PackGenome: Automatically Generating Robust YARA Rules for Accurate Malware Packer Detection

Anonymous Author(s)

ABSTRACT

Binary packing, a widely-used program obfuscation style, compresses or encrypts the original program and then recovers it at runtime. Packed malware samples are pervasive—they conceal arresting code features as unintelligible data to evade detection. To rapidly respond to large-scale packed malware, security analysts search specific binary patterns to identify corresponding packers. The quality of such packer patterns or signatures is vital to malware dissection. However, existing packer signature rules severely rely on human analysts' experience. In addition to expensive manual efforts, these human-written rules (e.g., YARA) also suffer from high false positives: as they are designed to search the pattern of bytes rather than instructions, they are very likely to mismatch with unexpected instructions.

In this paper, we look into the weakness of existing packer detection signatures and propose a novel automatic YARA rule generation technique, called `PackGenome`. Inspired by the biological concept of species-specific genes, we observe that packer-specific genes can help determine whether a program is packed. Our framework generates new YARA rules from packer-specific genes, which are extracted from the unpacking routines reused in the same-packer protected programs. To reduce false positives, we propose a byte selection strategy to systematically evaluate the mismatch possibility of bytes. We compare PackGenome with public-available packer signature collections and a state-of-the-art automatic rule generation tool. Our large-scale experiments with more than 640K samples demonstrate that PackGenome can deliver robust YARA rules to detect Windows and Linux packers, including emerging low-entropy packers. PackGenome outperforms existing work in all cases with zero false negatives, low false positives, and a negligible scanning overhead increase.

1 INTRODUCTION

Binary packing is recognized as one of the most popular software protection techniques [1]. It was originally designed to reduce the size of executable programs. Binary packers compress (or encrypt) the code and other necessary assets of the input program to packed data. It integrates an unpacking routine and packed data into the packed version. The unpacking routine takes charge of recovering and executing the original code at runtime. As a result, the original program's behaviors are hidden from static analysis. When combined with other code obfuscation and anti-analysis methods, packed programs can effectively impede reverse engineering attempts [2, 3]. Therefore, binary packing is not only favored by software developers but also has long been abused by malware authors. Recent studies [4–6] show that nearly 50% of packed programs (collected in the wild within the last five years) are benign, and 75% of malware samples are packed.

Intuitively, security analysts can dynamically monitor a packed program's execution to accurately discover the concealed behaviors. Unfortunately, the various evasion techniques adopted by advanced packers are significantly hindering dynamic analysis [7]. Security analysts have to adopt highly customized dynamic analysis (e.g., stealthy instrumentation [8] or hardware-assisted tracing [9]) as countermeasures. When processing large-scale programs, the cost of customized environments will become unacceptable [6]. For example, VirusTotal, the top malware online scan service, processes more than 868K new files daily [10]. Therefore, security analysts typically rely on efficient static analysis to prioritize packed programs that are worthy of expensive dynamic analysis [11].

Static packer detection has evolved into several variations. One heuristic method is to measure entropy: the compressed or encrypted data typically reveal higher entropy than the compiled code. Many research papers [4, 12–15] and industrial tools [16] regard the high entropy of sections as a sign of packed programs. However, Mantovani et al. [17] find that more than 30% of their 50K Windows malware datasets are low-entropy packed samples. These packed samples adopt multiple data encoding tricks to evade entropy-based detection. Another direction identifies packed programs based on the features extracted from executable binaries, such as PE header metadata, disassembly instructions, or the labels provided by VirusTotal [18–22]. The security community mainly uses these arresting static features to create signature rules or train machine learning models. However, Aghakhani et al. [6] point out that the machine-learning classifiers are not robust enough to detect packed malware variants in the wild.

As the most popular technique adopted by the security community, signature-based packer detection matches packers with predefined textual or binary patterns. The representative tool, YARA [23], has become the industry de facto standard to express malware characteristics. In academia, we surveyed the papers published in 12 major cyber security conference venues (e.g., IEEE S&P, USENIX Security, ACM CCS, and NDSS) over the past 16 years (detailed in Appendix A). There are 26 papers (12 of them are published in the top four venues) that rely on signature-based tools to identify packed programs in their experiments. According to whether directly using the signature-based tools in their experiments, these papers can be divided into two categories: (i) 24 papers directly use signature-based tools. For example, Ugarte-Pedrero et al.'s SoK paper [4] uses PEiD [24], Sigbuster, and F-Prot to classify off-the-shelf and custom packers. (ii) Two papers indirectly use signature-based tools. Downing et al. [25] and Park et al. [26] use unipacker [27] to unpack the samples of their datasets, while unipacker relies on YARA rules to detect packed programs.

The quality of the rules that describe packer features is the key to the effectiveness of signature-based detection. As research papers widely use signature-based detection tools to prepare ground-truth datasets [4], problematic rules might unintentionally pollute the dataset and lead to biased results. Unfortunately, the existing packer signature rules are mostly written and maintained by human analysts. After examining 10,249 publicly-available packer detection rules (detailed in Sec. 7), we find that the existing human-written rules are confronted with the following three problems.

P1: The cost of manually writing and maintaining rules is becoming unaffordable. To develop signature rules, security analysts have to put great effort into analyzing packed programs and summarizing common features. A recent study [28] shows that experienced analysts spend several hours to weeks on reverse engineering programs. When handling complex packers such as Themida [29], even skilled analysts need up to six months on understanding programs and developing unpackers [30]. Meanwhile, the number of packers grows faster than the rule development process. In addition to over 150 different off-the-shelf packers with multiple versions [31, 32], there are also a great number of custom packers, which are preferred by malware authors [4, 30]. Considering the evolution of packers, security analysts also have to periodically track packers' new updates. Furthermore, 99.88% of packer detection rules are created from x86 instructions. To support x64 packed program detection, security analysts have to repeat the tedious rule development process.

P2: The development of packer rules severely relies on human analysts' experience. Guided by reverse engineering experience, malware researchers manually extract the common patterns of packed programs as rules. We observe that nearly 85% of rules only describe the Portable Executable (PE) entry point's instructions or section names. However, these features make the rules vulnerable to adversary packers. For example, APT41 camouflages VMProtect-packed programs by changing the section name from "`.vmp`" to "`.UPX`" [33]. Furthermore, the choice of the rule lengths and special constructions (e.g., wildcards) may bring more uncertainty to the rule matching. For example, a YARA rule with eight wildcards causes VirusTotal to mistakenly recognize the 7z.exe file as the Armadillo-packed program [34] (detailed in Appendix B).

P3: Packer rules reveal high false positives caused by mismatching with unexpected instructions. Human analysts develop rules according to the bytes of expected instructions. However, signature-based detectors are based on the pattern matching, which operates on byte strings, regardless of instruction formats. Note that the byte length of an x86/x64 instruction varies from 1 to 15 [35]. As a result, human-written rules are very likely to mistakenly match parts of an irrelevant instruction, leading to high false positives (detailed in Sec. 2.3).

In this paper, we aim to mitigate the above problems by proposing *PackGenome*, an automatic YARA rule generation technique to advance packer detection. PackGenome is inspired by a biological fact that *species-specific genes* make humans different from chimpanzees [36]. PackGenome creates rules from *packer-specific genes*, which make the packed programs distinguished from the non-packed programs. We extract *packer-specific genes* from the unpacking routine instructions, because the unpacking routine is reused in the same-packer protected programs and does not exist in non-packed programs.

In particular, we first collect the unpacking routine instructions from packed binaries using a hybrid static-dynamic analysis. Since signature-based tools only scan programs statically, we dynamically extract the high-frequency instructions that are also visible to static analysis. Then, we identify packer-specific genes by calculating statistical similarity [37] of unpacking routines reused in the same-packer protected programs. At last, we propose a byte selection strategy to generate YARA rules. Our approach evaluates the mismatch probability of the generated rules when matching with unexpected instructions. This mismatch probability guides us to select appropriate bytes as rules.

We have conducted a set of experiments to evaluate the efficacy of PackGenome. We first apply PackGenome to automatically generate new YARA rules for popular off-the-shelf and custom packers. Our evaluation of over 640K samples shows that our generated rules outperform existing work, including public-available YARA rules, the YARA rules generated by the state-of-the-art automatic rule generation tool AutoYara [21], and sophisticated JavaScript-like rules from Detect it Easy [16]. Compared with these representative tools, our approach exhibits zero false negatives, much lower false positives, and a negligible scanning overhead increase. We also evaluate the scalability of PackGenome in real-world scenarios. The results show that PackGenome-generated rules are robust to recognize x86/x64 Windows and Linux packed programs, even the custom packers such as low-entropy versions modified from standard packers.

Contributions Our key contribution is to free security professionals from the burden of manually piecing together the tedious steps of packer signature generation. In fact, malware researchers utilizing PackGenome will enjoy a simpler and more streamlined YARA rules development process than ever before. In summary, this paper makes the following technical contributions:

- Our key observation is that packer-specific genes, extracted from unpacking routines, are ideal candidates as packer significant features. We develop a hybrid static-dynamic extraction method to obtain these genes from the same-packer protected programs.
- We propose an automatic YARA rules generation technique for packer detection. The generated rules are robust to detect off-the-shelf packers, even the custom versions.
- We design a novel byte selection strategy, which evaluates the mismatch probability of the given byte rules. It can guide both automatic rule generation tools and human analysts to reduce false positives significantly.

Open Source We release PackGenome's source code and generated YARA rules to facilitate reproduction and reuse, as all found at (*URL omitted for double-blind reviewing*).

2 BACKGROUND AND MOTIVATIONS

In this section, we provide the background information needed to understand our work's motivation. We first introduce binary packing and signature-based packer detection techniques. Then, we discuss the limitations of existing human-written rules and the challenges of developing robust packer detection rules, which motivate us to propose PackGenome.

2.1 Binary Packing

As shown in Fig. 1, the original program is statically rewritten by a packer and then gets self-unpacked at runtime. The packer treats instructions and other resources (e.g., "`.data`" section) of the input

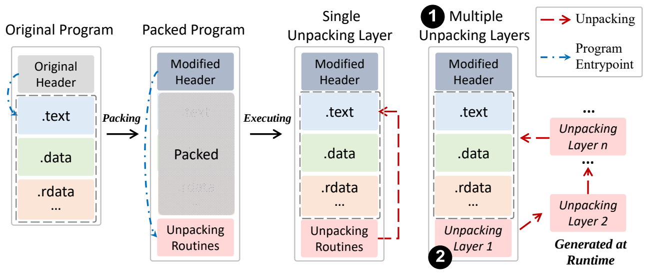


Figure 1: An illustration of the unpacking process.

```
rule UPX {
strings:
    $a = "UPX"
    $b = { 60 E8 00 00 00 00 58 83 E8 3D }
    $c = { EB ?? ?? ?? ?? ?? 8A 06 46 88 07 47 01 DB 75 07
         8B 1E 83 EE FC 11 DB }
    $d = { 60 E8 [4] 58 83 E8 3D 50 8D B8 [4] (57|87) 8D
        B0}
condition:
    $a and ($b at pe.entry_point or $d at pe.entry_point
        ) and $c
}
```

Figure 2: A YARA rule to detect UPX-packed programs.

program as data. It compresses or encrypts these data and rewrites the input program. Meanwhile, the packers can modify (or remove) any parts of the original program that are not required for normal execution. For example, the UPX-packed programs use the section name "`.UPX`" instead of "`.text`". The generated packed program typically contains the packed data and an unpacking routine.

The unpacking routine takes care of recovering the original code and driving the packed program to execute (i.e., the "written-then-executed" procedure [38]). To avoid breaking the functionality of the original program, the unpacking routine places unpacked original instructions and related resources at the original virtual addresses instead of arbitrary memory areas [39]. The reason is that the compiler-generated instructions access memory contents via specific address offsets, but recognizing and relocating memory addresses is still unsolved for static binary rewriting [40]. Furthermore, to complicate reverse engineering, the unpacking process may contain layers of "written-then-executed" code [2, 4] (❶ in Fig. 1). However, no matter how many unpacking layers exist, each packed program needs the first layer of the unpacking routine to release other layers, which means that part of the unpacking routine is always visible to static analysis (❷ in Fig. 1).

The core of the unpacking routine is the compression or decryption algorithm, which is typically reused from mature third-party libraries [41, 42]. For example, packed malware widely adopts the aPLib compression library [43]. Due to the performance concern, we observe that unpacking routines are usually protected by light-weight (or even no) obfuscation (detailed in Sec. 7.3).

2.2 Signature-based Packer Detection

Signature-based packer detectors search the predefined textual or binary patterns using their own pattern matching grammars. Hereinafter, we focus on YARA [23], which is the most widely used tool for specifying malware signatures and performing searches. Each YARA rule consists of two essential parts: `strings` and `condition`.

Table 1: Categorized public-available YARA rules for packer detection after removing duplicates. "Packers" means supported packers. "Meta" means the rules created from the PE header information [44] and text strings. "SC Bytes" means the rules use special constructions such as wild cards.

