Eureka: A Framework for Enabling Static Analysis on Malware

Technical Report Number:  SRI-CSL-08-01
SRI Project 17382

Prepared by:
Vinod Yegneswaran, Hassen Saidi, and Phillip Porras
Computer Science Laboratory
and
Monirul Sharif
College of Computing, Georgia Institute of Technology

Approved by:
Patrick Lincoln, Director
Computer Science Laboratory

William S. Mark, Vice President
Information and Computing Sciences Division
Eureka: A Framework for Enabling Static Analysis on Malware

Technical Report SRI-CSL-08-XX

Monirul Sharif (msharif@cc.gatech.edu), Vinod Yegneswaran, Hassen Saidi, and Phillip Porras
firstname.lastname@sri.com

Computer Science Laboratory
SRI International

and the

College of Computing
Georgia Institute of Technology

12 April 2008

We introduce Eureka, a framework for enabling static analysis on Internet malware binaries. Eureka incorporates a novel binary unpacking strategy based on statistical bigram analysis and coarse-grained execution tracing. The Eureka framework uniquely distinguishes itself from prior work by providing effective evaluation metrics and techniques to assess the quality of the produced unpacked code. Eureka provides several Windows API resolution techniques that identify system calls in the unpacked code by overcoming various existing control flow obfuscations. Eureka’s unpacking and API resolution capabilities facilitate the structural analysis of the underlying malware logic by means of micro-ontology generation that labels groupings of identified API calls based on their functionality. They enable a visual means for understanding malware code through the automated construction of annotated control flow and call graphs. We demonstrate Eureka’s capabilities using the Storm worm as a case study to reveal critical information such as spam capabilities, propagation methods, and command and control (C&C) protocols. Our evaluation on multiple datasets reveals that Eureka can simplify analysis on a large fraction of contemporary Internet malware by successfully unpacking and deobfuscating API references.
## Contents

1 Introduction .................................................. 2

2 Related Work .................................................. 5

3 Automated Unpacking with Informed Coarse-grained Execution Tracking ........................................... 6
   3.1 Heuristics-based unpacking .................................. 7
   3.2 Statistics-based unpacking .................................. 8

4 API Resolution Techniques .................................. 10
   4.1 Background: standard API resolution ....................... 11
   4.2 Resolving obfuscated APIs without the import tables and IAT ......................................................... 12
      4.2.1 Handling DLL obfuscations ......................... 13
      4.2.2 API resolution for statically identifiable targets ......................................................... 13
      4.2.3 API resolution for dynamically computed addresses ......................................................... 14
      4.2.4 More sophisticated cases ................................ 15

5 Evaluation Metrics and Graph Generation ............... 15
   5.1 Measuring analyzability ..................................... 16
   5.2 Malware call graph generation ............................ 17
   5.3 Micro-ontology labeling .................................... 18

6 Case Study: Analysis of the Storm Worm ................. 18
   6.1 Storm Logic’s Overview .................................... 20
   6.2 Overnet/eDonkey Communication Logic .................. 20

7 Experimental Results ......................................... 20
   7.1 Benign dataset evaluation: Goat test .................... 20
   7.2 Malicious data set evaluation ............................ 22
   7.3 Eureka: runtime performance results ..................... 23

8 Limitations and Future Work ................................. 24

9 Conclusion ....................................................... 25

References ......................................................... 26

SRI International is a nonprofit organization.
1 Introduction

Consider the challenges that arise in assessing the threat posed from a new malware binary strain that appears on the Internet or is discovered in a highly sensitive computing environment. Let us set aside the challenges of discovering the binary, whether it is caught in a honeynet [32], through the triggering of a signature [23], through some form of behavioral-based pattern detector [15, 36], or simply through human vigilance. Once it is discovered, there is a significant need to understand its capabilities, its infiltration methods, its impact on the local system, and its purpose for existence. Now multiply this challenge by the hundreds of new strains and repurposed malware variants that appear on the Internet yearly [21, 28, 3], and the need to develop automated tools to extract and analyze all facets of malware binary logic becomes clear.

Unfortunately, malicious software developers are also well aware of the efforts to reverse engineer their binaries, and employ a wide range of code and binary obfuscation techniques to deter analysis and reverse engineering. Some researchers are even exploring the degree to which advanced binary obfuscations can be created to thwart analysis [19, 37], further illustrating the growing depth of challenges that face the malware defense community.

Nevertheless, whether drawn by the deep need or the challenges, substantial efforts have been made in recent years to develop automated malware binary analysis systems. In particular, two primary approaches have dominated these efforts. Dynamic analyses refer to techniques to profile the actions of the malware binary at runtime [11, 5]. Static analyses refer to techniques to decompile and analyze the logical structure, flow, and data content stored within the binary itself. While both analysis techniques yield important (and sometimes complementary) insight into the capabilities and purpose of a malware binary, these techniques also have their unique advantages and disadvantages.

To date, dynamic analyses have arguably offered a better record and mind share among those working on malware binary analysis. Part of that success is attributable to the challenges of overcoming the formidable binary obfuscation techniques (or binary packing systems) that are widely utilized by contemporary malware developers. These obfuscation techniques are multi-layered, and include a gamut of protections, such as binary code and data segment encryption, function and API call obfuscation, and control flow obfuscations. While defeating these obfuscations is a requisite step to conducting meaningful static analyses, they can largely be ignored by those conducting dynamic analyses. However, dynamic analysis provides only a partial “effects-oriented” profile of the full potential of a given malware binary. Dynamic analyses cannot reveal the effects of programming logic that fails to execute during the runtime analysis. For example, the subject malware binary may include unsatisfied trigger conditions (e.g., logic revealed only when certain environmental or temporal conditions are satisfied), or suicide logic that can be triggered when process tracing is detected or when other self-protection conditions are met.

Static program analysis can provide complementary insights to dynamic analyses in those occasions where binary obfuscations can be sufficiently overcome. Static program analysis offers the potential for a more comprehensive assessment of the entire code and data of the program. For example, by analyzing the sequence of invoked system calls and APIs, performing control flow analysis, and tracking data segment references, it is possible to infer logical code bombs, temporal triggers, and other malicious system interactions. Features such as the presence of network communication logic, registry and OS manipulations, and object creations (e.g., files, processes, interprocess communication...
can be detected, whether or not these capabilities are exercised at runtime. Static analysis, when presented with a deobfuscated binary can complement and even inform dynamic program analyses with a more comprehensive picture of the program logic.

The Eureka Framework

In this paper, we introduce a malware binary deobfuscation framework referred to as Eureka, designed to maximally facilitate static code analysis. To date, there has been substantial work in automated unpacking techniques [10, 31, 22]. Eureka, too, presents yet another unique strategy for automatically unpacking packed and obfuscated malware binary programs, including an innovative statistical methodology for dynamically evaluating whether a runtime process has reached a stable unpacked state before its process image is dumped for disassembly. Eureka also incorporates a service that attempts to automatically overcome several sophisticated code-level obfuscation strategies to hide API invocation. (API resolution is a critical step in statically analyzing code semantics.) Eureka further distinguishes itself by being the first unpacking system to automatically evaluate whether the resulting binary image dumped is likely to yield a statically analyzable code and data image. We implement a service that collects and reports analyzability metrics that it produces from the dumped disassembled process image. These metrics provide meaningful insight into vetting whether the disassembled image will yield meaningful flow and control analyses before an analyst needs to invest effort in statically analyzing the dumped image. Finally, we present our code graph and ontology generation services that augment the disassembled image with labels, graphical structures, and structural simplifications that make the image more comprehensible to analysts and future potential post-processing systems.

Figure 1: The Eureka Malware Binary Deobfuscation Framework

Figure 1 presents an overview of the modules and logical work flow that compose the Eureka framework. The figure also identifies which section of this paper discusses the details of each.
module. The Eureka workflow begins with the subject-packed malware binary, which is executed in a virtual machine (VM) managed by Eureka. The malware process image will often interrogate its local environment in search of evidence that it is being traced, debugged, or stepped through in virtual time, which may lead the process to self-terminate. In practice, we experience self-termination in less than 5% of the malware binaries we examine through Eureka. Otherwise, the malware binary will typically enter a phase of unpacking and the eventual spawning of its core malware payload logic. As this self-unpacking stage is occurring, a parallel Eureka kernel driver is tracking the execution of the malware binary, periodically evaluating the process for signs that it has unpacked its image. In Section 3, we present Eureka’s coarse-grained execution tracking algorithm and introduce a novel binary n-gram statistical trigger for evaluating when the unpacked process image has reached a stable state.

