Mimimorphism: A New Approach to Binary Code Obfuscation

Zhenyu Wu, Steven Gianvecchio, Mengjun Xie*, and Haining Wang
The College of William and Mary
Williamsburg, VA 23187, USA
{adamwu, srgian, mjxie, hnw}@cs.wm.edu

ABSTRACT

Binary obfuscation plays an essential role in evading malware static analysis and detection. The widely used code obfuscation techniques, such as polymorphism and metamorphism, focus on evading syntax based detection. However, statistic test and semantic analysis techniques have been developed to thwart their evasion attempts. More recent binary obfuscation techniques are divided in their purposes of attacking either statistical or semantic approach, but not both. In this paper, we introduce mimimorphism, a novel binary obfuscation technique with the potential of evading both statistical and semantic detections. Mimimorphic malware uses instruction-syntax-aware high-order mimic functions to transform its binary into mimicry executables that exhibit high similarity to benign programs in terms of statistical properties and semantic characteristics. We implement a prototype of the mimimorphic engine on the Intel x86 platform, and evaluate its capability of evading statistical anomaly detection and semantic analysis detection techniques. Our experimental results demonstrate that the mimicry executables are indistinguishable from benign programs in terms of byte frequency distribution and entropy, as well as control flow fingerprint.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—Security and protection

General Terms

Security

Keywords

binary obfuscation, mimicry attack

*This author is currently affiliated with the Department of Computer Science at University of Arkansas at Little Rock and can be reached at mxxie@ualr.edu.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

1. INTRODUCTION

Real-time malicious binary detection is the first line of defense against malicious software. With the prevalence of anti-malware software nowadays, in order to be executed on a host computer, a piece of binary code is subjected to a number of detection scans during transportation and before execution. Consequently, evading real-time binary detection is critical for malware to succeed in propagation.

To date, real-time malware detection largely relies on static binary analysis, due to its significant speed and resource consumption advantages over dynamic executable analysis [1, 2, 3, 4, 5]. Malware mainly evades static analysis detections through binary obfuscations, namely oligomorphism, polymorphism, and metamorphism [6]. Oligomorphism is used to evade byte sequence signature detections on the malware functional code. It utilizes simple operations such as XOR to scramble malware functional code before propagation, and decodes it while executing. Evolved from oligomorphism, polymorphism encodes malicious code by "packing" (i.e., compressing or encrypting), and then camouflages the "unpacker" (the decompressing or decrypting code) by using binary mutation techniques, such as instruction substitution and register remapping. Instead of packing program binaries, metamorphism generates different instruction combinations to represent the same functional part of a malicious program in its variants. The major techniques employed by metamorphism are binary-level mutations and meta-level transformations. A meta-level transformation first translates binary code into a temporary representation called P-code, then manipulates P-code, and finally composes a new instance from P-code. In this way, metamorphic malware can significantly shuffle its program contents and escape substring signature based detections.

Although the classic polymorphism and metamorphism enable malware to generate many binary instances with different byte patterns, they cannot effectively disguise the presence of malicious code in terms of statistical properties and program semantics. Compression and encryption in polymorphism usually change the statistical characteristics of a program in such a dramatic manner that the malware program can be easily classified as suspicious and be further scrutinized. Exploiting this property, byte-frequency based detection methods such as [7, 8] and entropy based detection methods such as [9] have been proposed to uncover polymorphic malware. Additionally, because compressed or encrypted code segments are no longer executable, they can be easily identified by advanced disassemblers [10]. Such a filtering strategy has been applied to extract the unpackers of polymorphic worms [11]. Meanwhile, metamorphism preserves semantic equivalences between different variants. This property is thus exploited by semantic analysis techniques. For example, MetaAware [12] detects variants of meta-

morphic malware based on analysis of system call and library call instructions.

With the advancements in detection, state-of-the-art evasion techniques are moving beyond polymorphism and metamorphism. Targeting byte-frequency-based static anomaly detection, polymorphic blending attacks [13] manipulate the statistics of malware through byte padding and substitution. Designed to thwart semantic analysis, [14] mutates a program's control flow by transforming constants into an NP-complete problem. However, while flying under the radar of their targeted detection methods, these evasion techniques are ineffective against other analysis techniques. Polymorphic blending can hardly escape semantics based detection such as [11] because byte padding and substitution destroy the executable semantics, making it easy to single out the unpacker code. Similarly, encoding control flow with opaque constants induces identifiable syntax patterns, which can be used as signatures.

In this paper, we introduce mimimorphism, a new approach to binary obfuscation. Leveraging a stenographic technique—mimic function, mimimorphism can transform a malware executable into "mimic-binaries" that resemble ordinary benign programs. Mimimorphism is unique in that instead of targeting at a specific detection approach, it aims to camouflage malware binaries as legitimate executables and thus significantly increases malware's resistance against a range of static statistical tests and semantic analyses. To achieve this goal, we augment a high-order mimic function with customized assembler and disassembler, creating an instruction-syntax-aware mimic function—the core of the mimimorphic engine. The mimimorphic engine captures the high-order instruction-level characteristics of a given set of benign programs, and encodes malicious binaries based on the captured characteristics. As a result, a mimimorphic binary acquires highly similar statistical properties and semantic structures to those of ordinary benign programs.

We implement a prototype of the mimimorphic engine based on 7_{th}-order and 8_{th}-order mimic functions, and evaluate its performance in evading statistical anomaly tests and semantic analysis detections. For statistical anomaly tests, we apply byte frequency and entropy tests against both the mimicry executables and benign programs. The results show that the mimicry executables generated by both 7_{th}-order and 8_{th}-order mimimorphic functions are indistinguishable from benign programs in terms of byte frequency and entropy scores. For semantic analysis detection, we apply the control flow fingerprinting technique against both the mimicry executables and benign programs. The results show that both 7_{th}-order and 8_{th}-order mimimorphic engines introduce a large number of fingerprints that match benign programs, leading to high detection false positive rates. Moreover, even subjected to harder tests by training the detection system with a large number of mimicry executables for common fingerprints extraction, the 8_{th}-order mimimorphic engine can still introduce a significant quantity of benign-program-matching fingerprints.

The remainder of this paper is structured as follows. Section 2 covers related work on malware detection and evasion. Section 3 presents the concept of mimic functions. Section 4 details the design of our mimimorphic engine. Section 5 discusses the implementation issue. Section 6 evaluates the mimimorphic engine against different malware detection and signature generation techniques. Finally, Section 7 concludes the paper with future directions.

2. RELATED WORK

Attackers increasingly employ polymorphic and metamorphic techniques [6] to disguise their attacks and evade intrusion detection systems. The core of these techniques is to change the appearance of malicious code. Although the bit patterns of polymorphic attacks are distinctly different, their malicious functions remain the same. A number of tools have been developed for generating polymorphic shellcode [15, 16] and polymorphic executables [17, 18, 19]. Since polymorphic malware and metamorphic malware are able to significantly transform their contents in propagation, as mentioned in [4, 20], they can effectively circumvent the perimeter of the network intrusion detection systems that are based on contiguous byte string signatures [1, 2, 5].

