PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware

Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee
{paul.royal, mphalpin, dagon, edmonds, wenke}@cc.gatech.edu
College of Computing
Georgia Institute of Technology

Abstract

Modern malware often hide the malicious portion of their program code by making it appear as data at compile-time and transforming it back into executable code at runtime. This obfuscation technique poses obstacles to researchers who want to understand the malicious behavior of new or unknown malware and to practitioners who want to create models of detection and methods of recovery. In this paper we propose a technique for automating the process of extracting the hidden-code bodies of this class of malware. Our approach is based on the observation that sequences of packed or hidden code in a malware instance can be made self-identifying when its runtime execution is checked against its static code model. In deriving our technique, we formally define the unpack-executing behavior that such malware exhibits and devise an algorithm for identifying and extracting its hidden-code. We also provide details of the implementation and evaluation of our extraction technique; the results from our experiments on several thousand malware binaries show our approach can be used to significantly reduce the time required to analyze such malware, and to improve the performance of malware detection tools.

1. Introduction

A popular obfuscation mechanism used by modern malware (viruses, trojans, worms, etc.) is the runtime generation and execution of program code. When run, instances of this type of malware transform or *unpack* and then *execute* a block of program code that was obfuscated at compile-time. The transformation generating the code may be trivially simple (e.g., XORing a block of data into code) or reasonably complex (e.g., using the International Data Encryption Algorithm to decrypt a block of code [16]). Regardless of the degree of complexity, the consequence of successfully applying these techniques is that a static analysis of the program will view the obfuscated block as non-instruction data or omit its analysis entirely, thereby hiding the program's true intentions. As a concrete example, *encrypted viruses* are a class of *unpack-executing* malware that keep their method of runtime decryption constant between generations.

The ability of information security practitioners to implement models of detection and methods of recovery against malware are often stymied by instances of unpack-executing malware, such as encrypted and polymorphic viruses. Time must be invested to learn the mechanism by which a given instance of malware unpacks its compile-time obfuscated code (usually the malicious component) so that it can be extracted and studied. Some Computer Emergency Response Teams (CERTs) report that as many as 160 new viruses arrive each day [7], out of many hundreds of sample submissions. Given this volume, the process of unpacking alone (before any analysis is performed) can be overwhelming. Further, resources can be wasted in determining whether a new malware sample contains unpack-execute behavior, or when two or more new samples found turn out to be the same malware with well-differentiated unpacking mechanisms.

In this paper we present a behavior-based approach that uses a combination of static and dynamic analysis to *automate* the process of extracting the hidden-code of unpack-executing malware. We diverge from purely static techniques by focusing on the *results* (i.e., runtime-generated code execution) of unpack-execution rather than the unpacking mechanism used. This perspective yields a detection algorithm that bypasses the shortcomings of other approaches (e.g., [9, 3]) that require either prior knowledge of the exact unpacking technique or explicit programming of the semantic behavior capturing all instances in an unpacking class (e.g., decryption loop/jmp-based unpack-execution).

Our approach to automatically extracting hidden-code is based on the observation that sequences of unpacked code in a malware instance can be made *self-identifying* when the

instance is executed in an environment with knowledge of the instance's static code model. Making self-identification possible can be viewed as *enabling* a dynamic analysis of the malware instance with the ability to *query* for whether the current instruction sequence being executed exists in the static code model. The *absence* of that sequence in the static code model will identify it as unpacked code; an overview of the entire process is shown in Figure 1.

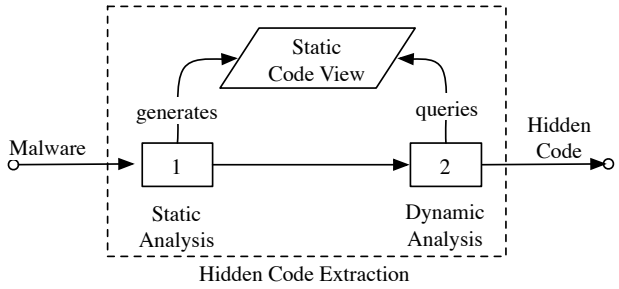

Figure 1: Detecting unpacked code using both static and dynamic analysis.

Starting with a malware instance, we begin by performing static analysis over it to acquire a model of what its execution would look like if it did not generate and execute code at runtime; this is depicted in Step 1. The statically derived model and the malware instance are then fed into the dynamic analysis component where the malware is executed in a *sterile*, *isolated* environment. The malware's execution is paused after each instruction and its execution context is compared with the static code model, as shown in Step 2. When the first instruction of a sequence not found in the static model is detected, representations of that unknown instruction sequence are written out and the malware's execution is halted.

Our approach does *not* determine whether a program is an instance of malware, but rather *supplements* modern malware research efforts and detection techniques, which can use a program's unpacked code to perform a more complete or expedited analysis. The implementation of our technique, a tool we call *PolyUnpack*, can output a plain-text disassembly of the unpacked code, a binary dump of the code, or a complete executable version which can be loaded into popular analysis tools such as IDA Pro [4].

