

OmniUnpack

Fast, Generic, and Safe Unpacking of Malware

Lorenzo Martignoni¹ Mihai Christodorescu² Somesh Jha³

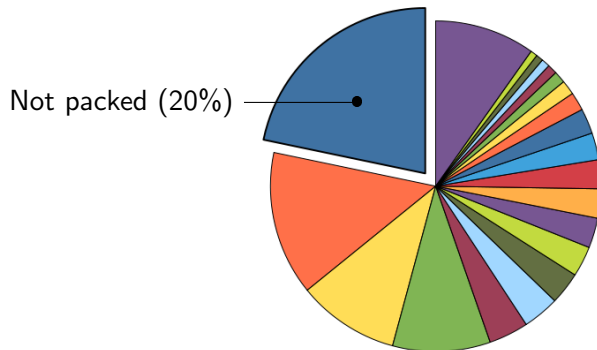
¹Università degli Studi di Milano

²IBM Research

³University of Wisconsin–Madison

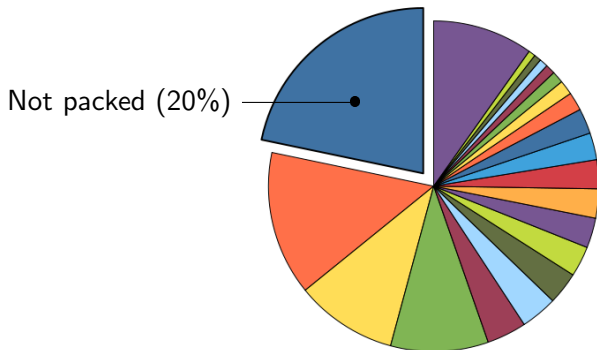
ACSAC 2007

Malware and packing



80% of new malware are packed with various packers

Malware and packing



80% of new malware are packed with various packers


50% of new malware samples are simply repacked versions of existing malware

Code packing

- ▶ A technique to hide the real code of a program through one or more layers of compression/encryption
- ▶ At run-time the unpacking routine restores the original code in memory and then executes it

Code packing

- ▶ A technique to hide the real code of a program through one or more layers of compression/encryption
- ▶ At run-time the unpacking routine restores the original code in memory and then executes it



Malicious
code

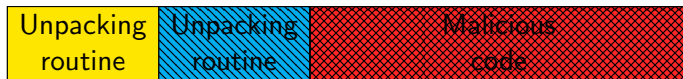
Code packing

- ▶ A technique to hide the real code of a program through one or more layers of compression/encryption
- ▶ At run-time the unpacking routine restores the original code in memory and then executes it



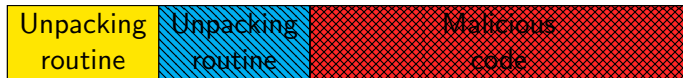
Code packing

- ▶ A technique to hide the real code of a program through one or more layers of compression/encryption
- ▶ At run-time the unpacking routine restores the original code in memory and then executes it



Code packing

- ▶ A technique to hide the real code of a program through one or more layers of compression/encryption
- ▶ At run-time the unpacking routine restores the original code in memory and then executes it



The effectiveness of malware detectors depends on the ability to recover the “real” malicious code, but recovery often fails!

Traditional approaches to deal with packed code

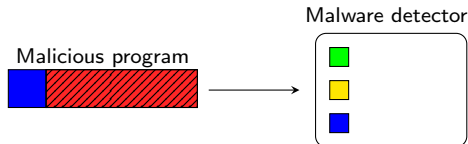
Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)

Traditional approaches to deal with packed code

Algorithmic unpacking

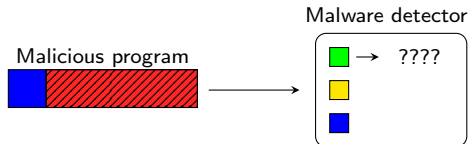
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

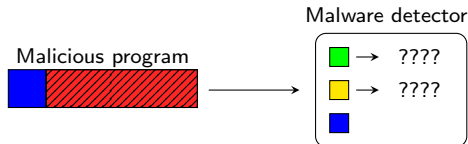
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

