Frankenstein:
Stitching Malware from Benign Binaries

Vishwath Mohan and Kevin W. Hamlen
School of Electrical and Computer Science
University of Texas at Dallas

Abstract—**This paper proposes a new self-camouflaging malware propagation system, *Frankenstein*, that overcomes shortcomings in the current generation of metamorphic malware. Specifically, although mutants produced by current state-of-the-art metamorphic engines are diverse, they still contain many characteristic binary features that reliably distinguish them from benign software.**

Frankenstein forgoes the concept of a metamorphic engine and instead creates mutants by stitching together instructions from non-malicious programs that have been classified as benign by local defenses. This makes it more difficult for feature-based malware detectors to reliably use those byte sequences as a signature to detect the malware. The instruction sequence harvesting process leverages recent advances in gadget discovery for return-oriented programming. Preliminary tests show that mining just a few local programs is sufficient to provide enough gadgets to implement arbitrary functionality.

I. Introduction

The underground economy associated with malware has grown rapidly in the last few years. Recent studies demonstrate that malware authors no longer need concern themselves with the distribution of their creations to end user systems; they can leave that task to specialized pay-per-install services [1]. In such an environment, the primary concern of malware is evading detection on infected machines as it carries out its malicious task.

End-user machines are protected by real-time detection systems that rely heavily on static analysis. Static analysis is favored because it is faster and consumes fewer resources relative to dynamic methods [2]–[6]. Resilience against static analyses is therefore a high priority for malware obfuscation technologies.

Oligormorphism, *polymorphism*, *virtualization-based obfuscation*, and *metamorphism* are the main techniques used to evade static analyses. Oligomorphism uses simple invertible operations, such as XOR, to transform the malicious code and hide distinguishing features. The code is then recovered by inverting the operation to deploy the obfuscated payload. Polymorphism is an advancement of the same concept that encrypts most of the malicious code before propagation, leaving only a decryption routine, which unpacks the malicious code before execution.

Both oligomorphism and polymorphism are statically detectable with high probability using statistical or semantic techniques. Encrypting or otherwise transforming the code significantly changes statistical characteristics of the program, such as byte frequency [7], [8] and entropy [9], prompting defenses to classify them as suspicious. Subsequent, more computationally expensive analyses can then be judiciously applied to these suspicious binaries to identify malware.

More advanced polymorphic techniques, such as polymorphic blending, try to overcome this weakness by modifying statistical information of binaries via byte padding or substitution [10]. However, the malware's decryption routine (which must remain unencrypted) is often sufficiently unique that it can be used as a signature to detect an entire family of polymorphic malware. Semantic analysis techniques can therefore single out and identify the unpacker to detect malware family members [11]. Virtualization-based obfuscators express malware as bytecode that is interpreted at runtime by a custom VM. However, this shifts the obfuscation burden to concealing the (usually large) in-lined, custom VM.

Metamorphism is a more advanced approach to obfuscation that, in lieu of encryption, replaces its malicious code sequences with semantically equivalent code during propagation. This is accomplished using a metamorphic engine that processes binary code and modifies it to output a structurally different but semantically identical copy. Since the mutations all consist of purely non-encrypted, plaintext code, they tend to exhibit statistical properties indistinguishable from other non-encrypted, benign software.

Simple metamorphic engines mutate by adding padding (e.g., dead code), permuting registers, or inserting semantic no-ops consisting of state-preserving instructions and loops. More advanced engines additionally perform function reordering, control flow modification or data structure modification.

Current metamorphic engines focus on achieving a high diversity of mutants in an effort to decrease the probability that the mutants share any features that can serve as a basis for signature-based detection. However, diversity does not necessarily lead to indistinguishability. For example, malware signatures that whitelist features (i.e., those that classify binaries as suspicious if they *do not* contain certain features) actually become more effective as mutant diversity increases. Similarly, reverse-engineering current metamorphic engines often reveals patterns that can be exploited to derive a suitable signature for detection.

Our system, Frankenstein, therefore adopts a different approach to metamorphism that is inspired by recent advances in return-oriented programming. Return-oriented programming searches the address spaces of victim binaries for *gadgets*—instruction sequences that end with the return instruction [12].