Once the execution tracker triggers a process image dump, Eureka employs the IDA-Pro disassembler [1] to disassemble the image, and then proceeds to conduct API resolution and prepare the code image for static analysis. In Section 4, we discuss Eureka’s API map recovery module, which provides several automated deobfuscation procedures to recover hidden API invocations that are commonly used to thwart static analysis. Once API resolution is completed, the code image is processed by Eureka’s analyzability metrics generation module. This module, presented in Section 5, produces a summary report of several critical attributes that may indicate whether the unpacked image contains coherent code and data segments, providing several useful indicators as to whether a static analysis of this unpacked image will yield useful results. We also describe the Eureka code and graph generation modules that produce call graphs, micro-ontology labels, and structural graph refinements that can usefully augment the static code analysis, and show how it allowed us to perform an in-depth analysis of the Storm worm [29]. Following the presentation of the Eureka framework, we further present a corpus evaluation (Section 7) to illustrate the usage and effectiveness of Eureka.

Research Contributions

Among the important themes that we hope to convey in this report is that we do not view the production of a process image dump as the primary end result of unpacking systems. Binary unpacking is one step in the greater investment of deciding to statically analyze a malware binary. Eureka is presented as a framework of modular steps that are applied to packed malware binaries to extract a process image dump, deobfuscate the API call structure, derive simple (yet useful) metrics that can indicate image analyzability, and generate meta-data that can substantially augment static analysis. We believe that the Eureka framework offers the following contributions:

- It introduces a novel unpacking technique based on coarse-grained execution tracing. Our evaluation shows that it successfully unpacks 95% of daily occurring real-world packed malware binaries.
- It develops methods for improved API resolution under various API and control flow obfuscations. To our knowledge, we are the first to handle API obfuscations after unpacking.
- It computes a set of basic metrics for assessing the quality of the unpacking process and a set of heuristics to improve the quality of the unpacked code base.
- It explores the use of structural analysis as means to evaluate analyzability of unpacked code.

©2005 SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025. SRI International is a nonprofit organization.
It employs a micro-ontology for automatically annotating the API calls of the function call graph. This labeling helps to illustrate the structural inter-relationship of code features and associate data sections during static analysis.

2 Related Work

The problem of obfuscated malware has confounded analysts for decades [35]. The first obfuscation techniques exhibited by malware in the wild include viral metamorphism [35] and polymorphism [33]. Several obfuscation approaches have since been presented in the literature [9], including opaque predicates [10] and recently opaque constants [25]. Packers and executable protectors [30] are often used to automatically add several layers of protection to malware executables. Recent packers and protectors also incorporate API obfuscations that make it hard for analyzers to identify system calls or calls to Windows APIs.

<table>
<thead>
<tr>
<th>System</th>
<th>Monitor Environ</th>
<th>Monitor Granularity</th>
<th>Trigger Types</th>
<th>Child Process Monitoring</th>
<th>Output Layers</th>
<th>Speed</th>
<th>Potential Evasions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyUnpack</td>
<td>Inside VM</td>
<td>Instruction</td>
<td>Model-based</td>
<td>No</td>
<td>1</td>
<td>Slow</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Renovo</td>
<td>Emulator</td>
<td>Instruction</td>
<td>Heuristic</td>
<td>Yes</td>
<td>many</td>
<td>Slow</td>
<td>2,4</td>
</tr>
<tr>
<td>OmniUnpack</td>
<td>Inside VM</td>
<td>Page</td>
<td>Heuristic</td>
<td>No</td>
<td>many</td>
<td>Fast</td>
<td>2,3</td>
</tr>
<tr>
<td>Eureka</td>
<td>Inside VM</td>
<td>System Call</td>
<td>Heuristic, Statistic-based</td>
<td>Yes</td>
<td>1,many</td>
<td>Fast</td>
<td>2,3</td>
</tr>
</tbody>
</table>

Table 1: Design space of unpackers. Evasions: (1) multiple packing, (2) partial code revealing multi-layered packing, (3) VM detection, (4) emulator detection

Automated unpacking: Table 1 summarizes the design space of automated unpackers that illustrates their strengths, differences, and common weaknesses. There have been several recent attempts at building automated and generic tools for unpacking malware, most notably PolyUnpack [31], Renovo [16], and OmniUnpack [22]. PolyUnpack first performs program analysis to build a static model of the program and uses fine-grained execution tracking to detect when an instruction is executed that is outside of the static model. PolyUnpack uses the Windows debugging API to single-step through the process execution. Like PolyUnpack, Renovo uses a fine-grained execution monitoring approach to track unpacking progress and considers the execution of newly written code as an indicator of unpack completion. Renovo is implemented using the QEMU emulator, which resides outside the execution environment of the malware and supports multiple layers of unpacking. OmniUnpack is most similar to Eureka in that it uses a coarse-grained execution tracking approach. However, their granularities are orthogonal: OmniUnpack tracks execution at the page level while Eureka tracks execution at the system call level. OmniUnpack uses page-level protection mechanisms available in hardware to identify when code is executed from a page that was newly modified.

Static malware analysis: Previous work in malware analysis that uses static analysis has primarily focused on malware detection approaches. Known malicious patterns are identified in [12]. The approach of using semantic behavior to thwart some specific code obfuscations was presented in [13]. Rootkit behavior detection was presented in [18], and [17] uses a static analysis approach to identify spyware behavior in Browser Helper Objects. Traditional program analysis techniques
have been investigated for binary programs in general and malware in particular. Dataflow
techniques such as Value Set Analysis aim at recovering the set of possible values that each data
object can hold at each program point.

**Dynamic malware analysis:** Automated malware analysis methods are becoming increasingly
critical to keep up with the hundreds of new malware strains that appear each month. Since static
analysis approaches are prone to several code obfuscation techniques such as packing, research efforts
have been biased toward dynamic analysis. CWSandbox and TTAnalyze execute programs
in a restricted environment and observe the sequence of system interactions (using system calls).
Pararoma uses system-wide taint propagation to analyze information flow, which it uses for
detecting malware. The single-path exploration restriction of pure dynamic analysis approaches
has been overcome in, which provides methods for exploring multiple paths during execution.
The approach saves execution state at conditional branches and tries each path in turn by solving
costs to consistently update memory variables that lead to the other branch. Bitscope incorporates symbolic execution-based static analysis to analyze malicious behavior.

**Statistical analysis:** Fileprint analysis studies statistical binary content analysis as a means
to identify malicious content embedded in files, finding that n-gram analysis is a useful means
to detect anomalous file segments. A further finding is that normal system files and malware
can be well classified using 1-gram and 2-gram analysis. While our methodology is similar, the
problem differs in that we use bi-grams to model unpacked code and it is independent of the code
being malicious. N-gram analysis has also been used in other contexts, including anomalous packet
detection in network intrusion detection systems such as PAYL and Anagram.

3 Automated Unpacking with Informed Coarse-grained
Execution Tracking

In general, all the current methods for binary unpacking start with some sort of dynamic analysis.
Unpacking systems begin their processing by executing the malware binary, allowing it to self-
decrypt its malicious payload logic and to then fork control to this newly revealed program logic.
One primary method by which unpacking systems distinguish themselves is in the approach each
takes to monitor the progression of the packed binaries’ self-decryption process. When the unpacker
determines that the process has sufficiently revealed the malicious payload logic, it will then dump
the malicious process image for use in static analysis.

Much of the variability in unpacking strategies comes from the granularity of monitoring that is
used to track the self-decryption progress of the packed binary. Some techniques rely on tracking
the progress of the packed process on a per-individual instruction basis. We refer to this instruction-
level monitoring as fine-grained monitoring. Other strategies use more coarse-grained monitoring,
such as OmniUnpack, which checkpoints the self-decryption progress of the malicious binary via
intercepts from the page-level protection mechanisms.