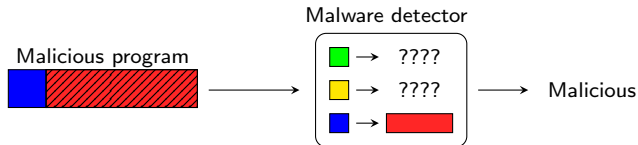
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

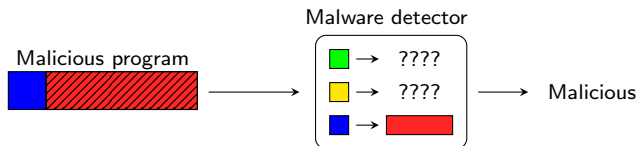
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



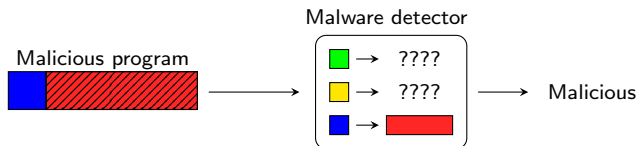
Generic unpacking

Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])

Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Generic unpacking

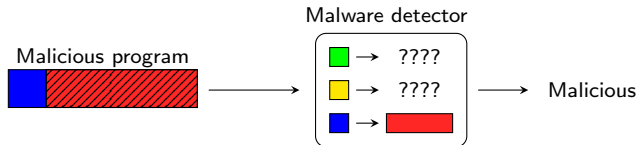
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])



Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Generic unpacking

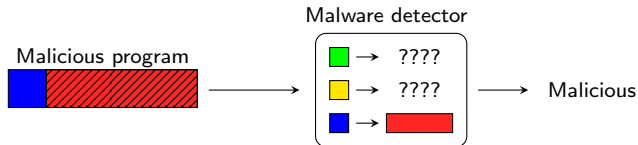
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])



Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Generic unpacking

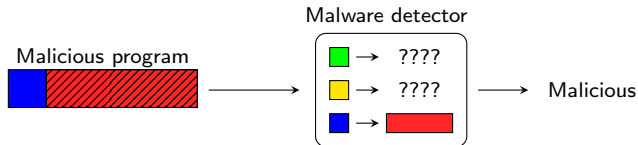
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])



Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Generic unpacking

Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])



Limitations of traditional approaches

Algorithmic unpacking

- ▶ Every new packer requires a dedicated unpacking algorithm
- ▶ New packers are created from existing ones at a rate of 10-15 per month

Limitations of traditional approaches

Algorithmic unpacking

- ▶ Every new packer requires a dedicated unpacking algorithm
- ▶ New packers are created from existing ones at a rate of 10-15 per month

Generic unpacking

- ▶ Unpacking is slow and is not suitable for end-user environments
- ▶ Effectiveness depends on the fidelity of the emulation environment (packers leverage anti-emulation techniques)

Limitations of traditional approaches

Algorithmic unpacking

- ▶ Every new packer requires a dedicated unpacking algorithm
- ▶ New packers are created from existing ones at a rate of 10-15 per month

Generic unpacking

- ▶ Unpacking is slow and is not suitable for end-user environments
- ▶ Effectiveness depends on the fidelity of the emulation environment (packers leverage anti-emulation techniques)

Algorithmic & generic unpacking

- ▶ The detection of the **termination of the unpacking routine is undecidable**
- ▶ The unpacker might fail to recover the entire malicious payload

Goals of OmniUnpack

- ▶ Generic unpacking with low-overhead by using existing hardware mechanisms
- ▶ Precise unpacking by running the program on the native OS
- ▶ A new malware detection strategy, independent of packing, where the malware detector analyzes new pieces of code before they are executed

Goals of OmniUnpack

- ▶ Generic unpacking with low-overhead by using existing hardware mechanisms
- ▶ Precise unpacking by running the program on the native OS
- ▶ A new malware detection strategy, independent of packing, where the malware detector analyzes new pieces of code before they are executed

Feature	ClamAV	PolyUnpack	OmniUnpack
Fast for interactive use	✓	—	✓
Handles unknown packers	—	✓	✓
Incremental detection	—	✓	✓
Resilient to anti-debugging	✓	—	✓
Resilient to SEH attacks	✓	—	✓

Outline

OmniUnpack

Implementation

Evaluation

Conclusion

A simple generic unpacker

- ▶ Track all memory writes and the program counter
- ▶ The execution of a previously written memory location denotes the end of an **unpacking stage**
- ▶ All written-then-executed memory locations should then be analyzed by a malware detector

