A Study of the Packer Problem and Its Solutions

Fanglu Guo Peter Ferrie Tzi-cker Chiueh

Symantec Research Laboratories

Abstract. An increasing percentage of malware programs distributed in the wild are packed by packers, which are programs that transform an input binary's appearance without affecting its execution semantics, to create new malware variants that can evade signature-based malware detection tools. This paper reports the results of a comprehensive study of the extent of the packer problem based on data collected at Symantec and the effectiveness of existing solutions to this problem. Then the paper presents a generic unpacking solution called Justin (Just-In-Time AV scanning), which is designed to detect the end of unpacking of a packed binary's run and invoke AV scanning against the process image at that time. For accurate end-to-unpacking detection, Justin incorporates the following heuristics: Dirty Page Execution, Unpacker Memory Avoidance, Stack Pointer Check and Command-Line Argument Access. Empirical testing shows that when compared with SymPack, which contains a set of manually created unpackers for a collection of selective packers, Justin's effectiveness is comparable to SymPack for those binaries packed by these supported packers, and is much better than SymPack for binaries packed by those that SymPack does not support.

1 The Packer Problem

1.1 Overview

Instead of directly obfuscating malware code, malware authors today heavily rely on packers, which are programs that transform an executable binary into another form so that it is smaller and/or has a different appearance than the original, to evade detection of signature-based anti-virus (AV) scanners. In many cases, malware authors recursively apply different combinations of multiple packers to the same malware to quickly generate a large number of different-looking binaries for distribution in the wild. The fact that more and more malware binaries are packed seriously degrades the effectiveness of signature-based AV scanners; it also results in an exponential increase in AV signature size, because when an AV vendor cannot effectively unpack a packed threat, it has no choice but to create a separate signature for the threat.

The percentage of malicious programs (malware) and benign applications (goodware) that are packed is hard to measure accurately. The numbers reported vary from vendor to vendor, but it is generally accepted that over 80% of malware is packed. These malware samples are often "wrapped" rather than packed,

because many packers alter the original form of input binaries in ways that don't necessarily involve compression.

A much smaller percentage of goodware is packed. We took a random sample of tens of thousands of executable files that were collected over a period of several months and were packed by packers that Symantec recognizes and knows how to unpack, and ran a set of commercial anti-virus (AV) scanners from multiple vendors against them. About 65% of these executable files are known malware. The remaining 35% most likely falls into the goodware category because these samples were collected more than a year ago and today's AV scanners should be able to capture most malware programs during that period of time. Clearly, not only are most malware samples packed, but the use of packers to protect goodware is quite common.

The number of known packers, both good and bad, is also hard to measure accurately. Symantec has collected a large number of packers - more than 2000 variants in more than 200 families. Among them, Symantec currently can identify the unpacker code in nearly 1200 packers spread among approximately 150 families. However, among the 150 packer families Symantec knows about, it only has the packer code for about 110 of them, which contain approximately 800 members. This means that Symantec has a backlog of approximately 1200 members in 90 families, and this number increases day by day.

Without doubt, UPX [9] remains the most widely used packer. The rest of the list depends on how files are collected, but it always includes the old favourites like ASPack [1], FSG [2], and UPack [4]. In addition to those known packers, analysis of the above randomly sampled file set revealed at least 30 previously unknown packers. Some were minor variations of known packers, but most were custom packers. Amazingly, some of the clean files were packed with these custom packers.

The typical way an AV vendor such as Symantec handles packers involves the following steps:

1. Recognize a packer. To recognise a packer is to assign it to a packer family. This is not as simple as it sounds. There are plenty of packers whose code is constant, and these can be recognized using simple strings. But many packers use polymorphic code to alter their appearance, and some packers intentionally use fake strings from other packers or standard compiler code, in order to fool the recognizers.
2. Identify a packer. This step goes beyond recognition. To identify a packer is to classify it into an existing version or assign it a new version. Being able to identify a packer is essetial for successful unpacking, because there can be enough variations among members of the same family such that an unpacker for one member of a family cannot be used for another member for the same family.
3. Create a recognizer. The previous two steps are usually handled by a human, or applications such as neural net programs that have been trained on packers that are assigned to known families. This step, in contrast, is the act of

writing a program whose function is solely to recognise that family, and perhaps that particular member.
4. Create an unpacker. Unlike the recognizer, whose goal is just to recognize the packer, the unpacker actually performs the reverse actions of the corresponding packer, and restores a packed binary as much as possible to its original form, including its metadata such as PE header for Win32 binaries.

It requires a non-trivial amount of efforts to develop packer recognizers and unpackers. As noted above, Symantec has a backlog of approximately 1200 members in 90 families. To add unpacking support for a typical packer takes about six hours, on average. This means that it would take five full-time engineers about six months to clear the backlog if two unpackers are developed per day. However, in the case of complex packers such as Themida [10], it alone may take an experienced engineer up to six months to develop its unpacker. Packers with this level of complexity are not rare.

