Thwarting Real-Time Dynamic Unpacking

Leyla Bilge, Andrea Lanzi, Davide Balzarotti
Institute Eurecom, Sophia Antipolis
{bilge,lanzi,balzarotti}@eurecom.fr

ABSTRACT

Packing is a very popular technique for obfuscating programs, and malware in particular. In order to successfully detect packed malware, dynamic unpacking techniques have been proposed in literature. Dynamic unpackers execute and monitor a packed program, and try to guess when the original code of the program is available unprotected in memory. The major drawback of dynamic unpackers is the performance overhead they introduce. To reduce the overhead and make it possible to perform dynamic unpacking at end-hosts, researches have proposed real-time unpackers that operate at a coarser granularity, namely OmniUnpack and Justin. In this paper, we present a simple compile-time packing algorithm that maximizes the cost of unpacking and minimizes the amount of program code that can be automatically recovered by real-time coarse grained unpackers. The evaluation shows that the real-time dynamic unpackers are totally ineffective against this algorithm.

1. INTRODUCTION

In recent years, packing has become a popular technique of choice for obfuscating malware code and for evading signature-based anti-virus (AV) scanners [18]. In general, the packing and unpacking process is simple. The packer modifies the original binary by encrypting the code and inserting an unpacking routine. When the packed binary is executed, the unpacking routine unpacks the code and transfers the control flow to the unpacked code. Since signature-based anti-virus scanners analyze suspicious programs statically, they cannot inspect the original binary code, and fail to detect when this code is malicious.

To deal with packed malicious code, several unpacking systems have been proposed. These systems have been developed to support AV scanners by providing them the unpacked binary code to scan. They originally operated statically by trying to recognize the packing scheme used to protect the code, and to recover the original code using a manually-written routine. Malware authors soon discovered that for defeating static unpackers it was sufficient to use a slightly different packing algorithm each time such that custom unpacking routines needed to be constantly adapted. For these reasons, static systems have been superseded by systems based on dynamic approaches.

Dynamic unpackers execute and monitor a packed program, and try to guess when it has finished unpacking itself so that they can locate the unprotected code in memory. To this end, the dynamic unpackers employ a number of heuristics and leverage the fact that the programs are typically packed as a whole. Depending on the deployment scenarios, the heuristics trade off the precision with performance, and vice-versa. Off-line dynamic unpackers typically run inside a sand-box (e.g., a virtual environment or an emulator) and are mainly used as a part of the malware analysis process [9, 20]. As a result, such off-line dynamic unpackers do not have strict time constraints. Hence, they can apply very precise monitoring techniques and heuristics to detect the execution of unpacked code.

One important drawback of the off-line dynamic unpackers is the performance overhead they introduce. Unfortunately, this performance overhead makes it impossible to install and run them as real-time systems on end-user machines. Therefore, it is desirable to have an effective real-time dynamic unpacker that is efficient enough to be run on an end-user machine.

To date, only two real-time dynamic unpackers have been proposed: Omniunpack [13] and Justin [8]. They both try to find the point in time where the binary is unpacked in the memory. Once they detect this point, they invoke the AV scanner for initializing the detection procedure. Omniunpack and Justin are able to detect malicious code that is packed by a wide range of packers [7, 2, 4, 6, 15, 16, 17, 23, 25].

To the best of our knowledge, there has not been any previous research that evaluates the effectiveness of the real-time dynamic unpackers against more sophisticated packing techniques. In this paper, we perform an analysis on the real-time dynamic unpackers proposed to date. To this end, we implemented a prototype compile-time packer designed to maximize the cost of unpacking and minimize the amount of program code that can be automatically recovered. The packing scheme we propose is based on the same principles of the Themida packer (function level packing) but with more finer packing granularity. The aim of our prototype is to have a tool that can perform different levels of packing (i.e. functions, basic block, instruction) to test real-time unpackers. The results of our experiments show that our packer makes existing real-time unpacking solutions ineffective.