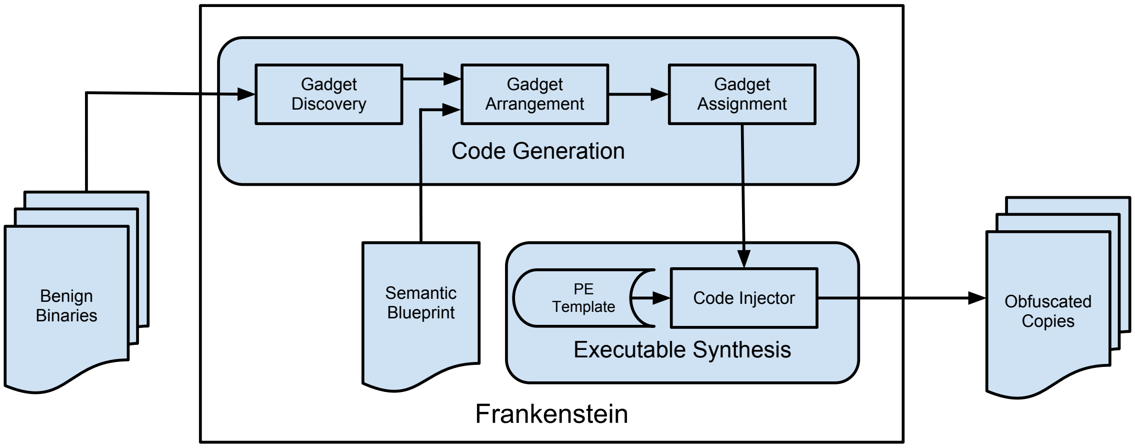

Fig. 1. High-level architecture of Frankenstein

Previous work has shown that a sufficiently large code base (such as the standard C library *libc*), suffices to find a Turing-complete set of gadgets that can be used to exploit known vulnerabilities [13]. Others have also shown that searching for gadgets is an automatable task [14].

We apply the idea of harvesting instructions to obfuscate malicious code. Rather than using a metamorphic engine to mutate, we stitch together harvested code sequences from benign files on the infected system to create a semantically equivalent binary. By composing the new binary entirely out of byte sequences common to benign-classified binaries, the resulting mutants are less likely to match signatures that include both whitelisting and blacklisting of binary features.

Our main contribution is a new method to obfuscate malware that works by synthesizing copies entirely from byte sequences that have already been classified as benign by local defenses. In doing so, we demonstrate a heretofore unrecognized synergy between research on metamorphic obfuscation and that on return-oriented programming.

As a proof-of-concept, we present a toy implementation consisting of a binary obfuscator that generates stand-alone x86 native code mutants from a specification (described in Section II), but that does not self-propagate. Experiments focus on obfuscating code whose size and functionality is representative of the small, unencryptable portion of malware (e.g., unpackers) rather than full malware payloads, to which alternative approaches are usually applicable (e.g., mimimorphism [15]). Thus, we envision our approach as a complement to these alternatives rather than a replacement.

In Section II we present a high-level overview of Frankenstein and discuss its constituent components. Section III contains the details of our prototype implementation, and Section IV reports experimental results. Related work is discussed in Section V. Finally, Section VI summarizes and discusses opportunities for future work.

II. Design

Frankstein searches programs on the local machine for gadgets, which it composes to form a semantically identical copy of itself. The different components of Frankenstein are diagrammatically represented in Figure 1. Below, we describe our notion of a gadget, which differs from its usual definition in the context of return-oriented programming, and then discuss each of the components in more detail.

A. Gadgets

Our definition of a gadget is a more relaxed version of that needed for return-oriented exploits [12], [14]. Return-oriented programming, which is a form of control-flow hijacking, relies on the `ret` instruction to transfer control from one sequence of instructions to the next. Thus, only sequences that end with a `ret` constitute viable gadgets. Since we statically stitch gadgets together, we are not bound by this constraint. For our purposes, a gadget is any sequence of bytes that are interpretable as valid x86 instructions.

A second difference from return-oriented programming is that, as a purely static approach, we need not restrict the search to the address space of a currently running process. Gadgets are therefore harvested from executable file images on victim systems. Both these facts afford us a much larger pool of potential gadgets from which to construct mutants. This is advantageous since we would like each copy of Frankenstein to differ as much as possible from all other copies.

Gadgets are categorized by their *type* (a semantic abstraction of the kind of task they perform) and by a set of *parameters* (instantiated with values that specialize the gadget to a particular task). For example, the **MovReg** type represents any gadget that moves a value from one register to another. All **MovReg** gadgets have two parameters, *InReg* and *OutReg*, that represent the operation $OutReg \leftarrow InReg$.