1.2 How Packers Work

Let's start with UPX, which arguably is among the most straightforward packers in use today. When UPX compresses a PE binary, it begins by merging all of its sections into a single section, with the exception of the resource section. Then a reformatted copy of the original binary's import table is reformatted is combined with the original binary's data section and PE header. The combined data is then compressed into a single section of the resulting packed binary. The original PE header enables UPX to optionally reverse the compression operation and produces an uncompressed file that is byte-for-byte identical to the original never-compressed file.

After UPX compresses a PE binary, it outputs a new PE binary containing three sections. The size of the first section is equal to the original host image size, less an amount that depends on the compressibility of the original binary's data - the less compressible are the data, the smaller the section will be. The reason for this is because UPX allows earlier decompressed data to be overwritten during later decompression. So, if more bytes are needed to decompress a block, the more bytes won't be needed later.

This first section is entirely virtual - it contains has no physical data and is simply a placeholder for the decompressed data. The second section contains the compressed data, followed immediately by the unpacker code. The entry point in the PE header is changed to point directly to this unpacker code. The third section contains the resource data, if the original binary had a resource section, and a partial import table, which contains some essential imports from `kernel32.dll` as well as one imported function from every DLL used by the original binary.

The first two sections are set permanently to read/write/execute, regardless of what they were before, and are not changed at run time. Therefore UPX is NX compatible, but it loosens up the protection for the original binary's read-only

sections. The third section is simply set to read/write, since no execution should happen within that section.

After unpacking, UPX write-enables the header of the resulting binary, then changes the first two sections to read-only, and write-protects the header again. This ensures compatibility with some application programs that check in-memory section tables instead of the actual section attributes, because these sections are supposed to be writable.

More sophisticated packers use a variety of techniques that virus writers use to defeat attempts to reverse-engineer, bypass, and disable the unpackers included in packed binaries. We discuss some of them in the following.

Multi-layer packing uses a combination of potentially different packers to pack a given binary, and makes it really easy to generate a large number of packed binaries from the same input binary. In practice, packed binaries produced by some packers may not be packed again by other packers. Also, the use of multi-layer packing itself could be used as an indication of malware, so the very presence of multiple layers - supported or not - could allow for a heuristic detection.

Anti-unpacking techniques are designed to make it difficult to uncover the logic of an unpacker, and fall into two major categories: passive and active. Passive anti-unpacking techniques are intended to make disassembly difficult, which in turns makes it difficult to identify and reverse the unpacking algorithm. Active techniques are intended to protect the running binary against having the fully unpacked image intercepted and extracted, and can be further classified into three subcategories: *anti-dumping*, *anti-debugging*, and *anti- emulating*. There are several commercial packers, such as Enigma and Themida, which promote their use of all of these techniques.

The simplest way to capture an unpacked image is to dump the address space of a running process. The simplest form of anti-dumping involves changing the value of the image size in the process environment block, and makes it difficult for a debugger to attach to the process or to dump the correct number of pages. More advanced anti-dumping methods include page-level protections, where each page is packed individually and unpacked only when accessed. It can even be packed again afterwards. This technique is used by packers such as Armadillo [11]. Shrinker [3] uses a variation of this method, by unpacking regions when they are accessed, but it is perhaps for performance reasons rather than an anti-dumping mechanism, since the unpacked pages remain in memory.

A very common way to capture an unpacked image is to use a debugger to step through the code, or to set breakpoints at particular locations. Two common forms of anti-debugging involve checking some values that the operating system supplies in the presence of a debugger. The first uses a public API, called `IsDebuggerPresent()`, which returns a Boolean value that corresponds to the presence or absence of a debugger. This technique is defeated by always setting the value to FALSE. The second anti-debugging technique checks if certain bits are set within the `NtGlobalFlag` field. The values of interest are heap tail checking (0x10), heap free checking (0x20), and heap pa-

rameter checking (0x40). They get their values from the `GlobalFlag` field of the `HKLM/System/CurrentControlSet/Control/Session Manager` registry key. A debugged process always has these values set in memory, regardless of the values in the registry. This technique can be defeated by clearing the bits in the process environment block.

Another way to capture an unpacked image to use an emulator to execute it in a protected environment. There are many ways to attack an emulator. The most common is to attempt to detect the emulator, since it is very hard to make an emulator whose behaviour matches closely to real hardware. However, each emulator has different capabilities, so there are multiple methods to detect different emulators [5].

Not all protection methods restore the host to its original form when executed. In particular, wrappers such as VMProtect [13] replace the host code with byte-code, and attach an interpreter to execute that byte-code. The result is that the original host code no longer exists anywhere, making it hard to analyse and essentialy impossible to reverse. In addition, the byte-code have different meanings in different files. That is, the value 0x01 might mean "add" in one VMProtect-packed binary, but "xor" in another, and only the corresponding embedded interpreter knows for sure.