Eureka, like OmniUnpack, tracks the execution progress of the packed binary image via coarse-
grained check pointing. However, rather than using page interrupts, Eureka tracks the malicious
process via the system call interface. Eureka’s coarse-grained execution tracker operates as a kernel
driver that dumps the malicious process into a binary image for disassembly when it believes that the malicious payload logic has been sufficiently revealed.

In the following sections we present our methods for deciding when to dump the malicious process image. First, in Section 3.1 we present a direct heuristics-based approach, which we later show works well on a large fraction of our experimental malware corpus. Indeed, this method works as accurately as many other contemporary unpacking methods at a fraction of the performance cost (Section 7.3). Second, we introduce a radically different approach to evaluating the completeness of the malicious process’s self-decryption via a periodic statistical n-gram analysis of the process image (Section 3.2).

3.1 Heuristics-based unpacking

Eureka’s principal method of unpacking is to follow the execution of the malware program by tracking its progress at the system call level. Among the advantages of this approach, the progression of the self-decrypting process image can be tracked with very little overhead. Each system call indicates that a particular interesting event is occurring in the executing malware. Eureka employs a Windows-driver-based unpacker that hooks the Windows SSDT (System Service Dispatch Table). The driver executes a callback routine when a system call is invoked from a user-level program. We use a filtering approach based on the process ID (PID) of the process invoking the system call. A user-level program initiates the execution of the malware and informs the Eureka driver of the malware’s PID.

The heuristics-based unpacking approach of Eureka exploits a simple strategy in which it uses the event of program exit as triggering the snapshot of the malware’s virtual memory address space. That is, the system call NtTerminateProcess is used to trigger the dumping of the malware process image, under the assumption that the use of this API implies that the unpacked malicious payload has been successfully decrypted, spawned, and is now ending.

We observe that the heuristic method works well for incremental unpackers, i.e., those that decrypt their process’s concealed segments on demand, but never erase already-decrypted segments. Thus, the later the memory snapshot is taken, the more unpacked code is revealed. Since our goal is to produce unpacked code that is amenable to static analysis, we can relax the requirement of identifying the point of execution when control flows from an already-revealed code region to a newly unpacked code region, because entry points into that code can be identified later by following control flow from the original entry point of the packed code. This relaxation allows Eureka to use coarse-grained execution that has the advantage of negligible time delays and at the same time use efficient heuristics and statistical analysis to determine when to dump an unpacked image. Incremental packers only introduce new code as execution proceeds, making it favorable to take the memory dump as late as possible.

Another noticeable behavior we found in a large number of malware programs was that the malware spawns its own executable as another process. We believe this is a widely used technique that detaches from debuggers or system call tracers that trace only the initial malware process. Thus, Eureka also employs a simple heuristic that dumps the malware during the execution of the NtCreateProcess system call; we found that a large fraction of current malware programs were successfully unpacked.

SRI International is a nonprofit organization.
A problem with the above heuristic is that not all malware programs exit and keep an executing version resident in memory. There are several weaknesses in this simple heuristics-based approach. Although the above two heuristics may work for a large fraction of malware today, it may not be the same for future malware. With the knowledge of these heuristics, packers may incorporate the features of including process creation as part of the unpacking process. This would mean that unpacking may not have completed when the `NtCreateProcess` system call is intercepted. Also, malware authors can simply avoid exiting the malware process, avoiding the use of the `NtTerminateProcess` system call. Nevertheless, these very basic and very efficient heuristics demonstrate that very simple and straightforward mechanisms can be effective in unpacking a significant fraction of today's malware (as much as 80% of malware analyzed in our corpus experiments, Section 7). Where these heuristics fail, our statistical-based n-gram strategy provides a more than sufficient complement to unpack the remaining malware.

### 3.2 Statistics-based unpacking

As an alternative to its system-call heuristics, Eureka also tracks the statistical distribution of executable memory regions. In developing such an approach, we are motivated by the simple premise that unpacked executables have fundamentally different statistical properties that could be exploited to determine when a malware program has fully unpacked itself. A Windows PE (portable executable) is composed of several different types of regions. These include file headers and data directories, code sections (typically labeled as .text), and data sections (typically labeled as .data). Intuitively, as the malware unpacks itself, we expect that the code-to-data ratio would increase. So we expect that tracking the volume of code and data in the executable would provide us with a measure of the progress of unpacking. However, several potential complications could arise that must be considered:

- Code and data are often interleaved, especially in malicious executables.
- Data directory regions such as import tables that have statistically similar properties to the data section (i.e., ASCII data) are often embedded within code sections.
- Properties of data sections holding packed code might vary greatly based on packers and differ significantly from data sections in benign executables.

To address these issues, we develop an approach that models statistical properties of unpacked code. Our approach is based on two fundamental observations. First, code has certain intrinsic properties that tend to be invariant across executables (e.g., certain opcodes, registers, addresses and instruction sequences) and are more prevalent than others. These statistical properties may be used to measure relative changes in the volume of unpacked code. Second, we expect that the volume of unpacked code would be strictly increasing as packed malware executes and unravels itself. Surprisingly, we find that both our assertions hold for the vast majority of malware and across most packers.

**Mining statistical patterns in x86 code:** As a means to study typical and frequently occurring patterns in x86 code, we began by looking at a small collection of benign PE executables. A natural way to search for such patterns is to use a simple n-gram analysis. Specifically, we were interested in using n-gram analysis to build models of sections of these executables that contained x86 instructions. Our first approach was to simply extract entire sections from the PE header that
was labeled as code. However, we found that large portions of these sections also contained long sequences of ASCII data from non-x86 instructions, e.g., data directories or dynamically loaded library (DLL) names, which biased our analysis. To alleviate this bias, we used the IDA Pro disassembler, to extract regions from these executables that were marked as functions by looking for arguments to the MakeFunction calls in the IDA Pro script (IDC file). We then performed bigram analysis on this data. We chose bigrams because x86 opcodes tend to be either 1 byte or 2 bytes. By looking at frequently occurring bigrams we are looking at the most common opcode pairs or 2-byte opcodes. Once we developed a list of the most common bigrams for the benign executable, we used objdump output to evaluate whether bigrams occur in opcodes or operands (addresses, registers). Intuitively, one expects the former to be more reliable than the latter. We provide a summary in Table 2. Based on this analysis, we selected FF 15 (pushl) and FF 75 (call) as two candidate bigrams that are prevalent in x86 code. We also looked for spaced bigrams (byte pairs separated by 1 or more bytes). We found that the call instruction with 1 byte opcode (e8) has a relative offset. The last byte of this offset invariably ends up being 00 or FF depending on whether it has a positive or negative offset. Thus, high frequencies of e8 00 and e8 ff are also indicative of x86 code.

To evaluate the feasibility of this approach, we examined bigram distributions on a corpus of 1291 malware instances. We first unpacked each of these instances using our heuristic-based unpacker and then evaluated the quality of unpacking by evaluating the code-to-data ratio in an IDA Pro disassembly. We found that the heuristic-based unpacker did not produce a useful unpacking in 201 instances (small amount of code and low code-to-data ratio in the IDA disassembly). Out of the remaining 1090 binaries, we labeled 125 binaries as being originally unpacked (significant amount of code and high code-to-data ratio in both packed and unpacked disassemblies) and 965 as being successfully unpacked (significant amount of code and high code-to-data ratio only in the disassembly of the unpacked executable). Using counts of the aforementioned bigrams, we were able to produce output consistent with that of IDA disassembly evaluation. We correctly identified all 201 instances of still-packed binaries, all 125 instances of originally unpacked binaries, and 922 (out of 965) instances of the successfully unpacked binaries. In summary, this simple bigram counting approach had over a 95% success rate in distinguishing between packed and unpacked malware instances.