The work we describe reflects the following contributions:

- A formal description of unpack-executing programs and an algorithm for behavior-based hidden-code extraction.
- Motivated by our formal description, implementation of an extraction tool that can be used to supplement malware analysis and detection techniques. An interactive stand-alone version of our technique is available for download at http://polyunpack.cc.gt.atl.ga.us/polyunpack.zip.
- Implementation and use of a framework for testing our technique against large sets of malware. Benefits observed by evaluating the results from testing include the following:

  Use of our tool can *meaningfully assist* a malware researcher faced with unpack-executing malware by removing the need to perform extraction manually. PolyUnpack *automates* the process of extraction without requiring knowledge of how the malware unpacks its hidden-code. In testing, PolyUnpack was shown to extract the hidden-code from many hundreds of actual malware samples that exhibited a wide variety of unpack-execute behavior.

  Automated unpacking can be used to *enhance the accuracy of malware detectors* as illustrated with our experiments using ClamAV [10] and McAfee Antivirus [13]. When presented with the hidden-code from samples processed with PolyUnpack, both detectors were able to identify samples previously classified as benign. The results (shown in Section 6) demonstrate a good reduction in false negatives using PolyUnpack.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents a formal overview of unpack-executing programs and the extraction algorithm. Section 4 provides details for implementing the algorithm. Section 5 details the experimental framework constructed to test large sets of malware for unpack-execute behavior, while Section 6 presents the results of our testing. Section 7 briefly provides some concluding remarks.

2. Related Work

In [3], Christodorescu et. al. further the detection of malware to include models based on semantic behavior. Called *templates*, these models leverage the power of context-free grammars (CFGs) to automatically identify classes (rather than instances) of malware. In such a framework, a template could be created for detecting malware instances that contain unpack-execute behavior based on the semantic class of their unpacking mechanism; indeed, one of the templates presented serves to capture the decryption loop of a polymorphic worm. In the context of our focus, if a malware instance is matched to a template describing a particular unpacking mechanism, the code segment matched could be used to unpack and extract the instance's hidden-code. Unfortunately, however, even the power of CFGs do not provide a comprehensive mechanism for extracting the hidden-code of all unpack-executing malware. All the malware writer needs to do to successfully evade

detection is find a different semantic mechanism of unpacking for which a template has not yet been written, such as pulling malicious code from a network.

Among pattern-matching based extraction approaches, the program Portable Executable (PE) Identifier (PEiD) [9] stands out as a widely used tool for detecting binaries that exhibit unpack-execute behavior. PEiD uses a signature database to determine if a binary contains packed-code. If a signature match is found, knowledge of the identified packing mechanism can be used to unpack and extract the hidden-code contained in the binary. The key limitations of PEiD are identical to that of a pattern-matching anti-virus tool: its signature database must be updated for it to detect new unpack-executing binary instances and can fail to detect even minor variations of an otherwise known packing method in the same semantic class.

The closest industry work to ours is the Universal PE Unpacker plugin [17], available for IDA Pro 4.9. The plugin uses the behavioral heuristic that a program will return to its original entry point once it starts unpacking. Although a good approach for handling compression-based packing techniques, there exists a straightforward possibility of evasion. That is, any program that begins execution *in* its entry point area, transforms a small portion of data in one of its data sections, then directs execution to the data section it transformed would evade the plugin's detection heuristic. Using the results of testing our approach as described in Section 6, we discovered several hundred malware instances sampled in the wild that evade the IDA Pro plugin's heuristic. Our approach does not make the same assumption, and represents a more general solution to the problem of unpacking. As an additional confirmation, we corresponded with Ilfak Guilfanov, the author of IDA Pro; he agrees that our technique provides a new approach not addressed by the Universal PE Unpacker plugin [6].

Finally, anti-virus companies have made reference to the notion of malware-instance-independent code extraction, occasionally referring to its implementation as a *generic decryption engine* [12]. Just like the authors in [2], we found it impossible to obtain further details about the workings of commercial AV tools. In 2004, however, Christodorescu and Jha demonstrated in [2] that encapsulation techniques used in modern polymorphic viruses prevented commercial anti-virus products from detecting otherwise functionally identical variants of known malware. Given the closed nature of commercial AV products, investigation of the efforts employed by these companies is difficult, and the degree of success in their implementations remains an open question.

3. Formalizing Unpack-Execution

The idea for an approach to *automating* the process of extraction stems from the observation that a program attempting to hide its behavior through obfuscation does so because the code it hides is malicious. That is, if the code in the obfuscated block was available to a static analyzer, the program would be identified as malware. As such, the code contained in the compile-time obfuscated block is not expected to appear anywhere in the statically-identifiable instruction code portions of the program. This observation gives rise to a formal behavioral definition for capturing both encrypted and polymorphic malware, and a method for automating the process of extracting their hidden-code independent of the unpacking mechanism used.

To help correspond formalism with practice, a running example of an Intel 80x86-based unpack-executing program will be used to clarify the context.

3.1. Basic Program Definition

In modern computers, programs can be conceptualized as a composition of sets of ordered sequences of instructions and non-instruction data. Let a program $P = (I, D)$ be a two-tuple representation of these sets. The tuple $I = \{i_0, i_1, ..., i_n\}$ is a set of ordered sequences of instructions, where $i_k = (o_0, o_1, ..., o_j)$, $0 \leq k \leq n$ is a particular ordered sequence of instructions, and $o \in i_k$ is an instruction within that sequence. The set $D = \{d_0, d_1, ..., d_m\}$ is similarly defined as a set of ordered sequences of non-instruction data.