2 Unpacking Solutions from the Anti-Virus Industry

The AV industry has developed several approaches to tackle the packer problem, which satisfy different combinations of the following requirements:

- Effective: An ideal unpacker should restore packed binaries to their original form.
- Generic: An ideal unpacker should cover as many different types of packed binaries as possible.
- Safe: Execution of an ideal unpacker should not leave any undesirable side effects.
- Portability: An ideal unpacker should be able to run on multiple operating systems.

The first requirement enables existing signature-based AV scanners to be directly applied to an unpackers output and detect the embedded malware if applicable. The second requirement decreases the amount of efforts required to keep up with new packers. The third requirement is crucial for at-rest file scanning, where the AV scanner initiates the unpacker and therefore has to be absolutely sure that the unpacker itself does not cause any harm. The final requirement is relevant for in-network scanning, where the unpacker and the AV scanner may need to run on different platforms than that required by the packed binaries.

The first solution to the packer problem is to manually create unpackers by reverse-engineering the unpackers in packed binaries. The SymPack library [12] from Symantec falls into this category. This solution is safe, portable, largely

effective but not generic. That is, one needs to develop a packer recognizer and an unpacker for each distinct packer. Given a set of packer recognizers and unpackers, one can classify packed binaries into four caregories: (1) packed binaries whose packer can be recognized and that can be unpacked, (2) packed binaries whose packer can be recognized but that cannot be unpacked, (3) packed binaries whose packer cannot be recognized, and (4) non-packed binaries. Assuming all the packers that goodware programs use fall into the first category, then one can black-list all packed binaries belonging to the second and third category. To be able to distinguish between packed and non-packed binaries, one needs a technique to detect packed files generically. This can be done by, for example, calculating the entropy [**?**] of a particular region of an executable binary that most likely contains compressed data.

However, this general approach of handling packed binaries has several problems. First, it entails significant investments in engineering efforts, and the level of investment required is expected to increase over time as more packers appear in the wild. Second, if goodware decides to use packers in the second and third category above, false positives in the form of blocking legitimate goodware may arise. The same problem may also occur when packer recognizers and unpackers contain design or implementation bugs that, for example, treat variants of packers used by goodware as unknown packers. Finally, this approach requires continuing maintainence for existing packer recognizers and unpackers. Old packers never die. They just get rediscovered and reused, and never quite go away. For example, the self-extractor stub for RAR - the world's second-most popular archiving format after ZIP - is packed by UPX v0.50, which dates from 1999.

The second solution to the packer problem is to run a packed binary inside an emulator for a sufficiently long period of time so that the embedded binary is fully unpacked, and then invoke signature-based AV scanners against the memory image to check if it contains any malware. XXX [**?**,**?**] take this approach. This solution is safe, portable and generic, but is not always effective for two reasons. First, a packed binary can terminate itself before the embedded binary is unpacked if it detects that it is running inside an emulator. Second, so far there is no good heuristic to decide when it is safe to stop the emulation run of a packed binary, because it is difficult to distinguish between the following two cases: (1) the embedded binary is benign and (2) the embedded binary is malicious but is not fully unpacked. Another disadvantage of this approach is that it takes a non-trivial amount of effort to develop a high-fidelity and high-performance emulator.

The third solution to the packer problem is to invoke AV scanning against a suspicious running process memory image either periodically or at certain security-sensitive events. Symantecs Eraser dump [**?**] takes this approach. This solution is generic, somewhat effective, but neither safe nor portable. Its effectiveness is compromised by the facts that certain information required by AV scanners, such as the main entry point, is not available, and that the memory image being scanned is not the same as that of an embedded binary immediately

after it is loaded. For example, a malware program may contain an encrypted string in its binary file, and decrypt it at run time. If the encrypted string is part of its signature, periodic memory scanning may fail to detect the malware because the encrypted string is no longer in its memory image.

OllyBone[6] is a semi-automatic unpacking for IA-32 packed binaries that is designed to track pages that are written and then executed at run time by overloading the user/supervisor bit and exploiting the separation of data TLB and instruction TLB in the X86 architecture. Saffron [**?**] combines OllyBones technique with Intels PIN to build a tool that detects control transfers to dynamically created or modified pages, and dumps memory images at that time.

Omniunpack [7] also relies on OllyBone for identifying executed pages and invokes AV scanning before every dangerous system call, which modifies the persistent system state, rather than periodically. In addition, it incorporates two additional optimizations to reduce the total number of AV scans. First, it invokes an AV scan only when there is a control transfer to a dynamically modified page between the previous and current dangerous system calls. Second, whenever an AV scanner is invoked, it only scans those pages that are modified since the last dangerous system call. Omniunpack is generic and largely effective, but neither safe nor portable. In particular, the fact that it requires whole-binary scanning is incompatible with almost all existing commercial AV scanners, which scans only a selective portion of each binary. Moreover, it only works for running processes, but is not suitable for at-rest file scanning, for the same reason as the emulator-based solution described above.

