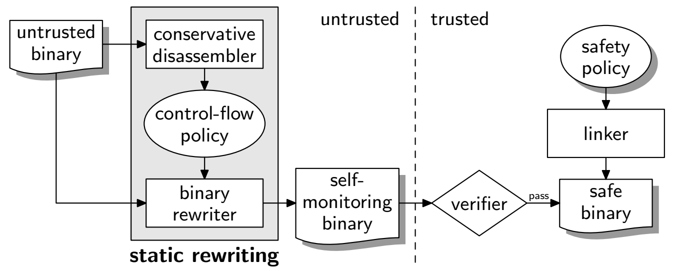

Fig. 1. System architecture of a binary code transformation system

IV. Conclusions and Future Work

While research on offensive software technologies has historically focused on anticipating future attacks, the rise of cyberspace as a theater of war now also demands that such research be additionally applied for purposes of active national defense through strong offensive capabilities. Next-generation software stealth technologies are likely to be an important part of tomorrow's cyber-arsenals.

To defend against this futuristic weaponry, there is a tremendous need for research on better malware detection paradigms that go beyond mere syntactic inspection or incomplete simulation of untrusted code. Automated, preemptive, binary-transformation is one promising alternative that offers numerous advantages over traditional approaches. By conservatively modifying potentially dangerous byte sequences in untrusted programs before they run, these transformation algorithms can successfully preclude malicious software behavior even when static analyses conservatively fail, and when simulations prove inconclusive. Recent advances have shown that the technology is applicable to large-scale, production-level software products even when their source code and implementation details are unavailable. Formal, automated verification provides the highest level of security assurance for the resulting transformed code.

Although these offensive and defensive technologies offer great promise, there is still much work that remains to be done. On the offensive front, malware stealth technologies like Frankenstein offer resistance to syntax-based detection but not semantic-based detection. Future work must therefore consider reactively adaptive semantic obfuscation of payloads.

On the defensive front, the effectiveness of code-transformation often remains contingent upon precise, comprehensive security policies that formally distinguish permissible from impermissible software behavior. Such policies are extremely difficult to manually formulate, even for domain experts. Therefore, automated inference of unsafe software and user behavior is needed as a support infrastructure to ensure that these techniques scale with the increasingly diverse range of software systems and security needs posed by government and industry. Our ongoing work on secure data- and stream-mining is introducing new, more powerful learning algorithms for these domains [25]–[27].

Acknowledgments

The author thanks Vishwath Mohan for his assemblage of the data reflected in Table I.

This material is based upon work supported by U.S. Air Force Office of Scientific Research (AFOSR) grant FA9550-10-1-0088, and by National Science Foundation (NSF) CAREER award #1054629. Any opinions, findings, conclusions, or recommendations expressed are those of the author and do not necessarily reflect the views of the AFOSR or NSF.

References

[1] F. Cohen, "Computer viruses," Ph.D. dissertation, University of Southern California, 1986.

[2] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro, "The Cornell commission: On Morris and the Worm," *Communications of the ACM*, vol. 32, no. 6, pp. 706–709, 1989.

[3] S. Kumar and E. H. Spafford, "A generic virus scanner in C++," in *Proc. 8th Computer Security Applications Conf.*, 1992, pp. 210–219.

[4] L. Bilge and T. Dumitras, "Before we knew it: An empirical study of zero-day attacks in the real world," in *Proc. 19th ACM Conf. Computer and Communications Security*, 2012, pp. 833–844.

[5] A. K. Sood and R. J. Enbody, "Crimeware-as-a-service—a survey of commoditized crimeware in the underground market," *Int. J. of Critical Infrastructure Protection*, vol. 6, no. 1, pp. 28–38, 2013.

[6] Trend Micro, "Zero-days hit users hard at the start of the year," TrendLabs Q1 2013 Security Roundup, April 2013.

[7] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *Proc. 16th ACM Conf. Computer and Communications Security*, 2009, pp. 635–647.

[8] A. Shalal-Esa, "Six U.S. Air Force cyber capabilities designated 'weapons'," *Reuters*, April 2013.

[9] Defense Advanced Research Projects Agency, "Foundational cyberwarfare (plan X)," DARPA-BAA-13-02, November 2012.

[10] D. E. Sanger, "Obama order sped up wave of cyberattacks against Iran," *The New York Times*, June 2012.

[11] D. Kushner, "The real story of Stuxnet," *IEEE Spectrum*, March 2013.

[12] M. Hypponen, "Why antivirus companies like mine failed to catch Flame and Stuxnet," *Wired*, June 2012.

[13] K. W. Hamlen, V. Mohan, M. M. Masud, L. Khan, and B. Thuraisingham, "Exploiting an antivirus interface," *Computer Standards & Interfaces*, vol. 31, no. 6, pp. 1182–1189, 2009.

[14] V. Mohan and K. W. Hamlen, "Frankenstein: Stitching malware from benign binaries," in *Proc. 6th USENIX Workshop on Offensive Technologies*, 2012, pp. 77–84.

[15] T. Cross, "A thing of threads and patches," *The Economist*, August 2012.

[16] J. Aron, "Frankenstein virus creates malware by pilfering code," *New Scientist*, August 2012.

[17] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *ACM Trans. Programming Languages and Systems*, vol. 30, no. 5, 2008.

[18] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annual Computer Security Applications Conf.*, 2007, pp. 421–430.

[19] C. Humble, "Java still vulnerable, despite latest patches," *InfoQ*, April 2013.

[20] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proc. 28th Annual Computer Security Applications Conf.*, 2012, pp. 299–308.

[21] ——, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. 19th ACM Conf. Computer and Communications Security*, 2012, pp. 157–168.

[22] K. W. Hamlen, M. M. Jones, and M. Sridhar, "Aspect-oriented runtime monitor certification," in *Proc. 18th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, 2012, pp. 126–140.

[23] M. Jones and K. W. Hamlen, "Enforcing IRM security policies: Two case studies," in *Proc. IEEE Int. Conf. Intelligence and Security Informatics*, 2009, pp. 214–216.

[24] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Computer and Communications Security*, 2007, pp. 552–561.

[25] P. Parveen, Z. R. Weger, B. Thuraisingham, K. W. Hamlen, and L. Khan, "Supervised learning for insider threat detection using stream mining," in *Proc. 23rd IEEE Int. Conf. Tools with Artificial Intelligence*, 2011, pp. 1032–1039.

[26] P. Parveen, J. Evans, B. Thuraisingham, K. W. Hamlen, and L. Khan, "Insider threat detection using stream mining and graph mining," in *Proc. 3rd IEEE Conf. Privacy, Security, Risk and Trust*, 2011, pp. 1102–1110.

[27] M. M. Masud, T. M. Al-Khateeb, K. W. Hamlen, J. Gao, L. Khan, J. Han, and B. Thuraisingham, "Cloud-based malware detection for evolving data streams," *ACM Trans. Management Information Syst.*, vol. 2, no. 3, 2011.