Define a conventional execution of P as an execution where P does not generate and execute new ordered sequences of instructions. At any time during a conventional execution, a special register called the *program counter*, or *pc*, will point to an $o \in i$ for some $i \in I$ (the 40xxxx numbers in Figure 2 correspond to the *pc* of each $o \in i$). Starting at o and incrementing the *pc* (non-uniformly) some t times will produce an ordered sequence $i_r = (o_0, o_1, ..., o_t)$, where i_r is a subsequence of one or more $i \in I$.

3.2. Execution-Time Data Transformations

During execution, P may have access to data outside of its own program data D; P may take in user input, read files on disk, open network connections, etc. Let this external set of data be $\{e_0, e_1, ..., e_l\}$, and define $E = \{d_0, d_1, ..., d_m, e_0, e_1, ..., e_l\}$ as the set of ordered sequences of data P can access when it is loaded and executed. From a semantic perspective, an ordered sequence of instructions i_p consisting of (possibly repeating) ordered instruction subsequences of members in the set I can represent a data transformation function π_p. Denoting an invocation of π_p in an execution of P to be $\pi_p(E)$, the function π_p uses members of E to produce transformed data d_p.

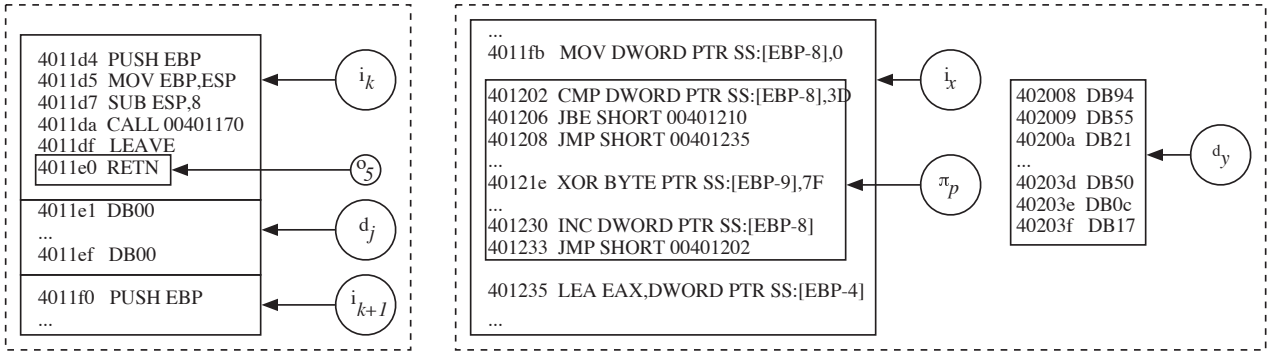

Figure 2: On the left, example program snippet partitioned into ordered instruction sequences i_k and i_{k+1}, non-instruction data sequence d_j; example of a single instruction $o_5 \in i_k$. On the right, a subsequence of ordered instruction sequence i_x comprises example data transformation function π_p. π_p operates over non-instruction data sequence d_y.

3.3. Behavioral Definition of a Unpack-Executing Program

Under a conventional execution of *P*, the result of a data transformation function, d_p, can be treated as only data, even if it represents a legitimate ordered sequence of instructions. Diverging from the model, however, introduces the possibility that d_p may indeed be executed. Assuming that d_p is a valid ordered sequence of instructions, this execution can occur from either the *pc* being incremented to point to an instruction $o \in \mathrm{d}_p$, or a control-transfer instruction (such as a call or jmp) o_{ct} in some $i \in I$ explicitly directing the *pc* to fetch its next instruction from d_p.

A sequence of execution-time generated instructions remains simply data if it is never executed. Combining this fact with the previous observations gives rise to a behavioral definition for an unpack-executing program.

Definition 3.1 A program *P* is said to be *unpack-executing* if, at some point during its execution, the *pc* points to *o*, an instruction in a to-be-executed instruction sequence i_p, where $\forall i \in I$, i_p is not a subsequence of *i*.

One immediate observation is that Definition 3.1 captures *both* simple unpack-execute and polymorphic classes of malware. Concisely, a polymorphic virus can vary its decryption method in subsequent generations, but such a mutation is orthogonal to the *results* of its unpack-execute behavior.

3.4. Algorithm for Unpacked Code Extraction

The unpacked code extraction algorithm EXTRACTUNPACKEDCODE described below operates by using the static code view of an input program *P* as a model of comparison when single-step executing *P*. Functionally, it serves as a technique for determining whether a given program exhibits behavior meeting the condition of Definition 3.1.

Posed as a decision problem, determining whether a program *P* will exhibit unpack-execute behavior when executed (subsequently, extracting its unpacked code) is *undecidable* through a reduction from HALT_{TM}; a proof is provided in Appendix A. In addition, an unknown malware instance may exhibit no unpack-execute behavior, meaning no amount of execution will yield runtime generated code. Therefore, in order to bring the problem of extracting unpacked code into the realm of decidability we introduce an additional input parameter *n* as the number of instructions of *P* to execute before halting.

Input: An input program *P* and instruction-execution bound *n*.
Output: An instruction sequence i_p representing runtime-generated code, or NIL if *P* halts without exhibiting unpack-execute behavior or the instruction-execution bound *n* is reached.