Sources	#Packers	Search Scope		Search Content			#Total
		Address-Based	Full-Binary	Meta	Bytes	SC Bytes	
[45–50]	492	9,582	667	31	6,672	3,549	10,249

A YARA rule to detect UPX-packed programs is shown in Fig. 2. The contents below the `strings` keywords are the expressions to be matched in binary code. YARA supports four types of strings, including text strings (`$a`), text strings with regular expression, hexadecimal strings (`$b`), and hexadecimal strings with special constructions (i.e., wildcard (`??` in `$c`), jumps (`[4]` in `$d`), and alternatives (`(57|87)` in `$d`)).

To produce appropriate rules of the `strings`, security analysts need to examine enormous packed programs and find the common salient expressions. They typically extract byte features of relevant instructions rather than textual features to develop packer detection rules, because binary packing can easily conceal or camouflage text strings of the packed program. As signature-based detection directly scans binary code instead of disassembly instructions, security analysts write hexadecimal rules based on the bytes of expected instructions. The contents of `condition` define two scopes to perform pattern matching: address-based vs. full-binary matching.

Address-Based Matching In this scenario, signature-based detectors only search given rules at specific addresses. For example, the rules `$b` and `$d` in Fig. 2 will only be searched at the entry point of PE files. We notice that more than 90% of packer detection rules only perform searches at specific addresses. However, the packer developers and malware authors can easily change the instructions at the entry point to bypass the address-based matching.

Full-Binary Matching To increase the robustness of rules, analysts can let signature-based detectors search through the entire binary (e.g., rule `$a` and `$c` in Fig. 2). However, signature-based detectors match the format of bytes regardless of the instruction encoding. Problematic whole-binary matching rules will introduce high false positives.

2.3 Challenges of Generating Packer Detection Rules

Developing high-quality packer detection rules is a long-standing problem. We collect 10,249 public-available YARA rules after removing duplicated ones, and we categorize them in Table 1. A notable trend is that, as the number of packers continues to grow, the cost of manually developing and maintaining YARA rules is becoming unaffordable. Generating robust packer detection rules is faced with the following two challenges.

First, the guidelines to generate packer detection rules are missing. Human analysts rely on their experience to select features and develop signature rules. Table 1 presents that 93.3% of human-written rules only consider the bytes of packed programs' entry point, while these rules can be easily bypassed via modifying the

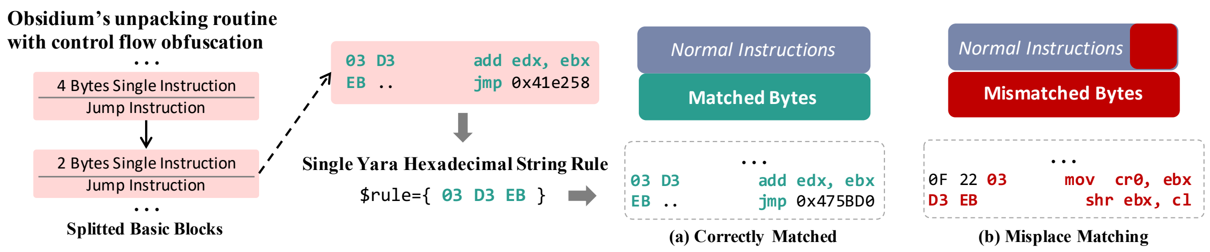


Figure 3: Comparison of different results matched by the YARA hexadecimal string rule: `$rule={03 D3 EB}`.

entry point instructions. An alternative way is to expand the search scopes to the whole binary, but the hexadecimal string rules with few bytes will result in high false positives. To counteract false positives, security analysts tend to create rules with long-length bytes—80% of rules in Table 1 are longer than 25 bytes. These rules contain multiple control transfer instructions such as `jmp` and `call`[1]. Unfortunately, they can still be easily thwarted by control-flow obfuscations such as basic block splitting.

The **second** obstacle is that packer rules mismatching with irrelevant instructions occurred often. The reason is that signature-based detection matches bytes without considering the format of instructions. This design shortcoming would lead to *misplace matching*, in which the matched bytes belong to parts of unexpected instructions. Fig. 3(b) shows an example of a hexadecimal string rule mismatching with parts of two sequential instructions; the matched instructions have different formats and semantics from the expected ones. Intuitively, the signature-based detection can support instruction matching based on static disassembly results, but performing disassembly for every program will incur extra overhead. As a result, the accumulated slowdowns of scanning large-scale packed programs will become unacceptable.

3 OVERVIEW

Our research aims to solve the challenge discussed in Sec. 2.3 and make packer detection rules generation less burdensome. In particular, we develop a new packer analysis framework to extract packer-specific genes and automatically generate YARA rules. The insight behind our approach is that the reused unpacking routine instructions are ideal candidates for packer-specific genes, because they recur in the same-packer protected programs. Furthermore, we propose a novel byte selection strategy to reduce the mismatch probability of generated YARA rules. PackGenome is effective in processing both Windows and Linux packers on x86/x64 platforms. As shown in Fig. 4, the workflow of PackGenome involves the following four steps.

❶ **Packed Program Preprocessing** This step prepares multiple same-packer protected programs for packer-specific gene extraction. By proactively interacting with packer tools, we traverse obfuscation configurations of packers to synthesize diversified packed programs with different unpacking routines. Then, we statically extract the section information from packed programs.

❷ **Packer-specific Gene Extraction** We first record the packed program's runtime information using dynamic instrumentation. Guided by the extracted section information, we adopt control flow analysis to discover unpacking routine instructions that are also visible to static analysis. Then, we find similar unpacking routine instructions that are reused in the same-packer protected programs. These instructions are candidates for packer-specific genes.

❸ **Rule Generation** At last, our framework automatically generates YARA rules from our extracted packer-specific genes. According to the information provided by the similarity analysis, it can adopt appropriate special constructions (e.g., wildcards) into YARA rules. In addition, the generation step interacts with a byte selection strategy to select the rules with a lower mismatch probability.

❹ **Byte Selection** Unlike existing packer rule development that relies on human analysts' experience, we systematically evaluate the misplace matching possibility to guide byte selection. We first convert the given bytes to possible mismatched instructions based on our predicting disassembly technique. Then, we use an N-gram technique to calculate the possibility that the converted instructions appear in real-world programs. It helps us to filter out the byte strings exhibiting a high mismatch probability.

4 PACKED PROGRAM PREPROCESSING

This preprocessing step prepares packed programs and collects necessary information from synthesized programs to assist the packer-specific gene extraction process.

Inspired by the chosen-instruction attack [51] learning knowledge through interaction with code virtualization obfuscators, we cover different unpacking routines of packers by proactively synthesizing packed programs. Note that the unpacking routine attached to the packed program is irrelevant to the semantics of the input program's instructions. The input program only needs to meet the requirements of the packing tool such as file size. The major factor determining the unpacking routine's diversity is the packer's obfuscation configurations, because the specific compression or decryption algorithm—the core of the unpacking routine, is controlled by the obfuscation configurations. To cover different unpacking routines, we traverse every configuration combination provided by the packer and synthesize corresponding packed programs.

To assist the discovery of statically visible unpacking routine instructions at runtime, we first collect the section information (i.e., name and address) of the packed programs. Because a notable feature of unpacking routines is that they need to place the unpacked original instructions and data back at the original virtual address. For example, the unpacked instructions have to be placed at the virtual address of the original non-packed program's "`.text`" section at runtime. With the help of the collected section information, we monitor the regions of packed program that are written then get executed by the statically visible, unpacking routine instructions; we also assign labels to these instructions during dynamic analysis.

[1]We show a long YARA rule with 238 bytes in Appendix B, Fig. 12.

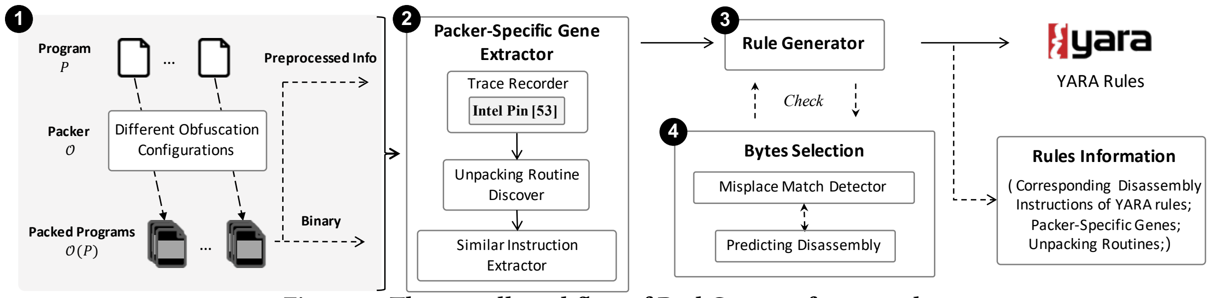

1
Program
P
...
Preprocessed Info
Packer
O
Different Obfuscation Configurations
Packed Programs
O(P)
...
Binary
2
Packer-Specific Gene Extractor
Trace Recorder
Intel Pin [53]
Unpacking Routine Discover
Similar Instruction Extractor
3
Rule Generator
Check
4
Bytes Selection
Misplace Match Detector
Predicting Disassembly
yara
YARA Rules
Rules Information
(Corresponding Disassembly Instructions of YARA rules; Packer-Specific Genes; Unpacking Routines;)

Figure 4: The overall workflow of PackGenome framework.

5 PACKER-SPECIFIC GENE EXTRACTION

In this section, we describe how to extract packer-specific genes from the unpacking routine instructions reused in the same-packer protected programs. We first record the execution trace of the first unpacking layer and assign labels to instructions. To discover unpacking routine instructions, we propagate the labels among the recorded basic blocks guided by the control-flow information and the execution numbers. At last, we extract packer-specific genes from similar instructions reused in unpacking routines.

5.1 Recording the First Unpacking Layer Execution Trace

Sophisticated packers usually adopt obfuscation (e.g., self-modifying code) to frustrate static disassembly [52]. It is difficult for static analysis to correctly extract unpacking routine instructions from the obfuscated binary. Therefore, we adopt the Intel Pin [53] framework to record the runtime information of the unpacking routine instructions that are visible to static analysis (i.e., the first unpacking layer). The reason is that YARA and other signature-based detectors only search patterns from the programs statically.

Our Pintool records the statically visible instructions that exist in the main executable and collects runtime information at the basic block level. The recorded trace information includes the memory address, the length of instruction bytes, the basic block's execution numbers, instruction bytes of basic blocks, and labels. To monitor the "written-then-executed" behaviors of instructions, we employ the runtime monitoring techniques used in Deep Packer Inspector [4]. During dynamic analysis, our Pintool assigns labels to instructions according to their runtime behaviors. If an instruction I writes unpacked instructions I' to the original code section and I' gets executed at runtime, this instruction I will be assigned a label. Note that sophisticated packed programs may adopt multiple unpacking layers [4], which iterate the procedure of writing to allocated memory and then executing the written memory. We need to monitor the "written-then-executed" region written by the instructions of the first unpacking layer (e.g., "Unpacking Layer 1" in Fig. 1) and assign labels to these instructions.

5.2 Discovering Unpacking Routine Instructions

Note that our Pintool only assigns labels to the instructions that are directly written to the monitored address such as the "written-then-executed" region. It ignores other parts of unpacking routine instructions that only decode unpacked data without writing to the monitored address at runtime. To find complete unpacking routine instructions, we propagate labels of instructions to related basic block B based on control flow analysis.

In particular, we only propagate labels among the B with similar execution numbers to avoid propagating labels to the entry point instructions, which can be easily diversified by packers. The reason is that the instructions pertaining to the decompression (or decryption) function are executed considerably more times than other instructions. Since the execution numbers of the recorded basic block N_B are mainly decided by the size of packed data, the N_B of different packed programs could vary greatly. Therefore, we compare the relative execution numbers of N_B. Formally, we define the relative execution numbers $REN(B_i)$ of the given basic block B_i as follows:

$$REN(B_i) = \frac{N_{B_i}}{\sum_{j=1}^{m} N_{B_j}} \tag{1}$$

where m is the total number of B, and B_i belongs to recorded basic blocks $\{B_1, ...B_m\}$ of a single trace. We use $REN(B_i)$ to find high-frequency labeled basic blocks B_i. Then, we strip off the self-modified instructions from the labeled B by comparing the recorded bytes of instructions with the bytes statically extracted from the same address. The rest of the statically visible unpacking routine instructions are candidates for packer-specific genes.

5.3 Extracting Packer-specific Genes

To extract the packer-specific genes from the unpacking routines instructions, we find similar instructions from the reused unpacking routines. We first calculate the similarity of labeled basic blocks $\mathcal{B}$ from multiple same-packer protected programs. Then, we use the similarity of $\mathcal{B}$ to guide the selection of packer-specific genes, and prepare the similarity information (e.g., the offset of different bytes) of syntactically similar instructions for our rule generator.

