Eureka: A Framework for Enabling Static Malware Analysis

Monirul Sharif[1], Vinod Yegneswaran[2], Hassen Saidi[2], Phillip Porras[2], and Wenke Lee[1]

[1] College of Computing, Georgia Institute of Technology
{monirul,wenke}@cc.gatech.edu
[2] Computer Science Laboratory, SRI International
{vinod,saidi,porras}@csl.sri.com

Abstract. 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. 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.

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. Now multiply this challenge by the hundreds of new strains and repurposed malware variants that appear on the Internet yearly [16,21], and the need to develop *automated tools* to extract and analyze all facets of malware binary logic becomes clear. Unfortunately, malware developers are also well aware of the efforts to reverse engineer their binaries, and employ a wide range of binary obfuscation techniques to deter analysis and reverse engineering.

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 [9,4]. *Static analyses* refer to techniques to decompile and analyze the logical structure, flow, and data content stored within the binary itself.

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 481–500, 2008.

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 analysis based approaches have arguably offered a better track record and mind share among those working on malware binary analysis. Part of that success is attributable to the challenges of overcoming the formidable obfuscation techniques [24,26], or packers [22] that are widely utilized by contemporary malware authors. These obfuscation techniques, including function and API call obfuscation, and control flow obfuscations along with a gamut of other protections proposed by the research community [8,19], have been shown to deter static analyses. While defeating these obfuscations is a prerequisite step to conducting meaningful static analyses, they can largely be overcome by those conducting dynamic analyses. However, traditional dynamic analysis provide only a partial "effects oriented" profile of the full potential of a given malware binary. Multipath exploring dynamic analysis [18] has the potential to improve traditional dynamic analysis by executing code paths for unsatisfied trigger conditions, but does not guarantee completeness.

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 and correlation of 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, and from these form higher level semantics about malicious behavior. Features such as the presence of network communication logic, registry and OS manipulations, object creations (*e.g.*, files, processes, inter-process communication) can be detected, whether these capabilities are exercised at runtime or not. 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.

1.1 The Eureka Framework

In this paper, we introduce a malware binary deobfuscation framework referred to as *Eureka*, designed to maximally facilitate static code analysis. Figure 1 presents an overview of the modules and logical work flow that compose the Eureka framework. The Eureka workflow begins with the subject-packed malware binary, which is executed in a VM managed by Eureka. After interrogating local environment for evidence of tracing or debugging, the malware process enters a phase of unpacking and the eventual spawning of its core malware payload logic while a parallel Eureka kernel driver tracks 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 course-grained execution tracking algorithm and introduce 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

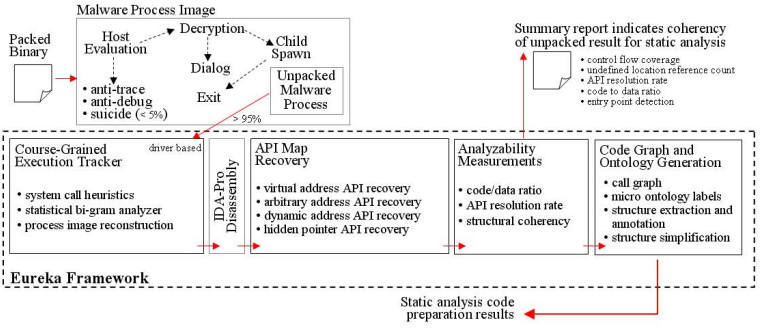

Fig. 1. The Eureka Malware Binary Deobfuscation Framework

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

System	Monitoring Environment	Monitoring Granularity	Trigger Types	Child Process Monitoring	Output Layers	Execution Speed	Potential Evasions
PolyUnpack	Inside VM	Instruction	Model-based	No	1	Slow	1,2,3
Renovo	Emulator	Instruction	Heuristic	Yes	many	Slow	2,4
OmniUnpack	Inside VM	Page	Heuristic	No	many	Fast	2,3
Eureka	Inside VM	System Call	Heuristic, Statistical	Yes	1,many	Fast	2,3

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 which compares several attributes to decide if static analysis of the unpacked image yields useful results. Following the presentation of the Eureka framework, we further present a corpus evaluation (Section 6) to illustrate the usage and effectiveness of Eureka.