A basic approach to detecting polymorphic worms is based on byte statistics, such as byte frequency [8] and byte entropy [9]. Wang *et al.* [8] developed a payload-based anomaly detector, PAYL, which profiles the byte distribution of packet payloads and detects the abnormal byte distributions of polymorphic worms. Lyda *et al.* [9] demonstrated that the byte entropy of executables can be used to effectively identify packed or encrypted malware. Tang *et al.* [21] introduced the position-aware distribution signature (PADS), which records a byte frequency distribution for each position in the signature "string" and is capable of detecting certain types of polymorphic worms.

There are several advanced polymorphic attacks [13, 15] designed to evade detection systems based on byte statistics. Detristan *et al.* [15] built a polymorphic engine, CLET, which uses byte padding to approximately match the normal byte distribution. In [13], Fogla *et al.* introduced an advanced polymorphic blending attack that exploits byte substitution and byte padding to achieve a very close match to normal profiles. The polymorphic blending attack is effective in evading 1-gram and 2-gram PAYL [8], and should be effective against other detection methods based on low-order byte distributions. In [22], the problem of generating optimal polymorphic blending attacks is shown to be NP-complete, and a near-optimal heuristic approach is described. A drawback to these mimicry attacks, similar to basic polymorphic attacks, is that the encrypted regions do not contain valid instruction sequences, while the attack vector and decryption routines are still executable, making these regions easily differentiated.

To counter mimicry attacks, higher-order byte patterns have been used in recent detection methods [7, 23]. In [7], Wang *et al.* presented a new anomaly detector, which is capable of detecting a modified polymorphic blending attack [13], called Anagram. Anagram employs a Bloom filter to reduce the computation and storage requirements for modeling higher-order n-grams, in particular, n-grams $2-9$ are chosen for experiments. While higher-order n-grams tend to produce better signatures, their training costs are much higher. In [23], Perdisci *et al.* proposed a multi-classifier system. It summarizes higher-order n-grams as pairs of non-consecutive bytes, reducing the dimensionality of fully modeling higher-order n-grams. A clustering algorithm, originally proposed for text classification, is also used to reduce the dimensionality. The experimental results demonstrate that the proposed detector is as robust against evasion as a hypothetical 7-gram PAYL.

A different approach for polymorphic worm detection is based on syntactic signatures composed of multiple invariant substrings. The rationale behind this approach is that invariant substrings such as protocol framing substrings and high-order bytes of overwritten addresses often occur in all variants of polymorphic malware. Polygraph [4] proposes three types of syntactic signatures and related automatic signature generation algorithms. Hamsa [3] shares a similar design principle and signature scheme with Polygraph, but is faster, more noise-tolerant and attack-resilient. Both Polygraph and Hamsa require innocuous and suspicious traffic pools for signature generation, and thus, are vulnerable to training attacks. Perdisci *et al.* [24] presented a noise injection attack, in

which injecting just one fake anomalous flow per real worm flow can prevent Polygraph from generating an accurate worm signature. Similarly, Newsome *et al.* [25] stated that malicious training can cause problems even when all of the training data are correctly labeled, and demonstrated that this type of attacks in general can be effective against both Polygraph and Hamsa. Gundy *et al.* [26] developed a polymorphic engine for PHP code and a polymorphic PHP-based worm that is able to evade Polygraph and Hamsa. Venkataraman *et al.* [27] presented the fundamental limits on the accuracy of a class of pattern-extraction algorithms for signature-generation in an adversarial setting.

More recent research has begun to focus on semantic analysis methods that extract higher-level meaning from executables[11, 12, 28, 29, 30]. Christodorescu *et al.* [28] proposed a semantic-aware malware detection system, which essentially exploits the uniform behavior exhibited by the variants of the same malware. The use of a template transforms the malware detection problem into a template matching problem. A program is classified as malicious if it contains a sequence of instructions exhibiting the behavior specified by a malware template.

In [11], Kruegel *et al.* proposed a polymorphic worm detection scheme by utilizing the structural information of polymorphic worms. Based on the facts that the decryption routines of polymorphic worms are usually executable and their control flow graphs (CFG) are fairly stable across worm mutations, the proposed method employs the static analysis and comparison of binary's CFG for worm detection. As another semantic analysis method, MetaAware [12] detects metamorphic worms by matching call instruction patterns. A pattern usually comprises multiple sub-patterns, each constituting library and system call instructions with corresponding parameter setting instructions.

Due to the fundamental roles of control flow and data flow analyses in static analysis, Moser *et al.* [14] designed a binary obfuscation scheme based on the concept of opaque constants. They demonstrated that advanced semantics-based malware detection methods (such as model checking [29]) can be effectively thwarted by scrambling control flow and hiding data locations and usage through obfuscation transformations. Barak *et al.* [31] discussed the theoretical limits of program obfuscation. In particular, they proved that it is impossible to hide certain properties of particular families of functions via program obfuscation.

The concept of mimicry has also been applied in other areas. In the context of system call monitors—a type of dynamic host based IDS, a mimicry attack is defined as a sequence of malicious system calls that is still considered as legitimate by the IDS program model [32]. Therefore, such attacks focus on discovering and exploiting flaws in the program model, which is specified by the form of automata. Traditionally, these attacks are manually constructed [32, 33], but recent research has shown that they can be automatically developed [34, 35, 36].

3. MIMIC FUNCTIONS

The idea of mimic functions was first introduced by Peter Wayner [37] as a steganographic technique. A mimic function transforms given input data into certain output that assumes the statistical properties of a different type of data, thereby concealing the true identity of the original data.

3.1 Regular Mimic Function

The Huffman mimic function [37], referred to as the "regular mimic function," is the functional inverse of the Huffman coding. The use of a mimic function involves three phases, digesting (i.e., Huffman tree building), encoding and decoding.

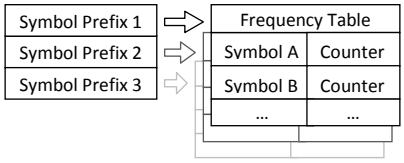


Figure 1: The Prefixed Symbol Tables

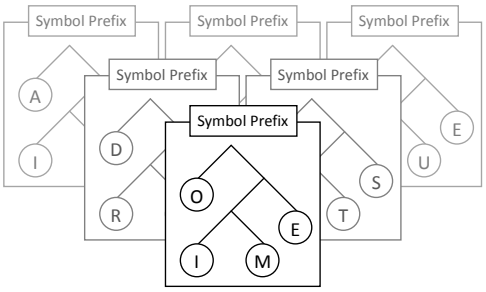


Figure 2: The Prefixed Huffman Forest

Like Huffman coding, a mimic function requires a Huffman tree to operate. In the digesting phase, a Huffman tree is constructed based on the frequency of each symbol appearing in a given piece of mimicry target data. In the encoding phase, the mimic function applies the *Huffman decoding* operation on the input data, and produces the mimicry output by referring to the Huffman tree. In the decoding phase, the mimic function applies the *Huffman encoding* operation, referring to the same Huffman tree, and uncovers the original input data from the mimicry output. In order to produce the mimicry output with a symbol frequency distribution similar to that of the mimicry target data, the input data must be random (i.e., follow uniform distribution). To meet this requirement, the input data can be randomized, such as XORing with a sequence of random numbers.