Given two packed programs P_a and P_b, we first collect the labeled basic blocks: $\{\mathcal{B}_{a1}, ..., \mathcal{B}_{an}\}$ and $\{\mathcal{B}_{b1}, ..., \mathcal{B}_{bn}\}$, respectively. To discover the $\mathcal{B}$ reused in packed programs, we compare the similarity of different $\mathcal{B}$ using the following two steps.

Bytes Given $\mathcal{B}_{ai}$ and $\mathcal{B}_{bj}$, we first directly compare their bytes. If their bytes are identical, we will skip the following comparison. Otherwise, we compare them at the slice level.

Slice To overcome the obfuscations adopted by the sophisticated packers, we compare the slices extracted from the $\mathcal{B}$. We first decompose $\mathcal{B}$ into slices S by performing the backward slicing starting from the outputs of $\mathcal{B}$. Then, we calculate the statistical similarities [37] of slices and lift slices' similarity into the similarity between

```
rule Packer_v1 {
strings:
  $a = {a4 eb} //Pa = 0.7
  $b = {21 41 3c e8 74} //Pb = 0.5
  $c = {8b 96 8c 00 00 00 8b c8 c1 e9 10 33 db 8a 1c 11
        8b d3 eb} //Pc = 0
condition:
  $a and $b and $c
}
rule Packer_v2 {
strings:
  $a = {a4 eb} //Pa = 0.7
  $b = {21 41 3c e8 74} //Pb = 0.5
condition:
  $a and $b
}
```

Figure 5: The example of YARA rules with calculated misplace matching probability.

$\mathcal{B}$. We define the similarity of slice pairs as follows:

$$SimSlice(S_a, S_b) = \sum_{\substack{k=1,l=1 \\ I_k \in S_a, I_l \in S_b}}^{n} SimIns(I_k, I_l) \Big/ n \qquad (2)$$

where S_a and S_b have the same output operands, n is the maximum instruction number of S_a and S_b, and the $SimIns(I_k, I_l)$ is used to compare the similarity of instructions, which will return 1 when the instruction format (i.e., mnemonic and operand types such as `REG`) of I_k and I_l are the same, otherwise return `0`. Then, we lift the slices' similarity into the similarity of $\mathcal{B}$ by the calculation defined as follows:

$$SimBS(\mathcal{B}_a, \mathcal{B}_b) = \sum_{\substack{i=1,j=1 \\ S_i \in \mathcal{B}_a, S_j \in \mathcal{B}_b}}^{n} SimSlice(S_i, S_j) \Big/ n \qquad (3)$$

where n is the maximum slice number of $\mathcal{B}_a$ and $\mathcal{B}_b$. If each slice group of the given two $\mathcal{B}$ is syntactically similar, the $\mathcal{B}_a$ and $\mathcal{B}_b$ are highly similar at the slice level.

According to the similarity of $\mathcal{B}$, we collect $\mathcal{B}$ as packer-specific genes to generate rules. The results of the above comparison can be divided into the following two equivalent scenarios.

Completely Equivalent If the bytes of given recorded basic blocks $\mathcal{B}_{ai}$ and $\mathcal{B}_{bj}$ are identical, we consider $\mathcal{B}_{ai}$ and $\mathcal{B}_{bj}$ are completely equivalent. For example, the compression packers (e.g., UPX) are reusing exactly the same unpacking routine instructions in each packed program (detailed in Sec. 7.3). The completely equivalent bytes can be directly used to generate YARA rules for packer detection.

Partially Equivalent When packers adopt obfuscation to protect unpacking routine instructions, we may find $\mathcal{B}_{ai}$ is only partially equivalent to $\mathcal{B}_{bj}$. It means that they have similar slices but different bytes. For example, the two slices "`mov ecx, 0x579; dec ecx;`" and "`mov ecx, 0x586; dec ecx;`" extracted from Enigma-packed programs are similar but have different bytes due to two different operand values of `mov` instructions. The $\mathcal{B}$ with a higher $SimBS(\mathcal{B}_a, \mathcal{B}_b)$ are preferred candidates for packer-specific genes.

6 YARA RULE GENERATION

Given packer-specific genes, we first generate hexadecimal string rules (HSR) from each basic block of the packer-specific genes based on the similarity information. If the bytes of packer-specific genes are completely equivalent, we directly convert these bytes to the HSR. Otherwise, we locate different bytes from the partially equivalent bytes, replace them with the elaborated special constructions (e.g., wildcards), and construct HSR. The minimum length of HSR is **two**. Next, we take a byte selection strategy to minimize misplace matching errors for the generated YARA rules.

6.1 Byte Selection

Our byte selection strategy calculates the misplace matching probability of the input YARA rules and guides the selection of YARA rules. Specifically, given an input YARA rule, we first calculate the misplace matching probability of each hexadecimal string rule P_{HSR}. Thanks to our predicting disassembly technique, we transform the mismatching probability into the occurrence probability of possible mismatched instructions (detailed in Sec. 6.2). Then, we multiply each P_{HSR} to compute the mismatch probability of a single YARA rule $\mathcal{P}_{rule}$, and filter out the rules with high mismatching probability. Taking the rules in Fig. 5 as an example, the misplace matching probability of the rule `Packer_v1` is $\mathcal{P}_{Packer\_v1} = P_a * P_b * P_c$. Since $P_c = 0$, $\mathcal{P}_{Packer\_v1} = 0$, which means the `rule Packer_v1` will not lead to misplace matching errors. In contrast, the rule `Packer_v2` has a higher misplace matching probability $\mathcal{P}_{Packer\_v2} = P_d * P_e = 0.35$. Therefore, our strategy will only retain the rule `Packer_v1`.

6.2 Calculating Misplace Matching Probability

According to the mismatch type of HSR, Fig. 6 shows how we calculate misplace matching probability of HSR P_{HSR} in two ways.

HSR entirely belongs to a single instruction. The P_{HSR} is the possibility of corresponding mismatched instruction I occurring in real-world programs. In this scenario, the length of possible mismatched HSR is in the interval $[2, 15]$, because the minimum length of HSR is 2 and the maximum byte length of x86/x64 instruction is 15. We first apply the predicting disassembly technique to find possible mismatched instructions. It synthesizes a set of possible mismatched instructions $\{I_1, ..., I_n\}$ from the given HSR (detailed in Sec. 6.3). Then, we compute the occurrence probability of each instruction p_I in the instruction database[2]. The P_{HSR} is the maximal probability of p_I. i.e., $P_{HSR} = max(p_{I_1}, ..., p_{I_n})$.

HSR partially belongs to a single instruction. The P_{HSR} is the occurrence probability of all *possible* mismatched instruction sequences IL that appear in real-world programs. In this scenario, the given HSR consists of x bytes ($x \geq 2$), and only the first i ($i \in [1, x-1]$, $i \leq 15$) bytes of HSR can be mismatched to the tail bytes of one single instruction. We first adopt the predicting disassembly to synthesize a set of possible mismatched instructions $U = \{I_1, .., I_n\}$ from the first i bytes of HSR. Then, we combine each instruction of U with the instruction sequences disassembled from the rest $x - i$ bytes as IL. To calculate the occurrence probability of possible mismatched instruction sequences p_{IL}, we apply a standard N-gram analysis to process each IL. Then, we search converted IL from our constructed N-gram database [2] and calculate the $P_{HSR} = max(p_{IL_1}, ..., p_{IL_n})$.

[2]The instruction database and N-gram database are created from the dataset *NPD*, including more than 20,000 samples collected in the real world (detailed in Sec. 7.1).

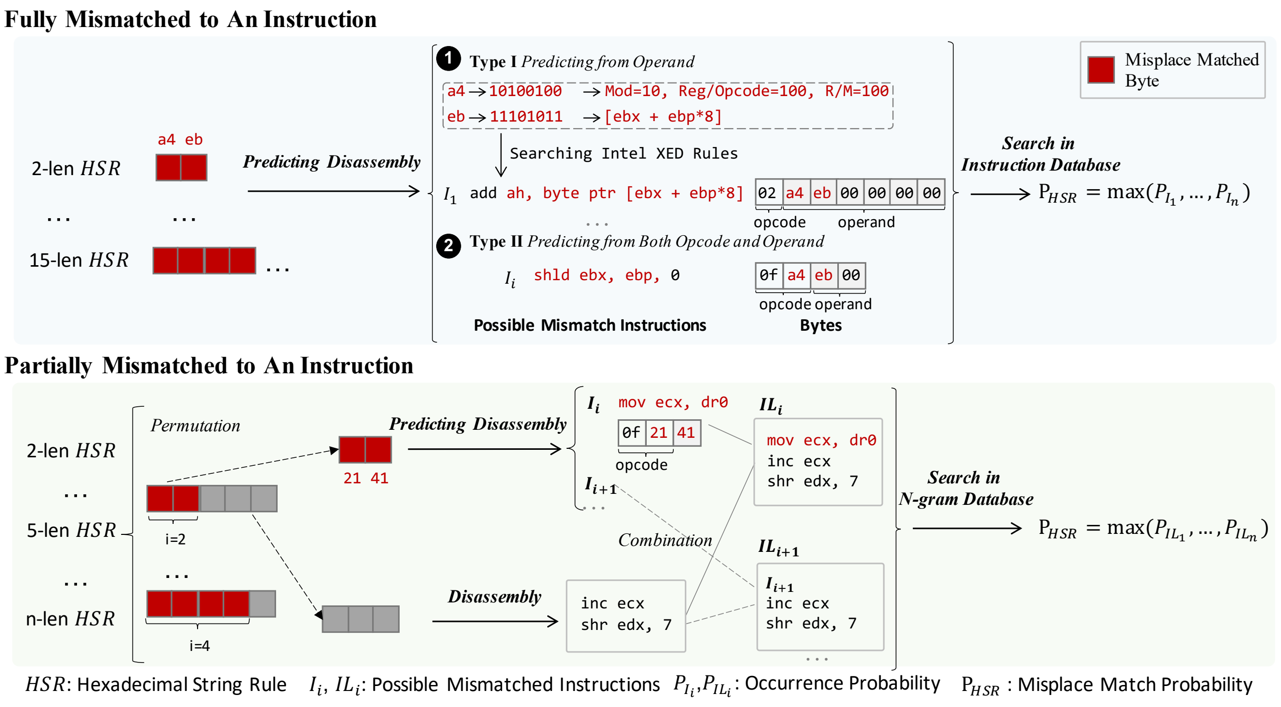

Fully Mismatched to An Instruction
Misplace Matched Byte
2-len HSR
a4 eb
15-len HSR
Predicting Disassembly
Type I Predicting from Operand
a4 → 10100100 → Mod=10, Reg/Opcode=100, R/M=100
eb → 11101011 → [ebx + ebp*8]
Searching Intel XED Rules
I1 add ah, byte ptr [ebx + ebp*8]
02 a4 eb 00 00 00 00
opcode operand
Type II Predicting from Both Opcode and Operand
Ii shld ebx, ebp, 0
0f a4 eb 00
opcode operand
Possible Mismatch Instructions
Bytes
Search in Instruction Database
P_HSR = max(P_I1, ..., P_In)
Partially Mismatched to An Instruction
2-len HSR
5-len HSR
n-len HSR
Permutation
21 41
i=2
i=4
Predicting Disassembly
Ii mov ecx, dr0
0f 21 41
opcode
Ii+1
ILi
mov ecx, dr0
inc ecx
shr edx, 7
Combination
ILi+1
Ii+1
inc ecx
shr edx, 7
Disassembly
inc ecx
shr edx, 7
Search in N-gram Database
P_HSR = max(P_IL1, ..., P_ILn)
HSR: Hexadecimal String Rule
Ii, ILi: Possible Mismatched Instructions
P_Ii, P_ILi: Occurrence Probability
P_HSR : Misplace Match Probability

Figure 6: The process of calculating the misplace matching probability of different hexadecimal string rule.

Different from the prior N-gram based techniques (e.g., MutantX-S [54]) that only extract the opcode of instructions, we use four components (i.e., prefix, opcode, mnemonic, and the format of operands) to represent an instruction during the N-gram analysis. The reason is that the opcode may not fully represent the semantics of instructions. The instructions with the same opcode could have totally different semantics. For example, two semantically different instructions "`add eax, 0x41`" and "`or eax, 0x41`" share the same opcode `0x81` (shown in Appendix Fig. 15).

Examples. The two-bytes HSR `$a` in Fig. 5 can be mismatched in two ways. For the HSR `$a` entirely belonging to a single instruction, we calculate the probability $P_a = 0.7$. For the HSR `$a` partially belonging to a single instruction, we calculate the probability $P_a = 0.3$. Since the maximum $P_a = 0.7$, the HSR `$a` should be combined with the HSR that has a low misplace matching probability when constructing YARA rules. Another example is the 19-bytes HSR `$c`. It can only partially belongs to a single instruction. Since the maximum $P_c = 0$, the HSR `$c` can be directly used in any YARA rules.