The paper is organized as follows. Section 2 briefly introduces the packers and their limitations, and surveys the various techniques proposed in the literature to recover packed code. Section 3 presents the packing algorithm we have developed to attack real-time unpackers. Section 4 discusses the results of the evaluation. Finally, Section 5 concludes the paper.

2. BACKGROUND

Packing is one of the program obfuscation techniques employed by malware authors to protect their software (i.e.., prevent detection and complicate the reverse engineering). This section briefly describes how packers work and discusses the techniques for recovering packed code.

2.1 Packing Schemes and Their Limitations

In the traditional form, packing is applied to the executable programs produced in output by compilers (as shown in Figure 1). Essentially, packing consists of taking the original content of the executable, packing this data by encrypting or compressing it, and in generating a new program composed by the packed executable image and the appropriate unpacking routine. When invoked, this new program will execute the unpacking routine first. The unpacking routine will restore the original image of the executable in memory, and then transfer the execution to the original entry point of the program. Thus, any attempt to statically analyze the packed program is made ineffective, since the real code of the program is only accessible at run-time. This form of packing is widely adopted because it is very easy to apply and because it can be used with any program, independently from the source language in which the program is written. The majority of the open-source and commercial packers follow this approach ([7, 2, 4, 6, 15, 16, 17, 23, 25]).

However, this type of packing is very weak and easy to defeat. At the end of the execution of the packing routine, the whole program image is accessible in memory in an unprotected form. This weakness is a consequence of the fact that the packer does not have any insight on the program itself. In fact, the analysis of executables or assembly programs cannot, in general, assure soundness and completeness [1]. Without any precise information about the control-flow and the data-flow of the program, it is difficult for the packer to reliably partition the program into fragments and construct an unpacking routine that unpacks these fragments incrementally at run-time. Hence, the normal approach consists in packing the entire program. There are, however, some exceptions to this behavior. For example, Armadillo [19] partitions the program into pages and unpack only the page to be executed. This approach is reliable since partitioning does not require to analyze the program, but just leverages hardware protection mechanisms to detect the page that is going to be executed. Although Armadillo unpacks the program on-demand, once a page is accessed and unpacked, the page remains unprotected in memory until the program terminates. Finally, we believe that the function level granularity, supported by Themida [24], is the finest granularity that can be achieved without performing any sophisticated (and error-prone) analysis on the executable.

Recently, obfuscation techniques that are based on code virtualization have become popular [24]. In few words, code virtualization consists of translating the instructions of the program into new instructions belonging to a custom instruction set, and executing them using an ad-hoc emulator, which is embedded directly into the program. Although code virtualization can be used to obfuscate binaries as packers do, the analysis of programs protected with code virtualization is a completely separate research problem [21], and it is outside the focus of this paper.

2.2 Unpacking Techniques

The historical method for unpacking a packed program is based on *algorithmic unpacking*. As the name indicates, this method employs a manually written algorithm that mimics the unpacking routine embedded in the packed program. This approach is very effective and efficient at unpacking programs that are packed with well-known packing schemes. However, its effectiveness has been decreasing since the development of a specific unpacking algorithm requires substantial expert knowledge, and since malware developers are continuously developing new packing schemes to exhaust experts' resources.

To overcome this limitation, a new unpacking technique, called *generic unpacking*, has been proposed. The main idea behind generic unpacking is to rely directly on the unpacking routine embedded in the packed program, and to execute this routine until the program is completely restored in memory. To this end, generic unpackers either emulate or monitor the execution of the program and adopt several heuristics to detect when the unpacking is completed and the original program can be identified in memory in an unprotected state. Heuristics are necessary because the detection of the end of the unpacking is an undecidable problem [20]. All generic unpackers are based on the same heuristics that detect an unavoidable behavior: the execution of *previously written code*. On the other hand, unpackers might implement this heuristic at a different granularity (e.g., at the byte or at the page granularity) and might combine it with additional ones to improve the accuracy and to optimize the performance.