However, the regular mimic function suffers from a limitation that the symbol frequency of the mimicry output is limited to negative powers-of-2, e.g., 0.5, 0.25, 0.125, and so on. There are several techniques to overcome this limitation and we choose to use a high-order mimic function.

3.2 High-order Mimic Function

High-order mimic function differs from regular mimic function mainly in the digesting phase. Instead of building a single Huffman tree, an n_{th}-order mimic function constructs a collection of Huffman trees for a detailed "profile" of the mimicry target. Specifically, as shown in Figure 1, each observed unique symbol prefix of length $n-1$ is associated with a frequency table, which records occurrences of symbols with the given prefix. At the end of the digesting phase, each table is converted into a Huffman tree. This results in a forest of Huffman trees, each labeled by its symbol prefix, as shown in Figure 2.

Correspondingly, in the encoding and decoding phases of an n_{th}-order mimic function, a symbol prefix cache of length $n-1$ is maintained, recording the sequence of symbols that have just been encoded or decoded. For each symbol to be encoded or decoded, the high-order mimic function first locates the Huffman tree whose label corresponds to the current symbol prefix, and then performs Huffman decoding and encoding operations respectively, using the located Huffman tree.

Compared to a single Huffman tree in a regular mimic function, the Huffman forest in a high-order mimic function contains more

Table 1: Mimicked English Text

Each of these historical reason, I don't recommend using gA(t) to choose the safe. These one-to-one encoded with n leaves and punctuation. The starting every intended to find the same order mimic files. A Method is to break the trees by constructing the mimics the path down the most even though, offer no way that is, in this paper. Figure will not overflow memory. These produced by truncating letter. This need to handle n-th ordered compartment of nonsense words cannot bear any resemblance to B because this task is a Huffman showed in [1], [2], [3] among others.

Table 2: Runtime Analysis for Mimic Function

Digesting = O(n)		
Table building	Reading a symbol	1
	Prefix lookup	1
	Recording frequency	1
	Input length	n
Tree conversion	Sort	C
	Construct tree	C
	Number of tables	C
Encoding / Decoding = O(n)		
For each symbol	Locate Huffman tree	1
	Huffman de(en)coding	1
	Input length	n

detailed symbol frequency distributions as well as interdependencies among a number of adjacent symbols. As a result, the output produced by an n_{th}-order mimic function consists of n-grams that are observed in the mimicry target; and the occurrence of each n-gram is close to that of the mimicry target.

3.2.1 Power of High-order Mimic Function

Compared to the polymorphic blending attack, the state-of-the-art payload mimicry polymorphism, the high-order mimic function holds two major advantages.

Structural and semantic mimicry: While polymorphic blending attack on large n-grams is a hard problem, high-order mimic functions are designed to perform multi-gram mimicry. The output of a high-order mimic function manifests structural and even semantic similarities to the mimicry target. Table 1 lists a sample output produced by a 6_{th}-order mimic function, using Wayner's paper [37] as the mimicry target. Without the concept of "word" or "grammar," the mimic function manages to produce the paragraphs with correctly spelled words and semi-sensible sentences. In addition, it also successfully reproduces the grammatical feature that every sentence starts with a capitalized letter. While a human reader may eventually realize that the output is mere mimicry, it is very difficult to differentiate the output from "normal" English text by using statistical tests, such as byte frequency (spectrum) and entropy. Some of the sentences can even trick a grammar parser.

Run-time efficiency: As shown in Table 2, the high-order mimic function has a linear time computational complexity. Let R denote the order of a mimic function, and M denote the number of possible symbols in a given language. In the digesting phase, collecting symbol usages and constructing symbol frequency tables take linear time, using a hash table for prefix lookup. Then, converting all symbol frequency tables into Huffman trees takes sub-linear time, with a constant bound—the total number of entries in each table is bounded by M and the total number of tables is bounded by $Min(n, R^M)$[1]. Overall, the digesting phase runs in linear time. The encoding and decoding phases essentially consist of a prefix lookup followed by a Huffman decoding or encoding, which are constant time operations for each input or output symbol. Therefore, the encoding and decoding phases run in linear time as well.

3.2.2 Enhancements to High-order Mimic Function

The high-order mimic function is a powerful evasion technique against statistical anomaly detection, because it ensures the transformation of any data into legitimate symbol sequences with approximately the same frequency of the mimicry target. However, without proper enhancements, the mimic function falls short against semantic analysis detection.

Compared to human languages, binary machine languages (i.e., executables) have higher density and less structural flexibility. Without the knowledge of instruction syntax, the mimic function is unable to generate continuous long sequences of legitimate instructions. The control flows in the mimicry output are very often interrupted by malformed instructions, and thus fail to reproduce semantic properties of the mimicry target. We resolve this problem in our proposed mimimorphic engine by helping the mimic function understand the machine language. We augment the mimic function with customized assembler / disassembler. The enhanced mimic function is aware of instruction syntax, and thus is capable of generating executable instructions as well as mimicking control flows.

[1]In theory, the constant R^M can be very large. However, the upper bound is reached only when the input data is completely random. For meaningful data such as English text or executable binaries, the actual bound is much lower because the number of possible fixed-length substrings is limited.

4. MIMIMORPHIC ENGINE

The mimimorphic engine consists of four major components: assembler, disassembler, high-order mimic function, and pseudo-random number generator (PRNG). In this section, we describe the function of each component and detail the three operation phases of the mimimorphic engine: digesting, encoding and decoding. Table 3 defines a few important terms used throughout the paper.

Table 3: Mimimorphic Terms

Terms	Description
Mimicry target	The target binaries to be mimicked
Mimicry digest	A high-order instruction "profile" produced by digesting the mimicry target
Mimicry output	The output of the mimic function
Mimicry instance	A fake executable composed from the mimicry output (contains malware encoded by the mimimorphic engine)

4.1 Digesting

In the digesting phase, the mimimorphic engine takes a set of binary executables as the mimicry target, and produces a mimicry digest—a high-order machine language "profile." Two components, the disassembler and the mimic (digesting) function, are involved in this phase, as shown in Figure 3.

Preparing for the digesting function, the disassembler decodes instructions in the executable binary streams into `CommonInst` structures, as shown in Figure 4. This structure is designed to provide a generalized abstraction from platform-specific machine instructions, making the mimimorphic engine easily deployable on

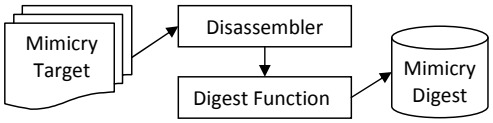


Figure 3: The Digesting Phase

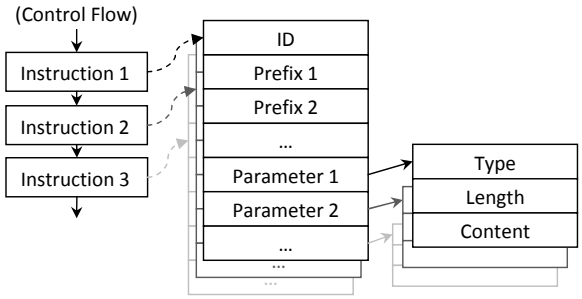


Figure 4: The `CommonInst` Structure

any instruction set architecture. The ID field contains an index to identify each unique instruction. The mimic function treats this field as a symbol in the machine language. The prefix fields, not to be confused with the "symbol prefix" of the mimic function, correspond to the fields within an instruction that alter the instruction behaviors, such as atomic memory operation and address size override. The parameter fields record instruction parameters. Each parameter further includes three different fields: type name, length, and content, indicating the type, size and content of a parameter, respectively.