EXTRACTUNPACKEDCODE(*P*, *n*)
begin
// *Step 1: Static Analysis*
// Disassemble *P* to identify code and data. Partition
// blocks of code separated by non-instruction data into
// sequences of instructions i_0, ..., i_n. These sequences
// form the set *I* (the static code view). *I* will be
// repeatedly queried in the dynamic analysis step to
// detect if *P* is executing unpacked code.

I = DISASSEMBLE(*P*)

// *Step 2: Dynamic Analysis*
// Execute *P* one instruction at a time. Pause execution
// after each instruction and acquire the current

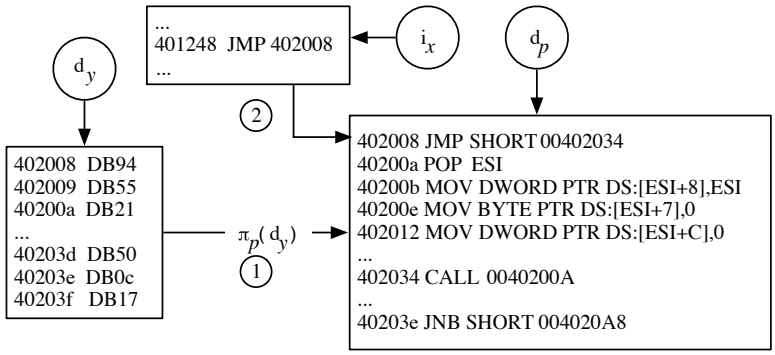

Figure 3: (1) Data transformation function π_p (shown in Figure 2) uses members of ordered data sequence d_y to create ordered instruction sequence d_p. (2) Later, a *jmp* control-transfer instruction in ordered instruction sequence i_x directs execution to d_p.

// instruction sequence by performing in-memory
// disassembly starting at the current value of the *pc*
// until non-instruction data is found. Compare the
// current instruction sequence with each instruction
// sequence in the set *I*. If the current instruction
// sequence is not a subsequence of any member of *I*,
// then it did not exist in the static code view of *P*
// (i.e., it is unpacked code being executed).

for 1 **to** *n* **do**
1. Execute an instruction of *P*.
2. Acquire the current *pc* of *P*, pc_{cur}.
3. Using pc_{cur}, perform in-memory disassembly until non-instruction data is found to acquire the instruction sequence i_{cur}.

if $\forall i \in I$, i_{cur} is not a subsequence of *i* **then**
return i_{cur} (the unpacked code)
done
return NIL
end

4. Implementing EXTRACTUNPACKEDCODE

Implementing the algorithm presented in Section 3 on actual hardware and operating systems requires care independent of the formalism motivating its implementation. This section details the implementation of the algorithm and where applicable, discusses additional considerations and the manner in which they were handled.

4.1. Hardware Platform and Operating System

We have implemented EXTRACTUNPACKEDCODE (i.e., PolyUnpack) as a command-line tool that operates over x86 Microsoft Windows executables (ideal for experimentation with large sets of malware). The tool uses software and hardware breakpoints, and Windows API calls to single-step execute a program; static and dynamic disassembly is performed using a 80x86 32-bit disassembler library [18]. For outputting the complete executable version of unpacked code found, we integrated the source of a debugged process memory dumper [5].

4.2. Dynamic Link Library (DLL) Calls

A naive step-through of any MS Windows binary execution will almost always result in the detection of code not found in the binary's static disassembly due to the presence of DLL calls. Stepping into a DLL call results in stepping through that DLL's code; such a case must be avoided. To prevent misidentification of DLL code, whenever a new DLL is loaded, PolyUnpack records the memory range the DLL occupies. During single-step execution, the program's *pc* is continually compared against all known memory areas. If the *pc* enters a region occupied by a DLL, PolyUnpack reads the return address from the stack and sets a breakpoint there, allowing single-step execution to resume after the call's return.

4.3. Enhancing Detection Accuracy and Speed

A straightforward implementation of the EXTRACTUNPACKEDCODE algorithm described in Section 3 would need to overcome significant obstacles in order to be viable. In terms of performance, efficiently testing whether the current instruction sequence exists in the static code view of the entire binary after the execution of every instruction requires considerable effort. More importantly, the implementation would have to mitigate threats stemming from the nature of 80x86 assembly, structural properties of the MS Portable Executable (PE) format, and the degree to which the MS Windows OS enforces the PE specification. Key

challenges arising from these threats include the ability to perform accurate disassembly and relatedly, the successful separation of code and data.

In the 80x86 Instruction Set Architecture (ISA), assembly instructions are of variable length, and a compiler or assembler which targets this platform can mix instructions and data together. These properties create an inherent problem of performing correct disassembly given the inability of a disassembler to statically determine the outcome of some 80x86 instructions (such as indirect branches). Although recent approaches have been proposed to significantly enhance the accuracy of disassembling x86 binaries [11], incorrect disassembly of even one instruction could cause EXTRACTUNPACKEDCODE to falsely report the existence of unpack-execute behavior.

