Frankenstein: A Tale of Horror and Logic Programming*

Vishwath Mohan and Kevin W. Hamlen
The University of Texas at Dallas

Abstract

Frankenstein is a new, more stealthy malware propagation system that evades feature-based detection through camouflage rather than mere diversity. Rather than mutating purely randomly as it propagates, it stitches together instruction sequences harvested from programs that have already been classified as benign by local defenses. The resulting mutants are each unique, yet fully composed of benign-looking code. This makes it hard for feature-based malware detectors to find a signature that reliably identifies all variants.

Frankenstein relies on concepts from constraint logic programming to correctly and quickly identify potential instruction sequence orderings that produce behavior semantically equivalent to the original malware. This article presents the context for Frankenstein's development, and explains how logic programming became the tool of choice for crafting a next-generation cyber weapon.

1 Introduction

In the last few years, the world has witnessed an emerging trend towards the use of state-sponsored malware for espionage and cyber-warfare. The wide media coverage of Stuxnet, DuQu, and Flame proves that well-designed malware can efficiently infiltrate well-defended, and even isolated networks in order to covertly monitor communications and financial transactions. Such cyber intrigues have already captured the public imagination, inspiring a recent Bond movie, for example [4].

Malware like Stuxnet requires a sophistication beyond that exhibited by the everyday malware that harasses typical end-users, because its targets are specific and strongly defended, and its missions are often longer and more complex. End-user systems present a comparatively wide target—malware authors are interested in infecting as many systems as they can, without much regard to

*This research was supported in part by AFOSR active defense award FA9550-10-1-0088 and NSF CAREER award #1054629. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR or NSF.

where or whom. Such malware need only be strong enough to defeat relatively weak defenses in order to infect many machines. In contrast, targeted attacks like Stuxnet are designed to infiltrate specific networks that are well-defended and often closed.

In addition, stealth is a high priority for malware. Everyday malware must only remain undetected for long enough to propagate and carry out a simple mission, such as deleting files. Targeted malware must often remain undetected behind enemy lines for months or years at a time in order to carry out longer-term missions. It is estimated that Flame operated in the wild for over 2 years before it was discovered [3]. The need for such stealth raises an interesting challenge for cyber-warriors:

> *How can cyber weapons like Stuxnet, which target closed, hostile environments, be crafted in such a way that even if one or a few instances are discovered, the rest continue to operate and evade discovery?*

Most modern malware approaches the stealth problem through *polymorphism*—it randomly mutates as it propagates in order to avoid making exact copies. The most common form of polymorphism is *packing*, in which the malware encrypts itself with a randomly chosen key each time it propagates. *Oligomorphic* malware adopts a similar approach, applying a reversible function, such as `xor`, to its payload with a randomly chosen one-time pad. Botnet infections achieve even higher diversity by periodically downloading completely new versions of themselves from a *command-and-control* (C&C) center. The most sophisticated form of polymorphism, called *metamorphism*, directly mutates its own binary code on propagation—for example, by reordering or inserting instructions, reallocating registers, or non-deterministically recompiling itself from an intermediate representation.

Unsurprisingly, anti-malware products have developed an array of detection technologies to identify malware despite these mutations. The obvious approach is *semantic detection*, which classifies programs as malicious or benign based on their behavior rather than their syntax, typically by simulating them in a secure VM. However, semantic detection is ineffective against previously unseen behaviors, such as zero-days; it cannot reliably detect time bombs, which unleash their malicious behaviors only after days or weeks of waiting; and is impractical to apply indiscriminately to the millions of software programs on large networks.

As a result, state-of-the-art malware detection still relies heavily upon syntax-based heuristics—particularly *signature-matching*—as a first step toward identifying suspicious programs worthy of greater scrutiny. Packed malware exhibits statistical anomalies, such as high entropy, that can serve as red flags to defenders. Oligomorphic obfuscations are reversible. C&C updating is potentially detectable at the firewall level and impossible in closed networks. Metamorphic obfuscators pose some of the greatest challenges, but are typically armed with only a limited set of code transforms. Once this set is known, it can be automatically reversed or normalized to detect all variants.

The ongoing efforts of attackers to create more stealthy obfuscations, and of defenders to respond with more general detection strategies, has resulted in

Table 1: Examples of logical predicates

Predicate	**Semantic Definition**	**Suitable Gadgets**
noop	—	NoOp
move(L_1,L_2)	$L_1 \leftarrow L_2$	All Loads/Stores
add(L_1,L_2,L_3)	$L_1 \leftarrow L_2 + L_3$	Arithmetic
sub(L_1,L_2,L_3)	$L_1 \leftarrow L_2 - L_3$	Arithmetic
jump(n, *Why*)	Jump n blueprint steps if *Why* holds	DirectBranch, ConditionalBranch

an ever escalating cyber arms race. Recently we introduced a new malware stealth technology—Frankenstein—that may be the next weapon of choice in this race [1, 2, 5]. Frankenstein is a next-generation metamorphic obfuscator that can re-implement itself fully automatically entirely from code fragments pilfered from other programs. By composing its mutants entirely out of code deemed benign by the host system, it is resilient against many feature-based detection mechanisms. It uses no encryption, so is immune to entropy-based detection; its mutants have no common byte features, thereby evading signature-matching; it performs no runtime self-modification, so is not identifiable by write-then-execute detection; and it does not rely on C&C communication, making it applicable to closed network targets. Moreover, its use of harvested code imbues it with an ever-expanding, potentially infinite corpus of available code transformations.