After the disassembly, the digesting function processes the decoded instructions in a sequential manner. Internal to the digest function, a sequence of most recently processed instruction IDs, called *InstPfx*, is maintained, acting as the "symbol prefix" of the mimic function. For each `CommonInst`, the digest function first tries to locate an *instruction digest table (IDT)* associated with the *InstPfx*. If absent, a new table is created. Then, the digest function records the information of the `CommonInst` into the *IDT*. Finally, it appends the current instruction ID to *InstPfx* before moving onto the instruction.

The *IDT* consists of *instruction digest records (IDRs)*, indexed by the instruction ID. Each record includes a frequency counter of the instruction, as well as frequency counters of each type of prefixes and parameters, in the form of nested tables. To record the information of a `CommonInst`, we locate the *IDR* (or create a new one) with the matching instruction ID and increment its frequency counters and all the frequency counters corresponding to each of the prefixes and parameters noted in the `CommonInst`. Figure 5 illustrates the structure of an *IDT* and its *IDRs*.

At the end of the digesting phase, each *IDT* is converted into an *instruction Huffman tree (IHT)*, based on the frequency counter of each *IDR* inside the table. Correspondingly, each *IDR* is turned into an *instruction encoding template (IET)* by converting all the frequency tables associated with the prefixes and parameters into Huffman trees.

4.2 Encoding

Utilizing the mimicry digest, the encoding phase transforms an arbitrary piece of binary into a sequence of executable instructions that resembles the mimicry target. Three components of the mimi-

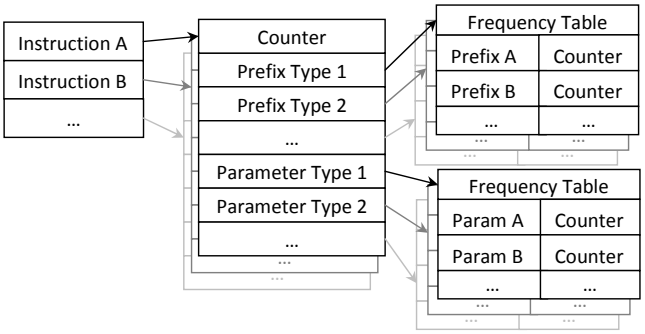


Figure 5: The Instruction Digest Table and Its Records

Algorithm 1 Mimimorphic Encoding Pseudo-code

Bin: Input binary data
$Digest$: Mimicry digest
$RSeed$: Pseudo-RNG seed

Initialize $InstPfx$;
$SBin = Randomize(Bin, RSeed)$;
while $SBin$ is not empty **do**
  $IHT = Lookup(Digest, InstPfx)$;
  $IET = TreeWalk(IHT, SBin)$;
  $Inst = InstEncode(IET, SBin)$;
  Append $Inst$ to $InstCollection$;
  Update $InstPfx$ with $Inst$;
end while
$Result = Assemble(InstCollection)$;

morphic engine—the PRNG, the mimic (encoding) function, and the assembler—are involved in this phase, as shown in Figure 6.

Algorithm 1 provides a high level overview of the mimimorphic encoding operations. Similar to the digesting function, the encoding function also maintains an *InstPfx*, recording the sequence of the most recently encoded instruction IDs. In the `Randomize` function, the input data (i.e., malicious binary) is randomized by XORing with a pseudo-random data stream generated by the PRNG. This randomization is a dual purpose operation: on one hand, it ensures that the input data satisfies the requirement of the mimic function (i.e., uniformly distributed); on the other hand, it completely erases all the characteristics of the original binary. The `TreeWalk` function searches for an *IET* from the *IHT* by "walking" down the Huffman tree from the root node, taking left or right branches according to the (randomized) input bits—this is essentially a Huffman decoding operation. Then, the `InstEncode` function constructs mimicry instructions based on the *IET*. Each *prefix* or *parameter* field in the *IET* is associated with a Huffman tree, and thus the generation of a prefix or parameter is essentially a Huffman decoding operation as well. The constructed mimicry instructions are stored in the form of `CommonInst` structures, which are later converted to binary machine instructions by the assembler.

Figure 7 shows an example of a 7_{th}-order mimimorphic engine generating an instruction in a function prologue. First, an *IHT* is looked up based on the six previously-generated instructions. Then the engine searches the tree branches according to the input bits, until a leaf node is reached. The leaf node is an *IET* of a "MOV" instruction, which contains the information of this instruction used after this particular prefix in the mimicry target. The encode function further leverages this information to generate a mimicry "MOV" instruction.

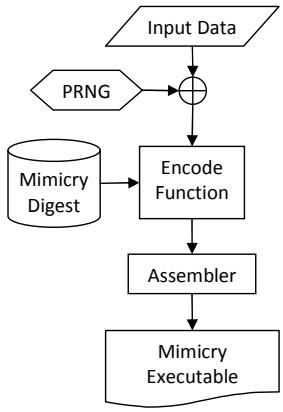


Figure 6: The Encoding Phase

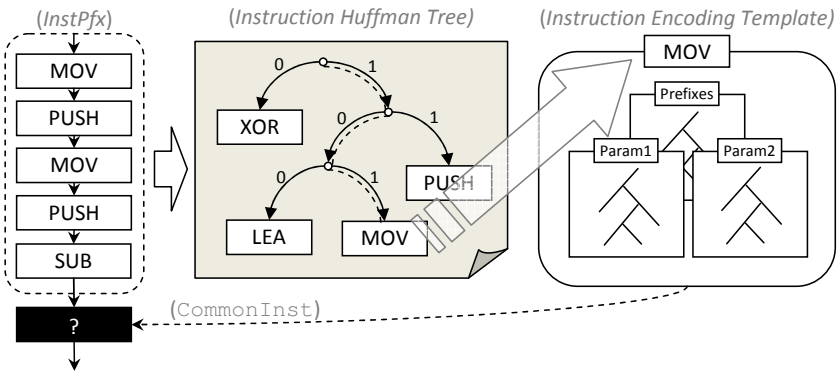


Figure 7: An Encoding Example

4.3 Decoding

The decoding phase is the inverse of the encoding phase, as shown in Figure 8. Based on the same mimicry digest, the decoding phase uncovers the input data from the mimicry output. There are three components of the mimimorphic engine involved in this phase: the disassembler, the mimic (decoding) function, and the PRNG.