One common heuristic shared among OllyBone, Saffron, Omniunpack and Justin (descibed in the next section) is that a necessary condition of the end of unpacking is a control transfer to a dynamically created or modified page. However, there are important differences between Justin and these previous efforts. First, Justin includes a more complete set of heuristics to detect the end of unpacking, including stack frame, unpacked code region make-up, and exception sequence. Second, Justin includes several counter-measures that are designed to fend off evasion techniques that existing packers use. Finally, Justin leverages NX support rather than overloads the supervisor/user bit, and is more portable across different versions of the Windows OS.

3 Justin: Just-in-Time AV Scanning

3.1 Design

Justin is designed to be generic, effective and safe, but is not portable. The key idea of Justin is to detect the end of unpacking during the execution of a packed binary and invoke AV scanning at that instant. In addition to triggering AV scanning at the right moment, Justin also aims to provide the AV scanner a more complete picture about the binary being scanned, specifically its main entry point.

A packed binary logically consists of three components, the unpacked, the packed binary, and the area to hold the output of the unpacker. Different pack-

ers arrange these components into one, two or three PE sections. The section containing the unpackers output typically is relatively easy to identify because its reserved size is larger than that of its initialized data contained in the binary.

The initial design goal of Justin is to enforce the invariant that no code page can be executed without being scanned first. Its design is relatively straightforward: it first scans a packed binary at load time, runs the binary, keeps track of pages that are dynamically modified or created, and scans any such page when the programs control is transferred to it. This design relies on an AV scanner that does whole-binary scanning, and is thus not compatible with existing commercial AV scanners, which employ a set of heuristics (e.g., file type) to select a portion of an unknown binary and scan only bytes in that portion.

To work with commercial AV scanners, the design goal of Justin is shifted to detecting the end of unpacking during the execution of a packed binary. In addition, it makes the following two assumptions about packers: (1) The address space layout of the program embedded within a packed binary after it is unpacked is the same as that if the program is directly loaded into memory, and (2) the unpacker in a packed binary completely unpacks the embedded program before transferring control to it. The majority of packers satisfy Assumption (1) because they are supposed to work on commercially distributed executable binaries, which generally do not come with a relocation table. They also satisfy Assumption (2) because they cannot guarantee 100% static disassembly accuracy and coverage [8]. Some packers do perform simple metamorphic transformation to the input binaries before packing them. These packers inherently can evade signature-based AV scanners even without packing and are thus outside the scope of Justin. These two assumptions make it feasible to apply standard file-based AV scanners with selective scanning to a packed binarys memory image at the end of unpacking.

When the unpacker in a packed binary completes unpacking the embedded program, it sets up the import address table, unwinds the stack, and transfers control to the embedded programs entry point. Therefore the necessary conditions for the execution of a packed binary to reach the end of unpacking are

- A control transfer to a dynamically created/modified page occurs.
- The stack is similar to that when a program is just loaded into memory.
- The exception sequence does not exhibit any anomaly.
- The command-line input arguments are properly set up on the stack.

Accordingly, Justin combines these conditions into a composite heuristic for detecting the end of unpacking during the execution of a packed binary as follows. Given a binary, Justin loads it, marks all its pages as executable but non-writeable, and starts its execution. During the execution, if a write exception occurs on a non-writeable page, Justin marks this page as dirty, turns it into non-executable and writeable and continues; if a execution exception occurs on a non-executable page, Justin invokes an AV scanner to scan the memory image, and turns the page into executable and non-writeable if the end-of-unpacking check concludes that the unpacking is not done. For a non-packed binary, because no code page is generated or modified during its execution, it is impossible

for an execution exception to occur on a dirty page and no AV scan will be triggered at run time. So the performance overhead of Justin for non-packed binaries is insignificant. The performance overhead of Justin for packed binaries, on the other hand, depends on the number of times in which the programs control is transferred to a newly created page during its execution.

The current Justin prototype leverages virtual memory hardware to identify control transfers to dynamically created pages. More specifically, it manipulates write and execute permissions of virtual memory pages to guarantee that a page is either writeable or executable, but never both. With write protection, Justin can track which pages are modified. With execute protection, Justin can detect which pages are executed. If a binary Justin tracks needs to modify the protection attributes of its pages in ways that conflict with Justins setting, Justin records the binarys intentions but physically keeps Justins own setting. If the binary later on queries the protection attributes of its pages, Justin should respond with the binarys intentions, rather than the physical settings.

Whenever a virtual memory protection exception occurs, Justin takes control and first checks if this exception is owing to its setting. If not, Justin simply delivers the exception to the binary being monitored; otherwise Justin modifies the protection attributes according to the above algorithm, and also delivers the exception to the binary if the binarys intention is the same as Justins setting. To ensure that Justin is the first to respond to an exception, the exception handler component of Justin must be the first in the binarys vectored exception handler list.

To ensure that the original program in a packed binary can execute in the same environment, most unpackers unwind the stack so that when the embedded program is unpacked and control is transferred to it, the stack looks identical to that when the embedded program is loaded into memory directly. For example, assume that the initial ESP at the time when a packed binary is started is `0x0012FFBC`, then right after the unpacking is done and the unpacked code is about to be executed, the ESP should point to `0x0012FFBC` again. This rule applies to many unpackers and is widely used in manually unpacking practice. Justin automates this method by recording ESPs value at the entry point of a packed binary, and compares the ESP at every exception in which the programs control is transferred to a dynamically created page. The exception context of an exception contains all CPU registers at the time when the binary raises the exception.