6.3 Predicting Disassembly

Please note that our goal is to find every possible instruction that can be fully misplace-matched by HSR. Intuitively, the analysts can brute-force traverse every combination of bytes that can be mismatched by HSR. However, they would face an ultra-large search space consisting of 256^{15} combinations of bytes. Because the maximum byte length of the x86/x64 instruction is 15 and the value of each byte is in the interval $[0, 255]$. For example, to find the instructions that can be fully misplace-matched by the shortest HSR (i.e., two bytes), the analysts have to traverse more than $256^{15-2} \approx 2 * 10^{31}$ combinations of bytes.

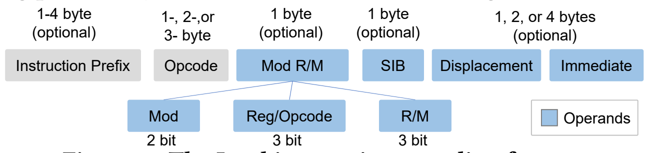


Figure 7: The Intel instruction encoding format.

Therefore, to efficiently predict every possible misplace-matched instruction of HSR, we propose the predicting disassembly technique. Given the input HSR, we first search the qualified Intel XED rules that can hold the full bytes of mismatched HSR. We choose Intel XED rules as they reveal each combination of Intel instruction encoding. To find the qualified XED rules, our approach transforms HSR into searchable formats based on the encoding grammar of XED rules. For example, as shown in ❶ of Fig. 6, the byte "`a4`" would be converted to the format `Mod=10, Reg/Opcode=100, R/M=100`. According to the components of XED rules matched by transformed HSR, the transformation and prediction process can be divided into the following three scenarios.

Predicting from Opcode (and Prefix) As defined in the Intel instruction encoding, the combination of opcode and prefix consists of predefined concrete values. After searching in the combinations of opcode and prefix, we observe that our generated HSR do not mismatch any opcode combinations of instructions. Because the last byte of our generated HSR is the opcode of control transfer instructions and the rest bytes of HSR are converted from normal disassembly instructions. Therefore, HSR cannot satisfy any combinations of opcode and prefix. Our approach only needs to process the following two types.

Type I *Predicting from Operand* Our approach transforms the given HSR into the operand encoding format of XED rules, and collects the qualified XED rules that have the same operand encoding format as the transformed HSR. As shown in Fig. 7, the components of the operand encoding include `Mod R/M`, `SIB`, `Displacement`, and `Immediate`.

For the given HSR mismatching the `Mod R/M` and `SIB`, we convert HSR to the operand encoding format and find the qualified Intel XED rules. As the example ❶ shown in Fig. 6, given HSR "`{a4 eb}`", we first convert the byte "`a4`" to `Mod=10, Reg/Opcode=100, R/M=100` based on the `Mod R/M` encoding scheme. After verifying the correctness of encoding, we convert the byte "`eb`" to the operand `[ebx+ebp*8]` based on the `SIB` encoding scheme. For the given HSR mismatching the `Displacement` and `Immediate`, we can directly convert HSR to the hexadecimal bytes. However, the probability of HSR mismatch in the `Displacement` and `Immediate` is negligible, because these components can be arbitrary hexadecimal bytes from `0x0` to `0xffffffff`. Finally, we synthesize 9,055 instructions that can be mismatched by "`{a4 eb}`".

Type II *Predicting from Both Opcode and Operand* We search the first several bytes of HSR from the combinations of opcode and prefix, and collect the qualified Intel XED rules. Then, we treat the rest bytes of HSR as Type I and search for the qualified rules from prior collected Intel XED rules. As the example ❷ shown in Fig. 6, HSR's first byte "`a4`" mismatches the tail of opcode "`0fa4`" and the second byte "`eb`" mismatches the head of the operand "`eb00`". In total, we synthesize one instruction that can be Type II mismatched by "`{a4 eb}`".

After collecting the qualified Intel XED rules, the predicting disassembly synthesizes the possible mismatched instructions from the collected rules. We describe the detailed process of predicting disassembly in Appendix Algorithm 1. Given the input HSR "`{a4 eb}`", we synthesized 9,056 instructions. Then, we calculate the occurrence probability of each synthesized instruction. The maximum occurrence probability $P_{HSR} = 0.7$, which means this HSR can be easily mismatched. When constructing the YARA rules, it should be combined with the HSR that has a low misplace matching probability.

7 EVALUATION

In this section, we evaluate PackGenome by answering the following four research questions (RQs).

- **RQ1:** Can PackGenome effectively generate detection rules for different types of packers?
- **RQ2:** How are the accuracy and efficiency of PackGenome's generated rules compared to human-written rules and other automated rule generators?
- **RQ3:** How is the scalability of PackGenome's generated rules on detecting packed samples?
- **RQ4:** How is the performance of PackGenome's generated rules when detecting programs in the wild?

To answer RQ1, we apply PackGenome to generate 70 YARA rules for 20 popular packers, and evaluate the contributions of our byte selection technique (Sec. 7.2). We also discuss the new findings of our study (Sec. 7.3). For RQ2, we design two experiments to measure the accuracy of different rules (Sec. 7.4). To evaluate the efficiency, we compare the running time of YARA and Detect it Easy (DIE) on samples with four different magnitudes (Sec. 7.5). For RQ3, we evaluate generated rules on the packed programs with multiple versions. We also measure the scalability of PackGenome on Linux packed programs, custom packers, and low-entropy samples (Sec. 7.6). For RQ4, we evaluate PackGenome on the real-world malware samples and perform case studies to show the feasibility of detecting in-the-wild custom packers and adversarial samples (Sec. 7.7 and Appendix C, D, E, F).

7.1 Experimental Setup and Datasets

Peer Rules for Comparison We first collect public-available human-written packer detection rules from GitHub, including 9,296 rules from six open-source YARA rule libraries [45–50], and 5,703 rules converted from PEiD and ExeInfo PE [55]. After removing duplicates, we obtained 10,249 unique rules. Among the collected rules, only 44 rules support x64 packed program detection. Meanwhile, we compare PackGenome with the state-of-the-art automatic rule generation tool, *AutoYara* [21], which combines a biclustering algorithm and large N-grams to generate high-quality rules from limited samples.

Packers for Rule Generation We select off-the-shelf packers from recent papers [6, 38]. Finally, we shortlist 20 packers (listed in Table 2) because they can work properly in modern operating systems. They are used to generate x86/x64 Windows packed programs. As the existing x64 human-written rules only support four packers (i.e., UPX, MPRESS, Themida, and Enigma), we also use these four packers to generate x64 packed programs.

Rule Generation Datasets (RGD) We traverse multiple versions and configurations of 20 off-the-shelf packers to generate packed programs (*RGD*). Given each configuration of packers, we generate three packed samples as the input of PackGenome during the experiment. The purpose here is to estimate the performance of PackGenome in the worst scenario that was pointed out by AutoYara, i.e., the number of the same-packer protected programs is limited in real-world scenarios. Similarly, we generate 16 packed samples, greater than most rule generation scenarios (i.e., ≤10 samples) reported in the AutoYara paper [21], as the input of AutoYara to generate rules for each configuration of packers.

Testing Datasets To construct the labeled packed samples dataset *LPD* that links to known packers, we first generated 38,663 x86 programs and 2,237 x64 programs by combining 20 off-the-shelf packers with multiple versions and configurations. We also constructed a non-packed samples dataset *NPD* to measure the false positive rates of rules. This dataset consists of 26,326 non-packed malware samples retrieved from the recent work [17], and 1,224 collected real-world benign programs such as system files. To evaluate the performance of rules in the real world, we collected 579,832 malware samples from VX-underground [56], VirusTotal, and GitHub [57, 58]. They are divided into three categories. We use 560,285 Windows APT and malware samples as *WD1* to evaluate the effectiveness of the rules. We also retrieved 18,288 packed and evasive samples from the low entropy dataset [17] (*WD2*). It helps us to evaluate the robustness of rules on adversarial samples. Furthermore, we retrieved 1,302 x86/x64 Linux malware as *WD3*, which is used to evaluate the scalability of our generated rules on different systems.

Testing Environment We run all experiments on a testbed machine with Intel i7-6700 CPU (4 cores, 3.40GHz), 32GB RAM, 1.8TB Hard Disk, running Windows 10.

Table 2: Comparing PackGenome with other rules on the *LPD* dataset. "Configurations" reports the obfuscation configurations of packers we use to generate packed programs. "Related" means the configuration that affects the generated unpacking routine instructions and "Total" means the total number of configurations used in the program generation process. "Obfuscation" reports the obfuscation adopted by the first layer of unpacking routines. "GR" reports the number of the generated rules. The order of column "FPR" in "Our approach" is (PackGenome-N, PackGenome).

Packers	# of Vers	Configurations		Obfuscation [1]	Our Approach				Human-Written Rules			AutoYara[21]			Detect It Easy [16]		
		Related	Total		GR	FPR [%]	FNR [%]	Time [s]	FPR [%]	FNR [%]	Time [s]	FPR [%]	FNR [%]	Time [s]	FPR [%]	FNR [%]	Time [s]
UPX	6	8	36	N	10	(13.5, 0)	0	1.9	100	0	5.7	22.7	68.0	1.2	0	0	729
Armadillo	3	5	33	EU;N	4	(100, 0)	0	8.5	79.3	6.29	28.4	100	42.3	4.2	0	0	592
MPRESS	3	1	10	N	2	(0, 0)	0	0.2	100	0	0.9	38.8	95.9	0.2	0	0	55
PECompact	2	5	46	N	5	(11.9, 0)	0	3.0	91.4	0	8.9	18.5	86.9	1.7	0	0	1032
ASPack	3	1	8	N	3	(14.2, 0)	0	1.3	100	0	4.3	19.5	70.7	0.9	0	0	503
VMProtect	2	6	10	VM	9	(96.4, 0)	0	11.6	15.9	0	17.6	19.0	88.7	5.4	0	0	545
FSG	1	1	1	N	1	(14.5, 0)	0	0.2	100	0	0.9	19.0	88.5	0.1	0	0	33
Obsidium	1	7	37	CF	7	(98.8, 0)	0	1.3	1.82	100	3.1	17.1	89.0	0.8	0	9.3	275
Petite	1	5	21	N	1	(12.5, 0)	0	0.5	3.03	0	1.8	19.6	98.7	0.4	0	0	166
kkrunchy	2	1	4	N	2	(26.0, 0)	0	0.2	2.33	1.95	0.9	31.9	73.5	0.1	0	0	42
MEW	1	2	9	N	2	(12.5, 0)	0	0.5	1.25	0	1.8	0.33	0	0.4	0	0.5	201
NsPack	3	1	15	N	1	(13.3, 0)	0	0.8	71.8	0	2.5	20.3	90.3	0.5	21.1	61.4	287
Themida	2	9	17	EU;SC; EU+VM;N	9	(1.0, 0)	0	12.6	10.7	35.8	26.3	19.5	67.5	6.5	5.33	0	639
ACProtect	2	2	14	N	3	(89.9, 0)	0	2.2	98.6	0	5.6	19.5	67.3	1.4	21.6	0	512
ZProtect	1	1	29	EU+CF	1	(100, 0)	0	1.1	5.08	24.9	2.9	19.2	17.8	0.6	0.08	0	212
Winlicense	1	2	31	EU+VM;EU	4	(0.1, 0)	0	8.1	19.7	0	16.4	19.5	78.1	4.1	8.28	0	413
Enigma	4	1	44	EU+SO	1	(100, 0)	0	7.1	100	0	12.7	8.42	0	4.2	0	0	698
MoleBox	1	1	10	N	1	(91.0, 0)	0	0.8	2.22	0	1.9	17.4	94.3	0.4	0	100	138
WinUpack	1	1	15	N	2	(12.0, 0)	0	0.3	100	0	1.3	18.5	92.5	0.2	0	0	138
expressor	1	2	15	N	2	(25.1, 0)	0	0.4	0	100	1.3	38.8	90.4	0.3	0	0	95

[1] "N" means not obfuscated, "VM" means code virtualization obfuscation, "CF" means control-flow obfuscation, "EU" means the first layer of unpacking instructions are encrypted, "SO" means single obfuscation such as junk instructions, "+" means using both obfuscation at the same time, ";" separates multiple types of unpacking instructions.

7.2 Rule Generation of PackGenome

We use the *RGD* dataset as the input of PackGenome to generate rules. To evaluate the effectiveness of PackGenome and the contribution of the byte selection technique, we generate rules under two different configurations: (i) **PackGenome**: PackGenome generates rules from programs packed in the same configuration of packers. (ii) **PackGenome-N**: PackGenome without byte selection technique. Then, we apply generated rules to the *LPD* dataset and compare the detection accuracy. As shown in the "FPR" column of "Our Approach" in Table 2, the byte selection technique can effectively reduce the mismatch possibility of our generated YARA rules. After inspecting the PackGenome-N generated rules, we find that most false positives are introduced by the hexadecimal string rules with a high mismatch possibility. For example, more than thousands of false positives are caused by a Winlicense detection rule, which contains 13 hexadecimal string rules with a mismatch possibility greater than 0.8.