Algorithm 2 Mimimorphic Decoding Pseudo-code

$Mimicry_{In}$: Mimicry instance
$Digest$: Mimicry digest
$RSeed$: Pseudo-RNG seed

Initialize $InstPfx$;
$InstCollection = Disassemble(Mimicry_{In})$;
for each $Inst$ in $InstCollection$ **do**
  $IHT = Lookup(Digest, InstPfx)$;
  $(IET, IData) = NodeLookup(IHT, Inst)$;
  $IData = InstDecode(IET, Inst)$;
  Append $IData$ to $Data_{Rand}$;
  Update $InstPfx$ with $Inst$;
end for
$Result = Derandomize(Data_{Rand}, RSeed)$;

The high level description of the mimimorphic decoding operations is given in Algorithm 2. Again, the *InstPfx* is used to record the most recently decoded instruction IDs. A mimicry instance is first disassembled into `CommonInst` structures, before being processed sequentially. The `NodeLookup` function locates the *IET* in the *IHT* with the matching instruction ID. Meanwhile, it produces a stream of data bits that corresponds to the branches taken from the root of the Huffman tree to the leaf node—this is essentially a Huffman encoding operation. The `InstDecode` function further retrieves the data bits encoded in each mimicry instruction by performing Huffman encoding operations for all the prefixes and parameters with their corresponding Huffman trees in the *IET*. Finally, the `Derandomize` function uncovers the original data by XORing the decoded data with a pseudo-random data stream, which are generated by the PRNG with the same seed used in the encoding phase.

Figure 9 shows an example of a 7_{th}-order mimimorphic engine decoding the instruction produced in the previous encoding example. First, an *IHT* is located based on the six previously-generated instructions. Then the engine looks up the leaf node *IET* that corresponds to the current instruction to be decoded, in this example, the "MOV" instruction. The path from the *IHT* root to this leaf node is then converted to data bits. The similar operations are performed for each of the prefixes and parameters of the "MOV" instruction, using the corresponding Huffman trees in the *IET* and producing a stream of data bits.

4.4 Design Issues

We now discuss a few important design issues in the digesting and encoding phases, which affect the quality of mimicry. These issues include (1) how to handle embedded data in digest binaries, (2) how to select a good random source, and (3) how to ensure valid control flow generation.

In the digesting phase, the mimicry target binaries are first disassembled into `CommonInst` structures before digesting. However, in most legitimate executable binaries, there are a small but non-negligible amount of embedded data, which mainly consist of constants and jump address tables. Simply ignoring these embedded data might cause the statistical properties of the mimicry output to deviate from those of the mimicry target, resulting in the degradation of mimicry quality. We resolve this problem by masquerading embedded data as special one-byte-no-parameter "instructions" and digesting them along with other real instructions.

Recall that, in the encoding phase, a regular mimic function requires input data to be uniformly distributed, so as to produce the mimicry output with the statistical properties approximating those of the mimicry target. Correspondingly, a high-order mimic function also requires the input data to be randomized on high-order. In our mimimorphic engine design, we select MT19937 PRNG [38], which claims to have equidistribution in 623 dimensions. Other PRNGs that can pass high dimensional distribution tests could be used as well.

Although the mimimorphic engine ensures valid instruction generation, it does not guarantee to produce valid control flows. This is because branch/call instructions use byte offset to redirect control flows. In addition that the lengths of instructions are not fixed, there is no prior knowledge of subsequent instruction generations when the mimimorphic engine produces a branch/call instruction. Thus, a byte offset could point to the middle of a following instruction, invalidating the control flow. We resolve this problem by performing control flow correction on the intermediate data after the encoding phase. Instead of outputting the binary as soon as each instruction is generated, we keep all the `CommonInst` structures in a linked list. Then, for each branch and call instruction, we inspect whether its referring offset aligns to an instruction, and make corrections if

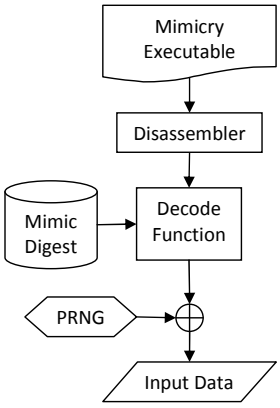


Figure 8: The Decoding Phase

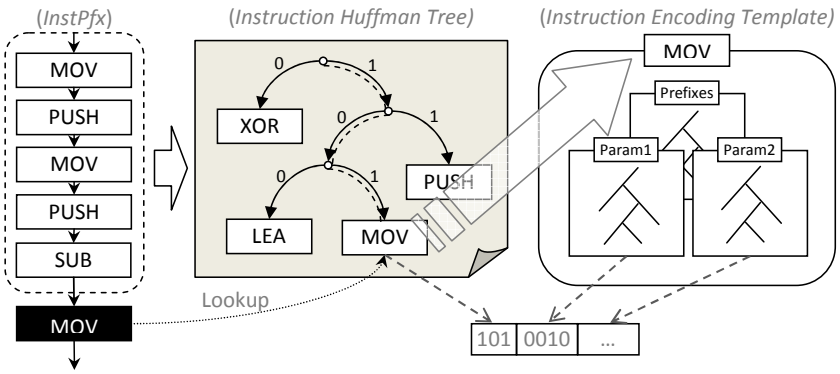


Figure 9: An Decoding Example

necessary. We have verified the effectiveness of the solution by performing control flow analysis and basic block identification [11] on the mimicry output with and without control flow correction. We have observed that the number of valid basic blocks increases by nearly seven times with control flow correction.

5. IMPLEMENTATION

We have implemented a prototype of the mimimorphic engine on the Intel x86 architecture. While the current implementation works on the Windows XP, its core component is OS-independent and can be easily ported to Unix variants. In the following, we briefly describe some non-trivial implementation details.

First, a mimicry target is required for the mimimorphic engine to perform transformations. We randomly select 100 executable files from the `system32` folder of the Windows XP, and extract their "text" sections to form a representative set of "normal" executables. System executables and libraries make good candidates of the mimicry target, because they exist on the majority of victim hosts, and they are also commonly delivered over the Internet (i.e., in forms of security patches).

Second, based on our observation of the basic block size of the mimicry target, we set the order of mimic functions to 7-8. Considering the unique feature of mimimorphism, we attempt to generate mimicked control flows that can be used to evade advanced semantic analysis detection. Because control flows are formed by basic blocks, the success of mimicking basic blocks is essential to the generation of mimicry control flows. We profile the basic block size of our selected mimicry target executable files, and observe that 89% of the executable files have the average basic block size less than or equal to eight instructions.

Third, we use a hash table to provide fast prefix lookup of *IHT*. Although the number of possible "symbol prefix" grows exponentially as the order of the mimic function increases, the number of observed unique prefixes is bounded by the size of input. With a relatively large hash table (22 bits), we are able to achieve reasonably low collision rate. In our experiments, the utilization rates of the hash table are below 20% and 25% for 7_{th}-order and 8_{th}-order mimic functions, respectively. For both mimic functions, 85% of entries are collision free, and over 99% of entries have less than or equal to one collision.

6. EVALUATION

We use 7_{th}-order and 8_{th}-order mimic functions in the mimimorphic engine (M_7 and M_8 for short). An 83KB executable file is used as a hypothetical malware program, on which we apply mimimorphism. For each M_7 and M_8, we generate 100 instances of the mimicry output, and each instance uses a different seed value for input data randomization. We evaluate the effectiveness of mimicry from two detection aspects: statistical test and semantic analysis test. Note that whether the executable file is a "real" malware or not is irrelevant to our evaluation. This is because (1) as stated in Section 4.2, the input randomization in the encoding phase has completely erased all the characteristics of the input data, thereby any input data would yield equivalent output; and (2) the detections we apply in our experiments are generic anomaly and similarity tests, instead of specific malware detections (such as commercial malware/virus scanners).