TABLE I
GADGET TYPES

Gadget Type (t)	**Input** (ℓ)	**Parameters** (p)	**Semantic Definition**
NoOp	—	—	No change to memory or registers
DirectBranch	Offset	—	$\mathit{EIP} \leftarrow \mathit{EIP} + \mathit{Offset}$
DirectConditionalBranch	Offset	$\bowtie_{cmp}$, Reg_1, Reg_2	$\mathit{EIP} \leftarrow \mathit{EIP} + \mathit{Offset}$ if $\mathit{Reg}_1 \bowtie_{cmp} \mathit{Reg}_2$
LoadReg	OutReg, InReg	—	$\mathit{OutReg} \leftarrow \mathit{InReg}$
LoadConst	OutReg, Value	—	$\mathit{OutReg} \leftarrow \mathit{Value}$
LoadMemAddr	OutReg, Addr	—	$\mathit{OutReg} \leftarrow [\mathit{Addr}]$
LoadMemReg	OutReg, $\mathit{AddrReg}$	Scale, Disp	$\mathit{OutReg} \leftarrow [\mathit{AddrReg} * \mathit{Scale} + \mathit{Disp}]$
StoreMemAddr	InReg, Addr	—	$[\mathit{Addr}] \leftarrow \mathit{InReg}$
StoreMemReg	InReg, $\mathit{AddrReg}$	Scale, Disp	$[\mathit{AddrReg} * \mathit{Scale} + \mathit{Disp}] \leftarrow \mathit{InReg}$
Arithmetic	OutReg, InReg_1, InReg_2	$\diamond_{aop}$	$\mathit{OutReg} \leftarrow \mathit{InReg}_1 \diamond_{aop} \mathit{InReg}_2$

TABLE II
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

```
hypotenuse_squared :-
   mov(L1, '[0x401248]'),
   mov(L2, '[0x40124C]'),
   mul(L3, L1, L1), mul(L4, L2, L2)
   add(L5, L3, L4), mov('EAX', L5).
```

Fig. 2. A semantic blueprint to compute the square of a triangle's hypotenuse

A complete set of gadget types, their associated parameters, and the semantic task that each encodes is given in Table I. The symbols $\bowtie_{cmp}$ and $\diamond_{aop}$ represent integer comparison and modular arithmetic operations, respectively. The collection is Turing-complete, and therefore suffices to build arbitrary computations. In contrast to gadget types for return-oriented programming [14], our collection includes types for conditional and non-conditional branches. These are unnecessary for return-oriented programming since in that context every gadget is reached via an appropriate return instruction injected into the stack. To better resemble benign software, Frankenstein uses more conventional control-flows that include standard branching instructions.

Every gadget is also associated with a *clobber list*, which represents secondary register and memory locations whose values the gadget modifies. The clobber list is used to find a sequence of gadgets that do not interfere with one another.

The types defined in the table are sufficient to carry out our initial experiments, but future work should consider extending the table to support obfuscation of more complex tasks.

B. Semantic Blueprint

Typical metamorphic malware recompiles itself from a bytecode intermediate language during propagation. Each mutant carries a freshly obfuscated intermediate form of itself for this purpose. (The intermediate form is data, which is easier to obfuscate than code.) In contrast, Frankenstein propagates by re-synthesizing itself from a more abstract *semantic blueprint*. The semantic blueprint is a sequence of abstract machine states, where each step in the sequence is represented as a logical predicate. Each predicate is a combination of an atomic term and zero or more *locations*. A location can be a specific register, memory address, immediate value, or a variable that refers to an arbitrary register or memory location. The **jump** predicate instead has arguments consisting of a relative offset into the blueprint's list of states and an optional condition.

A subset of logical predicates used by Frankenstein is shown in Table II. For example, the **move** predicate is an abstraction of the movement of a value from one location L_2 to another L_1. Thus, depending on the values of its locations, a **move** predicate might be satisfiable by any of the **Load*** or **Store*** gadget types. The flexibility of a predicate to match multiple gadget types allows the semantic blueprint to more abstractly encode *what* is computed rather than *how* the computation is carried out. This in turn allows for a diverse set of gadgets to match a given portion of the semantic blueprint.

Figure 2 shows how predicates can be chained together to form a clause. In the example, the memory locations `0x401248` and `0x40124C` contain the values of two sides of a right-angled triangle, and the calculated length of the square of the hypotenuse is stored in the `EAX` register. Variables L1–L5 can refer to memory locations or registers.

The level of abstraction (and with it the diversity of mutants) can be tuned by adjusting the granularity of the predicates in the blueprint. It is possible to create layers of predicates that can each be expressed as clauses of lower-layer predicates, each layer effectively abstracting higher-level operations. For example, two consecutive predicates that increment register r and then multiply it by 2 could be replaced by a single predicate that computes $2(r+1)$. The resulting predicate would be satisfied by new gadget sequences, such as one that first multiples by 2 and then adds 2. The trade-off is the greater search time required to discover implementations of more abstract gadgets. If the predicates are too abstract, the search becomes intractable.