Depending on the precision of the heuristics used by generic unpackers to detect the end of the unpacking, unpackers might or might not be suitable for real-time scenarios (i.e., on end-hosts). Examples of off-line generic unpackers, whose heuristics are too precise and expensive for real-time deployment, are Renovo and PolyUnpack [9, 20]. Both unpackers detect the end of the unpacking by emulating the execution of the program and monitoring all memory writes and instruction fetches, and consider all instructions fetched from previously written memory locations to be successfully unpacked.

Real-time packers, such as OmniUnpack and Justin [13, 8], in comparison, use coarse grained heuristics that are much less expensive to deploy. They can be used on end-hosts to recover, on behalf of the AV, the protected malicious code. More precisely, these unpackers track memory writes and instruction fetches at the page level, at a very low-cost, leveraging directly hardware protection mechanisms offered by the CPU. To compensate the coarseness of this heuristic, to increase the precision of their guess about the end of the unpacking, and to reduce the number of ineffective AV invocations (i.e., to scan memory locations containing code that is only partially unpacked), OmniUnpack and Justin leverage additional heuristics. That is, OmniUnpack does not consider the unpacking to be concluded until the execution of alleged unpacked code is followed by the invocation of a dangerous system call (i.e., a system call that could potentially alter the safe state of the system). The rationale is that the unpacking routine is typically executed in batch mode, immediately after the program is launched, and that dangerous interactions with the system (e.g., the creation of a new file) only take place after the malware is completely unpacked. The approach taken by Justin, instead, is to consider the unpacking concluded only when the stack layout during the execution of previously written memory locations is compatible with the layout expected in the absence of packing. The intent is to catch the instant of time at which the `main` function of the protected program is executed.

3. ON-THE-FLY (UN)PACKING

Figure 2 shows an overview of the packing scheme we implemented for our experiments. Compared to traditional schemes, our approach offers a *finer packing granularity*. In traditional packing schemes, the program is typically packed as a whole. In contrast, our approach partitions the program into multiple *packing units* (a

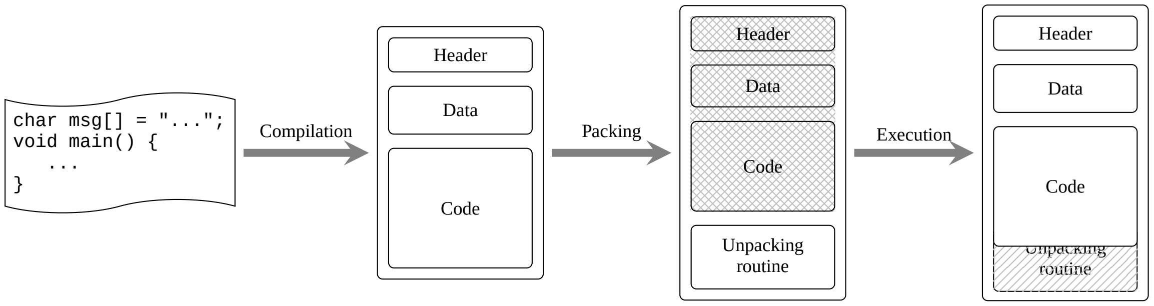


Figure 1: Overview of the traditional (un)packing scheme (cross-hatched regions are packed)

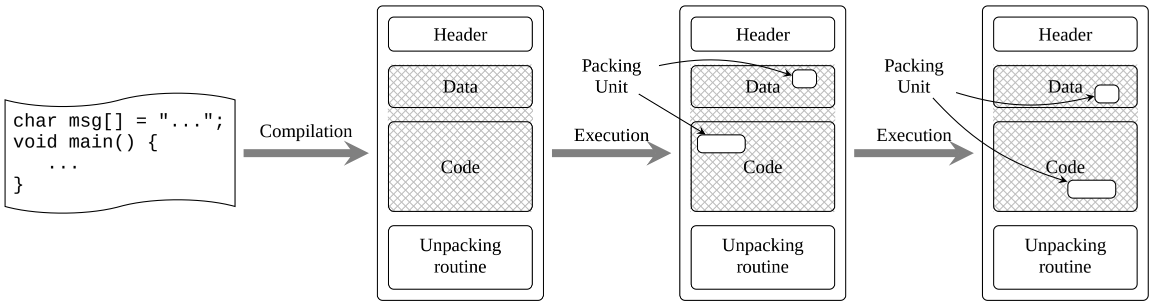


Figure 2: Overview of proposed on-the-fly (un)packing scheme (cross-hatched regions are packed; white regions inside the cross-hatched regions are not)