STOP – Statistical Test for Online unPacking: Inspired by the results from offline bigram counting experiments, Eureka incorporates STOP, an online algorithm for determining the terminating (or dumping) condition. We pose the problem as a simple hypothesis testing argument that checks for increase in mean value of bigram counts. Our null hypothesis is that the mean value of x86 instruction bigrams has not increased. We would like to conclude that the mean value has increased when we see a consistent and significant shift in the bigram counts. Let us assume that we have the prior mean (µ₀) for the candidate x86 instruction bigrams, and that we have a sample of N recent bigram counts. We assume that this sample is normally distributed with mean value (µ₁) and standard deviation (σ₁). We compute \( z_0 = \frac{\mu_1 - \mu_0}{\sigma_1} \). If \( z_0 > 1.645 \) then we reject the null hypothesis (with a confidence level of 0.95 for a normal distribution). We have integrated the STOP algorithm into our Eureka execution tracking module. STOP parameters include the ability to choose to compute the mean value of particular bigrams at each system call, every \( n \) system calls for a given value of \( n \), or only when certain anomalous system calls are invoked.

©2005 SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025. SRI International is a nonprofit organization.
4 API Resolution Techniques

User-level malware programs require the invocation of system calls to interact with the OS in order to perform malicious actions. Therefore, analyzing and extracting malicious behaviors from these programs requires the identification of system calls invoked from within the code. Although system calls in operating systems are predefined mechanisms for trapping to the kernel and asking for services, application programs may interact with other standard helper modules provided by the OS that eventually trap into the kernel. As an example, in Windows, the Win32 API is a collection of services provided by helper DLLs that reside in user space, while the native APIs are services provided by the kernel. In such a design, the user-level API allows a higher-level understanding of behavior because most of the semantic information is lost at the native level. Therefore, an in-depth binary static analysis requires the identification of all Windows API calls, and call sequences, made within the program.

Obfuscations that impede analysis by using various methods to hide API calls from within the code have become prevalent in malware. Analyzers such as IDA Pro [1] or OllyDbg [2] support the standard loading and linking of binaries with DLLs and allow the identification of API calls in a straightforward manner for legitimate binaries. Unfortunately, modern packers bypass standard link and load methods. Rather, they employ a variety of nonstandard techniques to link or connect call sites with the intended API function residing in a DLL.

We refer to the task of deobfuscating or identifying Windows API function targets from the image of a previously packed malware binary, no matter how they are referenced, as **obfuscated API resolution**. In this section, we first provide a background on how normal API resolution occurs in Windows, and then contrast this with how Eureka handles problems of obfuscated API resolution. These analyses are performed on IDA Pro’s disassembly of the unpacked binary, as produced by Eureka’s automated unpacker.

Understanding the issue of API resolution requires understanding how Windows executables refer to APIs from the various Windows libraries. The PE executable file format includes an import section that determines what DLLs are imported by the executable. The information contained in the import table is used by the loader at runtime to identify the addresses of the referred APIs so that whenever an API is called, a jump to the API code is executed.

Roughly speaking, the import table is an array where each element represents a DLL and the set of imported functions from that DLL. For each DLL, the corresponding APIs can be referenced either by name or by ordinal. In addition to the import table, an executable can at runtime load additional DLLs and refer to their function using `GetProcAddress` that retrieves the address of the given API.

Unpacked binaries are the result of taking a memory snapshot of the executable, at which time the
import table has already been built and might even have been corrupted by either erasing imported
API information or adding erroneous entries to the table to prevent the automated reconstruction
of the import table.

Figure 2 shows the typical states of the import table after the unpacked code is generated. On the
left, we consider the case where the loader has determined the addresses of the referred APIs. On
the right is the case where the import table has not been built, but the necessary information to
resolve the APIs is present.

![Figure 2: Example of import table after unpacking](image)

4.1 Background: standard API resolution

Understanding the challenges of obfuscated API resolution first requires an understanding of how
packers typically avoid the standard methods of linking API functions that reside in user-level
DLLs. We first briefly describe the standard method. Windows APIs are functions exported from
standard DLLs, such as \texttt{KERNEL32.DLL} or \texttt{USER32.DLL}. The Windows process loader and linker are
responsible for linking DLLs with a PE (Portable Executable) binary. Figure 3 illustrates the high-
level view of the mechanism. Each executable contains an import table directory, which consists
of entries corresponding to each DLL it imports. The entries point to tables containing names or
ordinals for functions that need to be imported from a specific DLL. When the binary is loaded,
the required DLLs are mapped into the memory address space of the application, and the export
table in the DLL is used to determine the virtual addresses of the functions that need to be linked.
A table called The Import Address Table (IAT) is filled in by the loader and linker with the virtual
addresses of each imported function. This table is referred to by indirect control flow instructions
in the program to call the functions in the linked DLL.

API resolution in benign executables: There are two potential ways a legitimate (unobfuscated)
program might invoke user-level APIs. The first method involves calling thunks, which are
small subroutines that contain an indirect \texttt{JMP} instruction using an entry in the IAT. The second
method does not use thunks, but places indirect CALL instructions directly in the code that use the entries in the IAT. These API invocations can easily be identified by referring to the corresponding entry in the import directory for an IAT entry used at a call site. In this scenario, the import directory works as an explicit mapping between a call site and its target API.

**API resolution in packed malicious executables:** Packers avoid using the standard linking mechanism by removing entries from the import directory of the packed binaries. For the program to function as before after unpacking, the logic of loading the DLLs and linking the program with the API functions is incorporated into the program itself. Among other methods, this may include explicit invocations to *GetProcAddress* and *LoadLibrary* API calls. The *LoadLibrary* API provides a method of mapping a DLL into a process’s address space during execution, and the *GetProcAddress* API returns the virtual address of an API function in a loaded DLL.

Earlier naive approaches rebuilt the import table as well as the IAT after unpacking. Achieving API resolution on the unpacked execution image of such binaries is trivial, as long as our image is taken after the IAT is rebuilt. Recent packers provide API call obfuscation as a feature and rely on sophisticated methods that do not rebuild the IAT or the import table. These packers either utilize proprietary table structures or employ various control flow obfuscation techniques that make identification of the targets of calls made to API functions difficult. Eureka includes API resolution techniques that are intended for these cases, which are now quite prevalent in captured malware corpuses.

### 4.2 Resolving obfuscated APIs without the import tables and IAT

Let us assume that the IAT defined in a malware executable’s header is incomplete, corrupt, or not used at all. Let us further assume that the unpacking routine may include entries in the IAT that are planted to mislead naive analysis attempts. Moreover, the malware executable has the power to recreate a similar table in any memory location of its choosing or use methods that may not require table-like data structures. The objective of Eureka’s API resolution module is to resolve APIs in such cases to facilitate the static analysis of the executable. In the following sections we outline the strategies used by the Eureka API resolution module to accomplish these deobfuscations, presented in the increasing of complexity.

---

1 In most cases, at least these two API functions are kept in the import table, or their addresses are hard coded in the program.

©2005 SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025. SRI International is a nonprofit organization.
4.2.1 Handling DLL obfuscations

**DLLs loaded at standard virtual addresses:** By default, DLLs are loaded at the virtual address specified as the image base address in the DLL’s PE header. The standard Windows Win32 DLLs specified bases that do not clash with each other. Therefore, unless intervened, the loader and linker can load all these DLLs at the specified base virtual addresses. By assuming that this is the case, a table of probable virtual addresses of each exported API function from these DLLs can be built. This simple method has been found to work for many unpacked binary malware images. For example, for Windows XP service pack 2, the `KERNEL32.DLL` has a default image base address of 0x7C800000. The RVA of the API `GetProcessId` is 0x60C75, making its default virtual address 0x7C860C75.

In such cases, Eureka’s analysis proceeds as follows to reconstruct API associations. For each Win32 DLL $D_i$, let $B_i$ be the default base address. Also, let there be $k_i$ exported API functions, where each function $F_{i,j}$ has the RVA (relative virtual address) $R_{i,j}$. Eureka builds a database of virtual addresses $V_{i,j} = B_i + R_{i,j}$ and their corresponding API functions. Whenever Eureka finds a call site $c$ with resolved target address $A(c)$, it searches all $V_{i,j}$ to identify the API function target. We find that this method works as long as the DLLs are loaded in the default base address.

**DLLs loaded at arbitrary virtual addresses:** To make identification of an API harder, there may be cases where a DLL is loaded into a nonstandard base address by system calls to explicitly map them into a different address space. As a result, the address found during analysis of the unpacked binary may not be found in the computed virtual address set. In this case, we can utilize some of the dynamic information captured by running malware (in many cases, this information can be harvested during Eureka’s unpacking phase). The idea is to use runtime information of native system calls that are used to map DLL and modules into the virtual address space of an application. Since our unpacker traces native system calls, we can look for specific calls to `NtOpenSection` and `NtMapViewOfSection`. The former system call identifies the DLL name, and the latter provides the base address where it is loaded. Eureka correlates these two calls using the handle returned by the first system call.