7.3 New Findings of Packers

We first apply PackGenome-generated rules to the labeled packed dataset *LPD* (shown in Table 2), and examine extracted packer-specific genes. The new findings of packers are described in the following paragraphs.

Configuration of Packers As the configuration of packers controls the decompression algorithms and obfuscations used in unpacking routines, we traverse the configurations provided by the packers. The total number of each packer's configurations is shown in the "Total" column of Table 2. We find that only parts of configurations affect generated unpacking routines (shown in the "Related" column of Table 2). Most of these configurations are the options of compression algorithms. For example, UPX provides four compression options (i.e., Nrv2d, Nrv2e, Nrv2b, and LZMA).

Packer-specific Genes We notice that most of the extracted packer-specific genes are the decompression (or decryption) functions. The classification of packer-specific genes is shown in Fig. 10. Many packers use similar unpacking algorithms, especially the standard decompression algorithms such as aPLib. For example, FSG v1.x and MEW v1.x use the same unpacking routine instructions.

Obfuscation of Unpacking Routines We find out that six packers' unpacking routines are obfuscated. ZProtect and Obsidium adopt control-flow obfuscation to split their unpacking routine into many small basic blocks. In their obfuscated unpacking routines, each chained basic blocks consist of only two instructions. However, thanks to our byte selection strategy, we can combine multiple short hexadecimal string rules to generate YARA rules with a low mismatch possibility. VMProtect randomly inserts junk instructions into the unpacking routines instructions. But our rules can use wildcards to escape these junk instructions. Oreans Technologies's Themida and Winlicense share the same instructions to encrypt their unpacking routines and decrypt the first unpacking layer at runtime. PackGenome can still capture their reused decryption instructions as packer-specific genes.

Answer to RQ1: We extract packer-specific genes and generate 70 rules for 20 off-the-shelf packers. Our byte selection technique can help PackGenome generate rules with a low misplace matching possibility.

Table 3: Comparing PackGenome with other rules on the *NPD* dataset. Due to space limitations, we summarize the detection results of 20 packers.

LPDrules	PackGenome		Human-Written Rules		AutoYara[21]		Detect It Easy [16]	
	FPR [%]	Time [s]	FPR [%]	Time [s]	FPR [%]	Time [s]	FPR [%]	Time [s]
Total (20)	0	40.8	22.8	73.2	18.9	26.6	0	5205

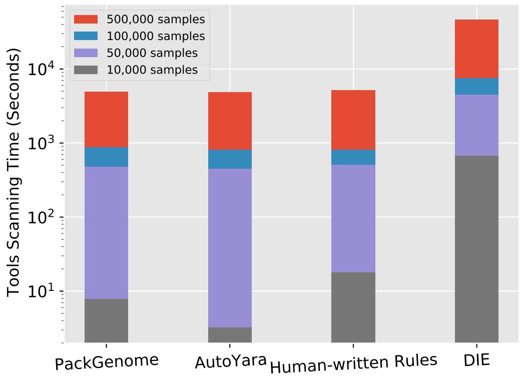


Figure 8: Scanning time comparison under four different sample magnitudes.

7.4 Accuracy

Our generated rules should accurately identify packed programs and ignore non-packed programs. To validate this hypothesis, we conducted the following two experiments.

Experiment I: Matching Labeled Packed Programs We apply each tool to the *LPD* dataset. As can be seen from Table 2, our rules outperform other rules. In contrast, AutoYara-generated rules can only detect a limited number of packed programs correctly, because they usually contain the common strings (e.g., "`GetProcAddress`") that are widely used by different packers. Meanwhile, AutoYara's large N-gram (n≥8) cannot capture the features of the unpacking routines protected by control-flow obfuscation. Another observation is that DIE performs better than other human-written rules. After examining the signature rules of DIE, we find out that many rules are created from the metadata of packers (e.g., the section name and unique strings). Unfortunately, they can be easily evaded or misled by in-the-wild custom packers (detailed in Sec. 7.7).

Experiment II: Matching Non-packed Programs We use the *NPD* dataset to measure the false positives rate that rules mistakenly match the non-packed programs. From the results shown in Table 3, we can see that our rules and DIE have zero false positive rate. In contrast, the human-written rules exhibit the highest false positive rate. Most false positives are introduced by the rules created from insignificant features. For example, an Armadillo packer detection rule mistakenly identifies 56 non-packed samples as packed (detailed in Appendix B).

7.5 Efficiency

To evaluate the efficiency of rules when processing massive programs, we compare our generated rules with human-written rules, AutoYara, and DIE using different amounts of programs randomly selected from the *WD1* dataset. During the experiments, we use four threads to execute YARA and DIE. The running times of YARA-based tools and DIE are shown in Fig. 8. The scanning overhead of our generated rules is on a par with the human-written YARA rules and the AutoYara-generated rules. In contrast, DIE performs worse than YARA-based tools, because the JavaScript-like grammar of DIE spends a lot of time on parsing and matching programs.

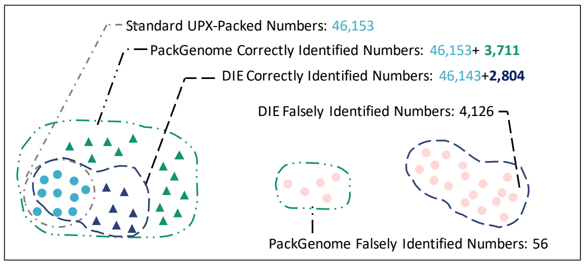

Standard UPX-Packed Numbers: 46,153
PackGenome Correctly Identified Numbers: 46,153+ 3,711
DIE Correctly Identified Numbers: 46,143+2,804
DIE Falsely Identified Numbers: 4,126
PackGenome Falsely Identified Numbers: 56

Figure 9: Comparison of UPX samples detected by PackGenome and DIE on the *WD1* dataset.

Answer to RQ2: Our generated rules outperform state-of-the-art human-written rules and an automatic rule generation tool on the labeled packed dataset and non-packed dataset. The scanning overhead of our generated rules is acceptable.

7.6 Scalability

Since the packer-specific genes are reused by the packers, our generated rules would be suitable for multiple scenarios such as detecting custom packers. We performed the following four experiments to validate this hypothesis.

Different Versions of Packers After examining the packer-specific genes extracted from 11 packers with multiple versions (detailed in Table 2), we find that nine packers reuse the unpacking routines across different versions. Our rules can directly detect multiple versions of packed programs that reuse the same unpacking routines. For example, as the classification of packer-specific genes shown in Fig. 10, four different versions of Enigma share the same unpacking routines but use completely different entry point instructions. A single PackGenome-generated rule is enough to detect different versions of Enigma-packed programs. In contrast, human analysts have to repeat the tedious rule development process when creating rules from the entry point instructions of packers. For example, we find 61 human-written YARA rules are developed for matching the entry point of Enigma-packed programs.

Different Systems A packer may support multiple OSs at the same time. For example, UPX supports different executable formats such as Linux and Windows programs. We use the Linux malware samples dataset *WD3* to evaluate whether our generated rules, created only from UPX-packed Windows programs, can also identify the UPX-packed Linux programs. Our evaluation shows that PackGenome-generated rules can successfully recognize all of 87 UPX-packed Linux programs. Because UPX reuses the same instructions of the unpacking algorithm in generated x86/x64 Linux and Windows packed programs. In contrast, only three human-written rules, created from compression algorithms (e.g., Nrv2x),

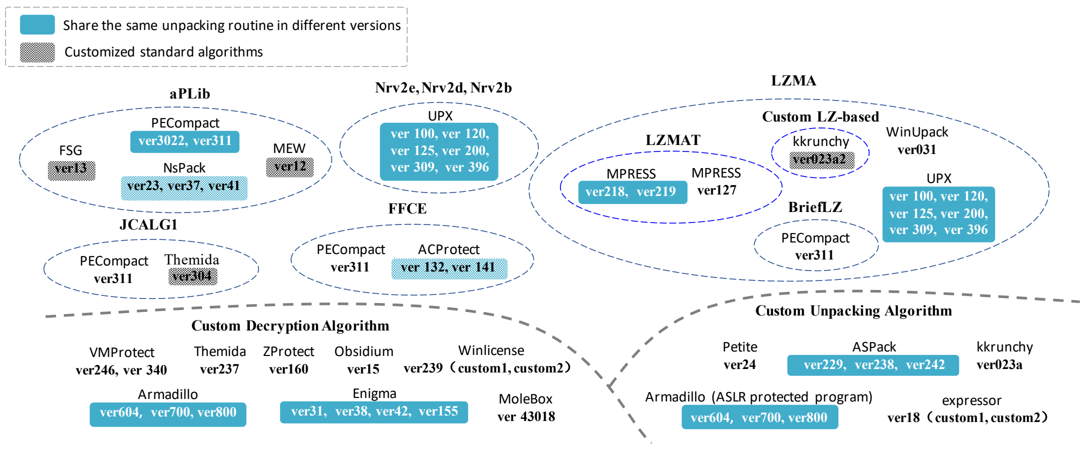


Figure 10: The classification of first layer unpacking routines by comparing with the packer-specific genes extracted from PECompact-packed programs.

can detect 34 UPX-packed programs. These long-length rules contain many consecutive basic blocks. They can be easily thwarted by the control-flow obfuscations (detailed in Appendix B). DIE can identify 64 programs. Because DIE's meta-information-based UPX detection rules are evaded by custom UPX packers.

Custom Packers This experiment evaluates the ability of our generated rules on detecting custom UPX variants. We choose UPX because it is the most widely used open-source packer and is usually customized by malware authors. Malware authors typically camouflage the features of standard packers by modifying the unique strings (e.g., "`UPX`") or the entry point instructions [12]. We first apply our generated rules and DIE on the *WD1* datasets, and filter out the programs packed by standard UPX. To identify standard UPX-packed programs, we use the rules created from the entry point instructions of the standard UPX-packed programs. Then, we manually inspect whether the detected samples are generated by custom UPX packers.

The experiment results show that our generated rules capture 907 unique custom packed programs with low false positives (shown in Fig. 9 and Table 4). After examining the packed samples, we find that these custom packers reuse the standard UPX's decompression algorithms. For example, we find that a packed sample from APT 29 can bypass the entropy-based detection and most human-written rules, including the rules created from the entry point instructions and rules of DIE (detailed in Appendix C).

Low Entropy Samples To demonstrate the robustness of our generated rules in detecting adversarial packed samples, we also apply our rules to the *WD2* dataset, which consists of the low entropy packed programs discovered by the study [17]. Mantovani et al.'s study [17] points out that the existing off-the-shelf packers detectors (e.g., DIE) are unable to identify low entropy packed samples. They attribute this to the finding that programs packed by off-the-shelf packers would have a high entropy. However, different from Mantovani et al. find only three samples packed by known packers, PackGenome-generated rules identify 47 samples packed by the off-the-shelf packers. These low entropy samples are protected by the off-the-shelf packer combined with the low-entropy technique.

Table 4: Comparing with DIE in the *WD1* dataset. We choose five popular packers as targets and verify the detection results. "#UD" reports the number of unique detected samples which cannot be discovered by another tool. "#FD" reports the number of falsely detected samples.

Packers	PackGenome			Detect It Easy [16]		
	#UD	#FD	#Total	#UD	#FD	#Total
Open Source Compression Algorithm						
UPX	907	56	49,920	0	4,117	53,083
MPRESS	197	0	791	0	48	642
Close Source						
ASPack	466	10	7,578	0	421	7,523
Obsidium	2	0	43	0	974	1,015
PECompact	9,449	138	16,440	0	12	6,805
Themida	35	1	1,422	0	2	1,388

For example, two samples with entropy less than 4.0[3] are packed by standard NsPack (detailed in Appendix D).

Answer to RQ3: Our generated rules are suitable for detecting different versions of packers. The rules created from packer-specific genes can directly detect custom packers that reuse the same unpacking routines.

7.7 Performance in the wild

To study the accuracy and robustness of our generated rules in the real world, we compare our rules with DIE on the *WD1* dataset. We choose DIE as it outperforms other human-written rules and AutoYara. Considering the sample number of *WD1* is more than 560K, which exceeds the ability of manually reverse engineering. We choose five popular packers (i.e., UPX, MPRESS, ASPack, Obsidium, PECompact, and Themida) as targets, and filter out the incomplete samples (e.g., unpacked failed samples).

[3]Existing work takes the entropy value of 7.0 or higher as the signal of a packed program [4, 17].