2 Related Work

The problem of obfuscated malware has confounded analysts for decades [26]. The first obfuscation techniques exhibited by malware in the wild include viral metamorphism [26] and polymorphism [24]. Several obfuscation approaches have since been presented in the literature [7] including, opaque predicates [8] and recently opaque constants [19]. Packers and executable protectors [22] 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.

Automated unpacking. There have been several recent attempts at building automated and generic tools for unpacking malware, most notably PolyUnpack [23], Renovo [13], and OmniUnpack [17]. Table 1 summarizes the design

space of automated unpackers that illustrates their strengths, differences, and common weakness. PolyUnpack, which was the first automated unpacking technique, builds a static model of the program and uses fine-grained execution tracking to detect when an instruction an instruction outside of the model is executed. 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 and dynamic malware analysis. Previous work in malware analysis that uses static analysis has primarily focused on malware detection approaches. Known malicious patterns are identified in [10]. The approach of using semantic behavior to thwart some specific code obfuscations was presented in [11]. Rootkit behavior detection was presented in [15], and [14] uses a static analysis approach to identify spyware behavior in Browser Helper Objects. Traditional program analysis techniques [20] have been investigated for binary programs in general and malware in particular. Dataflow techniques such as Value Set Analysis [3] aim at recovering the set of possible values that each data object can hold at each program point. CWSandbox [9] and TTAnalyze [4] are dynamic analysis systems that execute programs in a restricted environment and observe sequence of system interactions (using system calls). Pararoma [30] uses system-wide taint propagation to analyze information flow, which it uses for detecting malware. Bitscope [6] incorporates symbolic execution-based static analysis to analyze malicious behavior.

Statistical analysis. Fileprint analysis [25] 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 [29] and Anagram [28].

3 Informed and Coarse-Grained Execution Tracking

In general, all of 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 intercepting interrupts 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 image for disassembly when it believes that the malicious payload logic has been sufficiently revealed. In the following, we present two different methods for deciding when to dump the malicious process image, *i.e.,* a heuristic-based method which works for most contemporary malware and a statistical n-gram anlaysis method which is more robust.

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. 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.

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 6). 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 data sections (*i.e.,* ASCII data) are 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 observations. First, code has certain intrinsic properties that tend to be invariant across executables (*e.g.,* certain opcodes, registers, and instruction sequences 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 a 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 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 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 one byte opcode (e8) has a relative offset. The last byte of this offset invariably ends up being 00 or FF depending on whether 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 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.

Table 2. Occurrence summary of bigrams

Bigrams	calc (117 KB)	explorer (1010 KB)	ipconfig (59 KB)	lpr (11 KB)	mshearts (131 KB)	notepad (72 KB)	ping (21 KB)	shutdown (23 KB)	taskman (19 KB)
FF 15 (call)	246	3045	184	24	192	415	58	132	126
FF 75 (push)	235	2494	272	33	274	254	41	63	85
E8 - - - 0xff (call)	1583	2201	181	19	369	180	87	49	41
E8 - - - 0x00 (call)	746	1091	152	62	641	108	57	66	50

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 (μ_0) 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 (μ_1) and standard deviation (σ_1). We compute $z0 = \frac{\mu_1 - \mu_0}{\sigma_1}$. If $z0 > 1.645$ 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.

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 require the identification of invoked system calls. Besides the predefined mechanism of system calls that require trapping to kernel, application programs may interact with the operating systems via higher level shared helper modules. For 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 hiding API calls have become prevalent in malware. Analyzers such as IDA Pro [1] or OllyDbg [2] support the standard loading and linking method of binaries with DLLs, which modern packers bypass 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

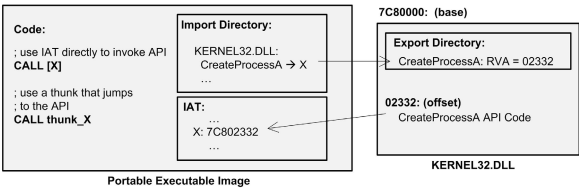

Fig. 2. Example of the standard linking mechanism of PE executables in Windows

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.

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. The Windows process loader and linker are responsible for linking DLLs with a PE (Portable Executable) binary. Figure 2 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.

4.2 Resolving Obfuscated APIs without the Import Tables and IAT

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.[1] 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.