Since gadget discovery is based on search, our proof-of-concept implementation expresses semantic blueprints as predicates written a logic programming language (Prolog).

Algorithm 1 Gadget discovery

```
Input: $\sigma_0$ (initial symbolic machine state), and
       $[i_1, \ldots, i_n]$ (instruction sequence)
Output: $G \subseteq T \times \Phi$ (matching gadget types)
  for $j = 1$ to $n$ do
    $\sigma_j \leftarrow \mathcal{E}[\![i_j]\!]\sigma_{j-1}$
  end for
  $G \leftarrow \emptyset$
  for all $t \in T$ do
    if $\mathcal{U}(t, \sigma_n)$ is defined then
      $\phi \leftarrow \mathcal{U}(t, \sigma_n)$
      $G \leftarrow G \cup \{(t, \phi)\}$
    end if
  end for
  return $G$
```

While a full Prolog search engine is obviously too heavyweight for inclusion in real malware, effective gadget search does not require the full capabilities of logic programming. We expect a combination of unification and simple depth-first search to suffice, and consequently believe that a much slimmer implementation is possible.

C. Gadget Discovery

The majority of the obfuscation process involves finding a suitable set of gadgets that can be used to implement the semantic blueprint. The search process proceeds similarly to gadget searches for return-oriented programming [14], but with several variations reflecting our different focus (obfuscation as opposed to finding one viable sequence from a limited code base).

In the discovery phase, Frankenstein searches the local file system for binaries. From our experiments, our experience has been that 2–3 binaries from the *system32* folder suffices to provide a code base from which to harvest a diverse, Turing-complete set of gadgets on Microsoft Windows systems. Frankenstein starts by collecting byte sequences from the code sections of these binaries using a variable-length sliding window. The sliding window approach is simpler than implementing (and obfuscating) a full disassembler, and it increases the pool of available gadgets by including for consideration the many misaligned instruction sequences that all benign programs contain (but rarely execute).

Each byte sequence is passed through an instruction decoder to produce an instruction sequence. Sequences containing invalid op-codes or undesirable branches (such as calls or returns) are discarded, and the remaining sequences are tested for gadget viability using Algorithm 1.

Frankenstein performs gadget discovery with the aid of a small abstract evaluator $\mathcal{E} : I \to \Sigma \to \Sigma$ that defines the effect of an instruction $i \in I$ upon a symbolic machine state $\sigma \in \Sigma$. Notation $\mathcal{E}[\![i]\!]\sigma$ denotes the resulting symbolic state, where states $\sigma : \ell \to e$ map locations ℓ (*viz.*, registers, flags, and memory addresses) to symbolic expressions e. For example, each register's initial content is encoded as a fresh symbol in the initial abstract state: $\sigma(eax) = EAX$, and so on.

After composing the effects of all instructions in a candidate sequence, the final symbolic output state is *unified* with each possible gadget type $t \in T$. A gadget type t is conceptually a state predicate, possibly containing uninstantiated parameters p. Unification $\mathcal{U}(t, \sigma')$ succeeds if there exists an instantiation $\phi : \Phi = p \to \mathit{ParamVals}$ of the parameters such that substituting t according to ϕ yields a concrete predicate satisfied by symbolic state σ'. In that case the unification returns the parameter instantiation ϕ. Otherwise $U(t, \sigma')$ is undefined (and the search continues). Two instruction sequences that match a particular gadget type are considered equivalent if they have identical instantiations of all parameters, excluding the clobber list.

It is common for a large instruction sequence to be recognized as multiple valid gadget types when considering its effect on different machine state variables. For example, the instruction sequence

```
mov ebx, dword ptr [eax*4 + 0xc]
mov ecx, eax
inc ecx
```

can be used as a **LoadReg** gadget representing $ecx \leftarrow ebx$, a **loadMemReg** gadget representing $ebx \leftarrow [eax * 4 + \texttt{0xc}]$, or as an **Arithmetic** gadget representing $ecx \leftarrow ecx + 1$. In each case, the clobber list includes all other state variables that are modified. We also include additional constraints for sequences where memory indirection is involved. In our example gadget above, the value of $eax * 4$ must be a valid memory address to ensure that it does not cause a crash when executed. These constraints are expressed in the form of logical clauses as part of the arrangement layer (described below), which ensures that we only find valid solutions.