The code section of a PE binary compiled by modern compilers on the Windows platform is typically loaded in an address space region that starts at a fixed location. More concretely, for a Win32 PE binary O consisting of a code section OC and a data section OD, its base loading address is `0x00400000` (as specified by Microsoft PE COFF specification), its OC starts from `0x00401000` and its OD starts with the page immediately following its OC. When a packer packs O, it compresses or encrypts O into O_{packed}, and forms the final packed binary B by combining the unpacker, O_{packed} and the area for holding O at run

time into one, two or three PE sessions. The entry point of B is set to that of the unpacker. The same principle applies when a packer packs a packed binary.

Let's use CBA to refer to the starting address of a non-packed PE binary's code section, typically at `0x00401000`. Given a packed binary B, let's call the region between CBA and the entry point page of the unpacker as TA (tracked area), which is the area that Justin monitors. For a recursively packed binary, the size of TA keeps shrinking during the binary's execution as each layer of unpacking unfolds. Because packers preserve the load image layout of the original binary, its code region, which starts with CBA, should be filled at run time after it is fully unpacked. Based on this reasoning, a possible end-of-unpacking heuristic is the following: The first several consecutive pages surrounding CBA have been written and contain "meaningful code. A simple way to determine if a page contains "meaningful code is to compute its entropy [**?**], which could easily detect zero-filled pages. A more reliable way is to disassembles the page in question. If it encounters disassembly errors, the page is not a code page. Even if there is no disassembly error, if the numbers of *dead* registers, destination registers whose values are not used later, and *uninitialized* registers, source registers that are not assigned before being used, in the resulting assembly code of a page are higher than a threshold, that page is not a meaningful code page.

When a PE binary is run with a set of command-line arguments, these arguments are first placed in heap by the loader and later copied to the stack by a piece of compiler-generated code included in the binary at the program start-up time. Based on this convention, one can detect the start of execution of the original binary embedded in a packed binary, which occurs temporally after the end of unpacking.

When an unpacker in a binary B reads in O_{packed}, unpacks it to produce the same address space image of the embedded original binary (O), and jumps to Os entry point, it should trigger an exception sequence of the form WWWWKWE under Justin, where W is a write exception and E is an execution exception. For a recursively packed binary, the resulting exception sequence is WWW..WEE..EW W..WEE..EWW..WE. Any deviations from these exception sequences indicate an attempt to evade Justins detection and therefore may not correspond to a true end of unpacking.

3.2 Implementation Details

The core logic of Justin is implemented in an exception handler that is registered in every binary at the time when it starts. In addition, Justin contains a kernel component that intercepts system calls related to page protection attribute manipulation and query and "lies properly so that its page status tracking mechanism is as transparent to the binary being monitored as possible. Justin leverages NX support [**?**] in modern Intel X86 processors and Windows OS to detect pages that are executed at run time. In theory, it is possible to use other bits such as supervisor bits for this purpose, as is the case with OmniUnpack [7] and OllyBone [6].

Because Justin enforces the invariant that a page is either executable or writeable but not both, it could lead a program that contains an instruction which modifies data residing in the same page as the instruction to a live lock V an infinite loop of interleaved execution and write exceptions. To address this issue, Justin checks if a memory-modifying instruction and its target address are in the same page when a write exception occurs. If so, Justin sets the page writeable, single-steps this instruction, and sets the page non-writeable again. This mechanism allows a page to be executable and writeable simultaneously for one instruction, but after that Justin continues to enforce the invariant.

One way to escape Justins invariant is to map two virtual pages to the same physical page, and set one of them as executable and non-writeable and the other as writeable and non-executable. With this set-up, the unpacker can modify the underlying physical page through the writeable virtual page and jump to the underlying physical page, without triggering exceptions. To defeat this evasion technique, Justin makes sure that the protection attributes of virtual pages which are mapped to the same physical page are physically set in the same way.

Instead of a PE section, an unpacker can put its output in a dynamically allocate heap area. To prevent unpacked binaries from escaping Justins tracking, Justin tracks pages in the heap, even when it grows. Similarly when a file is mapped into a processs address space, the mapped area needs to be tracked as well.

Instead of a PE section, an unpacker can put its output in a file, and spawns a process from the file later on. In this case, Justin will not detect any execution exception, because the generated code is invoked through a process creation mechanism rather than a jump instruction. Fortunately, standard AV scanners can detect this unpacked binary file when it is launched.

After recreating the embedded binary, some packers fork a new process and in the new process jumps to the embedded binary. This evasion technique is effective because page status tracking of a process is not necessarily propagated to all other processes it creates. Justin defeats this technique by tracking the page protection status of a process and that of all of its descendant processes.