6.1 Statistical Tests

We run our mimimorphic output, M_7 and M_8 files, against statistical tests, namely the Kolmogorov-Smirnov and byte entropy tests. The Kolmogorov-Smirnov test is a general purpose statistical test, whereas the byte entropy test is proposed for detecting packed or encrypted malware [9]. Although the Kolmogorov-Smirnov test is more powerful, it can only determine if a sample is anomalous, whereas the byte entropy can determine if a sample is, with high probability, a compressed or encrypted file.

The Kolmogorov-Smirnov test determines whether or not two samples (or a sample and a distribution) differ by measuring the maximum distance between two empirical distribution functions:

$$KSTEST = \max \mid S_1(x) - S_2(x) \mid$$

where S_1 and S_2 are the empirical distribution functions of the two samples. This test is distribution free—in other words, the test statistic is not dependent on a specific distribution, and thus, is very general in applicability. The Kolmogorov-Smirnov test is frequently used in steganalysis—the analysis of steganographic techniques, so its usage in evaluating the mimimorphic attack, a steganographic method, is pertinent. For our experiments, we perform Kolmogorov-Smirnov test between samples, mimicry or legitimate files, and a database of legitimate files. If the test statistic is low, the sample is classified as normal, otherwise the sample is classified as suspicious.

The mean and standard deviation of the Kolmogorov-Smirnov test scores are listed in Table 4. For the legitimate files, the mean score is 0.074 and the standard deviation is 0.045. For M_7 and M_8 files, the mean scores are 0.109 and 0.093, respectively. The standard deviation of the mimicry files is very low compared to that of the legitimate files. This is mainly due to the size of the

Table 4: Kolmogorov-Smirnov Results

	Mean	Std. Dev.
Legitimate	0.074	0.045
M_7	0.109	0.007
M_8	0.093	0.006

mimicry files. M_7 and M_8 files are approximately 2.4MB and 3.3MB, whereas the legitimate files range from 1KB to 0.5MB. As smaller files are statistically more likely to vary from the expected value, the variance of the mimicry files, whose sizes are larger on average, is very small. Although the test scores of the mimicry files are higher on average, the majority of these test scores fall within one standard deviation of the legitimate mean, 0.074 ± 0.045 (or 0.019 to 0.119). Therefore, the Kolmogorov-Smirnov test is unable to reliably differentiate mimicry files from legitimate files.

The byte entropy test is based on the randomness of compressed or encrypted files. The byte entropy test [9] measures the randomness of the distribution of bytes:

$$entropy(X) = -\sum_{X} P(x) log P(x)$$

where X is a byte sample and $P(x)$ is the probability $P(X = x)$. For our experiments, we measure the byte entropy of different test samples, either mimicry or legitimate files. If the entropy is high, then the sample is suspected as compressed or encrypted malware, which may be further examined by unpacking via emulation or other dynamic analysis. However, if the entropy is low, i.e., in the range of typical executables, then the sample is classified as uncompressed and unencrypted.

Table 5: Byte Entropy Results

	Mean	Std. Dev.
Legitimate	6.353	0.258
M_7	6.545	0.021
M_8	6.528	0.021

The mean and standard deviation of the byte entropy test scores are listed in Table 5. For the legitimate files, the mean score is 6.353 and the standard deviation is 0.258. For M_7 and M_8, the mean scores are 6.545 and 6.528, respectively. Like the Kolmogorov-Smirnov results, the standard deviation of the mimicry files is very low, again due to their file sizes. In comparison to those of legitimate files, the test scores of M_7 and M_8 are slightly higher on average, but fall well within one standard deviation of the legitimate mean, 6.35 ± 0.258 or 6.095 to 6.881. Based on these results, the byte entropy test is unable to differentiate mimicry files from legitimate files. Moreover, packed and encrypted executables have byte entropies over 7 [9], so M_7 and M_8 are successful in disguising their packed content as normal executables.

6.2 Semantic Analysis Test

We use M_7 and M_8 to evaluate mimimorphic attacks against semantic analysis detection, particularly, the detection based on control flow fingerprinting [11]. This detection technique analyzes the control flows of binaries, and generates "fingerprints" for those control flows. To detect polymorphic malware, the system compares the fingerprints for suspicious network traffic against the fingerprints of known malware instances. If a sufficient number of fingerprints match, the detection system asserts with high confidence that the traffic contains malware.

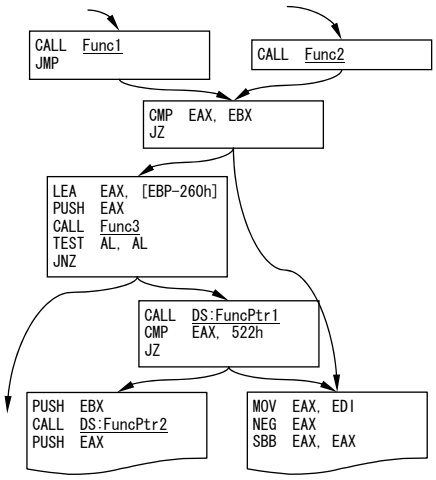


Figure 10: A Sample of M_7 Control Flow Graph

Mimimorphism attacks the control flow fingerprinting detection by introducing a large number of mimicked control flows to resemble those of legitimate binaries. The detection system generates a number of fingerprints from a database of malware, i.e., M_7 and M_8 instances. The fingerprints generated by the detection system can be described as "good" or "badi." A good fingerprint matches only malware files, but a bad fingerprint matches both malware and legitimate files. Figure 10 presents an example of "mimicry control flow graph" in an M_7 instance. Except for the underlined function addresses, the instruction sequence matches that of a system library file. As a result, the fingerprint generated from this segment of code is "bad." When the majority of fingerprints generated by the detection system are bad, it would suffer high false positives.

As a basic test, we first measure fingerprints that are common in the original hypothetical malware program and the M_7/M_8 instances. We observe that only one file from each set of instances, M_7 and M_8, has one or more common fingerprints with the hypothetical malware. The M_7 file shares three common fingerprints, while the M_8 file shares only one. Thus, overall M_7 and M_8 are successful in erasing the signatures from the original malware. We then proceed to measure the number of bad fingerprints produced from M_7 and M_8 instances, and the number of good and bad fingerprints shared by all M_7 and M_8 instances.

Table 6: Bad Fingerprints for M_7 and M_8 Instances

	M_7			
	Mean	Std. Dev.	Max.	Min.
Bad fprts.	1856.46	372.5	3321	1057
Matched files	72.93	14.53	92	44
	M_8			
	Mean	Std. Dev.	Max.	Min.
Bad fprts.	11407.99	912.42	14216	9606
Matched files	81.37	4.06	91	70

Table 6 presents the results of fingerprint comparisons in terms of mean, standard deviation, maximum and minimum counts between the legitimate files and M_7/M_8 files, respectively. The "bad fingerprints" row shows the number of bad fingerprints. The "matched files" row shows the number of legitimate files that share one or more fingerprints with a mimicry file. For all rows, larger numbers indicate that mimicry attacks are more successful.