4.2.2 API resolution for statically identifiable targets

One way to identify an invocation of an API function without relying on the import directory of the unpacked image is by testing targets of call sites to see whether they point to specific API functions. We assume that a call site may use an indirect call or a jump instruction. Such instructions may involve a pointer directly or may use a register that is loaded with an address in an earlier instruction. To identify targets in a generic manner, Eureka uses static analysis on the unpacked disassembly.

Eureka starts by performing control flow analysis on the program. The use of IDA Pro disassembly simplifies analysis by marking subroutine boundaries and inter-procedural control flows. Furthermore, control flow instructions that have statically identified targets that reside within the program are also resolved. In addition, IDA Pro identifies any valid API calls through the import directory and the IAT. Eureka’s analysis task then is to resolve unknown static or statically resolvable target addresses in control flow instructions. These are potential calls to API functions residing in DLLs.

©2005 SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025. SRI International is a nonprofit organization.
Our algorithm proceeds as follows. First, Eureka identifies functions in the disassembly (marked as subroutines using the SUB markers). For each function, the control flow graph is built by identifying basic blocks as nodes and static intra-procedural control flow instructions that connect them as edges. Eureka then models inter-procedural control flow by observing CALL or JMP instructions to subroutines that IDA already identifies. It selects any remaining such instructions with an unrecognized target as potential API call sites. For these instructions, Eureka uses static analysis to identify the absolute memory address to which they will transfer control.

We now use a simple notation to express the x86 instructions that Eureka analyzes. Let the set of all instructions be \( I \). For any instruction \( i \in I \), we use the notation \( S(i) \) as the source operand if one exists, and \( T(i) \) as the target operand. The operands may be immediate values, memory pointer indirection, or a register. Suppose the set of potential API call instructions is \( C \subseteq I \). Our goal is to find the target address of a potential API call instruction \( c \), which we express by \( A(c) \).

For instructions with immediate addresses, \( A(c) \) can be found directly from the instruction. For indirect control transfers using a pointer, such as CALL \[ X \], Eureka considers the static value stored at address \( X \) as a target. Since Eureka uses the disassembly generated by IDA, the static value at address \( X \) is included as data definition with the name \texttt{dword X}.

For register-based control transfers, Eureka needs to identify the value loaded in the register at the point of initiating the transfer. Some previous instruction can load the register with a value read from memory. A generic way to identify the target is to extract a sequence of instructions that initially loads a value from a specific memory address to a register and subsequently is loaded to the register that is used in the control-transfer instruction. Eureka resorts to dataflow analysis for solving these cases.

Using standard dataflow analysis at the intra-procedural level, Eureka identifies def-use instruction pairs. A def-use pair \((d, u)\) is a pair of instructions where the latter instruction \( u \) uses an operand that is defined in \( d \), and there is a control flow path between these instructions with no other definitions of that operand in between. For example, a MOV ESI, EAX followed by CALL ESI instruction with no other redefinitions of ESI forms a def-use pair for the register ESI. To find the value that is loaded in the register at the call site, starting from a potential call site instruction, Eureka identifies a chain of def-use pairs that end at this instruction involving only operands that are registers. Therefore, the first pair in the chain contains a def that loads to a register a value from memory or an immediate value, which is subsequently propagated to the call site. Figure 4(a) illustrates these cases. The next phase is to determine whether the address \( A(c) \) for a call site \( c \) is indeed an API function, and if so Eureka resolves its API name.

### 4.2.3 API resolution for dynamically computed addresses

In some cases, the resolved target address \( A(c) \) can be uninitialized. This may happen if the snapshot is taken at a point during the execution when the resolution of the API address has not taken place in the malware code. It may also be the case that the address is supposed to be returned from a system call such as GetProcAddress, and thus is not contained in the unpacked memory image. In such cases, Eureka attempts to analyze the malware code and extract the portion of code that is supposed to update this address by identifying instructions that write to the memory location that contained \( A(c) \). For each of these instructions, Eureka constructs def-use chains and identifies where they are initiated. If in the control flow path there is a call to the GetProcAddress,
Eureka identifies the arguments pushed onto the stack before calling the service. Since it is one of the arguments, Eureka can directly identify the name of the API whose address is returned and stored in the pointer. Figure 4(b) illustrates a sample code template and how our analysis propagates results of GetProcAddress to call sites.

4.2.4 More sophisticated cases

A packer may incorporate more sophisticated schemes in the malware code to resolve APIs at runtime. Besides MOV instructions, a sequence of PUSH and POP instructions can transfer values from one register to another. Although a simple sequence of a PUSH followed by a POP can be treated as a MOV instruction, an arbitrary number of these sequences require modeling the program stack during dataflow analysis, which is costly but possible. Uses of pointers can complicate identification because precise pointer analysis is undecidable and approximate algorithms tend to be costly.

To hide DLL base addresses from analyzers, packers may map a DLL into one portion of memory and then copy the contents to another allocated memory region. This action will not be revealed while intercepting system call sequences. The problem of such a technique is that for every DLL, the unpacker has to allocate memory for the DLLs in the process address space, which is not needed when an already-loaded DLL is mapped into the virtual address space of the application. Even if such a technique is used by an unpacker, all allocated virtual addresses can be scanned for PE header structures conforming to the API DLLs at the point when the unpacking snapshot is taken. Eureka does not handle these sophisticated cases at the moment, but we feel that some of these could be addressed using symbolic execution [14] or value-set analysis (VSA) [4].

5 Evaluation Metrics and Graph Generation

We consider the problems of measuring and improving analyzability after API resolution. First, we implement two metrics to evaluate the viability of static analysis on the unpacked code. We then describe additional static analysis on the malware code that can help visualization and analysis of
the behavior of a single malware sample or highlight dissimilarities across samples that are close variants. We describe how to generate a simplified and annotated call graph of the malware code and the use of micro-ontology labeling to generate labeled call graphs that illustrate capabilities. We illustrate these capabilities in Section 6 using the Storm worm as a case study.

5.1 Measuring analyzability

It is important to evaluate the output of the unpacking and API resolution phases in terms of the utility of these results to drive follow-on static analyses. Although a manual inspection can determine the quality of the output and its suitability for applying static analysis, in a large corpus of thousands of malware programs, automated methods for performing this step are essential. Technically, without the knowledge of the original malware code, it is impossible to precisely conclude how successfully the obfuscations applied to a code have been removed. Nevertheless, several heuristics can aid malware analysts and other post-unpacking static analysis tools in deciding which unpacked binaries can be analyzed successfully, and which require further attempts at deobfuscation. Poor analyzability metrics could further help detect when previously successful malware deobfuscation strategies are no longer successful, possibly due to new countermeasures employed by malware developers to thwart the unpacking logic. Here we present heuristics that we have incorporated in Eureka to express the quality of the disassembled process image, and its potential analyzability in subsequent static analyses.

**Code-to-data ratio:** An observable difference between packed code and unpacked code is the amount of identifiable code and data found in the binary. Although differentiating between code and data on x86 variable-length instructions is a known hard problem, in practice the state-of-the-art disassemblers and analyzers such as IDA Pro are quite capable of identifying code by recursively passing through code and by taking into account specific valid code sequences. However, these methods tend to err on the side of detecting data as code, rather than the other way around. Therefore, if code is identified via IDA Pro, it can be taken with confidence that it is actual code. The amount of code that is identified in and provided from an unpacker can be used as a reasonable indication of how completely the binary was unpacked.