unit is a fragment of the program's code or data), each of which is packed and unpacked independently. More precisely, in our scheme, a packing unit is unpacked on-the-fly, just before being used. In addition, it is repacked immediately after use. Therefore, the whole program is never entirely in memory in an unprotected fashion. Rather, only a small unit at a time is unprotected. The only way to recover the entire program is to force it to unpack each of its units (e.g., by forcing the program to execute a particular region of code or to access a particular region of data). Moreover, as we demonstrate in Section 4, this approach effectively causes real-time generic unpackers to perform excessive trashing [5], and can also prevent them to recover the entire program.

The fine packing granularity is possible in our scheme as the packing is performed *transparently* at compile time. Thus, we have access to high-level program information, which is not available to traditional packers working directly on compiled executables (as described in Section 2.1). In our scheme, thanks to the high-level information we have access to, we can partition the program into multiple packing units and protect these separately. By relying on conservative information, we have the guarantee that the transformations we apply to the program preserve its semantics. Depending on the desired level of security, a packing unit can be a module of the program, a function, a basic block, and even a single instruction, or a variable.

Although our packing scheme share some similarities with the obfuscation scheme presented by Sharif et al. [22], our scheme generalizes the obfuscation and is intended to be applied to whole programs.

3.1 (Un)Packing Mechanism

The steps required for packing a program for on-the-fly (un)packing are shown in Figure 3. In short, the usual compilation process is extended with two additional steps in order to make the program capable of unpacking and packing its code and data at run-time, and for the initial packing of the compiled executable.

The source code is translated into a high-level intermediate form by the front-end of the compiler (Step 1 in Figure 3). The intermediate form is analyzed with the aim of identifying the packing units, and is then instrumented to introduce `unpack` and `pack` operations respectively at entry and exit points of each unit (Step 2 in Figure 3). After instrumentation, the intermediate form is optimized and is then compiled into an executable object (Step 3 in Figure 3). Finally, the instrumented executable is packed (Step 4 in Figure 3).

The `unpack` and `pack` operations are responsible, respectively, for unpacking a packing unit just before usage, and for packing it again after the usage. In other words, these operations guarantee that the code or the data region represented by the packing unit are accessible in memory only when the memory region is effectively executed or accessed. Practically speaking, a packing unit representing a fragment of code (e.g., a function) is unpacked only when the fragment is being executed. On the other hand, a packing unit representing a global variable is unpacked only when the variable is being read or written.

The instrumentation performed in the second step of the compilation process then depends on the desired granularity of packing and on the type of packing unit. For example, for on-the-fly packing at function granularity, each function of the program is treated as a different packing unit. Thus, the instrumentation consists of wrapping all function calls and of adding `pack` and `unpack` operations to pack the caller and to unpack the callee. In contrast, for packing at basic block granularity, the instrumentation consists of extending each basic block with a prologue and an epilogue. The prologue unpacks the code of the basic block at the entry, while the epilogue packs the basic block at the exit. Finally, for packing a variable, the instrumentation consists of inserting an `unpack` operation before each usage of the variable, and of inserting a `pack` operation after each usage. To detect all program instructions that ac-

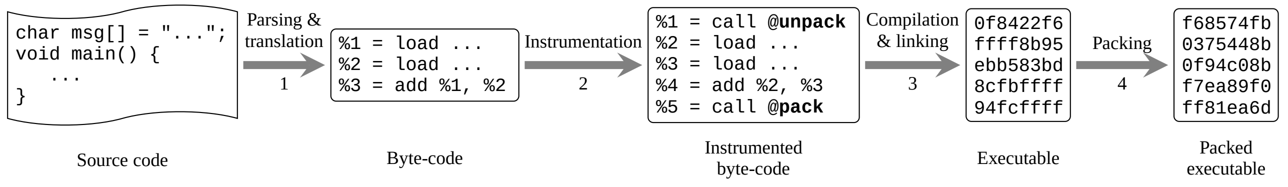


Figure 3: Overview of the compilation and packing process

cess a particular variable, we rely on alias information made available by the compiler.