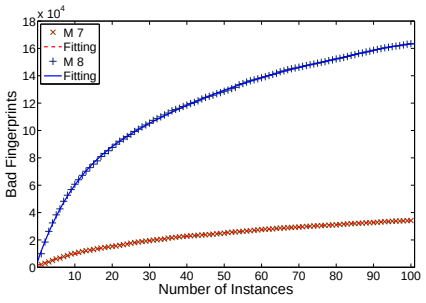
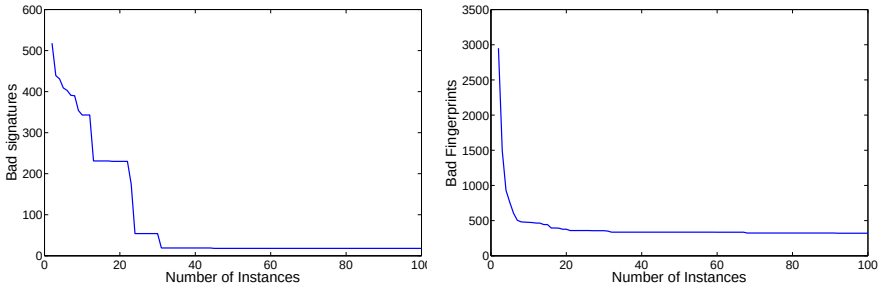


Figure 11: Bad Fingerprints in Collections of M_7 and M_8 Instances



(A) (B)

Figure 12: Shared Bad Fingerprints in Collections of M_7 and M_8 Instances

On average an M_7 mimicry file contains 1856.46 bad fingerprints, and shares one or more bad fingerprints with 72.93% of the legitimate files. The most successful M_7 mimicry instance shares one or more fingerprints with 92 legitimate files, while the least successful instance reproduces fingerprints in only 44 legitimate files. On average, an M_8 mimicry file contains 11407.99 bad fingerprints, and shares one or more fingerprints with 81.37% of the legitimate files. The most successful M_8 mimicry instance shares fingerprints with 91 legitimate files, while the least successful instance reproduces fingerprints in only 70 legitimate files.

Figure 11 illustrates the total number of bad fingerprints contained in a collection of N ($1 \leq N \leq 100$) M_7 and M_8 files. It highlights the mimimorphic engine's capability to mimic fingerprints from the legitimate files. The dashed and solid lines are curve fittings of the M_7 and M_8 data points, respectively. We can see that for both M_7 and M_8 files, as the number of instances increases, the total number of bad fingerprints increases, following a polynomial distribution.

Results in Table 6 and Figure 11 indicate that both M_7 and M_8 are successful in mimicking control flows of the mimicry target binaries. An M_7 or M_8 mimimorphic malware instance contains thousands to tens of thousands of bad fingerprints. As a result, the high false positive rates make it impractical to use the control flow fingerprints of a mimimorphic malware instance for detecting the other instances.

With greater efforts, a number of mimimorphic malware instances can be collected and analyzed, and their shared fingerprints can be extracted. However, our results show that such an approach can only achieve limited improvements on detecting mimimorphic malware. Figures 12 (A) and (B) present the results of the fingerprint comparisons between the legitimate files and a collection of N instances ($2 \leq N \leq 100$) of M_7 and M_8 files, respectively. The line signifies the number of bad fingerprints. While the bad fingerprint counts for $N = 2$ decrease dramatically compared to the results in Table 6, the decrease slows down and the bad fingerprints stabilize at a non-zero value. More specifically, when $N = 100$, for M_7 files, there are 18 bad fingerprints; for M_8 files, there are 321 bad fingerprints.

While the above two figures indicate very positive results for both M_7 and M_8, the number of shared fingerprints among 100 mimicry files presented in Table 7 gives us some surprises. When $N = 100$, there are 161 fingerprints shared by all M_7 files, but only 18 match legitimate files. This implies that the M_7 mimimorphic engine generates 143 fingerprints that can be used to identify the mimimorphic instances! However, the results are much better for M_8 files. Whereas there are 339 shared fingerprints in all M_8 files, 321 of them match legitimate files, leaving only 18 additional fingerprints. The polymorphic instances of a malware normally have tens to hundreds of shared fingerprints [11]. Thus, even with 100 instances of M_8 mimimorphic malware, there are still comparable amount of bad signatures mixed with the good signatures of the malware. Therefore, even given a large number of identified instances, M_8 mimimorphic malware can still maintain enough bad fingerprints to render the control flow fingerprints unusable.

Table 7: Shared Fingerprints of All M_7 and M_8 Instances and Good/Bad Fingerprints in Them

	Shared fprnts.	Bad fprnts.	Good fprnts.
M_7	161	18	143
M_8	339	321	18

6.3 Discussion

Artifact generation: Table 7 shows an interesting phenomenon that the mimimorphic engine produces shared fingerprints in all instances that do not belong to any legitimate file. We call this phenomenon persistent artifact generation. It is caused by digesting data with limited order mimic functions. When the mimic function digests two sequences of symbols that share a common subsequence with interdependencies longer than the order of the mimic function, the interdependencies will be partially merged. Thus, it is possible for the mimicry output to contain long symbol sequences constructed by mixing and matching the original parts. As a more intuitive example, when examining the mimicry output of the 6_{th}-order English mimic function, we sometimes encounter erratic words, such as "operationale" and "instructural", which are the combinations of the words "operational" and "rationale", and "instruction" and "structural", respectively. Because the combinations are limited by the number of such long symbol sequences, the mimic function tends to reproduce the same erratic words persistently. This problematic phenomenon can be greatly reduced by increasing the order of the mimic function, as demonstrated by the M_8 files, because of the increased symbol prefix length. Back to the previous example, a 7_{th}-order English mimic function would not produce the word "instructural" because "instruction" and "structural" do not have any common 7-grams.

Robustness against other detection approaches: Mimimorphism is also robust against other static analysis detection methods, such as automatic n-gram signature generation, and certain types of semantic analysis techniques. An N_{th}-order mimimorphic engine digests the mimicry target binaries in units of N adjacent instructions, and thus its mimicry output consists of series of N-instruction-grams, i.e., N consecutive instructions, observed from the mimicry target. As a result, for $n \leq N \times b$, where b is the

instruction length in bytes, the mimicry output should not contain any n-byte-grams that do not match the mimicry target. Based on our observation, the average length of an Intel x86 instruction is between 2.1-2.8. Thus, in theory, a 16-gram byte test would be needed to reliably generate signatures for an M_8 mimimorphic malware. Semantic analysis techniques, which make decision based on short-range semantic similarities [12], are also vulnerable to mimimorphic attacks, due to the large number of randomly generated control flows that are similar to legitimate binaries.

In our current implementation, the mimimorphic engine has limited ability to mimic program-level high-order syntactic and semantic characteristics, such as function boundaries, prologue and epilogue. Without the high-level concept of "functions," the mimimorphic engine can only capture and generate the related syntactic and semantic properties in a probabilistic and best-effort manner. We manually inspected an M_8 file, and found that about 45% of "functions" miss function prologue or epilogue sequence, and some relative jumps go across function boundaries. While it is possible to take advantage of these common program-level properties to identify our mimimorphic instances, we do not consider it a viable detection approach. Because those properties exist only by convention, and there are many programs that deviate from the norm, especially on copyright protected executables that employ non-conventional protection techniques [39]. Detection based on non-conforming of conventions would suffer high false positive rates.