Since there is no ground truth on the amount of code in the original malware binary prior to its packing, we have no absolute measures from which we can compare the quality of the unpacked results. However, empirically, we find that the ratio of code to data found in the unpacked binary is a useful analyzability metric. Usually, any sequence of bytes that is not identified as code is treated as data by IDA Pro. In the disassembled code, these data are represented using the data definition assembler mnemonics — `db`, `dw` or `dd`. We use the ratio of identified code and data by IDA Pro as an indication of unpacking quality. The challenge with this measurement is in identifying the threshold above which we can conclude that packing was successful. We used an empirical approach to determine a suitable threshold for this purpose. When experimenting with packed and unpacked binaries of benign programs, we observed that the amount of identified code is very low for almost all different packer-generated packed binaries. There were slight variations depending on the unpacking code inserted by the packer. Still, we found the ratio to be well below 3% in all cases. Although the ratio of code vs. data increased significantly after unpacking, it was not equal to the original benign program prior to packing, because the unpacked code still contained the packed data in the memory image, which appeared as data definitions in the disassembly. We
found that most of the successfully unpacked disassemblies had code-to-data ratios well above 50%. Eureka uses the 50% threshold as the value of valid unpacking.

**API resolution success:** When attempting to conduct a meaningful static analysis on an unpacked binary, one of the most important requirements is the proper identification of control flow, whether it relates to Windows APIs or to the malware’s internal functions. Incomplete control flow can adversely affect all aspects of static analyses. One of the main culprits of control flow analysis is the existence of indirect control flow instructions whose targets are not statically identifiable and can be derived only by dynamic means.

In Section 4, our presented API resolution method tries to identify the targets of call sites that were not identified by IDA Pro. If the target is not resolvable, it may be a call to an API function that was successfully obfuscated beyond the reversal techniques used by Eureka, or it may be a dynamically computed call to an internal function. In both cases, we lose information about the control flow behavior from that point in the program. By taking success and failure scenarios into account, we can compute the ratio of resolved APIs and treat it as an indication of quality of subsequent static analysis.

Our API resolution quality is expressed as a percentage of total number of API calls that have been resolved from the set of all potential API call sites, which are indirect or register-based calls with unresolved target. A higher value of $p$ indicates that the resulting deobfuscated Eureka binary will be suitable for supporting static analyses that support more in-depth behavioral characterization.

### 5.2 Malware call graph generation

A call graph is generated using static analysis, where each node is a procedure or function in a program and the edges denote calls from one function made to another. Call graphs are useful for identifying co-relations or dependencies among different modules of a program. In the case of source programs and legitimate binaries compiled in the clear, the programmer-provided names of functions can allow the call graph to provide a high-level view and layout of logical program structure. Unfortunately, for packed (and obfuscated) binaries, missing function names and the lack of debug information render traditional call graphs as less useful depictions of the program layout.

In the call graphs that we generate in Eureka post unpacking and API resolution, we include additional information and perform simplifications in order to enhance the suitability of the call graph for malware analysis. The set of calls to APIs from each function in the malware code can help express high-level behavior exhibited from specific portions of the code. We incorporate the set of API functions called from a function as its attributes in the call graph. The attributes in nodes together with the edges in the call graph can also correlate and find dependencies between APIs distributed across functions. We perform simplification of the call graph by removing nodes that do not provide any information. Many malware samples contain hundreds of functions. Since the goal of the call graph is to express high-level behavior together with structure, any nodes that do not contain API functions as attributes do not add information to the behavior exhibited by the graph. We remove such nodes and connect all inbound and outbound edges together, so that connectivity between callers and callees is maintained.

©2005 SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025. SRI International is a nonprofit organization.
The simplified and annotated call graph is not only useful for understanding code semantics, but also serves as a way to identify malware families and highlight dissimilarities between close malware variants. Graph comparison has been used earlier for comparing malware code samples for similarities. The problem of subgraph isomorphism is known to be NP-hard. However, with enough labeling, approximate solutions can provide very useful results very efficiently. By finding the common subgraph between two labeled call graphs, a common code structure may be identified between close variants. Moreover, the remaining nodes and edges can help highlight differences that may be new or extra features added to existing malware families.

5.3 Micro-ontology labeling

One goal of the Eureka framework is to simplify analysis by automatically labeling certain malware code blocks and isolating segments of high interest. As part of our effort to address this, our system uses structure abstraction and subroutine annotation to automatically produce a labeled subgraph from an unpacked, API-resolved disassembly. This labeling feature provides the equivalent of a bird’s eye view of the malware constituents and their interactions.

We classify the common Microsoft Windows user-level API functions, i.e., a list of 3140 functions into 103 categories. Most of these categories were harvested directly from Microsoft’s classifications of API functions [26], while a few others (around ten) were added manually. Popular and intuitive categories used in the labels include “Common File System”, “Random”, “Process and Thread”, “File Management”, “Socket”, “Time”, and “Registry”. We call this subroutine annotation process micro-ontology labeling. Eureka produces hyperlinked HTML and PNG call graphs. By clicking on these annotated subroutines, the analyst is traversed directly to the subroutine call graphs where API calls and data associations can be examined in greater detail. In the future, we plan to extend our micro-ontologies to higher-level classifications (macro-ontologies) that can automatically recognize and classify frequently shared and recurring code segments [20] in malware such as DoS, spam, command-and-control, and data harvesting.

6 Case Study: Analysis of the Storm Worm

The Storm worm appeared on the Internet in January 2007 and has since infected thousands (potentially millions of computers worldwide throughout several outbreaks. Many variations of Storm have been released with each outbreak, and static analysis of the different released binaries provides an efficient way to discover logical similarities, as well as newly introduced changes to Storm’s logic. We focus our analysis here on the version released on Labor day (September 2, 2007) in the form of the executable labor.exe. Along with other variants of Storm, labor.exe is packed using a custom packer employing known encryption routines. An elaborate discussion of the cryptographic code and rootkit techniques employed in Storm is provided in [7]. A more substantial in-depth analysis of Storm is available in [29].

Storm code executes in three stages. The first stage is the first level of unpacking that executes an XOR decryption of a data block. The second stage decrypts more code that constitutes the third and final stage, where the bulk of Storm’s logic is executed. It creates a copy of itself called spooldr.exe, modifies a legitimate driver tcpip.sys, and creates a new driver spooldr.sys. The
created file `spooldr.exe` is an exact copy of the malware in its encrypted form, but the `spooldr.sys` driver can be created in either its encrypted or decrypted form, depending on the version of Storm (different versions of Storm might infect drivers other than the tcpip driver). Unpacking Storm with Eureka reveals the third stage of execution that represents the bulk of Storm’s logic including the antidebugging mechanisms, spam and DoS capabilities, and a peer-to-peer protocol for C&C. The different versions of Storm differ in their implementations of checks to detect debuggers, virtual environments such as VMware and Virtual PC, that lead the code into an infinite loop whenever such environments are detected. The versions we have analyzed implement roughly the same logic, modulo the anti-debugging and anti-malware analysis techniques employed. Also, these different versions are created from a core common code base that is customized by the malware’s author(s). Understanding Storm’s logic required us to generate a clean assembly that would allow us to build a CFG of its code, recover all API calls, and identify their arguments.

We have used Eureka to analyze and facilitate the reverse engineering of Storm’s logic. The Eureka unpacker provided an unpacked binary image in which both the code-to-date ratio and bigram statistical analysis indicated that Storm was unpacked properly. Eureka’s API resolution techniques also properly rebuilt the import table and achieved a 100% API resolution rate. The successful unpacking and API resolution phases allowed us to build a CFG of Storm’s unpacked code and subjected such graph to further static analysis to reverse engineer Storm’s logic.
6.1 Storm Logic’s Overview

Figure 5 illustrates a high-level annotation of the different blocks of Storm’s code. Building the CFG allowed us to recognize the address 0x403318 as the entry point of the unpacked code. API resolution allowed us to determine which APIs are invoked throughout the different code blocks and allowed us to label code blocks. For instance, in a block where `CreateFile` is invoked with a string argument `spooldr.ini` and `WSAStartup` is invoked, we determine that such a block corresponds to the creation of the file `spooldr.ini` is the startup of the network service. Similarly, we determined code blocks that correspond to SPAM delivery, email propagation, binary update modules, infinite loops, and process exit points.

6.2 Overnet/eDonkey Communication Logic

Storm’s capabilities are coordinated with infected peers using the eDonkey peer-to-peer protocol. Based on the API resolution, we identified `send` and `recv` API calls and their arguments and determined which portion of Storm’s unpacked code corresponded to the eDonkey protocol. The eDonkey protocol is executed in a block of instructions at address 0x004033B.