Some unpackers include anti-emulation techniques that attempt to determine if they run inside an emulator or are being monitored in any way. Because Justin modifies page protection attributes in ways that may differ from the intentions of these unpackers, sometimes it triggers their anti- emulation techniques and results in program termination. For example, one unpacker detects if a page is writeable by passing a buffer that is supposedly writeable and Justin marks as non- writeable into the kernel as a system call argument. When the kernel attempts to write to the buffer, a kernel-level protection exception occurs and the program terminates. Justin never has a chance to handle this exception because it is a kernel-level exception and never gets delivered to the user level. To solve this problem, Justin intercepts this kernel-level protection exception, modifies the page protection attribute appropriately to allow it to continue, and changes it back before the system call returns. This example convinces us that it

is important to avoid tracking pages that hold the unpacker code. Thats why we set TA to be the area between CBA and the entry point page of the unpacker.

When Justin detects the end of unpacking, it treats the target address of the control transfer instruction as the entry point of the embedded binary that just gets unpacked. However, some packers obfuscate the main entry point by replacing the first several instructions at the main entry point with a jump instruction, say Y, to a separate piece of code, which contains the replaced instructions and a jump back to the instruction following Y. Because an unpacker can only safely replace the first N instructions, where N is smaller than 10, Justin can single-step the first N instructions at the supposedly entry point to specifically detect this evasion technique.

Some packers significantly transform an input binary before packing it. In general, these transformations are not always safe, because it requires 100which is generally not possible. Therefore, although these packers may evade signature-based AV scanners after Justin correctly produces the unpacked binary, we generally consider these packers to be too unreliable to be a real threat.

Packers	**Packed**	**Justin Unpack Failure**	**Justin Detection Failure**	**Justin Detection**	**SymPack Detection**	**Justin Detection Improvement**
ASPack	182	4	0	178	182	-4
BeroPacker	178	0	4	174	161	13
Exe32Pack	176	32	0	144	176	-32
Mew	180	1	8	171	171	0
PE-Pack	176	1	0	175	171	4
UPack	181	1	5	175	173	2

Table 1. *Effectiveness comparison between Justin and manually created unpackers from SymPack when they are used together with an AV scanner*

4 Evaluation

4.1 Effectiveness of Justin

To assess the effectiveness of Justin, we collect a set of known malware samples that are not packed by any known packers, then use different packers to pack them, and run the packed binaries under Justin and Symantec's AV scanner to see they together can detect these samples. As a comparsion, we used the same procedure but replaced Justin with Symantec's SymPack library, which contains a set of unpacker routines created manually by reverse engineering the logic of known packer programs. This experiment tests if Justin can unpack packed binaries to the extent that AV signatures developed for non-packed versions of malware samples still work.

There are totally 183 malware samples used in this study. As shown in Table 1, most packers cannot pack every malware program in the test suite successfully. So only successfully packed malware programs are unpacked. The number of successfully packed malware programs for each packer is listed in Column 2 of Table 1.

Justin cannot pack certain packed samples. By manually examining each failure case, we identify two reasons. First, some samples simply cannot run any more after being packed. Being a run-time detection technology, Justin cannot unpack something that does not run. From malware detection's standpoint, these packed samples are no longer a threat as they won't be able to cause any harm. Second, the packer Exe32Pack sometimes doesn't really modify the original binary when it produces a packed binary. For these packed binaries, no unpacking occurs at run time and Justin does not have a chance to step in and trigger the AV scan. From malware detection's standpoint, these packed samples are not a threat either if the AV scanner scans them before passing them to Justin. The number of packed malware programs that Justin fails to unpack Justin is listed in Column 3 of Table 1.

Among those malware samples that Justin successfully unpacks, not all of them can be detected. By manually analyzing these undetected samples, we find that most detection failures arise because signatures developed for non-packed versions of malware programs do not work for their unpacked versions. Although Justin can detect the end of unpacking, the unpacked result it produces is not exactly the same as the original program. Because some AV signatures are too stringent to accommodate these minor differences, they fail to detect Justin's outputs. For the same reasons, most of these undetected samples cannot be detected by SymPack either. The number of unpacked but undetected samples is listed in Column 4 of Table 1.

Overall, Justin's detection rate (Column 5) is slightly higher than SymPack's (Column 6) among the malware samples that can be successfully unpacked, because Justin relies on the unpackers embedded in the packed binaries, which are generally more reliable than the manually created unpackers in Sympack, to capture the execution state of a malware before it starts to run.

To test Justin's generic unpacking capability, we select a set of 13 packers that not supported by SymPack. Justin can successfully unpack binaries packed by 12 out of these 13 packers. The packer whose packed binaries Justin cannot unpack detects Justin's API call interception and terminates the packed binary's execution without unpacking the original program. We also test a set of malware samples packed by a packer that is not well supported by SymPack against Justin and an AV scanner. The number of these packed malware samples that can be detected by Justin/AV scanner is almost twice the number of SymPack/AV scanner.