Besides the need to accurately disassemble known code sections of an x86 binary, the manner in which MS Windows handles the execution of a program introduces the possibility that *non-code* regions must also be examined. To elaborate, while the PE header describes whether each program section is readable, writable, or executable, only the readable and writable flags are enforced. As a consequence, there may exist non-obfuscated code in sections marked non-executable, including the PE header itself. As a program can either begin or immediately direct execution to these regions, a faithful implementation of the algorithm should correctly *tag* them as code. However, accurate identification (and subsequently, disassembly) of these *extra* code sections can be exceptionally difficult given the aforementioned properties of the 80x86 ISA.

To overcome the disassembly and identification challenges required for implementing EXTRACTUNPACKEDCODE, the problem of detecting the execution of unpacked code via instruction subsequence existence can be mapped into a series of statically assigned and dynamically created bounds checks that test whether the current value of the *pc* points to a location statically or dynamically identified as code. Applying this observation was critical in implementing an *efficient*, *accurate* version of EXTRACTUNPACKEDCODE that emits no false positives in its search for unpack-execute behavior.

4.4. Evasion

PolyUnpack, like most instrumentation tools, is not transparent to the malware being processed. Therefore, there exists the possibility that an instance of malware being executed in PolyUnpack may detect that it is being instrumented and alter its behavior (e.g., halting its execution instead of generating hidden-code) in order to evade extraction of its unpacked code. While there is no comprehensive solution to identifying *all* attempts by malware to detect the presence of instrumentation tools short of implementing an entire virtualized environment, preventing common-case evasion attempts do not require difficult-to-implement solutions. As an example, in our implementation we unset a bit in the *thread information block* (TIB) which indicates that the program is being instrumented or debugged. This modification causes calls to IsDebuggerPresent() and its assembly equivalents to return false.

Finally, the decision to attach an instruction-execution bound n in EXTRACTUNPACKEDCODE, although bringing the problem of detecting unpack-execute behavior into the realm of decidability, introduces the possibility that a program's execution will be halted before it begins executing unpacked code. One approach to minimizing the risk of prematurely terminating a malware's execution is to select sufficiently large values for the instruction execution-bound n and testing multiple malware instances simultaneously (each in its own isolated environment).

4.5. Multiple Unpacking

Some instances of unpack-executing malware further complicate the process of extracting their unpacked code by having the unpacked code perform additional unpacking. The primary consequence of processing malware that uses this technique is that the (partially unpacked) code extracted contains code yet to be unpacked. In addition, the portion of still-obfuscated code may itself perform unpacking once unpacked; we refer to this behavior as *multiple unpacking*.

For instances of malware that perform multiple unpacking, PolyUnpack can be used to acquire the innermost body of unpacked code by leveraging its ability create a complete executable version of the hidden-code extracted. In this scenario a version of the malware being instrumented is first written out; execution in the new binary is changed to begin at the first instruction of the unpacked code. The new binary can then be tested for unpack-execute behavior in the exact same manner as the original malware instance. This two-step process is performed until some k^{th} version of the executable produces no unpacked code; this version represents the final body of the unpacked code.

5. Experimentation Framework

Security organizations such as Anti-Virus (AV) companies are faced with a flood of new samples being submitted by users, sensors, honeypots, and mail filters. The volume of the samples received makes manual analysis and reverse engineering of unpack-executing malware a very time-consuming and laborious task. In such an environment, the primary goal of PolyUnpack is to determine whether a given sample exhibits unpack-executing behavior and if so, to automatically extract its hidden-code for use with existing analysis techniques.

To evaluate how well PolyUnpack assists malware reverse engineering and analysis we would ideally have access to the inner workings of several AV company labs. However, because of trade secrets and the highly competitive nature of the AV industry, this type of access was simply not possible. Most AV companies will not even discuss how their commercial tools work in any detail, much less the operation of in-house tools built for Research and Development (R&D).

5.1. Malware Analysis System Design

To overcome the inaccessibility of commercial AV lab resources we created a simple malware analysis system to evaluate PolyUnpack. The workflow of this system consists of the arrival of a sample, its initial classification, exportation of unidentified samples to a robot machine farm, and re-analysis using the results of hidden-code extraction. This entire process occurs in a pipelined fashion, as shown in Figure 4.

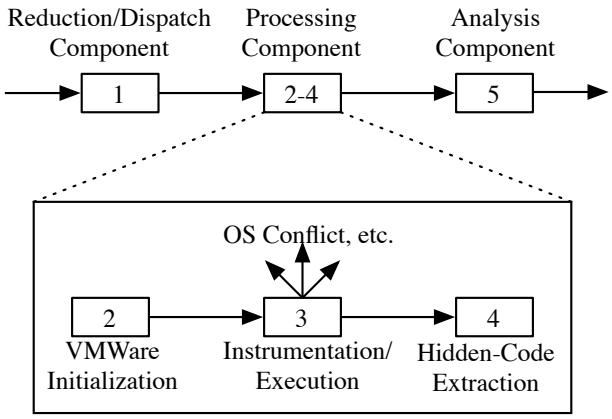

Figure 4: Workflow used to process samples.

Starting with Step 1, a sample arrives from one of various feed sources. It is analyzed by malware detectors and its MD5 value is computed and checked against previously seen instances. In a production environment no further processing occurs if it is found to be malicious or if a previously analyzed sample with the same MD5 is found. This step comprises the reduction component, which eliminates unnecessary work.