A simple generic unpacker

- ▶ Track all memory writes and the program counter
- ▶ The execution of a previously written memory location denotes the end of an **unpacking stage**
- ▶ All written-then-executed memory locations should then be analyzed by a malware detector

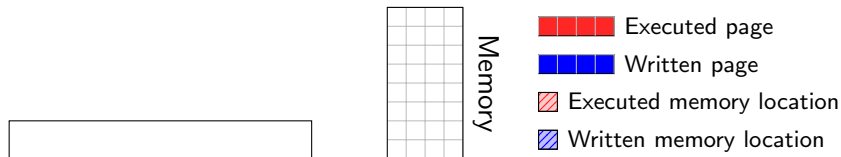
Extend this idea to design an iterative unpacking algorithm that achieves low overhead yet does not compromise the security of the system

Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

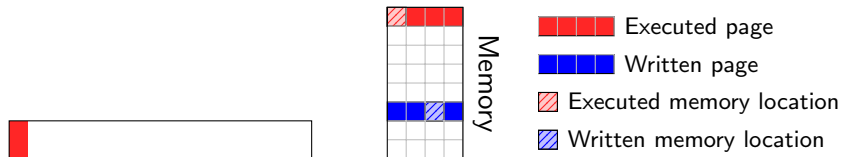
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms



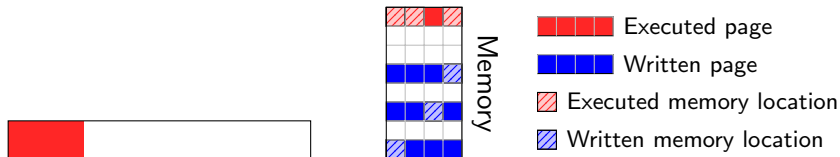
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms



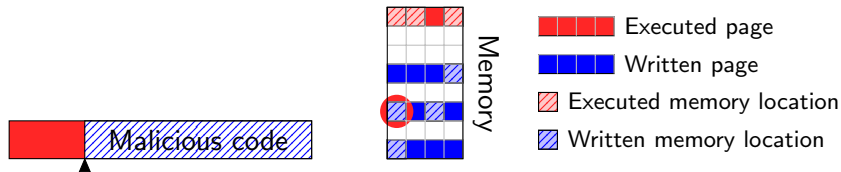
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms



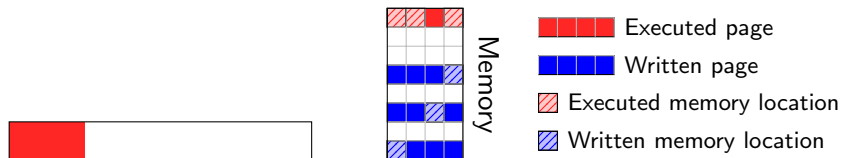
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms



Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

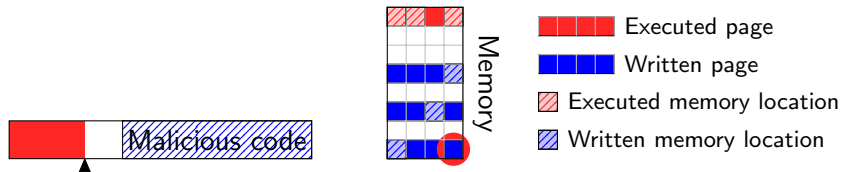


Unfortunately...

- ▶ Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- ▶ Coarse-grained memory accesses tracking further increases the chances to detect **spurious unpacking stages** (up to hundreds of thousands stages)

Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

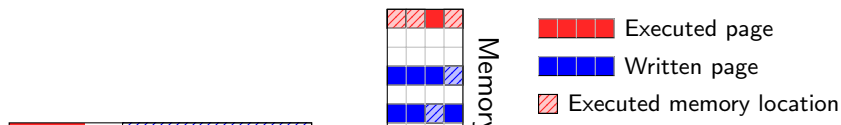


Unfortunately...