Figure 6 shows the control flow graph of the eDonkey protocol handler and illustrates how Storm dialog sequences are generated.

7 Experimental Results

We now evaluate the effectiveness of Eureka using three different datasets. First, we measure how Eureka and other unpackers handle various common packers using a dataset of packed benign executables. Next, we evaluate how Eureka performs on two recent malware collections: a corpus of 479 malicious executables obtained from spam traps and a corpus of 435 malicious executables obtained from our honeynet deployment. Finally, we provide some runtime performance measurements.

7.1 Benign dataset evaluation: Goat test

We evaluated Eureka using a dataset of packed benign executables. Specifically, we used several common packers to pack an instance of the popular Microsoft Windows executable, `notepad.exe`. An advantage of testing with a dataset of custom-packed benign executables is that we have ground truth for what the malware is packed with and we know exactly what is expected after unpacking. This makes it easier to evaluate the quality of unpacking results. We compared the unpacking capability of Eureka to that of PolyUnpack (using a limited distribution version obtained from the author) and Renovo (by submitting to BitBlaze malware analysis service [6]). We were unable to acquire OmniUnpack for our test results.

These results are summarized in Table 3. In cases where an output was found, we used Eureka’s code-to-data ratio heuristic to determine whether it was successfully unpacked and manually also verified the results of the heuristic. For Renovo, we compare with the last layer that was produced...
in the case of multiple unpacked layers. The results show that Eureka performs well compared to other unpacking solutions. Eureka was successful in all cases except Asprotect, which interfered with Eureka’s driver, and Themida, where the output was an altered unpacking with API calls emulated. In Figure 7 we illustrate how the bigram counts change as Eureka executes for three of the packers. We find that in most cases the bigram counts change synchronously and very sharply (similar to ASPack) making it easy to determine appropriate points for snapshotting execution images. We find that Eureka is also robust to packers that naively employ multiple layers such as MoleBox and some incremental packers such as Armadillo.

In this comparison study, PolyUnpack failed in many instances including cases where it unveiled just a single layer of packing while the output still remained packed. We suspect that aggressive implementation of anti-debugging features might be impairing its current success. Renovo, on the other hand, provided several unpacked layers in all cases except for Obsidium. Further analysis of the output, however, revealed that in some cases the binary was not completely unpacked. Finally,
Table 3: Evaluation of Eureka, PolyUnpack and Renovo: √ = unpacked; ⊗ = partially unpacked; × = unpack failed

<table>
<thead>
<tr>
<th>Packer</th>
<th>PolyUnpack Unpacking</th>
<th>Renovo Unpacking</th>
<th>Eureka Unpacking</th>
<th>Eureka API Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armadillo</td>
<td>×</td>
<td>⊗</td>
<td>√</td>
<td>64%</td>
</tr>
<tr>
<td>Aspack 2.12</td>
<td>⊗</td>
<td>√</td>
<td>√</td>
<td>99%</td>
</tr>
<tr>
<td>Asprotect 1.35</td>
<td>⊗</td>
<td>√</td>
<td>×</td>
<td>2%</td>
</tr>
<tr>
<td>ExeCryptor</td>
<td>√</td>
<td>⊗</td>
<td>√</td>
<td>97%</td>
</tr>
<tr>
<td>ExeStealth 2</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>0%</td>
</tr>
<tr>
<td>FSG 2.0</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>97%</td>
</tr>
<tr>
<td>MEW 1.1</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>97%</td>
</tr>
<tr>
<td>MoleBoxPro</td>
<td>×</td>
<td>√</td>
<td>√</td>
<td>98%</td>
</tr>
<tr>
<td>Morphine 1.2</td>
<td>√</td>
<td>⊗</td>
<td>√</td>
<td>0%</td>
</tr>
<tr>
<td>Obsidium</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>99%</td>
</tr>
<tr>
<td>PeCompact 2</td>
<td>×</td>
<td>⊗</td>
<td>√</td>
<td>99%</td>
</tr>
<tr>
<td>Themida</td>
<td>×</td>
<td>⊗</td>
<td>⊗</td>
<td>–</td>
</tr>
<tr>
<td>UPX 3.02</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>99%</td>
</tr>
<tr>
<td>WinUPack 3.99</td>
<td>⊗</td>
<td>√</td>
<td>√</td>
<td>99%</td>
</tr>
<tr>
<td>Yoda 3.53</td>
<td>⊗</td>
<td>⊗</td>
<td>√</td>
<td>97%</td>
</tr>
</tbody>
</table>

Figure 7: Bigram counts during execution of goat file packed with Aspack(left), Molebox(center), Armadillo(right).

our results show that Eureka’s API resolution technique determined almost all APIs for most packers and failed considerably in some others. Particularly, we found ExeCryptor and FSG to use a large amount of code rewriting for obfuscating API calls, including use of arbitrary combinations of complex instruction sequences to dynamically compute the targets.

7.2 Malicious data set evaluation

Spam corpus evaluation: We begin by evaluating how Eureka performs on a corpus of 481 malicious executables obtained from spam traps. The results are very encouraging. Eureka successfully unpacked 470 of 481 executables. Of the 470 executables from this spam corpus, 401 were successfully unpacked simply using the heuristic-based unpacker; the remainder could only be unpacked using Eureka’s bigram statistical hypothesis test. We summarize Eureka’s results in Tables 7.2 and 7.2. Table 7.2 illustrates the various packers used in this dataset (as classified by PeID) and describes how effectiveness of Eureka varies across the packers. Table 7.2 classifies the dataset based on antivirus (AV) labels obtained from Virus-Total [38] and illustrates how Eureka’s effectiveness varies across malware families. The successful API resolutions validate the quality of Eureka’s unpacking.


**Table 4:** Eureka performance by packer distribution on the spam malware corpus.

<table>
<thead>
<tr>
<th>Packer</th>
<th>Count</th>
<th>Eureka Unpacking</th>
<th>Eureka API Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>186</td>
<td>184</td>
<td>85%</td>
</tr>
<tr>
<td>UPX</td>
<td>134</td>
<td>132</td>
<td>78%</td>
</tr>
<tr>
<td>Warning:Virus</td>
<td>79</td>
<td>79</td>
<td>79%</td>
</tr>
<tr>
<td>PEX</td>
<td>18</td>
<td>11</td>
<td>70%</td>
</tr>
<tr>
<td>MEW</td>
<td>12</td>
<td>11</td>
<td>70%</td>
</tr>
<tr>
<td>Rest (10)</td>
<td>52</td>
<td>46</td>
<td>83%</td>
</tr>
</tbody>
</table>

**Table 5:** Eureka performance by malware family distribution on the spam malware corpus.

<table>
<thead>
<tr>
<th>Malware Family</th>
<th>Count</th>
<th>Eureka Unpacking</th>
<th>Eureka API Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRSmall</td>
<td>98</td>
<td>98</td>
<td>93%</td>
</tr>
<tr>
<td>TRDldr</td>
<td>63</td>
<td>61</td>
<td>48%</td>
</tr>
<tr>
<td>Bagle</td>
<td>67</td>
<td>67</td>
<td>84%</td>
</tr>
<tr>
<td>Mydoom</td>
<td>45</td>
<td>44</td>
<td>99%</td>
</tr>
<tr>
<td>Klez</td>
<td>77</td>
<td>77</td>
<td>78%</td>
</tr>
<tr>
<td>Rest(39)</td>
<td>131</td>
<td>123</td>
<td>78%</td>
</tr>
</tbody>
</table>

**Honeynet corpus evaluation:** Next, we evaluate how our system performed on a corpus of 435 malicious executables obtained from our honeynet deployment. We found that 178 were packed with Themida. In these cases, Eureka obtained an altered execution image. These results highlight the importance of building better analysis tools that can deal with this problem. Out of the remaining 257 binaries, 20 were binaries that did not execute on Windows XP (either because they were corrupted or because we could not determine the right execution environments). Eureka successfully unpacked 228 of the 237 remaining binaries and produced successful API resolutions in most cases.