From the results shown in Table 4, we can find that our generated rules perform better than DIE. Our generated rules have few to no false positives. In contrast, DIE can be easily misled by in-the-wild custom packers. The reason is that many rules of DIE rely on parsing the meta-information of packed programs. But custom packers can effortlessly eliminate these features, and even camouflage as standard packers by using the same meta-information (e.g., the custom packer camouflages as UPX detailed in Appendix E). In contrast, our generated rules can accurately identify packed programs that reuse the unpacking algorithms of standard packers.

Answer to RQ4: Our generated rules created from the packer-specific genes are robust to detect custom packers in the wild with low false positive rates.

8 RELATED WORK

We have summarized the literature of packer detection in Sec. 1 and Sec. 2.2. This section describes the related work on YARA improvement and binary unpacking.

YARA Improvement We categorize the existing works of improving YARA rules into two directions: (i) automatically producing YARA rules to reduce manual efforts, and (ii) optimizing YARA rule search performance. *YaraGenerator* [59] generates rules based on the most common textual features (e.g., strings) shared across malware families. *yabin* [60] uses the fix-length bytes of function prologues to generate rules. *yarGen* [61] creates rules from salient text and hexadecimal strings, which are filtered from several pre-built "good string" databases. *AutoYara* [21] combines large n-grams search and biclustering algorithm to generate rules. It outperforms the aforementioned rule generators and even skilled analysts. However, as admitted by AutoYara's authors [21], binary packing can impede all of the above YARA automation tools, because they create rules from common textual strings or bytes of malware payload. We have demonstrated that AutoYara's performance on packed programs is poor. An orthogonal work, *YARIX* [62], builds a preprocessed inverted malware file index to efficiently search for YARA rules. PackGenome-generated rules can also benefit from the search engine of YARIX, and we expect several orders of magnitudes performance boost on packed program detection.

Binary Unpacking Over the past two decades, this is a long-standing challenge in malware analysis. Due to the rise of machine-learning-based malware classifiers, binary unpacking has recently undergone a renaissance [4, 6, 9, 17, 38, 63]. The classic way, represented by *Deep Packer Inspector* [4], dynamically monitors the "written-then-executed" unpacking layers to identify the original entry point (OEP). The recent innovations are to propose a new heuristic to quickly determine the end of unpacking [38] or take advantage of hardware features [9]. For example, *BinUnpack* [38] monitors the API calls based on kernel-level DLL hijacking techniques; it can quickly locate OEP by capturing the "rebuilt-then-called" feature of import address tables. *API-Xray* [9] leverages hardware-assisted tracing to defeat API obfuscation schemes and then reconstruct API import tables, so that the unpacked malware payload can be executed independently. Facing millions of malware samples, PackGenome is an appealing complement to generic unpacking tools: once PackGenome rapidly identifies packed executable files, they can be flagged as high priority for further binary unpacking.

9 DISCUSSION

Missing Brand-new Packers Like other signature-based approaches, PackGenome bears with a similar limitation: it may miss brand-new packers that reveal totally different signatures. If the brand-new packer is accessible, PackGenome can still generate robust rules from proactively synthesized packed programs. As for the inaccessible packers, one approach is to use PackGenome directly generates rules from the manually collected packed programs that are potentially protected by the same packer. Our experiments show that PackGenome can successfully generate robust detection rules for inaccessible packers (detailed in Appendix F). Another possible countermeasure is to leverage the unpacking routine's side channel information. For example, the unpacking process performs iterations of decryption or decompression, which can incur identifiable deviations in hardware events [64]. We will explore the direction of modeling hardware performance counter values to detect packers.

Unavoidable Byte Mismatch As discussed in Sec. 2.3, due to the performance concern, signature-based detection tools mainly search for bytes rather than the expected form of instructions. Especially under the scope of full-binary matching, some YARA rules will introduce false positives. On the other side, performing binary disassembly and instruction searches are too expensive to process large-scale programs. PackGenome attempts to reduce the mismatch rate via our proposed byte selection strategy, which strikes a delicate balance between byte mismatch and performance.

Heavyweight Obfuscation Another limitation of signature-based detectors is that they cannot handle heavyweight obfuscation by nature. YARA rules are like a piece of programming language but only with limited grammar expression power, and we have already adopted special constructions such as wildcards to overcome lightweight obfuscations such as junk code. Determined attackers can obfuscate packer-specific genes using syntactically different instructions. Like our response to brand-new packers, a promising countermeasure is to explore tamper-resistant hardware features. We leave it as our future work.

10 CONCLUSION

Over the past two decades, packed malware in circulation is a tremendous amount. Security analysts rely on signature-based detection to quickly determine the packing technique/tool used; after that, unpacking a malware sample becomes easier. However, existing work on packer signature generation heavily depends on human analysts' experience, which makes the process of writing and maintaining rules painful, error-prone, and tedious. In this paper, we develop PackGenome, an automatic YARA rule generation framework for packer detection. We harvest packer detection rules from the unpacking routine, which is reused by the same-packer protected programs. Furthermore, we propose the first model to systematically evaluate the mismatch probability of bytes rules. Our large-scale experiments show that PackGenome outperforms existing human-written rules and peer tools with zero false negatives, low false positives, and a negligible scanning overhead increase.

REFERENCES

[1] Trivikram Muralidharan, Aviad Cohen, Noa Gerson, and Nir Nissim. 2022. File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements. *ACM Computing Surveys (CSUR)* (April 2022).

[2] Kevin A. Roundy and Barton P. Miller. 2013. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys (CSUR)* 46, 1 (2013), 1–32.

[3] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. 2021. An Inside Look into the Practice of Malware Analysis. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 3053–3069.

[4] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 659–673.

[5] Babak Rahbarinia, Marco Balduzzi, and Roberto Perdisci. 2017. Exploring the Long Tail of (Malicious) Software Downloads. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 391–402.

[6] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[7] Christian Wressnegger, Kevin Freeman, Fabian Yamaguchi, and Konrad Rieck. 2017. Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM, 587–598.

[8] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer Cham, 73–96.

[9] Binlin Cheng, Jiang Ming, Erika A. Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean Yves Marion. 2021. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association, 3451–3468.

[10] VirusTotal. VirusTotal - Stats. https://www.virustotal.com/gui/stats (accessed on 2022-12-09).

[11] Erin Avllazagaj, Ziyun Zhu, Leyla Bilge, Davide Balzarotti, and Tudor Dumitras. 2021. When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association, 3487–3504.

[12] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 161–175.

[13] Robert Lyda and James Hamrock. 2007. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security and Privacy* 5, 2 (2007), 40–45.

[14] Guhyeon Jeong, Euijin Choo, Joosuk Lee, Munkhbayar Bat-Erdene, and Heejo Lee. 2010. Generic Unpacking using Entropy Analysis. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE'10)*. IEEE, 114–121.

[15] Munkhbayar Bat-Erdene, Taebeom Kim, Hyundo Park, and Heejo Lee. 2017. Packer Detection for Multi-Layer Executables Using Entropy Analysis. *Entropy* 19, 3 (2017), 1–18.

[16] Horsicq. Detect-It-Easy. https://github.com/horsicq/Detect-It-Easy (accessed on 2022-12-07).

[17] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. 2020. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[18] Fabrizio Biondi, Michael A. Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. 2019. Effective, Efficient, and Robust Packing Detection and Classification. *Computers & Security* 85 (2019), 436–451.

[19] Fabian Kaczmarczyck, Bernhard Grill, Luca Invernizzi, Jennifer Pullman, Cecilia M. Procopiuc, David Tao, Borbala Benko, and Elie Bursztein. 2020. Spotlight: Malware Lead Generation at Scale. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*. ACM, 17–27.

[20] Erik Bergenholtz, Emiliano Casalicchio, Dragos Ilie, and Andrew Moss. 2020. Detection of Metamorphic Malware Packers Using Multilayered LSTM Networks. In *Proceedings of the 22nd International Conference on Information and Communications Security (ICICS)*.

[21] Edward Raff, Richard Zak, Gary Lopez Munoz, William Fleming, Hyrum S. Anderson, Bobby Filar, Charles Nicholas, and James Holt. 2020. Automatic Yara Rule Generation Using Biclustering. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security (AISec@CCS 2020)*. ACM, 71–82.

[22] Xianwei Gao, Changzhen Hu, Chun Shan, and Weijie Han. 2022. MaliCage: A Packed Malware Family Classification Framework based on DNN and GAN. *Journal of Information Security and Applications* 68 (2022), 2214–2126.

[23] Victor Manuel Alvarez. YARA — The Pattern Matching Swiss Knife for Malware Researchers. https://virustotal.github.io/yara/ (accessed on 2022-12-09).

[24] Aldeid. PEiD. https://www.aldeid.com/wiki/PEiD (accessed on 2022-12-09).

[25] Evan Downing, Yisroel Mirsky, Kyuhong Park, and Wenke Lee. 2021. DeepReflect: Discovering Malicious Functionality through Binary Reconstruction. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association, 3469–3486.

[26] Kyuhong Park, Burak Sahin, Yongheng Chen, Jisheng Zhao, Evan Downing, Hong Hu, and Wenke Lee. 2021. Identifying Behavior Dispatchers for Malware Analysis. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM, 759–773.

[27] Unipacker. Unpacking PE files using Unicorn Engine. https://github.com/uniPacker/uniPacker (accessed on 2022-12-09).

[28] Daniel Votipka, Seth M. Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle M. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. USENIX Association, 1875–1892.

[29] Oreans Technologies. Themida Overview. https://www.oreans.com/themida.php (accessed on 2022-12-09).

[30] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. 2008. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th Recent Advances in Intrusion Detection (RAID)*. Springer Berlin, Heidelberg, 98–115.

[31] Dhondta. Awesome Executable Packing. https://github.com/dhondta/awesome-executable-packing (accessed on 2022-12-09).

[32] Ange Albertini. Packers. https://corkami.blogspot.com/ (accessed on 2022-12-09).

[33] Rufus Brown, Van Ta, Douglas Bienstock, Geoff Ackerman, and John Wolfram. Does This Look Infected? A Summary of APT41 Targeting U.S. State Governments. https://www.mandiant.com/resources/apt41-us-state-governments (accessed on 2022-12-09).

[34] Cisco Talos Intelligence Group. New Research Paper: Prevalence and impact of low-entropy packing schemes in the malware ecosystem. https://blog.talosintelligence.com/2020/02/new-research-paper-prevalence-and.html (accessed on 2022-12-09).

[35] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (accessed on 2022-12-09).

[36] Ajit Varki and Tasha K. Altheide. 2005. Comparing the human and chimpanzee genomes: Searching for needles in a haystack. *Genome Research* 15, 12 (2005), 1746–1758.

[37] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 266–280.

[38] Binlin Cheng, Jiang Ming, Jianming Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-yves Marion. 2018. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 395–411.

[39] Arne Swinnen and Alaeddine Mesbahi. 2014. One Packer to Rule them All: Empirical Identification, Comparison and Circumvention of Current Antivirus Detection Techniques. In *BlackHat USA*. BlackHat, 1–55.

[40] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[41] Tomislav Pericin. 2011. Reversing software compressions: Tale of dragons and men who slay them. In *REcon 2011*. REcon.

[42] the MITRE Corporation. Obfuscated Files or Information: Software Packing. https://attack.mitre.org/techniques/T1027/002/ (accessed on 2022-12-09).

[43] Thomas Barabosch. The malware analyst's guide to aPLib decompression. https://0xc0decafe.com/malware-analysts-guide-to-aplib-decompression (accessed on 2022-12-09).

[44] Microsoft. PE Format. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (accessed on 2022-12-09).

[45] Yara-rules. rules. https://github.com/Yara-Rules/rules (accessed on 2022-12-09).

[46] Avast. retdec. https://github.com/avast/retdec/tree/master/support/yara_patterns/tools (accessed on 2022-12-09).

[47] JusticeRage. Manalyze. https://github.com/JusticeRage/Manalyze (accessed on 2022-12-09).

[48] Godaddy. yara-rules. https://github.com/godaddy/yara-rules/ (accessed on 2022-12-09).

[49] AlienVault-OTX. OTX-Python-SDK. https://github.com/AlienVault-OTX/OTX-Python-SDK (accessed on 2022-12-09).

[50] X64dbg. yarasigs. https://github.com/x64dbg/yarasigs (accessed on 2022-12-09).

[51] Shijia Li, Chunfu Jia, Pengda Qiu, Qiyuan Chen, Jiang Ming, and Debin Gao. 2022. Chosen-Instruction Attack Against Commercial Code Virtualization Obfuscators. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[52] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 732–744.

[53] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM Press, 190–200.

[54] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. 2013. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 187–198.

[55] A.S.L. EXEINFO PE. http://www.exeinfo.byethost18.com (accessed on 2022-12-09).

[56] Vx-underground team. vx-underground. https://samples.vx-underground.org/ (accessed on 2022-12-09).

[57] Cyber-research. APTMalware. https://github.com/cyber-research/APTMalware (accessed on 2022-12-09).