To summarize, as long as a packed binary can run and requires unpacking at run time, Justin can unpack it successfully. Moreover, for the same malware samples packed by packers supported in SymPack, the unpacked outputs produced by Justin are more amenable to AV scanning than those produced by SymPack,

thus resulting in a higher detection rate than SymPack. Finally, Justin is able to detect twice as many packed malware samples than SymPack when they are packed by packers not supported in SymPack.

Packers	Dirty Page Execution	Unpacker Memory Avoidance	Stack Pointer Check	Command-Line Argument Access
ACProtect	186	11	1	2
ASPack	96	12	2	3
ASProtect	1633	12	12	3
Exe32Pack	394	11	1	2
eXPressor	15	11	1	2
FSG	12	12	1	2
Molebox	3707	11	1	2
NsPack	19	11	1	2
Obsidium	not work	14	4	6
PECompact	16	12	2	3
UPack	442084	12	2	3
UPX	11	11	1	2
WWPack	12	11	1	2

Table 2. *Comparison among four heuristics in their effectiveness to detect the end of unpacking, as measured by the number of times it thinks the packed binary run reaches the end of unpacking. The last three heuristics, Unpacker Memory Avoidance, Stack Pointer Check and Command-Line Argument Detection, are used together with the first heuristic, which monitors first control transfers to dirty pages.*

4.2 Number of Spurious End-of-Unpacking Detections

When Justin detects an end of unpacking during a packed binary's execution, it invokes the AV scanner to scan the process image at that instant. The main heuristic that Justin uses to detect the end of unpacking is to monitor the first control transfer to a dirty page (called *Dirty Page Execution*). Unfortunately this heuristic triggers many spurious end-of-unpacking detections for binaries packed by certain packers and thus incurs a significant AV scanning overhead even for goodware packed by these packers. The same observation was made by Martignoni et al. [7]. Their solution to this problem is to defer AV scanning until the first "dangerous" system call. Even though this technique drastically decreases the number of spurious end-of-unpacking detections, it also loses the entry-point information, which plays an important role for commercial signature-based AV scanners.

Instead, Justin incorporates three addition heuristics to reduce the number of spurious end-of-unpacking detections. *Unpacker Memory Avoidance* limits the Dirty Page Execution technique to pages that are not likely to contain the unpacker code. *Stack Pointer Check* checks if the current stack pointer at the time of a first control transfer to a dirty page during a packed binary's run is the same

as that at the very start of the run. *Command-Line Argument Access* checks if the command-line arguments supplied with a packed binary's run is moved to the stack at the time of a first control transfer to a dirty page. Each of these three heuristics is meant to work in conjunction with the Dirty Page Execution heuristic.

We apply a set of packers to a set of test binaries, run these packed binaries under Justin, and measure the number of end-of-unpacking detections. Table 2 shows the average number of end-of-unpacking detections for each of these four heuristics. Used together, the three additional heuristics in Justin successfully reduces the number of spurious end-of-unpacking detections to the same level as Martignoni et al. [7], but in a way that still preserves the original program's main entry point information.

Although the number of spurious end-of-unpacking detections produced by Unpacker Memory Avoidance is higher than the other two heuristics, it is more reliable and resilient to evasaion. If Justin mistakes a non-unpacker page as a packer page, it will not monitor this page, and the worst that can happen is that Justin loses the original program's entry point if this page happens to contain the main entry point. If Justin mistakes a unpacker page as a non-packer page, it will monitor this page, and the worst that can happen is additional spurious end-of-unpacking detections. Currently, Justin is designed to be err on the conservative side and therefore is tuned to treat unpacker pages as non-unpacker pages rather than the other way around.

We test the Stack Pointer Check heuritic using the packers listed in Table 2, Column 4 of Table 2 shows the average number of end-of-unpacking detections for each packer tested is decreased to just one or two for most packers. Unfortunately, this heuristic generates false negatives but no false positive. A false positive occurs when a certain execution point passes the stack pointer check but it is not the end of unpacking. This happens when the unpacker intentionally manipulates the stack pointer to evade this heuristic. None of the packers we tested exhibit this evasion behavior. A false negative happens when Justin thinks an execution point is not the end of unpacking when in fact it is. This happens when the unpacker does not clean up the stack to the exactly same state when the unpacker starts. The unpacker in ASProtect-packed binaries doesn't completely clean up the stack before transferring control to the original binary. It is possible to loosen up the stack pointer check, i.e., as long as the stack pointers are roughly the same, to mitigate this problem, but this is not a robust solution and may cause false positives.

The key idea in Command-Line Argument Access is that when the original binary embedded in a packed binary starts execution, there is a piece of compiler-generated code that will prepare the stack by fetching command-line arguments. Therefore, if at an execution point the command-line arguments supplied to a packed binary's run are already put on the stack, that execution point must have passed the end of unpacking. This command-line argument access behavior exists event if the original binary is not designed to accept any command-line arguments. Because Justin gets to choose the values for command-line arguments,

it detects command-line argument access by searching the stack for pointers that point to values it chooses as command-line arguments.