Figure 4 shows a fragment of a sample program (in assembly) and compares the code of the program without packing and with packing at basic block granularity (Figures 4(a) and 4(b) respectively). The code of a basic block in the packed program is very similar to that in the unprotected copy of the program. In the packed program, each basic block begins with an `unpack` operation and ends with a `pack` operation. The `pack` operation is followed by an optional indirect control transfer instruction that transfers the execution to the next non-adjacent basic block. The two packing operations delimit the real code of the basic block (the crosshatched region in the Figure). This code is almost identical to the code in the unprotected block. The only difference is in the control transfer instructions used to transfer the execution to the next basic block (conditional and unconditional jumps and calls). In unprotected blocks (Figure 4(a)), control transfers are direct. In protected blocks (Figure 4(b)), control transfers are indirect, and the address of the next block to execute is stored in a special variable (`nextbb` in Figure 4(b)), which is set locally. The intent is to further complicate reverse engineering by preventing the adversary to guess how basic blocks are linked together, and thus to guess the structure of the program.

It is worth noting that the proposed scheme supports flawlessly dynamically linked programs, where shared libraries are not packed. In fact, when the execution flows from the program to a shared library, no portion of the program is left unprotected.

At a first sight, `unpack` operations might seem easy to detect (e.g., using pattern matching). If detected, they could be executed to unpack the associated packing unit. Our claim is that multiple pairs of `pack` and `unpack` operations can be used in the same program. Potentially, each packing unit could be protected using a different pair of operations and, in case of extreme paranoia, they could be even obfuscated using standard—but hard to reverse—obfuscation techniques (e.g., [11]).

3.2 Implementation Details

We have implemented a prototype of the proposed packing scheme using LLVM, an open-source, and language-independent compiler [10]. The prototype consists of three components. The first component is a LLVM "pass", which takes the intermediate bytecode generated by the compiler and instruments it to add on-the-fly (un)packing capabilities, at the requested granularity level. The second component is the real packer; it takes the final executable produced by the compiler, identifies the various packing units using symbols information, and packs them. The third component is a script that allows (malicious) users to pack their programs with no effort. Essentially, the script is meant to substitute the original compiler and to translate the source code into LLVM bytecode, to apply the instrumentation "pass", to compile and link the bytecode, and finally, to pack the resulting executable. In summary, (malicious) users can protect their programs without changing a single line in the source code, and by compiling their code using the script we provide. For example, programs compiled using Make can be packed simply by instructing Make to use a different compiler (e.g., `make CC=otfpcc`).

The LLVM "pass" traverses all the functions in the program, detects the appropriate instrumentation points (according to the requested packing granularity), and inlines the operations for on-the-fly (un)packing. From the LLVM prospective, our "pass" is a user-written optimization, and can be hooked into the compilation chain. Given the high-level API offered by LLVM, the instrumentation is trivial to perform. That is, less than a hundred lines of C++ are sufficient to analyze the bytecode, and to augment it with on-the-fly (un)packing capabilities at function, or basic block granularity. Currently, our prototype does not support on-the-fly (un)packing of data.

In the current prototype, the packing units are encrypted by XORing their content with a variable length key. Each packing unit is encrypted with a different key and the keys are embedded separately in each unit, within the code we insert during the instrumentation "pass". Obviously, more sophisticated and multiple encryption schemes can be used to protect the program. Furthermore, inlined operations for packing and unpacking are reentrant: they check whether the packing unit is already being used (and thus, unpacked) by another thread. The packing unit is unpacked by the first thread that accesses it and repacked when the last thread ceases to use it.