D. Gadget Arrangement

The next step involves finding a suitable combination of gadget types that match the semantic blueprint. In Frankenstein, gadget arrangement is a natural consequence of the way that the semantic blueprint is defined. The logical predicates that we define in Table II also happen to be the lowest level in the layered approach to constructing predicates described previously. We call this the *arrangement level* because all possible gadget arrangements are expressed in terms of these predicates.

Given a clause defined in terms of higher-level predicates, logic programming can be used to reduce them to multiple clauses composed entirely of arrangement layer predicates, such that the definition of each clause in turn represents one or more potential gadget arrangements. We assume that malware authors have access to arbitrary high-level representations of the code, including requirements, design, and implementation of the payload. They can therefore use this information to express the malware in terms of the higher-level predicates.

At present, our prototype of Frankenstein does not have support for higher level predicates, and expresses blueprints using predicates from the arrangement level only. However, adding higher-level predicates is not conceptually difficult, and we plan to include this feature in future versions of our system.

E. Gadget Assignment

In the last phase, we use the discovered gadgets to find satisfiable assignments for each generated gadget arrangement. We leverage the unification process of logic programming for this purpose, which is well-suited to this problem. Frankenstein begins by converting each discovered gadget into an extended version of one of the predicates defined in Table II. The extension adds two terms to each predicate: a list of clobbered locations and an identification number. The identification number associates each of these predicates with the instruction sequence that the gadget represents, while the clobber list facilitates discovery of sequences of non-interfering gadgets.

Next, the predicates that form the definition of each of the reduced clauses obtained in the gadget arrangement phase above are also extended to include variables that represent a clobber list and identification numbers. To each definition, we also add a generated list of constraints that prevent the parameters of predicates from interfering with one another. Finally, we use constraint logic programming to solve each clause. The full set of solutions obtained represent all the possible gadget assignments that implement the original semantic blueprint.

F. Executable Synthesis

For each successful gadget assignment, Frankenstein masks all external calls in the code by converting them into computed jumps. As a result, Frankenstein's mutants have no noteworthy system calls in their import address tables, concealing them from detectors that rely upon such features for fingerprinting.

The last step is injecting the finished code into a correctly formatted binary so it can be propagated. Frankenstein has a binary parsing component and a seed binary that it uses as a template. For each mutant, it injects the code into the template file and updates all relevant metadata in the header. At this point the new mutants are natively executable programs.

III. Implementation

To test the viability of our approach, we created a prototype stand-alone obfuscator that takes a gadget arrangement as input and produces a working portable executable (PE) file as output. The prototype searches the local system for programs, mines them to discover gadgets, finds a suitable gadget assignment, and realizes it as a PE file. The prototype was implemented in a combination of Python and Prolog. The experiments were performed on a quad-core virtual machine with 3 GB RAM running 64-bit Windows 7. The host machine is an Intel i7 Q6500 quad-core laptop running 64-bit Windows 7.

The gadget discovery, gadget assignment, and duplication phases implement the algorithms described in the previous section. However, the abstract evaluator that analyzes and discovers gadgets currently supports only a limited subset of instructions—about 8 different instructions excluding branches. Even though this greatly reduces the number of gadgets available for incorporation into mutants, it nevertheless suffices to find more than enough gadgets to implement our sample programs.

The discovery module, implemented in Python, takes a set of binaries and a semantic blueprint as input. It outputs a series of Prolog predicates that define each discovered gadget, as well as a Prolog query that represents a viable combination of the gadget types specified in the arrangement. This is delivered to the assignment module, implemented in Prolog, which outputs all discovered solutions to the query. Each solution is then converted into its equivalent byte sequence by the executable synthesis module, implemented in Python, which injects the byte code into a template PE file. For ease of testing, the duplication module contains pre-fabricated templates for function prologues and epilogues, which are used to modularize the synthesized byte sequence as a stand-alone function. We note that function prologues and epilogues could easily be synthesized using the gadget discovery mechanism instead, if so desired.

IV. Experimental Results

We tested our prototype by discovering gadgets in some common Windows binaries. For our results, we only chose gadgets that contained 2–6 instructions. Our results are tabulated in Table III. We recorded the number of gadgets found and time taken both with and without using the sliding window protocol discussed in Section II. Surprisingly, we found that using the sliding window protocol to discover misaligned sequences increased the gadget count by only 34% on average but increased discovery time by 794%, a trade-off that does not seem worthwhile. We conjencture that increasing the number of instructions supported by the abstract evaluator will help balance these ratios somewhat, but that a better strategy is likely to be one that searches for gadgets using a simple fall-through disassembly while increasing the number of binaries mined. All results we discuss hereafter are based on the numbers for the non-sliding window algorithm.