We test the Command-Line Argument Access heuritic using the packers listed in Table 2 and Column 5 of Table 2 list the average number of end-of-unpacking detections for each packer tested, which is generally higher than the Stack Pointer Check heuristic for the following reason. To put command-line arguments on the stack, the original binary has to be executed. So the number of code pages the original binary has to execute to put command-line arguments on the stack is exactly the additonal number of end-of-unpacking detections that the Command-Line Argument Access heuritic will report compared with the Stack Pointer Check heuristic. Even though its reported number of end-of-unpacking detections is slightly higher, the Command-Line Argument Access heuritic does not generate any false positive or false negative. For example, it can accurately detect the end of unpacking for ASProtect-packed binaries, but the Stack Pointer Check heuristic cannot.

Packers	**Number of AV Scans**	**Original Delay (msec)**	**Extra Delay (msec)**	**Extra Delay %**
ACProtect	2	46	4.2	9.1
ASProtect	3	62	9.0	14.6
eXPressor	2	62	5.5	8.8
FSG	2	62	4.2	6.8
Molebox	2	31	4.2	13.5
NsPack	2	46	4.5	9.9
Obsidium	6	31	12.1	38.7
PECompact	3	62	5.8	9.3
UPX	2	31	4.1	13.1

Table 3. *The average additional start-up delays for Microsoft Internet Explorer (IE) when it is packed by a set of packers and run under Justin and an AV scanner. The additional delay is dominated by AV scanning, which is mainly determined by the number of AV scans invoked during a packed binary's run.*

4.3 Performance Overhead of Justin

Justin is designed to work with an AV scanner to monitor the execution of non-known-good executable binaries. Its performance penalty comes from two sources: (1) additional virtual memory protection exceptions that are triggered during dirty page tracking, and (2) AV scans invoked when potential ends of unpacking are detected. We packed Microsoft Internet Explorer, whose binary size is 91KB, with a set of packers, ran the packed version, and measured its start-up delay with and without Justin on a 3.2GHz Pentium-4 machine running Windows XP. The start-up delay is defined as the interval between when the IE process is created and when it calls the the Win32 API CreateWindowEx function, which creates the first window. The start-up time excludes the program load time, which involves disk access, so that we can focus on the CPU overhead.

After the first window is created, a packed GUI application must have been fully unpacked, and there will not be any additional protection exceptions or AV scans from this point on. The AV scanner used in this study runs at 40 MB/sec on the test machine, and is directly invoked as a function call.

Table 3 shows the base start-up delay and the additional start-up delay for IE when it is packed by a set of packers and runs under Justin and an AV scanner. Overall, the absolute magnitude of the additional start-up delay is quite small. Justin only introduces several milliseconds of additional delay under most packers. The largest additional delay occurs under Obsidium and is only around 12 msec. At least for IE, the additional start-up delay that Justin introduces is too small to be visiable to the end user.

Most of the additional start-up delay comes from AV scanning, because the additional delay becomes close to zero when the AV scan operation is turned into a no-op. This is why the additional start-up delay correlates very well with the number of AV scans invoked. More specifically, the additional start-up delay for a packer is the product of the AV scanning speed, the number of scans, and the size of the memory being scanned. Because the amount of memory scanned in each AV scan operation may be different for binaries packed by different packers, the additional delay is different for different packers even though they invoke the same number of AV scans.

We also try other GUI programs such as Microsoft NetMeeting, whose binary is around 1 MB, and the additional delay results are consistent with those associated with IE. The performance overhead associated with additional protection exceptions is still negligible. Because of a larger binary size, the performance cost of each AV scan is higher. On a typical Windows desktop machine, more than 80% of its executable binaries is smaller than 100 KB. This means that the additional start-up delay when they are packed and run under Justin will be similar to that of IE and thus not noticeable. Finally, for legitimate programs that are not packed, no AV scanning will be triggered when they run under Justin, so there is no performance overhead at all.

5 Conclusion

References

1. ASPACK SOFTWARE. ASPack for Windows, 2007.
2. bart. FSG: [F]ast [S]mall [G]ood exe packer, 2005.
3. Blinkinc. Shrinker 3.4, 2008.
4. Dwing. WinUpack 0.39final, 2006.
5. Peter Ferrie. Attacks on Virtual Machines. In *Proceedings of AVAR Conference*, 2006.
6. Joe Stewart. OllyBonE v0.1, Break-on-Execute for OllyDbg, 2006.
7. Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

8. Susanta Nanda, Wei Li, Lap chung Lam, and Tzi cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. In *Proceedings of the 4th IEEE/ACM Conference on Code Generation and Optimization (CGO'06)*, 2006.
9. Markus F.X.J. Oberhumer, Lszl Molnr, and John F. Reiser. UPX: the Ultimate Packer for eXecutables, 2007.
10. Oreans Technology. Themida: Advanced Windows Software Protection System, 2008.
11. Silicon Realms. Armadillo/SoftwarePassport, 2008.
12. Symantec Corporation, 2008.
13. VMProtect. VMProtect, 2008.