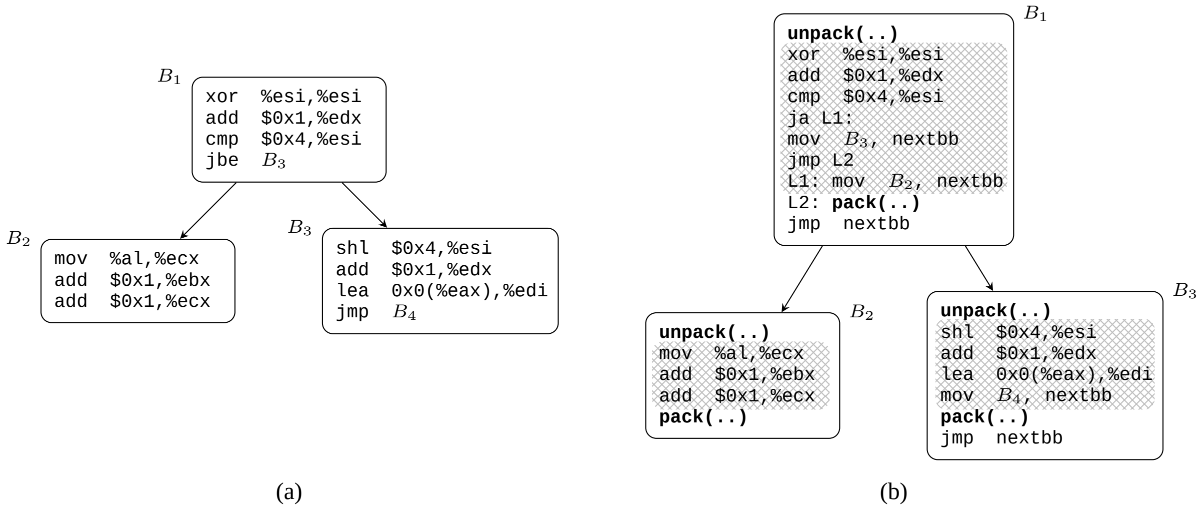
3.3 Limitations

Although the proposed packing scheme is simple and easy to implement, there are few situations in which it cannot be safely applied. For example, a program that uses self-checksumming to guarantee untampered execution will fail to execute because the checksummed regions of the programs will most likely be packed.

Although the packing of data would be very easy to implement using LLVM, our prototype currently does not support such a feature. In any case, packing could be applied only to global and statically allocated variables because the packing of dynamically allocated variables (e.g., in the stack or in the heap) would require to instrument library code as well.

4. EVALUATION

The purpose of our evaluation is to demonstrate that our packing scheme is very effective in protecting packed code and that it renders real-time generic unpackers such as OmniUnpack and Justin completely ineffective. Indeed, by continuously unpacking and repacking each packing unit, the heuristics adopted by OmniUnpack and Justin keep triggering and unpacking becomes ineffective because most of the resources of the CPU are consumed to perform useless analysis. Moreover, our packing scheme violates the assumption, made by OmniUnpack and Justin, that a block of code sufficiently big for reliable detection is available in memory unprotected. In fact, AVs need to analyze a certain number of functions in order to guess the maliciousness of the code with a low



```
xor  %esi,%esi
add  $0x1,%edx
cmp  $0x4,%esi
jbe  B3
```

```
mov  %al,%ecx
add  $0x1,%ebx
add  $0x1,%ecx
```

```
shl  $0x4,%esi
add  $0x1,%edx
lea  0x0(%eax),%edi
jmp  B4
```

```
unpack(..)
xor  %esi,%esi
add  $0x1,%edx
cmp  $0x4,%esi
ja L1:
mov  B3, nextbb
jmp L2
L1: mov  B2, nextbb
L2: pack(..)
jmp  nextbb
```

```
unpack(..)
mov  %al,%ecx
add  $0x1,%ebx
add  $0x1,%ecx
pack(..)
```

```
unpack(..)
shl  $0x4,%esi
add  $0x1,%edx
lea  0x0(%eax),%edi
mov  B4, nextbb
pack(..)
jmp  nextbb
```

Figure 4: Sample fragment of a program: (a) without packing and (b) with packing (at basic block granularity).

false-positive rate.