2 Bringing Frankenstein To Life

The idea of reusing a program's instructions for unintended purposes is not in itself new. *Return-oriented Programming* (ROP) attacks [8] find and abuse *gadgets*—short code sequences that are part of a victim process's code and that end with a return instruction. The attacker exploits a stack vulnerability to inject addresses of these gadgets onto the stack, causing the victim to execute its own code in an attacker-specified order to cause damage. Recent work has shown that large binaries typically contain Turing-complete gadget collections [6], affording attackers arbitrary, malicious functionality.

Frankenstein generalizes this idea to generate obfuscated mutants. It mines benign binaries for gadgets that are potentially useful for implementing its own functionality. Semantic correctness of mutants is ensured by matching gadget candidates to a *semantic blueprint*. The semantic blueprint describes the malicious payload's desired behavior as a sequence of abstract machine states. Each abstract step is represented as a logical predicate consisting of an atomic term and zero or more locations (i.e., registers or memory addresses). For example, the *move* predicate encodes the semantic task of moving a value from one location to another.

Table 1 shows some of the lowest-level (i.e., least abstract) predicates available

Table 2: Gadget types

Gadget Type (t)	**Input** (ℓ)	**Params** (p)	**Semantic Definition**
NoOp	—	—	no state change
Branch	*off*	—	$EIP \leftarrow EIP + off$
CondBranch	*off*	$\bowtie_{cmp}$, r_1, r_2	$EIP \leftarrow EIP + off$ if $r_1 \bowtie_{cmp} r_2$
LoadReg	r_1, r_2	—	$r_1 \leftarrow r_2$
LoadConst	r_1, *val*	—	$r_1 \leftarrow val$
LoadMemAddr	r_1, *addr*	—	$r_1 \leftarrow [addr]$
LoadMemReg	r_1, r_2	s, d	$r_1 \leftarrow [r_2 * s + d]$
StoreMemAddr	r_1, *addr*	—	$[addr] \leftarrow r_1$
StoreMemReg	r_1, r_2	s, d	$[r_2 * s + d] \leftarrow r_1$
Arithmetic	r_1, r_2, r_3	$\diamond_{aop}$	$r_1 \leftarrow r_2 \diamond_{aop} r_3$

to malware authors when expressing payloads as semantic blueprints. Frankenstein can implement blueprints containing more abstract predicates by searching for combinations of the primitive building blocks in the table. This turns the mutation problem into a search problem: blueprint steps can be thought of as breadcrumbs through a semantic maze. To generate a mutant, Frankenstein searches for a path through the maze that visits all the breadcrumbs (i.e., abstract states) and reaches the exit (i.e., implements the attack). Breadcrumbs that are wider spaced increase mutant diversity, but also make the search more difficult.

Logic programming is the natural paradigm in which to implement such a semantic search. To create mutants, Frankenstein first constructs a database of gadgets from randomly selected benign files on the host system. It does this by extracting and running candidate instruction sequences from the binaries through an abstract evaluator and measuring their effects on a symbolic machine state. This state is unified against a set of *gadget types* [7] that Frankenstein knows about. Table 2 lists some of the types and their meanings. Unification allows Frankenstein to recognize instruction sequences as potential instantiations of gadget types needed to implement the blueprint, as well as to find suitable arrangements of the gadgets for the final payload.

The search for an adequate path through the maze frequently requires some backtracking. For example, many gadgets have undesired side-effects that clobber locations that are auxiliary to the gadget's desired effect. To compensate, Frankenstein must extend the predicates in the blueprint to include variables representing clobber lists, as well as generated constraints that prevent the parameters of predicates from interfering with each other.

At the end of the process, the selected gadget sequence is stamped out into an executable binary file format, resulting in a completely unique instantiation of the malware composed entirely from stolen code fragments.

3 Conclusion

Frankenstein is a new approach to malware obfuscation designed for extended, autonomous stealth. Since its mutation relies on using existing code from benign binaries, it effectively converts mutation into a search problem consisting of two phases: gadget discovery and gadget arrangement. By formalizing these phases as unification with backtracking search, Frankenstein's core can be viewed as a constraint logic program—one that can be used to hide malware in plain sight.

References

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

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

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

[4] J. A. Kaplan. James Bond film 'Skyfall' inspired by Stuxnet virus. *Fox News*, November 2012.

[5] V. Mohan and K. W. Hamlen. Frankenstein: Stitching malware from benign binaries. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, August 2012.

[6] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[7] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, 2011.

[8] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.