[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.

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, we outline the strategies used by the Eureka API resolution module to accomplish these deobfuscations, presented in the increasing order of complexity.

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 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 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 (relative virtual address) 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.

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. 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 `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

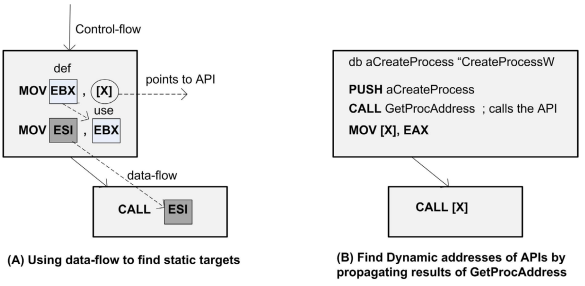

Fig. 3. Illustration of Static Analysis Approaches used to Identify API Targets

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 3(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.

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 3(b) illustrates a sample code template and how our analysis propagates results of `GetProcAddress` to call sites.

5 Evaluation Metrics

We consider the problems of measuring and improving analyzability after API resolution. 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.

6 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.

6.1 Benign Dataset Evaluation: Goat Test

We evaluate Eureka using a dataset of packed benign executables. Specifically, we used several common packers to pack an instance of the popular Microsoft

Table 3. Evaluation of Eureka, PolyUnpack and Renovo: $\surd$ = unpacked; $\otimes$ = partially unpacked; $\times$ = unpack failed

Packer	PolyUnpack Unpacking	Renovo Unpacking	Eureka Unpacking	Eureka API Resolution
Armadillo	$\times$	$\otimes$	$\surd$	64%
Aspack 2.12	$\otimes$	$\surd$	$\surd$	99%
Asprotect 1.35	$\otimes$	$\surd$	$\times$	–
ExeCryptor	$\surd$	$\otimes$	$\surd$	2%
ExeStealth 2	$\times$	$\surd$	$\surd$	97%
FSG 2.0	$\surd$	$\surd$	$\surd$	0%
MEW 1.1	$\surd$	$\surd$	$\surd$	97%
MoleBoxPro	$\times$	$\surd$	$\surd$	98%
Morphine 1.2	$\surd$	$\otimes$	$\surd$	0%
Obsidium	$\times$	$\times$	$\surd$	99%
PeCompact 2	$\times$	$\surd$	$\surd$	99%
Themida	$\times$	$\otimes$	$\otimes$	–
UPX 3.02	$\surd$	$\surd$	$\surd$	99%
WinUPack 3.99	$\otimes$	$\surd$	$\surd$	99%
Yoda 3.53	$\otimes$	$\otimes$	$\surd$	97%

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 compare 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 [5]). 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 4, 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 just unveiled 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, our results show that Eureka's API resolution technique was able to determine 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.

6.2 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 was able to successfully unpack 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 4 and 5. Table 4 illustrates the various packers used (as classified by PeID) and describes how effectiveness of Eureka varies across packers. Table 5 classifies the dataset based on antivirus (AV) labels obtained from Virus-Total [27] illustrating how Eureka's effectiveness varies across malware families and validating the quality of Eureka's unpacking.

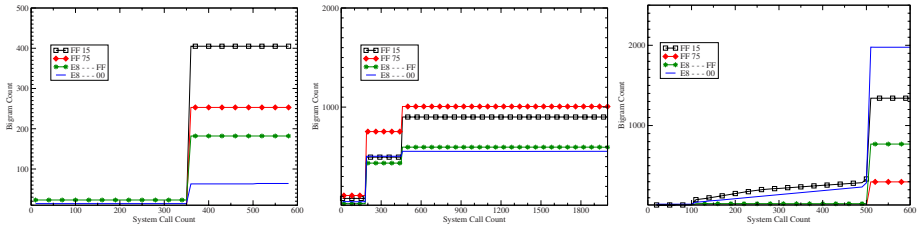

Fig. 4. Bigram counts during execution of goat file packed with Aspack(left), Molebox(center), Armadillo(right)

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

Packer	Count	Eureka Unpacking	Eureka API Resolution
Unknown	186	184	85%
UPX	134	132	78%
Warning:Virus	79	79	79%
PEX	18	18	58%
MEW	12	11	70%
Rest (10)	52	46	83%