For the evaluation, we developed a cross-platform dynamic binary instrumenter on top of PIN [12]. This application traces an arbitrary program, logs all memory accesses (i.e., fetches, loads, and stores) that occur during its execution, and takes complete snapshots of the content of the memory at will. We used this application to inspect the content of the memory during the execution of several experimental packed programs, and also to simulate the heuristics adopted by OmniUnpack and Justin for detecting the end of the unpacking. In particular, we simulated the W⊕X policy at the page level (i.e., a page can be either writable or executable). Note that this policy is used by both unpackers to track the execution of previously written code. Furthermore, we programmed our application to take full snapshots of the content of the memory at each system call and each time the constraints used in Justin's heuristic were satisfied.

In the first experiment, we tested how OmniUnpack and Justin are able to deal with our packer. Using our packer, each packing unit is unpacked before execution and packed immediately after. Thus, each time a packing unit is executed, at least two violations of the W⊕X policy occur, each of which results in a page-fault exception. When the packing unit is unpacked, the memory page that contains it is marked as being writable and not executable. When the unpacking is concluded and the execution begins, the first page-fault occurs, and the page is marked as being executable and not writable. Similarly, when the execution is concluded and the unit is repacked, another page-fault occurs. Therefore, the number of page-faults caused by the W⊕X policy is at least $2u$, where u is the number of times the program executes a different packing unit. This is a lower bound on the number of page-faults since a packing unit overlapping adjacent pages would cause multiple faults.

Thus, it is imaginable that the number of page-faults can easily explode to the point where most of the CPU time is spent in responding to these faults. To prove our assumption, we used the PIN-based application to measure the number of executed packing units in simple packed applications. Table 1 reports some results of our experiments. As an example, we packed the `tar` utility (at function granularity) and used it to compress the content of the `/etc` directory on a common Linux distribution. During the execution of the packed utility, we observed that more than 105,000 packing units were executed. In such a situation, the W⊕X policy adopted by OmniUnpack and Justin would have caused more than

Program	# executed packing units	# page-faults (estimated)
`tar`	∼105,000	∼210,000
`gzip`	∼143,000	∼286,000
`sed`	∼60,200	∼120,400
`grep`	∼26,100	∼52,200
`wget`	∼139,000	∼278,000

Table 1: Estimated number of page-faults that would be caused by the W⊕X policy used by OmniUnpack and Justin (programs packed at function granularity)

210,000 page-faults. By adopting a finer packing granularity (e.g., at the basic block level), the number of page-faults would have been even higher.

Note that, OmniUnpack and Justin invoke an AV scanner such that it scans the unpacked memory, instead of taking the snapshots of the content of the memory. Similarly to OmniUnpack and Justin, a human expert, with the help of an emulator or a debugger, might try to reconstruct the unpacked program by taking arbitrary snapshots of the content of the memory. Our packing scheme is resistant against all such attempts. That is, there is no instant in time where the program that is being executed is in completely unpacked state in memory. Further, the amount of code that is unpacked at a given time solely depends on the granularity of the packing.

In order to measure the amount of code exposed by our unpacker, we packed several sample applications at the function level and we ran them. For each execution of the program, we took snapshots of the content of the memory at two different instants in time: system call time (conservative approximation of OmniUnpack's heuristics) and at the start of the `main` function (Justin's heuristics). As expected, in all cases, we never found the entire program in the memory, but only one packing unit (the function) being executed at the time of the snapshot. In any of the snapshots taken using OmniUnpack's heuristics, we did not find any function in unprotected form in the memory. The explanation is simple: at every function call, the caller is packed right before entering the callee. Afterwards, the callee is unpacked only if it is one of the functions of the packed program. Otherwise, if the callee is either a library function or a system call, obviously it is not unpacked. Since OmniUnpack triggers the AV scanner only on specific system calls (i.e. dangerous system calls), at the time the AV scanner is called no unprotected