If the sample remains unidentified it is sent to one of several machines on a robot farm. The robot starts a virtual machine (VM) that has been configured to make the sample available to the VM environment and then instruct PolyUnpack to begin executing it (Steps 2, 3). If PolyUnpack detects the sample exhibiting unpack-execute behavior it writes out a plain-text disassembly, binary dump, and complete executable version of the unpacked code and then halts the sample's execution (Step 4). This information is then sent to Step 5 (the analysis component), where the complete executable version of the unpacked code is analyzed by malware detectors and its MD5 value is calculated.

If the sample's unpacked code is not identified by malware detectors or MD5 value it is forwarded to a human analyst for further analysis. Similarly, samples that terminate without exhibiting unpack-execute behavior, which cannot run in the guest OS of the virtual environment, or which exceed a preset time limit (i.e., the instruction execution bound) when being executed by PolyUnpack are also forwarded.

5.2. Implementation

The layout of the malware analysis system consisted of a cluster of three machines (the robot machine farm) running Linux and VMWare Server and one Linux machine running a small daemon we created to handle the workflow process. Each robot has a virtual machine with Windows 2000 installed and is configured to start in a state immediately before processing a sample using PolyUnpack. The start state (called a *snapshot*) represents a *paused* copy of the system, which includes the contents of disk, memory, and the CPU state. Using snapshots, we can save considerable time by not needing to reboot the virtual machine for each new sample. Relatedly, returning to the state of the snapshot discards all changes (e.g., disk writes) made by the previous sample, allowing us to provide the same *sterile*, *isolated* environment for each sample.

When a sample arrives at the workflow handler's input queue, the daemon dispatches the sample to a robot and issues a command to start its VM. After being loaded from its snapshot state, the Windows 2000 OS mounts a network share (where the sample has been placed) and directs PolyUnpack to begin single-step executing it. If the sample exhibits unpack-execute behavior, PolyUnpack writes out versions of its unpacked code and processing information to the network share, halts the sample's execution, and directs the daemon to terminate the VM session. After stopping the VM session, the daemon harvests information from the directory corresponding to the network share, processes and archives it, then dispatches the next sample to the (now available) robot machine.

While single-step executing a sample, PolyUnpack periodically sends status messages via the network to the workflow daemon. The absence of three such messages implies that the sample may have frozen or crashed PolyUnpack or the Windows OS and will trigger the daemon to terminate the VM session. Additionally, the daemon places a time limit (in our implementation, four hours) on the running time of each VM session, which is the implementation's equivalent of the instruction execution bound described in Section 3. If this limit is exceeded the daemon halts the

virtual machine regardless of its current state.

6. Evaluation

6.1. Preliminaries

To provide input to the malware analysis system we acquired 3,467 samples from the OARC [8] malware suspect repository. Its contents are semi-public and available to qualified academic and industry researchers upon request. The samples it contains were captured in the wild from September 2005 to January 2006 by mail traps, user-submissions, honeypots and other sources aggregated by the OARC; each sample is a unique binary according to its MD5 value. While we were able to confirm that each of the samples tested is indeed malware, their real-world nature made it difficult to determine the exact percentage of what samples are packed or unpacked.

6.2. Hidden-Code Extraction

For evaluating the ability of PolyUnpack to successfully extract hidden-code from malware without knowing the ground truth of which samples were packed, we compared its performance to that of PEiD. As mentioned in Section 2, PEiD is a popular and often-used reverse engineering tool that uses a highly specific set of signatures to identify whether a binary will exhibit unpack-execute behavior. It does not unpack a sample, but simply tries to identify what packing tool was used.

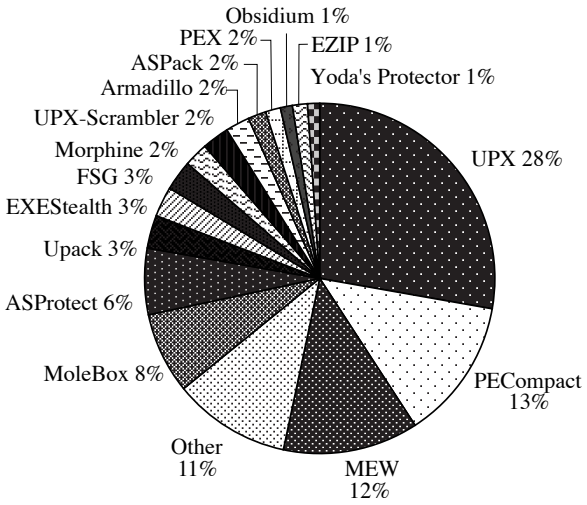

Figure 5: Distribution of code packing tools used in samples PolyUnpack successfully processed.

Using the malware analysis system, PolyUnpack found 1,754 samples to be unpack-executing and extracted their hidden-code. In contrast, PEiD identified only 1,482, which suggests that PolyUnpack performs competitively well. Figure 5 displays a breakdown of the well-known tools used to create hidden-code in the samples PolyUnpack successfully processed. These results show that PolyUnpack is indeed capable of extracting unpacked code in a *obfuscation-independent* fashion (i.e., without needing any knowledge of how the runtime code is generated) over a wide variety of real-world malware.