We summarize results of analyzing the remaining 237 binaries in Tables 6 and 7. Table 6 illustrates the distribution of the various packers used in this dataset (as classified by PeID) and describes how the effectiveness of Eureka varies across the packers. Table 7 classifies the dataset based on AV labels obtained from Virus-Total and illustrates how the effectiveness of Eureka varies across malware families.

**Table 6:** Eureka performance by packer distribution on the honeynet malware corpus minus Themida

<table>
<thead>
<tr>
<th>Packer</th>
<th>Count</th>
<th>Eureka Unpacking</th>
<th>Eureka API Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyEne</td>
<td>109</td>
<td>109</td>
<td>97%</td>
</tr>
<tr>
<td>FSG</td>
<td>36</td>
<td>35</td>
<td>94%</td>
</tr>
<tr>
<td>Unknown</td>
<td>33</td>
<td>29</td>
<td>67%</td>
</tr>
<tr>
<td>ASPack</td>
<td>23</td>
<td>22</td>
<td>93%</td>
</tr>
<tr>
<td>tElock</td>
<td>9</td>
<td>9</td>
<td>91%</td>
</tr>
<tr>
<td>Rest(9)</td>
<td>27</td>
<td>24</td>
<td>62%</td>
</tr>
</tbody>
</table>

**7.3 Eureka: runtime performance results**

We compared runtime performance of Eureka when using heuristic unpacking and statistical unpacking. Eureka’s statistical unpacker processed 100 binaries in 69 minutes or about 1 binary every

---

SRI International is a nonprofit organization.
41.4 seconds (including the time to revert Eureka’s (VM)). The remaining Eureka workflow (disassembly, API resolution and graph generation) added a total of 30 minutes or about 18 seconds per binary. The heuristic unpacker was marginally faster at 67 minutes or about 40.1 seconds per binary. Eureka’s performance represents a dramatic speedup in binary processing time in comparison to PolyUnpack, which required 5 to 20 minutes per binary, and Renovo, which runs within an emulator. Eureka’s unpacking performance, ignoring VM revert costs, is comparable to that of OmniUnpack (tens of seconds per binary).

8 Limitations and Future Work

The nature of the malware analysis game dictates that malware deobfuscation and analysis is a perennial arms race between the malware developer and the malware analyst. We expect new challenges to emerge as adversaries learn of and adapt to Eureka. In the near term, we plan to explore various strategies to overcome some of our current known limitations. The following briefly describes limitations and outlines some potential solutions we may pursue.

**Challenges in malware unpacking:** Partial code-revealing packers pose a significant problem for all automated unpackers. These packers implement thousands of polymorphic layers, revealing only a portion of the code during any given execution stage. Once the code section is executed, the packer then re-encrypts this segment before proceeding to the next code segments. At the moment, the favored approach to counter this packing strategy is to dump a continuous series of execution images, which must be subsequently analyzed and reassembled into a single coherent process image. However, this approach offers few guarantees of coverage or completeness. We plan to investigate new methods to extend Eureka to address this important problem. Another challenge is that malware authors will adapt their packing methods to detect Eureka or to circumvent Eureka’s process tracking methods. For example, malware could detect Eureka by looking for kernel API hooking. This is not a fundamental problem with our approach, but rather a weakness in our implementation. One potential solution is to move Eureka’s system call monitoring capability outside the kernel, into the host OS (e.g., via a kernel virtual machine). Knowledgeable adversaries could also design malware that suppresses Eureka’s triggers. A malware author who is aware of the heuristics and thresholds used by Eureka’s statistical models could explicitly engineer malware to evade these triggers, e.g., by avoiding certain system calls that trigger the heuristics or limit the use of certain instructions. We believe that some of this concern could be addressed by parameterizing features of the statistical model to introduce uncertainty in deciding what thresholds the malware must avoid. Malware could alternatively choose to purposely induce Eureka to image dump too soon, prior to performing its process unpacking. To counter this threat, Eureka could produce multiple

<table>
<thead>
<tr>
<th>Malware Family</th>
<th>Count</th>
<th>Eureka Unpacking</th>
<th>Eureka API Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Korgo</td>
<td>70</td>
<td>70</td>
<td>86%</td>
</tr>
<tr>
<td>Virut</td>
<td>24</td>
<td>24</td>
<td>90%</td>
</tr>
<tr>
<td>Padobot</td>
<td>21</td>
<td>21</td>
<td>82%</td>
</tr>
<tr>
<td>Sality</td>
<td>17</td>
<td>17</td>
<td>96%</td>
</tr>
<tr>
<td>Parite</td>
<td>15</td>
<td>15</td>
<td>96%</td>
</tr>
<tr>
<td>Rest(19)</td>
<td>90</td>
<td>81</td>
<td>90%</td>
</tr>
</tbody>
</table>

Table 7: Eureka performance by malware family on the honeynet malware corpus minus Themida
binary images, evaluating each dumped image to choose the image with the optimal analyzability metrics.

**Challenges in post-unpacking analysis:** A natural way to measure the analyzability of static disassembly is to interrogate the call graph structure of unpacked code, *e.g.* by looking at the portion of code that is reachable from the start function knowing the size of the largest connected component, or counting the number of dangling subroutines. A potential challenge here is to separate junk code that might be strategically inserted by the packer as a code obfuscation enhancement. We intend to evaluate the feasibility of such graph-based metrics. Furthermore, malware might include legitimate code that may never be reachable, perhaps because of a jump condition that will never be true. Such cases require a reachability analysis of the code using state exploration techniques, such as model checking. Such state exploration is computationally expensive and might be imprecise in nature because of the large state-space to be explored.

### 9 Conclusion

We have presented the Eureka malware deobfuscation framework, to assist in the automated preparation of malware binaries for static analysis. Eureka distinguishes itself from existing unpacking systems in several important ways. First, it introduces a new methodology for automated malware unpacking, using coarse-grained NTDLL system call monitoring. The unpacking system is robust, flexible, and very fast relative to other contemporary unpacking strategies. The system provides support for both statistical and heuristic-based unpacking triggers and allows child process monitoring. In particular, Eureka’s statistical bigram-based triggering algorithm offers a highly successful methodology for providing an informed basis from which to execute process image dumping. Eureka also includes an API resolution system that is capable of overcoming several contemporary malware API address obfuscation strategies.

Another important aspect of the Eureka framework is the inclusion of a system to measure various attributes of the disassembled process image and API resolution map. We introduced Eureka’s analyzability assessment module, which reports several key disassembly attributes that can assist the analyst in determining whether static analysis investment is warranted, before that investment is incurred. Finally, Eureka simplifies graph structure and automatically generates and annotates nodes in the call graph with ontology labels based on API calls and data references. While the post-unpacking analyses are novel to our system, they are complementary and could be readily integrated into other unpacking tools.

We also evaluate several important aspects of Eureka efficacy. First, we demonstrate the utility of Eureka’s analysis using a Storm worm binary as a case study. Using Storm as a specific example, we show how Eureka’s automated annotations and simple transformations can significantly improve code analyzability. Second, we compare the performance of Eureka with other unpacking systems and its ability to handle common packers using a corpus of packed benign executables. Our results demonstrate that Eureka successfully unpacks the majority of packers (13 of 15) and that its performance is comparable to other automated unpackers. Furthermore, Eureka is able to resolve most API references and produce binaries that result in analyzable disassemblies. We evaluate Eureka on two collections of malware: a spam malware corpus and a honeynet malware corpus. We find that Eureka is highly successful in unpacking the spam corpus (470 of 481 executables),
reasonably successful in unpacking the honeynet corpus (complete dumps for 228 of 435 executables
and partial dumps for 178 of 435 executables) and produces useful API resolutions. Finally, our
runtime performance results validate that the Eureka workflow is highly streamlined and efficient,
capable of unpacking more than 90 binaries per hour. The Eureka service is now available at

This material is based upon work supported by the Army Research Office under Cyber-TA Grant
No. W911NF-06-1-0316 and by National Science Foundation Grant No. CNS-0716612.

References

efficient approach to collect malware. In Proceedings of Recent Advances in Intrusion Detection,
of the 15th Annual Conference of the European Institute for Computer Antivirus Research
(EICAR), 2006.
(POPL98), January 1998.
from zero-day polymorphic and metamorphic worm exploits. In Proceedings of the 12th ACM


SRI International is a nonprofit organization.