Program	% of unprotected code	
	OmniUnpack	Justin
tar	0%	1.34%
gzip	0%	2.9%
sed	0%	2.9%
grep	0%	6.0%
wget	0%	2.54%

Table 2: Percentage of program's code that would be recovered by OmniUnpack and Justin (programs packed at function granularity)

code is available in the memory. On the other hand, the heuristics employed by Justin work differently: They are based on the assumption that the program is packed as a single block and it attempts to detect when the unpacking routine transfers the control to the original entry point of the program (e.g., the `main`). The results of our experiments show that at the time Justin's heuristics are triggered, the percentage of unprotected code is very small, as it can be seen from Table 2.

Table 2 reports some results of our experiments. For example, in common Linux utilities such as `wget` and `tar`, packed with our packing scheme at function granularity, the percentage of recovered unprotected code using OmniUnpack's and Justin's heuristics was respectively 0% and 2.54% for the first utility, and 0% and 1.34% for the second one.

In conclusion, we speculate that the only way to fully recover a program packed with our packer is to rely on off-line unpackers that can execute all possible paths, either manually or by using multi-path exploration techniques (e.g., [3, 14]) that can take a snapshot of the memory each time a previously unseen packing unit is unpacked. That means that new techniques for unpacking at the end-host, in real-time, must be developed.

5. CONCLUSIONS

In this paper, we performed an evaluation on the real-time dynamic unpackers. We implemented a prototype compile-time packer designed to maximize the cost of unpacking and minimize the amount of program code that can be automatically recovered. By our evaluation we showed the proposed unpacking scheme makes real-time dynamic unpacking solutions ineffective. We believe that in the future hardware solutions could be used to design more resilient unpackers.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n 257007.

6. REFERENCES

[1] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Working Conference on Verified Software: Theories, Tools, Experiments*, Zurich, Switzerland, Oct. 2005.

[2] Bitsum Technologies. PECompact. `http://www.bitsum.com/pecompact.php`, 2009.

[3] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Towards Automatically Identifying Trigger-based Behavior in Malware using Symbolic Execution and Binary Analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University, 2007.

[4] Danilo Bzdok. Yoda's Crypter. `http://yodap.sourceforge.net`, 2010.

[5] P. J. Denning. Thrashing: Its Causes and Prevention. In *Fall Joint Computer Conference*, 1968.

[6] Dwing. UPack. `http://dwing.cjb.net`, 2010.

[7] Fast Small Good (FSG). `http://www.woodmann.com/collaborative/tools/index.php/FSG`, 2009.

[8] F. Guo, P. Ferrie, and T. cker Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the Recent Advances in Intrusion Detection Symposium*, 2008.

[9] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode*, 2007.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2004.

[11] C. Linn, S. Debraydepartment, and C. Science. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM Press, 2003.

[12] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceeding of ACM Conference on Programming Language Design and Implementation*. ACM Press, 2005.

[13] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference*, 2007.

[14] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2007.

[15] North Star Software. NsPack. `http://www.nsdsn.com/eng/index.htm`, 2009.

[16] Nullsoft Inc. NSIS. `http://nsis.sourceforge.net`, 2009.

[17] M. Oberhumer. UPX, 2010.

[18] Panda Security. `http://www.pandasecurity.com/homeusers/media/press-releases/viewnews?noticia=8612`, 2007.

[19] S. Realms. SoftwarePassport: Armadillo. `http://www.siliconrealms.com/software-passport-armadillo.html`, 2010.

[20] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the Annual Computer Security Applications Conference*, 2006.

[21] M. Sharif, A. Lanzi, J. Giffin, , and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of The 2009 IEEE Symposium on Security and Privacy*, 2009.

[22] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[23] StarForce. ASPack. `http://www.aspack.com/`, 2009.

[24] O. Technology. Themida: Advanced Windows Software Protection System. `http://www.oreans.com/`, 2008.

[25] The EGOiSTE/TMG. tElock. `http://programmerstools.org/node/164`, 2009.