- ▶ Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- ▶ Coarse-grained memory accesses tracking further increases the chances to detect **spurious unpacking stages** (up to hundreds of thousands stages)

Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms



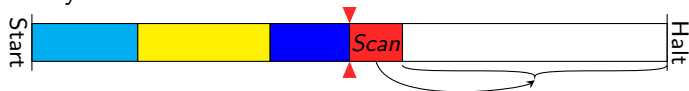
The overhead introduced by invoking the malware detector on every time a written page is executed is prohibitive!

Unfortunately...

- ▶ Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- ▶ Coarse-grained memory accesses tracking further increases the chances to detect **spurious unpacking stages** (up to hundreds of thousands stages)

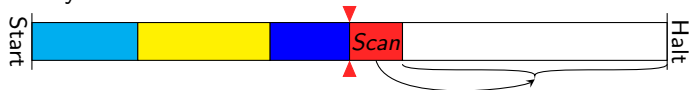
Better approximating the end of an unpacking stage

Ideally:

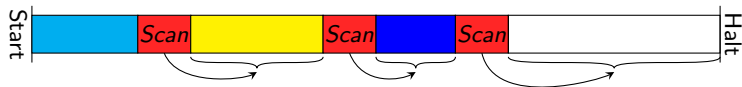


Better approximating the end of an unpacking stage

Ideally:

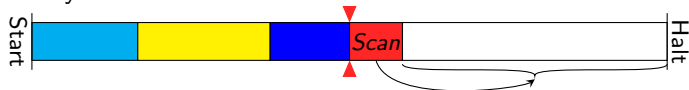


With coarse-grained memory access tracking:

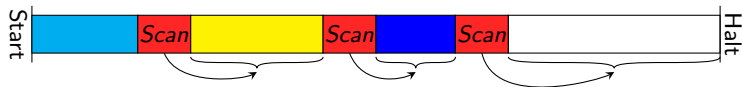


Better approximating the end of an unpacking stage

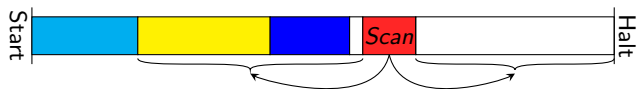
Ideally:



With coarse-grained memory access tracking:



Mitigate the imprecision of the coarse-grained memory accesses tracking by considering an unpacking stage concluded when the execution of a previously written page is followed by a **dangerous system call**



Dangerous system calls

To achieve its malicious goals, the malware has to interact with the system (through system calls)

Dangerous system calls

To achieve its malicious goals, the malware has to interact with the system (through system calls)

Only few system calls are dangerous

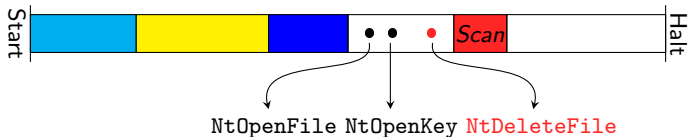
A system call is dangerous if its execution can leave the system in an unsafe state

Dangerous system calls

To achieve its malicious goals, the malware has to interact with the system (through system calls)

Only few system calls are dangerous

A system call is dangerous if its execution can leave the system in an unsafe state

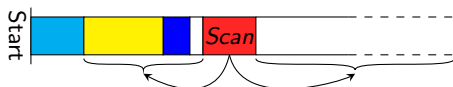


Continuous monitoring of the execution

- ▶ OmniUnpack implements a **continuous monitoring approach**, where the execution is observed in its entirety
- ▶ Continuous monitoring is the solution to the undecidability of the problem of detecting the real end of an unpacking stage
- ▶ The low overhead introduced by the system allows for continuous monitoring in an end-user environment

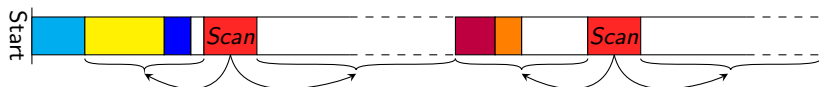
Continuous monitoring of the execution

- ▶ OmniUnpack implements a **continuous monitoring approach**, where the execution is observed in its entirety
- ▶ Continuous monitoring is the solution to the undecidability of the problem of detecting the real end of an unpacking stage
- ▶ The low overhead introduced by the system allows for continuous monitoring in an end-user environment



Continuous monitoring of the execution

- ▶ OmniUnpack implements a **continuous monitoring approach**, where the execution is observed in its entirety
- ▶ Continuous monitoring is the solution to the undecidability of the problem of detecting the real end of an unpacking stage
- ▶ The low overhead introduced by the system allows for continuous monitoring in an end-user environment



OmniUnpack algorithm

Input: an execution trace $\langle e_0, e_1, \dots \rangle$

where a trace event can be:

$w(p)$ write access to a memory page p

$x(p)$ instruction execution from a memory page p

s invocation of the system call s

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1		
2		
...		