[58] MalwareSamples. Linux-Malware-Samples. https://github.com/MalwareSamples/Linux-Malware-Samples (accessed on 2022-12-09).

[59] Xen0ph0n. YaraGenerator. https://github.com/Xen0ph0n/YaraGenerator (accessed on 2022-12-09).

[60] AlienVault-OTX. yabin. https://github.com/AlienVault-OTX/yabin (accessed on 2022-12-09).

[61] Neo23x0. yarGen. https://github.com/Neo23x0/yarGen (accessed on 2022-12-09).

[62] Michael Brengel and Christian Rossow. 2021. YARIX: Scalable YARA-based Malware Intelligence. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association, 3541–3558.

[63] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, 745–756.

[64] Jay Mayank Patel. 2019. *On the Feasibility of Malware Unpacking with Hardware Performance Counters*. Master's thesis. University of Texas at Arlington.

[65] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2015. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*. USENIX Association, 1057–1072.

[66] Kent Griffin, Scott Schneider, Xin Hu, and Tzi Cker Chiueh. 2009. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th Recent Advances in Intrusion Detection (RAID)*. Springer Berlin, Heidelberg, 101–120.

[67] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*. ACM Press, 611.

[68] David Dagon, Cliff Changchun Zou, and Wenke Lee. 2006. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2–13.

[69] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, and Saurabh Shintre. 2021. Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*. ACM, 744–758.

[70] Lamine Noureddine, Annelie Heuser, Cassius Puodzius, and Olivier Zendra. 2021. SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution. In *Proceedings of the 11th ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 281–292.

[71] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, Anthony D. Joseph, and J. D. Tygar. 2016. Reviewer Integration and Performance Measurement for Malware Detection. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer Cham, 122–141.

[72] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. 2008. McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*. IEEE, 301–310.

[73] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 289–300.

[74] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. MBA-Blast: Unveiling and simplifying mixed boolean-arithmetic obfuscation. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. USENIX Association, 1701–1718.

[75] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th Usenix Security Symposium (USENIX Security)*. USENIX Association, 253–270.

[76] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Network and Distributes System Security (NDSS)*. Internet Society.

[77] Guanhua Yan, Nathan Brown, and Deguang Kong. 2013. Exploring Discriminatory Features for Automated Malware Classification. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer Berlin, Heidelberg, 41–61.

[78] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. 2009. PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In *Proceedings of the 12th Recent Advances in Intrusion Detection (RAID)*. Springer Berlin, Heidelberg, 121–141.

[79] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. 2008. Eureka: A framework for enabling static malware analysis. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS)*. Springer Berlin, Heidelberg, 481–500.

[80] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions Artem. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*. ACM Press, 51.

[81] Markus Kammerstetter, Christian Platzer, and Gilbert Wondracek. 2012. Vanity, Cracks and Malware Insights into the Anti-Copy Protection Ecosystem. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS)*. ACM Press, 809.

[82] Zynamics. zynamics.com - BinDiff. https://www.zynamics.com/bindiff.html (accessed on 2022-12-09).

[83] Fortinet. Packers: What's in the Box? https://www.fortinet.com/blog/threat-research/custom-packer-tool-frenchy (accessed on 2022-12-09).

[84] Tuts 4 You. Tuts 4 You. https://tuts4you.com/ (accessed on 2022-12-09).

[85] Mi-Jung Choi, Jiwon Bang, Jongwook Kim, Hajin Kim, and Yang-Sae Moon. 2019. All-in-One Framework for Detection, Unpacking, and Verification for Malware Analysis. *Security and Communication Networks* 2019 (2019), 1–16.

Table 5: Signature-based packer detection is widely used in the existing research work.

Usage	Directly		Indirectly
	Signature-based detectors (e.g., PEiD)	Labels from anti-virus engines (e.g., VirusTotal)	Unpacker
Preprocessing Dataset	[38], [54], [65], [66], [67], [68]	-	[25], [26]
Creating Ground-Truth Dataset	[69], [70], [38], [71], [4], [72], [73]	[72], [69], [74], [75], [19], [76]	-
Classifying Packers	[6], [4], [77], [78], [79], [80]	[81], [5]	-
Comparison	[17]	-	-

A THE USAGE OF SIGNATURE-BASED PACKER DETECTION IN RESEARCH PAPERS

We systematically surveyed the papers published in 12 major cyber security conference venues (IEEE Security & Privacy, USENIX Security, ACM CCS, NDSS, USENIX ATC, ACSAC, ESORICS, ASIACCS, DSN, RAID, DIMVA, and CODASPY) over the past 16 years. As shown in Table 5, we find out that 24 papers are directly adopting signature-based detection in the experiments. Most works use signature-based detectors such as PEiD. There are two works [6, 72] also combine multiple detectors to identify packed programs. For example, Aghakhani et al. [6] adopt a hybrid packer detection approach, which combines the detection results from multiple antivirus engines, deep packer inspector [4], and signature-based detectors.

From Table 5 we can find that most works adopt signature-based detection in the process of preprocessing the dataset or constructing the ground-truth dataset. There are eight papers that apply signature-based detectors to classify packers. For example, Ugarte-Pedrero et al.'s SoK paper [4] adopts three detectors (i.e., PEiD, Sigbuster, and F-Prot) to classify the off-the-shelf packers and custom packers. However, the accuracy of the existing human-written rules is unsatisfactory. Inappropriate rules will greatly influence the correctness of the conclusion.

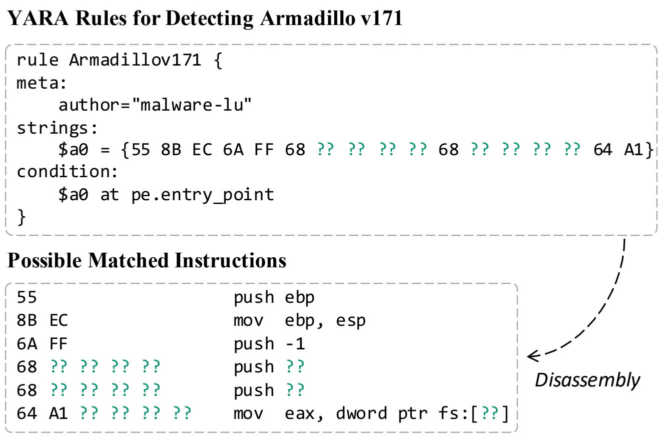
Furthermore, some tools (e.g., unpacker) may also influence the experiment results due to indirectly relying on YARA rules. Two recent papers [25, 26] indirectly use the signature-based detectors: they adopt unipacker [27] to unpack the samples of their datasets, while unipacker detects packed programs based on YARA rules.

B ERROR-PRONE HUMAN-WRITTEN RULES

The reasons for error-prone human-written rules can be divided into the following three categories.

Insignificant Features. If the rules are created from insignificant features, they will suffer from a high false positive rate. An example of Armadillo v171 detection rule adopted by PEiD is shown in Fig. 11. This rule is developed to match the entry point instructions of Armadillo-packed programs. However, the expected matched instructions are also used by compiler-generated programs. For example, by applying this rule `Armadillov171` in our *NPD1* dataset, we find that 56 non-packed samples are mistakenly identified as packed by Armadillo.

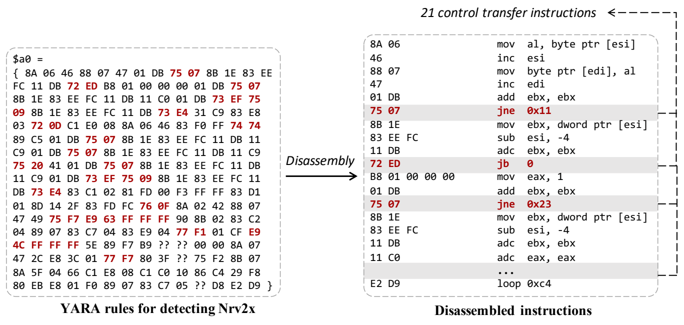
Rules with Long Length. To reduce the false positive rate of rules, the analysts prefer to develop the rules with a long length. But it usually causes the rules susceptible to control flow obfuscations. An example of Nrv2x detection rules adopted in x64dbg YARA database[50] is shown in Fig. 12. The Nrv2x is a decompression algorithm widely adopted by packers such as UPX. From the disassembled results shown in Fig. 12 we can find that the expected matched instructions contain 21 jump instructions. If the packer split the basic blocks of the Nrv2x algorithm, this rule can be easily evaded.


YARA Rules for Detecting Armadillo v171
Possible Matched Instructions
Disassembly

```
rule Armadillov171 {
meta:
    author="malware-lu"
strings:
    $a0 = {55 8B EC 6A FF 68 ?? ?? ?? ?? 68 ?? ?? ?? ?? 64 A1}
condition:
    $a0 at pe.entry_point
}
```

```
55                  push ebp
8B EC               mov  ebp, esp
6A FF               push -1
68 ?? ?? ?? ??      push ??
68 ?? ?? ?? ??      push ??
64 A1 ?? ?? ?? ??   mov  eax, dword ptr fs:[??]
```

Figure 11: Human-written rules to detect Armadillo packers.


21 control transfer instructions
Disassembly
YARA rules for detecting Nrv2x
Disassembled instructions

```
$a0 =
{ 8A 06 46 88 07 47 01 DB 75 07 8B 1E 83 EE
FC 11 DB 72 ED B8 01 00 00 00 01 DB 75 07
8B 1E 83 EE FC 11 DB 11 C0 01 DB 73 EF 75
09 8B 1E 83 EE FC 11 DB 73 E4 31 C9 83 E8
03 72 0D C1 E0 08 8A 06 46 83 F0 FF 74 74
89 C5 01 DB 75 07 8B 1E 83 EE FC 11 DB 11
C9 01 DB 75 07 8B 1E 83 EE FC 11 DB 11 C9
75 20 41 01 DB 75 07 8B 1E 83 EE FC 11 DB
11 C9 01 DB 73 EF 75 09 8B 1E 83 EE FC 11
DB 73 E4 83 C1 02 81 FD 00 F3 FF FF 83 D1
01 8D 14 2F 83 FD FC 76 0F 8A 02 42 88 07
47 49 75 F7 E9 63 FF FF FF 90 8B 02 83 C2
04 89 07 83 C7 04 83 E9 04 77 F1 01 CF E9
4C FF FF FF 5E 89 F7 B9 ?? ?? 00 00 8A 07
47 2C E8 3C 01 77 F7 80 3F ?? 75 F2 8B 07
8A 5F 04 66 C1 E8 08 C1 C0 10 86 C4 29 F8
80 EB E8 01 F0 89 07 83 C7 05 ?? D8 E2 D9 }
```

```
8A 06            mov  al, byte ptr [esi]
46               inc  esi
88 07            mov  byte ptr [edi], al
47               inc  edi
01 DB            add  ebx, ebx
75 07            jne  0x11
8B 1E            mov  ebx, dword ptr [esi]
83 EE FC         sub  esi, -4
11 DB            adc  ebx, ebx
72 ED            jb   0
B8 01 00 00 00   mov  eax, 1
01 DB            add  ebx, ebx
75 07            jne  0x23
8B 1E            mov  ebx, dword ptr [esi]
83 EE FC         sub  esi, -4
11 DB            adc  ebx, ebx
11 C0            adc  eax, eax
                 ...
E2 D9            loop 0xc4
```

Figure 12: An illustration of a long YARA rule for detecting Nrv2x decompression algorithm.

C CASE STUDY I: A CUSTOMIZED UPX SAMPLE FROM APT 29

This section discusses the evasion techniques of customized packers to bypass existing human-written rules. We use an APT 29 packed sample (SHA256: 55ba0c04d488903e07f0747407ed56319f0d9aac113c7f9c62287442f1f78c45) from our detected custom UPX-packed samples. This sample uses normal section names such as "`.text`" instead of the suspicious section name "`.UPX`". It also hides any suspicious strings of packers to bypass the rules created from textual features. By manually inspecting the samples, we observe that this sample hides the packed data and the unpacking routine in the "`.rdata`" section. Different from the standard UPX-packed programs, the entry point instructions of this sample are the trampoline code used to jump to the real UPX unpacking routine in the "`.rdata`" section. It can easily bypass the rules created from the entry point instructions. However, this sample uses the same unpacking routine as in standard UPX without any obfuscations.

At the runtime, this sample will place unpacked instructions in the "`.data`" section, which is an empty section ranging from `0x401000` to `0x418000`. Furthermore, according to the DIE's report, the entropy of this sample is less than 1. This sample uses bytes padding technique to reduce the entropy of "`.text`" section to 0.03. Only the entropy of the "`.rdata`" section is greater than 7. This sample can bypass the anti-virus engines that measure the entropy of executable files.