6.3. Processing Time

Analysts have reported that manually unpacking a given malware instance takes between 15 and 60 minutes [15]. With hundreds of suspect samples arriving each day, there exists a clear need to create efficient tools that assist in streamlining the process of unpacking. To determine the efficiency of PolyUnpack, we recorded the processing time (including the 30 seconds for VMWare's startup and shutdown) for each sample. Of the samples from which PolyUnpack extracted unpacked code, the average time was 1,020 seconds, or less than 20 minutes; over 60% took less than five minutes. These results suggest that PolyUnpack is capable of processing samples in an automated fashion *without* sacrificing efficiency, and can save malware researchers considerable amounts of time by removing the need to perform extraction manually.

6.4. Post-Extraction Code Duplicates

In order to determine the rate of unpacked code duplicates, we began with the complete executable versions of hidden-code extracted by PolyUnpack. We grouped these instances by their MD5 value to acquire the total number of *unique* instances; there were 1,260. Therefore, there were 494 duplicates, which accounted for 28% of the total set of unpacked codes. This result indicates that there are a non-trivial percentage of duplicates, and that PolyUnpack can help prevent time spent manually unpacking a sample that appears to be different, but once unpacked, is identical to a sample previously processed.

6.5. AntiVirus Detections

As each OARC binary sample went through the analysis system, it was scanned with up-to-date AV tools ClamAV and McAfee Antivirus to identify known malware instances (for testing purposes, a sample was not discarded if it was found to be malicious). ClamAV identified 2,746 of the samples as malware, while McAfee identified 3,138. Among the samples deemed benign by ClamAV, PolyUnpack found 252 to be unpack-executing. For McAfee, 83 of

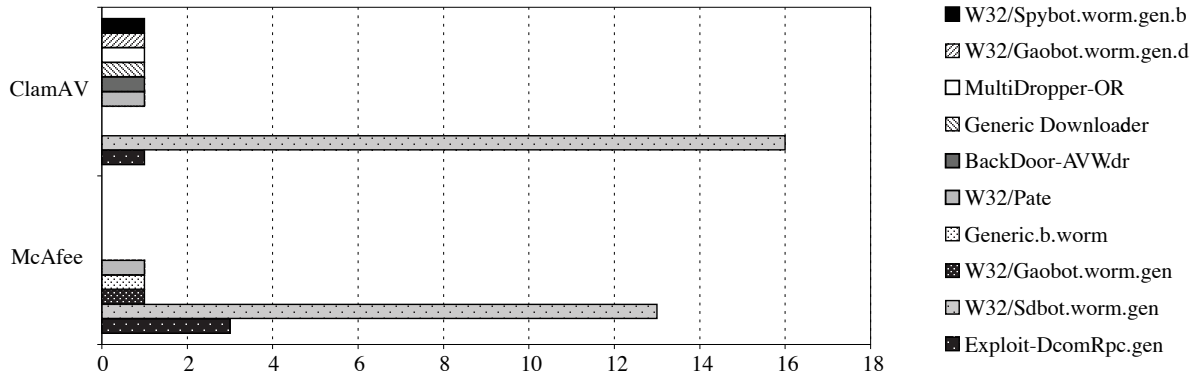

Figure 6: Distribution of malware samples found to be benign initially, but malicious after unpacking.

the samples it identified as harmless were found to contain hidden-code.

When the complete executable versions of the unpacked code were presented to ClamAV, an additional 23 were successfully identified as malware. For McAfee, an additional 19 were identified. To elaborate, in packed form, these instances evaded both signature detection and heuristic analysis of the AV tools. Once unpacked, however, they were successfully detected; Figure 6 shows a grouped distribution of these samples by malware type. Relative to the number of benign samples for each tool found to be unpack-executing, this result shows that hidden-code extraction provided false negative reductions of 9.1% and 22.8% for ClamAV and McAfee, respectively.

The above results suggest ways to improve AV tools. PolyUnpack is able to automatically identify and extract unpacked code because it executes a given binary in an isolated, virtual environment. Provided a similar solution is implemented with care, we believe client-side AV tools can do the same to improve detection rate. That is, rather than just statically scanning the binaries on a client computer, an AV tool could run a suspect binary in a small, dedicated virtual environment on a local computer using a technique similar to that used by PolyUnpack, then perform additional analysis on its output (unpacked code). Alternatively, to reduce overhead on a client computer, the client-side AV tool could instead send the suspect binary to a central AV server, which can then run it in a virtual environment and send the results back to the client computer for further analysis.

7. Conclusion

The analysis and detection of malware that hides malicious code as data can be a very time-consuming and challenging task. In this paper we have described PolyUnpack, an approach to automatically identifying and extracting the hidden-code bodies of unpack-executing malware. Our approach is based on the observation that sequences of packed or hidden code in a malware instance can be identified when its execution is checked against its static code model.

We have derived from this observation a formal definition for the unpack-executing behavior of a program. We have implemented our hidden-code extraction algorithm as PolyUnpack, a command-line tool on the x86 MS Windows platform. We have evaluated PolyUnpack using more than 3,400 known malware binaries. The results of our experiments showed that PolyUnpack identifies more unpack-executing malware than PEiD, a popular tool for identifying unpack-executing programs. Our results also showed that PolyUnpack can perform extraction efficiently, and can be incorporated into a malware analysis workflow to achieve significant automation, yielding savings of both time and effort. Finally, our results demonstrated that PolyUnpack (or rather, the idea and algorithm behind it) can be used to improve the performance of malware detection tools.