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1		
2		
...		

The memory page 0 is executed

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1		
2	•	
...		

The memory page 2 is written
The page is recorded in the set W of written pages

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

s_0 is NtOpenFile

Memory pages status

Page #	Access	
	W	WX
0		
1		
2	•	
...		

The system call s_0 is executed (not dangerous and WX is empty)

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1	•	
2	•	
...		

The memory page 1 is written

The page is recorded in the set W of written pages

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1	•	•
2	•	
...		

The memory page 1 is executed

The page is recorded in the set WX of written-then-executed pages

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

s_1 is NtOpenKey

Memory pages status

Page #	Access	
	W	WX
0		
1	•	•
2	•	
...		

The system call s_1 is executed (not dangerous)

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1	•	•
2	•	•
...		

The memory page 2 is executed

The page is recorded in the set WX of written-then-executed pages

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

s_2 is NtDeleteFile

Memory pages status

Page #	Access	
	W	WX
0		
1	•	•
2	•	•
...		

The system call s_2 is executed (dangerous)

The malware detector is invoked to scan all the memory pages in W

OmniUnpack algorithm

Execution trace

$\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \dots \rangle$

Memory pages status

Page #	Access	
	W	WX
0		
1		
2		
...		

If the program is not malicious the sets W and WX are emptied and the execution is resumed

Outline

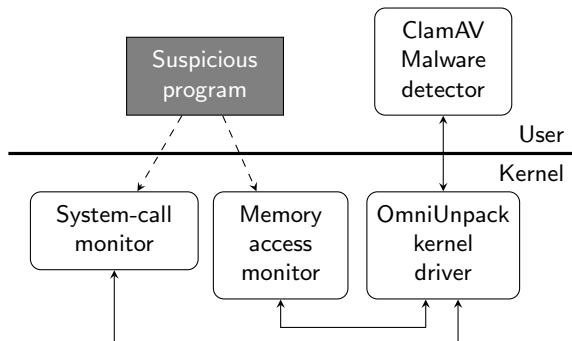
OmniUnpack

Implementation

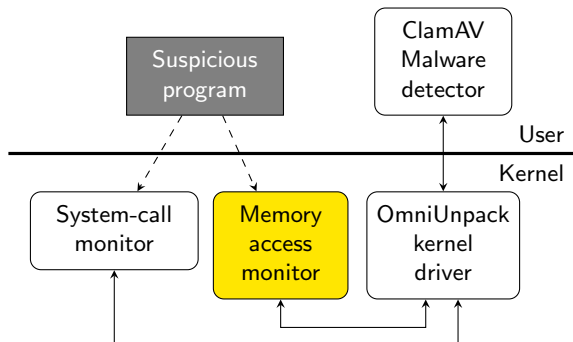
Evaluation

Conclusion

OmniUnpack for Microsoft Windows XP

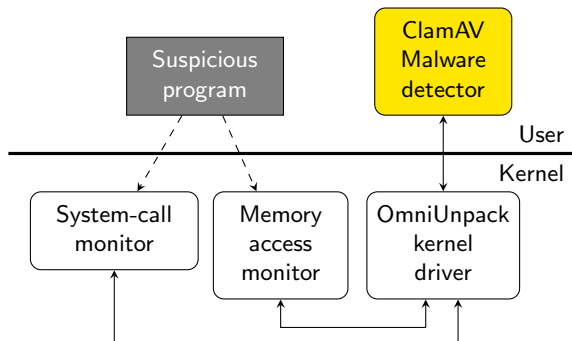


OmniUnpack for Microsoft Windows XP



- ▶ The $W \oplus X$ policy is enforced on the memory pages of the suspicious program
- ▶ Page-fault exceptions are trapped by OmniUnpack
- ▶ Non executable pages can be emulated via software

OmniUnpack for Microsoft Windows XP



- ▶ Any malware detection strategy can be used to scan the code generated during the previous stage