D CASE STUDY II: AN EXAMPLE OF LOW-ENTROPY SAMPLE PACKED BY NSPACK

From the low entropy dataset provided by [17], we find two low entropy packed samples are protected by NsPack. We use the sample (SHA256:aac73c9374127b9045724ba9a843314e2dc5a1edd25e11d1ae450cba13d3e30b) as the example. Our generated rules identify that this sample is packed with NsPack. However, the entropy of this sample is 3.9, which is lower than the entropy of packed programs (usually higher than 7). Compared to the programs packed by standard NsPack 2.4, we observe that the low entropy sample adopts the same unpacking routines as the standard NsPack packer. The comparison result based on BinDiff [82] is shown in Fig. 13. We can find that nearly 99% of recognized functions are similar. The only difference is introduced by two basic blocks in the "`start`" (i.e., the entry point of packed programs) function which is the first blue colored lines in Fig. 13. After manually examining the sample, we find that this sample adopts a large null segment to decrease the entropy.

E CASE STUDY III: A CAMOUFLAGED CUSTOM PACKERS

This section discusses the evasion technique of customized packers to mislead existing standard packer detection rules. We present a packed sample (SHA256: 001350bb9e5cacf470bc075a7433252d59c1e3516c7804959f9688a790a9abdf6) camouflaged as packed by standard UPX from our *WD1* dataset. The detection results of VirusTotal and DIE reveal that this sample is packed by UPX 3.10. We find out that the detection rules of DIE resolve the meta-information of the sample as the standard UPX. However, as shown on the official website of UPX [4], the UPX 3.10 version is nonexistent. This sample also cannot be unpacked by the "`upx -d`" command. After manually inspecting the samples, we observe that this sample uses a custom decoding algorithm at the first layer unpacking routines. The comparison result based on BinDiff [82] is shown in Fig. 14. We can discover that the similarity score between camouflaged samples and the standard UPX-packed program is only 18%. Only at the runtime, this sample will unpack and execute the real UPX unpacking routines. We discover 3,425 similar samples from our *WD1* dataset. These camouflaged samples will easily mislead the existing standard packer detection rules and analysts. If replacing the unpacked standard unpacking routines with custom ones, these custom packers can effectively impede the analysts.

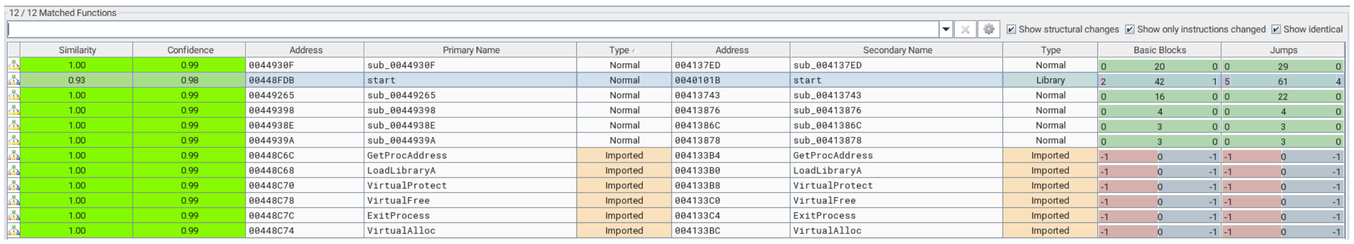
F INACCESSIBLE PACKERS

Malware authors may use custom packers to evade detection; they also adopt old packers that are no longer available in the market. For example, malware collected in recent years still uses many old packers (e.g., Diminisher) designed over 20 years ago [6]. In this scenario, we cannot synthesize packed programs from inaccessible packers. Fortunately, close malware variants within the same family are very likely to reuse the same packers [83]. We can manually collect the packed programs that are potentially generated by the same packer. Although we cannot traverse all configurations of inaccessible packers, we can still find similar unpacking routines when given sufficient same-packer protected samples.

When generating rules from the inaccessible packer protected programs, the workflow of PackGenome is similar to the original process shown in Fig. 4. To circumvent potential anti-instrumentation techniques used in packed programs, we integrate our Pintool with the anti-evasion framework ARANCINO [8]. We first preprocess packed programs, record the "written-then-executed" instructions at the first unpacking layer, and discover unpacking routines. Then, we extract packer-specific genes by collecting similar instructions from reused unpacking routines. During the rule generation process, PackGenome will evaluate the accuracy of generated rules. If the generated rules reveal high false negatives, we need to collect additional packed programs as input samples and reproduce rules. The reason is that the manually collected packed programs may contain completely different unpacking routines, which will make PackGenome cannot find similar instructions.

To evaluate the performance of PackGenome on generating rules for detecting inaccessible packer protected programs, we collect 2,392 programs from Tuts4you [84] and peer work [85] (*LPD2*). We randomly select 15 packed programs protected by five inaccessible packers, and generate eight rules in total. Then, we apply generated rules, human-written rules, AutoYara, and DIE to the dataset *LPD2*. As shown in Table 6, our rules have a small false negative rate.

[4]https://github.com/upx/upx/blob/devel/NEWS

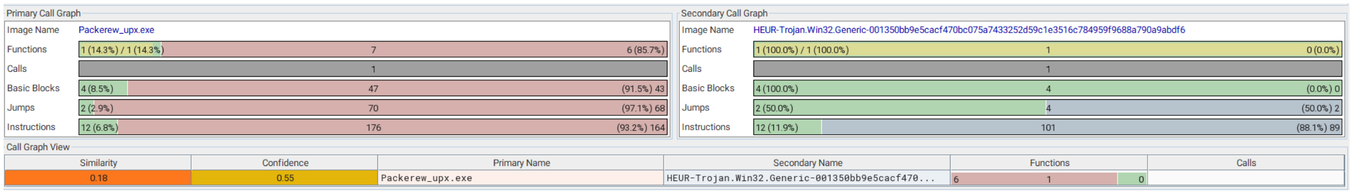


12 / 12 Matched Functions

☑ Show structural changes ☑ Show only instructions changed ☑ Show identical

Similarity	Confidence	Address	Primary Name	Type	Address	Secondary Name	Type	Basic Blocks	Jumps
1.00	0.99	0044930F	sub_0044930F	Normal	004137ED	sub_004137ED	Normal	0 20 0	0 29 0
0.93	0.98	00448FDB	start	Normal	0040101B	start	Library	2 42 1	5 61 4
1.00	0.99	00449265	sub_00449265	Normal	00413743	sub_00413743	Normal	0 16 0	0 22 0
1.00	0.99	00449398	sub_00449398	Normal	00413876	sub_00413876	Normal	0 4 0	0 4 0
1.00	0.99	0044938E	sub_0044938E	Normal	0041386C	sub_0041386C	Normal	0 3 0	0 3 0
1.00	0.99	0044939A	sub_0044939A	Normal	00413878	sub_00413878	Normal	0 3 0	0 3 0
1.00	0.99	00448C6C	GetProcAddress	Imported	004133B4	GetProcAddress	Imported	-1 0 -1	-1 0 -1
1.00	0.99	00448C68	LoadLibraryA	Imported	004133B0	LoadLibraryA	Imported	-1 0 -1	-1 0 -1
1.00	0.99	00448C70	VirtualProtect	Imported	004133B8	VirtualProtect	Imported	-1 0 -1	-1 0 -1
1.00	0.99	00448C78	VirtualFree	Imported	004133C0	VirtualFree	Imported	-1 0 -1	-1 0 -1
1.00	0.99	00448C7C	ExitProcess	Imported	004133C4	ExitProcess	Imported	-1 0 -1	-1 0 -1
1.00	0.99	00448C74	VirtualAlloc	Imported	004133BC	VirtualAlloc	Imported	-1 0 -1	-1 0 -1

Figure 13: BinDiff comparison between a low entropy sample and a program packed by standard NsPack 2.4.



Primary Call Graph

Image Name	Packerew_upx.exe		
Functions	1 (14.3%) / 1 (14.3%)	7	6 (85.7%)
Calls		1	
Basic Blocks	4 (8.5%)	47	(91.5%) 43
Jumps	2 (2.9%)	70	(97.1%) 68
Instructions	12 (6.8%)	176	(93.2%) 164

Secondary Call Graph

Image Name	HEUR-Trojan.Win32.Generic-001350bb9e5cacf470bc075a7433252d59c1e3516c784959f9688a790a9abdf6		
Functions	1 (100.0%) / 1 (100.0%)	1	0 (0.0%)
Calls		1	
Basic Blocks	4 (100.0%)	4	(0.0%) 0
Jumps	2 (50.0%)	4	(50.0%) 2
Instructions	12 (11.9%)	101	(88.1%) 89

Call Graph View

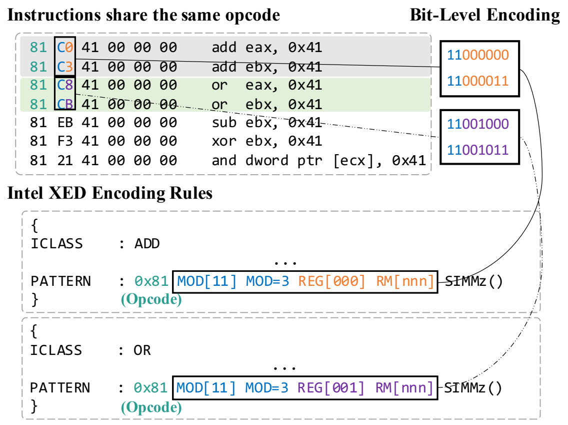
Similarity	Confidence	Primary Name	Secondary Name	Functions	Calls
0.18	0.55	Packerew_upx.exe	HEUR-Trojan.Win32.Generic-001350bb9e5cacf470...	6 1 0	

Figure 14: BinDiff comparison between a Standard UPX-packed sample and a program camouflaged as packed by standard UPX.

Table 6: The experiment results of applying our generated rules, human-written rules, AutoYara, and DIE to dataset *LPD2*. "Obfs" reports the obfuscation adopted by the unpacking routines. "GR" reports the number of the generated rules.

Packers	# of Vers	Configurations		Obfs [1]	Our Approach				Human-Written Rules			AutoYara[21]			Detect It Easy [16]		
		Relate	Total		GR	FPR [%]	FNR [%]	Time [s]	FPR [%]	FNR [%]	Time [s]	FPR [%]	FNR [%]	Time [s]	FPR [%]	FNR [%]	Time [s]
NoeLite	1	-	-	N	5	0	0	0.1	1.77	2.65	1	30.1	62.8	0.1	0	1.85	23
BeRoEXEPacker	1	-	-	N	3	0	0	0.2	100	0	1.1	0	60.7	0.2	0	6.03	21
exe32pack	1	-	-	N	1	0	0	0.1	0	0	0.7	0	50.4	0.1	0	0	17
JDPack	2	-	-	N	1	0	0	0.1	98.4	98.4	0.8	30.7	72.4	0.1	0	0	19
packman	1	-	-	N	1	0	0	0.2	100	0	1.3	0	54.2	0.2	0	0	20

[1] "N" means not obfuscated.



```
81 C0 41 00 00 00    add eax, 0x41
81 C3 41 00 00 00    add ebx, 0x41
81 C8 41 00 00 00    or  eax, 0x41
81 CB 41 00 00 00    or  ebx, 0x41
81 EB 41 00 00 00    sub ebx, 0x41
81 F3 41 00 00 00    xor ebx, 0x41
81 21 41 00 00 00    and dword ptr [ecx], 0x41
```

```
11000000
11000011

11001000
11001011
```

```
{
ICLASS    : ADD
                      ...
PATTERN   : 0x81 MOD[11] MOD=3 REG[000] RM[nnn] SIMMz()
}
```

```
{
ICLASS    : OR
                      ...
PATTERN   : 0x81 MOD[11] MOD=3 REG[001] RM[nnn] SIMMz()
}
```

Figure 15: An illustration of instructions with different functions but sharing the same opcode.

Algorithm 1: Predicting Disassembly

```
Input: HSR Hexadecimal String Rules
Result: INL List of Misplace Matched Instructions
1  Function B2Opcode(HSR) := Converting HSR to the same
     byte format as the opcode of XED rules.
2  Function B2Operand(HSR) := Transforming HSR to operand
     based on the grammar of XED rules.
3  Function InsGen(rule) := Generating instructions from the
     Intel XED rules.
4  $OPC[opcode]$ := The XED rules that satisfy opcode.
5  $OPE[operand]$ := The XED rules that satisfy operand.
6  INL ← {}
7  if the length of HSR > 15 then
8      return False
9  end
   /* Type I Predicting from operand */
10 if B2Operand(HSR) ≠ {} then
11     for each code in B2Operand(HSR) do
12         INL ← INL ∪ InsGen(OPE[code])
13     end
14 end
   /* Type II Predicting from both opcode and operand */
15 for each index i from the start of len to end do
16     sopcode := B2Opcode(HSR[:i])
17     if sopcode ≠ {} then
18         soperand := B2Operand(HSR[i:])
19         if soperand ≠ {} then
20             INL ← INL ∪ InsGen(OPC[sopcode],
                 OPE[soperand])
21         end
22     end
23 end
24 return INL
```