7.1. Future Work

In future work, we will improve our ability to remove duplicate samples. Currently, we remove duplicates using MD5 signatures. This technique is simple and effective, but may still result in double counting *functionally identical* samples. We expect to achieve better reductions using graph flow analysis (e.g., [1]), and heuristics to identify programs that are functionally the same.

Finally, although we briefly addressed making instrumentation transparent to an instance of malware being processed, malware can also evade through detection of the virtualized environment. For example, virtualization problems with some instructions in the x86 architecture [14] provide simple ways for a program to detect that is running inside a VM. In future work, we intend to investigate techniques (such as clever application of virtualization hardware extensions) for making the virtualized environment more transparent to programs that may check for a VM's presence.

Acknowledgements. This work is supported in part by NSF grant CCR-0133629 and Office of Naval Research grant N000140410735. The authors would like to thank H. Venkateswaran, Monirul Sharif, and Richard Hoelsher for their advice and feedback in the creation of this paper.

References

[1] E. Carrera and G. Erdélyi. Digital genome mapping: Advanced binary malware analysis. *Virus Bulletin*, 2004.

[2] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.

[3] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, May 2005.

[4] DataRescue, Inc. The ida pro disassembler. www.datarescue.com/idabase, 2006.

[5] Gigapede. OllyDump v. 3.0, 2006.

[6] I. Guilfanov. Personal correspondence. January 30 2006.

[7] C. Hoepers. Phishing's Cutting Edge: Brazil and the Future of Phishing. www.antiphishing.org/events/apwg_nov_05_montreal.html, 2005.

[8] Internet System Consortium. Isc oarc. oarc.isc.org, 2006.

[9] Jibz, Qwerton, snaker, and xineohP. PEiD. peid.has.it, 2005.

[10] T. Kojm. Clam antivirus. www.clamav.net, 2006.

[11] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security 2004*, pages 255–270, 2004.

[12] McAfee, Inc. Advanced virus detection scan engine and dat. www.mcafee.com/us/local_content/white_papers/wp_scan_engine.pdf, 2002.

[13] McAfee, Inc. Antivirus. www.mcafee.com, 2006.

[14] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[15] R. Russell. Personal correspondence. January 29 2006.

[16] P. Szor. Bad idea. *Virus Bulletin*, pages 18–19, April 1998.

[17] P. Vandevenne. Using the universal PE plug-in in IDA Pro 4.9 to unpack compressed executables. www.datarescue.com/idabase/unpack_pe/unpacking.pdf, 2005.

[18] O. Yuschuk. 80x86 32-bit disassembler and assembler. www.ollydbg.de/srcdescr.htm, 2006.

A. Undecidability of Detecting Unpack-Execution

A standard computer program P has at least one defined (immutable) *code section* and can also have one or more (mutable or immutable) *data sections*. In addition, while running P can write data to one of these sections and *then* direct execution to what it wrote. Formally, this ability makes P a *universal Turing machine* (UTM), which can simulate other Turing machines it reads (or writes) on its input tape.

Key structural equivalencies between programs and UTMs are listed below.

- The immutable code section of a program corresponds to the immutable control states of a UTM.
- The mutable data section(s) of a program corresponds to the mutable input tape(s) of a UTM.

The key functional equivalence is as follows.

- A program directing its execution to one of its data sections (exhibiting unpack-execute behavior) corresponds to a UTM running a Turing machine on its input tape.

Based on these equivalencies, we formally define the problem of determining whether a universal Turing machine M simulates a Turing machine on its input tape (i.e., whether a program exhibits unpack-execute behavior) as follows.

Definition $\text{UNPACKEX}_{TM} = \{\langle M, w\rangle \mid M \text{ is an UTM and } M \text{ simulates a Turing machine on its input tape in its computation of } w\}$.

Theorem UNPACKEX_{TM} is undecidable.

Proof: We will prove that UNPACKEX_{TM} is undecidable by describing a mapping reduction which shows that $\text{HALT}_{TM} \leq_m \text{UNPACKEX}_{TM}$ Define a function f that takes as input $\langle M, w\rangle$ and outputs $\langle M', w'\rangle$, where $\langle M, w\rangle \in \text{HALT}_{TM}$ if and only if $\langle M', w'\rangle \in \text{UNPACKEX}_{TM}$. The following machine F computes f.

F="On input $\langle M, \text{w}\rangle$, a valid encoding of a Turing machine M and input string w:

1. Construct a Turing machine T.
  T="On input x:
    1. Ignore x and halt."
2. Construct the following UTM M' from M.
  M' is the same as M, except:
    for all $q \in Q$, $\gamma \in \Gamma$
      if $\delta(q, \gamma)$ goes to a halting state (that is, (if $\delta(q, \gamma) = (q_{\{accept,reject\}}, -, -)$) then
        Replace this transition with one that begins simulation of T on the input tape. That is, change the transition to $\delta(q, \gamma) = (q_{start,T}, -, -)$.
3. Output $\langle M', \langle T, w\rangle\rangle$."

The output of F, a UTM M', will execute a Turing machine T in all and only those cases where M will halt on w. A decider for UNPACKEX_{TM} could decide if M' will execute T and therefore decide HALT_{TM}. But HALT_{TM} does not have a decider, and so a decider for UNPACKEX_{TM} cannot exist. Therefore, UNPACKEX_{TM} is undecidable. □