Outline

OmniUnpack

Implementation

Evaluation

Conclusion

Experimental evaluation

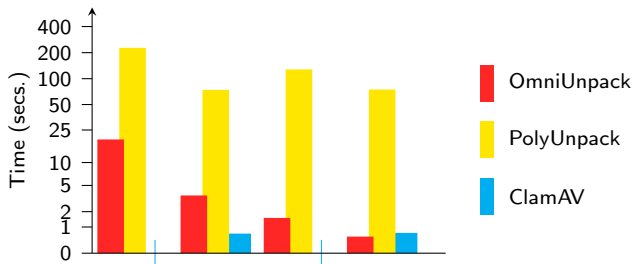
- ▶ OmniUnpack successfully monitored all the analyzed samples and detected the end of each unpacking stage (independent of the packing algorithm)
- ▶ Reduction of the average number of unpacking stages from 2089.79 to 7.14

Experimental evaluation

- ▶ OmniUnpack successfully monitored all the analyzed samples and detected the end of each unpacking stage (independent of the packing algorithm)
- ▶ Reduction of the average number of unpacking stages from 2089.79 to 7.14
- ▶ Reduced overhead on benign programs
- ▶ Improved detection rate

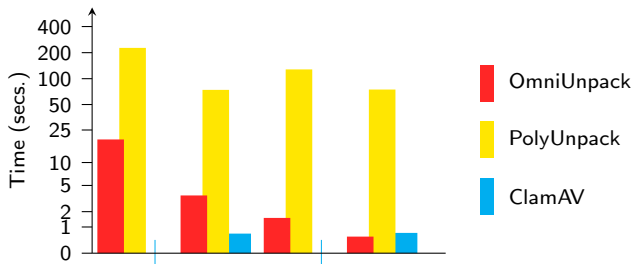
Malware detector	Unpacking	Detection rate
ClamAV	25%	15%
OmniUnpack-enhanced ClamAV	—	80%

Time to unpack w.r.t. ClamAV and PolyUnpack



See the paper for detailed results

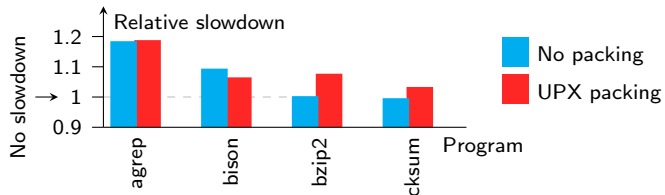
Time to unpack w.r.t. ClamAV and PolyUnpack



- ▶ OmniUnpack was significantly faster than PolyUnpack (e.g., with UPX OmniUnpack was 20.56 times faster)
- ▶ In $\sim 42\%$ of the cases PolyUnpack timed out (300 secs.)
- ▶ OmniUnpack was within one order of magnitude of ClamAV (e.g., with UPX OmniUnpack was only 5 times slower)
- ▶ In 75% of the cases ClamAV failed to unpack

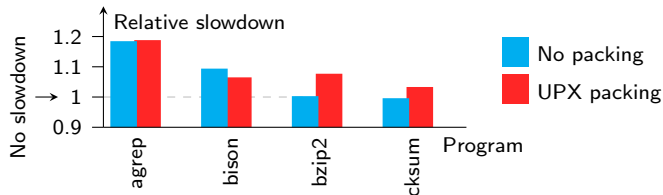
See the paper for detailed results

Overhead with benign programs



See the paper for detailed results

Overhead with benign programs



- ▶ ~6% average overhead on non-packed applications
- ▶ ~11% average overhead on packed applications (with UPX)

See the paper for detailed results

Outline

OmniUnpack

Implementation

Evaluation

Conclusion

OmniUnpack implements a new unpacking technique addressing the shortcomings of the current unpackers

- ▶ Generic unpacker
- ▶ Suitable for use on end-user environments (minimize the overhead by using existing hardware mechanisms and by working directly in the OS)
- ▶ Safe in the presence of multiple unpacking stages (incremental invocation of the malware detector)
- ▶ Improved detection rate (any malware detector can delegate to OmniUnpack the recovery of the packed code)

OmniUnpack

Fast, Generic, and Safe Unpacking of Malware

Thank you!
Questions?