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

Malware Family	Count	Eureka Unpacking	Eureka API Resolution
TRSmall	98	98	93%
TRDldr	63	61	48%
Bagle	67	67	84%
Mydoom	45	44	99%
Klez	77	77	78%
Rest(39)	131	123	78%

Honeynet corpus evaluation. Next, we evaluate how our system performs on a corpus of 435 malicious executables obtained from our honeynet deployment. We found that 178 were packed with Themida. In these cases, Eureka is only able to obtain an altered execution image.[2] These results highlight the importance of building better analysis tools that can deal with this important 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 is able to successfully unpack 228 of the 237 remaining binaries and produce 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 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.

7 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

[2] As we see from Table 3, this class of packers also poses a problem for the other unpackers.

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

Packer	Count	Eureka Unpacking	Eureka API Resolution
PolyEne	109	109	97%
FSG	36	35	94%
Unknown	33	29	67%
ASPack	23	22	93%
tElock	9	9	91%
Rest(9)	27	24	62%

Table 7. Eureka performance by malware family on the honeynet malware corpus minus Themida

Malware Family	Count	Eureka Unpacking	Eureka API Resolution
Korgo	70	70	86%
Virut	24	24	90%
Padobot	21	21	82%
Sality	17	17	96%
Parite	15	15	96%
Rest(19)	90	81	90%

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.

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 on 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, for example, by avoiding certain system calls that trigger the heuristics or limit the use of certain instructions. We believe 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 binary images, evaluating each dumped image to choose the one with maximal analyzability.

To thwart API resolution, 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. 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. 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 these could be addressed using symbolic execution [12] or value-set analysis (VSA) [3].

8 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. Second Eureka includes an API resolution system that is capable of overcoming several contemporary malware API address obfuscation strategies. Finally, Eureka includes an analyzability assessment module, 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 integrated into other unpacking tools.

Our results demonstrate that Eureka successfully unpacks 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 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 altered 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. Eureka is now available as a free Internet service at `http://eureka.cyber-ta.org`.

References

1. IDA Pro Dissasember, `http://www.datarescue.com/ida.htm`
2. Ollydbg, `http://www.ollydbg.de`
3. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)

4. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A tool for analyzing malware. In: EICAR (2006)
5. BitBlaze, http://bitblaze.cs.berkeley.edu
6. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Bitscope: Automatically dissecting malicious binaries. In: CMU-CS-07-133 (2007)
7. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland (July 1997)
8. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL 1998) (January 1998)
9. Willems, C.: CWSandbox: Automatic Behaviour Analysis of Malware (2006), http://www.cwsandbox.org/
10. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the Usenix Security (2003)
11. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware malware detection. In: Proceedings of the IEEE Symposium on Security and Privacy (2005)
12. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: The proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005) (2005)
13. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: Proceedings of WORM (2007)
14. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based spyware detection. In: Proceedings of the Usenix Security Symposium (2006)
15. Kruegel, C., Robertson, W., Vigna, G.: Detecting kernel-level rootkits through binary analysis. In: Yew, P.-C., Xue, J. (eds.) ACSAC 2004. LNCS, vol. 3189. Springer, Heidelberg (2004)
16. Malfease Malware Repository, https://malfease.oarci.net
17. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697. Springer, Heidelberg (2007)
18. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the IEEE Symposium of Security and Privacy (2007)
19. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697. Springer, Heidelberg (2007)
20. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
21. Offensive Computing, https://www.offensivecomputing.net
22. Realms, S.: Armadillo protector, http://www.woodmann.com/crackz/Packers.htm#armadillo
23. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186. Springer, Heidelberg (2006)
24. Pearce, S.: Viral polymorphism. VX Heavens (2003)
25. Stolfo, S.J., Wang, K., Li, W.-J.: Fileprint analysis for malware detection. In: ACM CCS WORM (2005)
26. Szor, P.: The Art of Computer Virus Research and Defense. Symatec Press (2005)

27. Virus Total Inc., http://www.virus-total.com
28. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A content anomaly detector resistant to mimicry attack. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
29. Wang, K., Stolfo, S.: Anomalous payload-based network intrusion detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 203–222. Springer, Heidelberg (2004)
30. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM conference on Computer and Communications Security (2007)