The results show that even with the limited capacity of our prototype, 2–3 binaries are sufficient to bring the number of gadgets above 100,000. On average we discovered about 46 gadgets per KB of code, finding approximately 2338 gadgets per second.

Next, we tested the prototype's ability to synthesize working code. We chose two algorithms for our experiments: insertion sort and a loop that XORs an array of bytes using a one-time pad. Both programs contain operations commonly found within the packers used by conventional malware. The semantic blueprints for these programs are shown in Figures 3 and 4.

The semantic blueprints were reproduced as Prolog queries, with extensions to predicates and added constraints to ensure non-interference between gadgets as detailed previously. In both cases, only gadgets harvested from `explorer.exe` were used. The queries produced over 10,000 viable gadget assignments each, with an average speed of 3 assignments per second.

TABLE III
GADGET DISCOVERY STATISTICS FOR SOME WINDOWS BINARIES

		Without Sliding Window		With Sliding Window	
Binary Name	**File Size** (KB)	**Gadgets Found**	**Time Taken** (s)	**Gadgets Found**	**Time Taken** (s)
gcc.exe	1327	82885	29.70	97163	172.24
calc.exe	758	41914	22.09	60390	189.86
explorer.exe	2555	89617	40.31	127859	429.56
cmd.exe	295	17514	7.17	25008	88.34
notepad.exe	175	4512	1.82	6974	24.39

Input:

L1 = address of data,
L2 = address of one-time pad,
L3 = array length,
L4 = address of encrypted output

Blueprint:

```
xor_encryption :-
  move(L5, 0),
  jump(7, L5 = (L3-1)),
  move(L6, [L1+L5*4]),
  move(L7, [L2+L5*4]),
  xor(L8, L6, L7),
  move([L4+L5*4], L8),
  add(L4, L4, 1),
  add(L5, L5, 1),
  jump(-7, always).
```

Fig. 3. Semantic blueprint for a simple XOR oligomorphism

Input:

L1 = address of array
L2 = length of array

Blueprint:

```
insertion_sort :-
  move(L3, 1),
  jump(14, L3 = L2),
  move(L4, L3),
  move(L5, [L4*4+L1]),
  jump(8, L4 =0),
  jump(7, [L1+(L4-1)*4] < L5),
  sub(L4, L4, 1),
  move(L6, [L4*4+L1]),
  add(L4, L4, 1),
  move([L4*4+L1], L6),
  sub(L4, L4, 1),
  jump(-7, always),
  move([L4*4+L1], L5),
  add(L3, L3, 1),
  jump(-13, always).
```

Fig. 4. Semantic blueprint for insertion sort

This high diversity can be attributed to multiple satisfiable sub-arrangements of gadgets, which can each be combined with every variation of all other sub-arrangements, leading to a combinatorially high number of unique overall arrangements. Although this might appear to produce a large number of similar variants, diversity can be ensured by harvesting gadgets from different sets of binaries and additionally by only selecting assignments that have no gadgets in common with each other.

To better understand the size increase induced by our approach, we compared the sizes of 100 mutants generated by Frankenstein for the one-time pad XOR algorithm against its corresponding compiler generated code. The compiled code was generated with C++ using Visual Studio 2010 with basic security checks turned on and optimization set to full. The mean of the sizes of the generated mutants was 48 bytes compared to the 25 bytes produced by Visual Studio. The variance in size between the generated samples was 16. This shows that the size of a mutant can be expected to be slightly less than double the size of its optimized compiler-generated version, an increase that we feel is an acceptable cost for the benefit of obfuscation.

To assess the binary distribution of the generated mutants, we generated 20 implementations of the XOR blueprint after mining donor program `explorer.exe` for gadgets, and counted the number of n-grams that do not appear in the donor program and were shared by at least m mutants. Our results are tabulated in Table IV. Only about 20 such n-grams are common across 25% of our mutant population, and no n-grams are common across more than 35% of the population. In addition, all the common n-grams are relatively short; no n-grams of length $n \geq 11$ were shared. These are encouraging results because they indicate that few binary n-gram features are relevant for distinguishing malware instances from the benign programs used for gadget harvesting.

Our experimental results are promising, and suggest that developing a more comprehensive Frankenstein tool is a worthwhile endeavor. Specifically, we conjecture that a more comprehensive abstract evaluator that can analyze a greater number of instructions can potentially find far more gadgets, and thus produce mutants that exhibit even greater diversity.