Constraints: There are two constraints for applying mimimorphism: memory consumption and payload size increase. Currently, the M_7 and M_8 mimimorphic transformations on average consume 600MB and 1.2GB memory, and increase the payload size by 20 and 30 times, respectively. However, both constraints can be effectively mitigated. To reduce memory consumption, we could implement an on-demand, disk based Huffman forest structure, which only loads Huffman trees into memory as needed[2]. With mimimorphic transformations on executable of reasonable size, only a small portion of Huffman trees will be traversed (bounded by input size) and thus the memory consumption is significantly reduced. To limit the payload size increase, we can apply compression to the input data before randomization. Because the size growth only occurs at the encoding phase, which takes already highly randomized data, compressing data before randomization does not affect the inflation ratio of the mimic function. Assuming compressing plain executables decreases their size by 30% [40, 41], applying compression before mimimorphic transformation thus can also lower the payload size increase by 30%.

Decoder: Like polymorphic malware, mimimorphic malware requires to ship its decoder with the payload. The decoder needs to be directly executable and thus cannot be transformed into non-executable data. As a result, the decoder is the common weakness of polymorphic malware, because its packed payloads have abnormal statistical properties and are *not executable*, making the decoder binaries easily extracted and analyzed. In contrast, mimimorphism by design provides much improved protection for the decoder. Because the mimimorphic payloads have similar statistical and semantic properties of *executable binaries*, correctly identify the decoder binary for signature extraction is no longer an easy task. We plan to explore techniques that "blend" the decoder control flow into the payload, side-by-side with hundreds of thousands of fake-but-legit-like mimicry control flows, effectively thwarting attempts to extract the decoder statically.

[2]The "dehydrated" Huffman trees stored on disk use much less space than in RAM. The file sizes of digest data for M_7 and M_8 mimimorphism are only about 150MB and 300MB, respectively.

7. CONCLUSION

In this paper we have introduced a novel binary obfuscation technique, called mimimorphism. Mimimorphism can transform a binary executable into a mimicry executable, with statistical and semantic characteristics highly similar to those of the mimicry target. Exploiting mimimorphism, malware can successfully evade certain statistical anomaly detections, automatic substring signature generation, as well as some state-of-the-art semantic analysis techniques.

We have implemented a prototype of the mimimorphic engine on the Intel x86 platform. Our experimental results validate its efficacy in evading statistical anomaly detection—the byte frequency distribution test and entropy test—and a semantic analysis technique—the control flow fingerprinting. In the byte frequency distribution test and entropy test, the mimicry output produced by the mimimorphic engine falls within the ranges of normal executable files. For control flow fingerprinting, the mimicry output produced by the mimimorphic engine induces a large number of fingerprints that match legitimate binaries, leading to high detection false positives.

In our future work, we would like to refine our mimimorphic engine and evaluate its performance on real scenarios. We are also interested in the study of effective counter-measures against mimimorphism. For instance, we will explore the possibility of detecting mimimorphism through higher-order mimic functions and higher level semantics.

Acknowledgments

The authors would like to thank the anonymous reviewers for their detailed and valuable comments. This work was partially supported by NSF grant 0901537 and ONR grant N00014-09-1-0746.

8. REFERENCES

[1] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proceedings of 13th USENIX Security Symposium*, 2004.

[2] C. Kreibich and J. Crowcroft, "Honeycomb: creating intrusion detection signatures using honeypots," in *Proceedings of 2nd Workshop on Hot Topics in Networks (Hotnets-II)*, 2003.

[3] Z. Li, M. Sanghi, B. Chavez, Y. Chen, and M.-Y. Kao, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, 2006.

[4] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P)*, 2005.

[5] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[6] P. Szor, *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[7] K. Wang, J. J. Parekh, and S. J. Stolfo, "Anagram: A content anomaly detector resistant to mimicry attack," in *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2006.

[8] K. Wang and S. Stolfo, "Anomalous payload-based network intrusion detection," in *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

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

[10] C. Kruegel, W. K. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[12] Q. Zhang and D. S. Reeves, "MetaAware: Identifying metamorphic malware," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 411–420.

[13] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[14] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 421–430.

[15] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, "Polymorphic shellcode engine using spectrum analysis," *Phrack Issue 0x3d*, 2003.

[16] S. Macaulay, "Admmutate: Polymorphic shellcode engine," http://www.ktwo.ca/security.html.

[17] M. Khafir, "Trident polymorphic engine," http://vx.netlux.org/lib/vx.php?id=et06.

[18] F. Perriot, P. Ferrie, and P. Szor, "Striking similarities: Win32/simile," http://securityresponse.symantec.com/avcenter/reference/simile.pdf.

[19] Z0mbie, "Automated reverse engineering: Mistfall engine," http://vx.netlux.org/lib/vzo21.html.

[20] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong, "On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 235–248.

[21] Y. Tang and S. Chen, "Defending against internet worms: a signature-based approach," in *Proceedings of the 24th INFOCOM*, 2005.

[22] P. Fogla and W. Lee, "Evading network anomaly detection systems: formal reasoning and practical techniques," in *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, 2006, pp. 59–68.

[23] R. Perdisci, G. Gu, and W. Lee, "Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems," in *Proceedings of the Sixth International Conference on Data Mining*, 2006, pp. 488–498.

[24] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, 2006.

[25] J. Newsome, B. Karp, and D. X. Song, "Paragraph: Thwarting signature learning by training maliciously." in *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2006.

[26] M. V. Gundy, D. Balzarotti, and G. Vigna, "Catch me, if you can: Evading network signatures with web-based polymorphic worms," in *Proceedings of 1st USENIX Workshop on Offensive Technologies*, 2007.

[27] S. Venkataraman, A. Blum, and D. Song, "Limits of learning-based signature generation with adversaries," in *Proceedings of the 15th Annual Network and Distributed Systems Security Symposium (NDSS)*, 2008.

[28] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P)*, 2005.

[29] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Proceedings of the 2nd International Conference Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2005, pp. 174–187.

[30] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[31] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)*, 2001.

[32] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, 2002, pp. 255–264.

[33] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion, "Undermining an anomaly-based intrusion detection system using common exploits," in *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[34] J. Giffin, S. Jha, and B. Miller, "Automated discovery of mimicry attacks," in *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[35] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[36] C. Parampalli, R. Sekar, and R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *Proceedings of the 2007 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

[37] P. Wayner, "Mimic functions," *Cryptologia*, vol. 16, no. 3, pp. 193–214, 1992.

[38] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[39] T. E. Dube, B. D. Birrer, R. A. Raines, R. O. Baldwin, B. E. Mullins, R. W. Bennington, and C. E. Reuter, "Hindering reverse engineering: Thinking outside the box," *IEEE Security and Privac*, vol. 6, no. 2, pp. 58–65, 2008.

[40] S. Debray, "Code compression," in *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, 2005.

[41] S. Debray and W. Evans, "Profile-guided code compression," in *Proc. SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI)*, 2002.