V. RELATED WORK

Gadget-based obfuscation is related to past and present research in the areas of software security and compiler optimization. Below we describe some related work.

TABLE IV
THE NUMBER OF FRESH n-GRAMS SHARED BY AT LEAST m MUTANTS

	mutant subset size (m out of 20)				
n	3	4	5	6	7
2	0	0	0	0	0
3	5	4	4	2	0
4	14	5	4	2	0
5	19	8	4	2	0
6	23	11	4	1	0
7	26	12	3	1	0
8	26	9	1	1	0
9	24	9	1	1	0
10	23	9	1	1	0
11	0	0	0	0	0
total	160 (2.3%)	67 (1.0%)	22 (0.3%)	11 (0.2%)	0 (0%)

A. Return-Oriented Programming

As mentioned previously, Frankenstein borrows the idea of gadgets from return-oriented programming (RoP). RoP is the latest in the evolution of code injection attacks. Such attacks find bytes within a binary's address space that correspond (either intentionally or unintentionally) to a sequence of instructions that perform a specific computation and end with the return instruction. Each such sequence forms a gadget. By searching through a binary for carefully chosen gadgets, it is possible to chain them together to perform arbitrary Turing-complete computations [12].

This chaining of gadgets is achieved by loading the stack with the starting address of each gadget in the chain and transferring control to the first gadget. Every subsequent return instruction then transfers control to the next gadget in the chain. RoP is thus a form of control-flow hijacking, and depends on a known exploit for a given binary in order to smash its stack and fill it with the appropriate addresses.

B. Metamorphic Engines

Metamorphic malware changes the structure of its payload with each generation to evade discovery. Metamorphic engines typically do this using a bottom-up approach: Starting with a disassembler to recover assembly code for the payload, they perform a series of obfuscation phases, followed by application of an assembler to generate mutated native code. Most engines use a combination of the following five phases to obfuscate their payloads [16]: *Garbage insertion* adds unreachable code to the original code. *Code substitution* replaces opcodes with functionally equivalent but structurally different opcodes. *Code insertion* in-lines semantically ineffectual code sequences or harmless computations. *Register swapping* reallocates registers, and *control flow scrambling* adds jumps and reorders function calls.

Frankenstein's gadget-based obfuscation is a more principled approach to metamorphism. It both widens the pool of possible mutations for greater diversity and tailors its mutations to local defenses for more targeted attacks. For example, code substitution in a metamorphic engine is typically performed by comparing instruction opcodes against a fixed table of alternative sequences and then randomly choosing one from amongst them. This induces a degree of randomness with respect to generated code sequences, but does not ensure that the generated sequences are vastly different, nor that they do not contain features widely recognized as malicious.

Frankenstein uses a more top-down approach. By starting with a high-level representation of the payload logic and searching benign files for viable gadgets, it implicitly combines all 5 phases described above. This combination gives it the ability to create mutants with a greater diversity than standard bottom-up approaches.

C. Program Equivalence

Reasoning about program equivalence arises in connection with translation validators and certifying compilers. A translation validator shows that compiler optimizations are semantics-preserving by proving the semantic equivalence of the original program and its compiler-optimized counterpart. Approaches include instrumenting the compiler [17], verifying a simulation relationship between the two programs [18], and constructing value-graphs of the two programs and proving their syntactic equivalence [19], [20].

Certifying compilers prove that object code respects the semantics of the higher-level source code whence it was generated. Most certifying compilers do not prove full program equivalence but instead reduce the complexity by considering only a subset of verifiable properties, such as type-safety [21], [22]. Certifying compilers output object code, type specifications, and code annotations. The annotations and type specifications can then be fed into a certifier which either outputs a proof of correctness or a counterexample that violates type safety.

Our work differs from these related fields in that it does not attempt to provide any formal evidence of semantic equivalence for mutants. That is, although all mutants satisfy the abstract specification whence they were generated, the mutator is under no obligation to provide any evidence of semantic preservation or equivalence. Thus, there is neither validation nor certification. Frankenstein does, however, leverage many theoretical foundations underlying this past research, including pre- and post-conditions for semantic blueprint specification, abstract (symbolic) interpretation for gadget discovery, and abstract machine semantics for gadget analysis and arrangement.

D. Superoptimizing Compilers

Superoptimization refers to the transformation of a loop-free code sequence into the most optimal set of assembly-level instructions. Optimality in this context is decided by the speed of the generated sequence, and hence superoptimizing compilers attempt to find the fastest sequence of assembly instructions that are equivalent to the input code. Such compilers use a lookup table populated with parametrized replacement rules to perform their optimizations. The lookup table can be generated manually as is the case with peephole optimizers, or generated automatically based on a training set of binaries [23].

In an abstract sense, metamorphic obfuscation can be viewed as a superoptimization problem where the model for optimality is not the speed of the generated code, but its dissimilarity to previous versions of the code. However, since similarity of programs is a high-dimensionality metric, there is no one unique solution to the obfuscation problem. (In fact, discovering ever more solutions is a goal of most obfuscation.)

Gadget-based obfuscation can also be compared to superoptimization, but with a subtle difference. The model for optimality in this case is the generated sequence's similarity to existing benign code. Additionally our technique does not use a pre-defined set of replacement rules, forgoing them for a top-down approach to finding viable sequences from benign files.

VI. Conclusion

We presented a new way of obfuscating malware that is fundamentally different from existing metamorphic malware approaches. Rather than recompiling the code purely randomly during propagation, which leads to diverse but potentially distinguishable binary features, our system searches non-malicious programs on the local system for byte sequences that function as the building blocks for semantically equivalent but syntactically new copies. Our experiments showed that mining a few files is both sufficient to obtain high mutant diversity, and fast enough to be a practical mutation strategy.

By creating new copies entirely from byte sequences obtained from benign files, we argue that it becomes significantly more difficult for defenders to infer adequate signatures that reliably distinguish malware from non-malware on victim systems. In particular, signatures that include feature-whitelisting are less effective against our framework than against more conventional forms of obfuscation.

For future work, we intend to implement a more comprehensive system and experiments to verify and extend our preliminary results. If successful, our gadget-stitching approach will constitute a powerful tool for active defense (e.g., offensive cyber-operations), and will highlight the need for stronger, purely semantics-based defenses that place less reliance on syntactic feature detection for early warning.

Acknowledgments

This research was supported in part by AFOSR active defense award FA9550-19-1-0088 and NSF CAREER award #1054629. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect those of the Air Force or National Science Foundation.

References

[1] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution," in *Proc. USENIX Security Sym.*, 2011.

[2] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proc. Conf. on USENIX Security Sym.*, vol. 13, 2004, pp. 271–286.

[3] C. Kreibich and J. Crowcroft, "Honeycomb: Creating intrusion detection signatures using honeypots," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 51–56, 2004.

[4] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Proc. IEEE Sym. on Security & Privacy (S&P)*, 2006, pp. 32–47.

[5] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proc. IEEE Sym. on Security & Privacy (S&P)*, 2005, pp. 226–241.

[6] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proc. Conf. on Sym. on Opearting Systems Design & Implementation (OSDI)*, vol. 6, 2004, pp. 45–60.

[7] K. Wang, J. J. Parekh, and S. J. Stolfo, "Anagram: A content anomaly detector resistant to mimicry attack," in *Proc. Int. Conf. on Recent Advances in Intrusion Detection (RAID)*, 2006, pp. 226–248.

[8] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *Proc. Int. Conf. on Recent Advances in Intrusion Detection (RAID)*, 2004, pp. 203–222.

[9] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[10] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *Proc. Conf. on USENIX Security Symposium*, vol. 15, 2006.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. Int. Conf. on Recent Advances in Intrusion Detection (RAID)*, 2005, pp. 207–226.

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

[13] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. on Information and System Security (TISSEC)*, vol. 15, no. 1, 2012.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. USENIX Security Sym.*, 2011.

[15] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proc. ACM Conf. on Computer and Communications Security (CSS)*, 2010, pp. 536–546.

[16] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[17] A. Kanade, A. Sanyal, and U. Khedker, "A PVS based framework for validating compiler optimizations," in *Proc. IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2006, pp. 108–117.

[18] G. C. Necula, "Translation validation for an optimizing compiler," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2000, pp. 83–94.

[19] J.-B. Tristan, P. Govereau, and G. Morrisett, "Evaluating value-graph translation validation for LLVM," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011, pp. 295–305.

[20] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proc. ACM SIGPLAN-SIGACT Sym. on Principles of Programming Languages (POPL)*, 2009, pp. 264–276.

[21] G. C. Necula and P. Lee, "The design and implementation of a certifying compiler," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1998, pp. 333–344.

[22] Y. Chen, L. Ge, B. Hua, Z. Li, and C. Liu, "Design of a certifying compiler supporting proof of program safety," in *Proc. Joint IEEE/IFIP Sym. on Theoretical Aspects of Software Engineering (TASE)*, 2007, pp. 127–138.

[23] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *Proc. USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 177–192.