PoMMaDe: Pushdown Model-checking for MAlware DEtection*

Fu Song
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University
Shanghai, P.R.China
fsong@sei.ecnu.edu.cn

Tayssir Touili
LIAFA, CNRS and University Paris Diderot
Paris, France
touili@liafa.univ-paris-diderot.fr

ABSTRACT

We present PoMMaDe, a Pushdown Model-checking based MAlware DEtector. In PoMMaDe, a binary program is modeled as a pushdown system (PDS) which allows to track the stack of the program, and malicious behaviors are specified in SCTPL or SLTPL, where SCTPL (resp. SLTPL) is an extension of CTL (resp. LTL) with variables, quantifiers, and predicates over the stack (needed for malware specification). The malware detection problem is reduced to SCTPL/SLTPL model-checking for PDSs. PoMMaDe allows us to detect 600 real malwares, 200 new malwares generated by two malware generators NGVCK and VCL32, and prove benign programs are benign. In particular, PoMMaDe was able to detect several malwares that could not be detected by well-known anti-viruses such as Avira, Avast, Kaspersky, McAfee, AVG, BitDefender, Eset Nod32, F-Secure, Norton, Panda, Trend Micro and Qihoo 360.

Categories and Subject Descriptors

F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs—*Malware Detection*

General Terms

Security, verification

Keywords

Model-Checking, Malware Detection, Pushdown Systems

1. INTRODUCTION

Malwares are one of the most prevalent security threats on the Internet. These programs are used to attack organizations and countries, e.g., the notorious malware *Flame* has been used for targeted cyber espionage in Middle Eastern countries and was not detected by any anti-virus for more than five years. Thus, malware detection is a critical topic in computer security. Classic antivirus techniques: code emulation (dynamic analysis) and signature (pattern)-based techniques become insufficient. Indeed, code emulation techniques monitoring only few traces of programs in a limited time span may miss some malicious behaviors. Signature-based techniques using patterns of programs' codes to characterize malwares can detect only known malwares and are easy to get around.

Addressing these limitations, many works proposed to apply model-checking to detect malware [5, 9, 10, 14, 13, 18, 8, 3, 4], as model-checking checks the behavior (not the syntax) of the program in a *static* way. However, all these works are not able to model the stack of the programs. Being able to track the stack is important for malware detection, since many obfuscation techniques rely on operations over the stack to evade anti-viruses [16].

To solve this problem, in our previous works [21, 19, 22], we proposed to model a binary program as a pushdown system (PDS). This allows to track the stack of the program. We introduced two new logics SCTPL [21, 19] and SLTPL [22] which are extensions of CTL and LTL with variables, quantifiers, and predicates over the stack. We reduced malware detection to SCTPL/SLTPL model-checking for PDSs. SCTPL/SLTPL allow us to express malicious behaviors in a more succinct manner and specify properties on the stack content which is important for malware detection. In [21, 19, 22], we showed that SCTPL/SLTPL model-checking for PDSs allow to have efficient and robust malware detection techniques.

In this paper, we present PoMMaDe, a malware detector based on SCTPL/SLTPL model-checking for PDSs. PoMMaDe takes as inputs a binary program and a set of SCTPL/SLTPL formulas specifying malicious behaviors. PoMMaDe outputs **Yes** if the binary program satisfies one of the formulas. It means that the program may be a malware. Otherwise, PoMMaDe outputs **No**, meaning that the program is benign.

PoMMaDe first uses a preprocessor to automatically unpack the program if it is packed. The (resulting) program is disassembled based on Jakstab [15] and IDA Pro [12]. Jakstab performs static analysis of the binary program, provides an assembly program and the values of the registers and memory addresses. We use IDA Pro to obtain API functions' informations which cannot be obtained by Jakstab. Then, the assembly program is translated into a PDS. Nex-

*This work is partially funded by ANR grant ANR-08-SEGI-006, Shanghai Knowledge Service Platform for Trustworthy Internet of Things No. ZF1213 and NSFC Project No.91118007.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2494599

t, PoMMaDe applies the SCTPL/SLTPL model-checking algorithms of [21, 19, 22] to check whether or not the PDS satisfies one of the formulas. If this is the case, PoMMaDe outputs **Yes**, meaning that the program may be a malware. Otherwise, PoMMaDe outputs **No**, meaning that the program is benign. Moreover, in PoMMaDe, we implement three optimization strategies which improve the performance when checking benign programs.

Our tool PoMMaDe allows to detect 600 real malwares, all of 200 new malwares generated by the best two malware generators NGVCK and VCL32. Several of these new malwares could not be detected by several well-known antiviruses such as: Avira, Avast, Kaspersky, McAfee, AVG, BitDefender, Eset Nod32, F-Secure, Norton, Panda, Trend Micro and Qihoo 360. NGVCK and VCL32 are the best malware generators as shown in [24]. Moreover, PoMMaDe can prove that benign programs are benign and detect the notorious malware Flame which was not detected for more than 5 years. The results show the efficiency and the applicability of our tool. PoMMaDe is downloadable at `http://www.liafa.univ-paris-diderot.fr/~song/pommade`.

Related Work. BitScope [7] is applied to detect trigger-based behaviors in malware. Panorama [11] is a tool for spyware detection. However, [7, 11] are dynamic analysis based tools. Thus, they can miss malware behaviors if they appear after the fixed time interval of the dynamic analysis. PyEA [8] is a malware detector that uses control flow graph matching. PyEA cannot track the stack behaviors which is important for malware detection. Other software model-checkers, such as SLAM [2], Blast [6] and PuMoC [20], dedicated to source code verification, cannot detect malware, since malwares are usually given in binary codes. Binary code analyzers such as CodeSurfer/x86 [1] and McVeto [23] check reachability and cannot detect malware. Indeed, many malicious behaviors are more complex than reachability. Jakstab [15] only considers disassembly and static analysis of binary codes rather than malware detection. In PoMMaDe, we implemented the model-checking algorithms of [21, 19, 22]. In these previous works, we had a preliminary implementation that was able to detect 270 malwares. We had to manually check whether a binary program is packed or not. If this is the case, we have to manually unpack the binary program. This task is completely automatic in PoMMaDe. Moreover, PoMMaDe is able to detect more than 600 real malwares.

2. BACKGROUND

2.1 Pushdown Systems

A *pushdown system* (PDS) $\mathcal{P}$ is a tuple (P, Γ, Δ), where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. A configuration of $\mathcal{P}$ is a pair $\langle p, \omega \rangle$ where $p \in P$ and $\omega \in \Gamma^*$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $(p, \gamma, q, \omega) \in \Delta$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. An execution (path) of $\mathcal{P}$ from c_0 is a sequence of configurations $\pi = c_0 c_1 ...$ s.t. for every $i \geq 0$, $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$.

We use the approach of [19] to translate a binary program into a PDS. Intuitively, a binary program is first disassembled into an assembly program by Jakstab and IDA Pro. Then, the assembly program is translated into a PDS in

```
push m
push f
call ReadFile
...
push m
push c
call send
```

Figure 2: A fragment of a Data-stealing malware.

which the control locations correspond to the control points of the program, the stack alphabet consists of the return addresses and the values of the operands of all the push statements. Each transition rule encodes a statement of the program. Thus, a path of the PDS mimics an execution trace of the program.

2.2 SCTPL and SLTPL

In PoMMaDe, malicious behaviors are specified in SLTPL or SCTPL. SLTPL (resp. SCTPL) can be seen as an extension of LTL (resp. CTL) with variables, quantifiers and predicates over the stack content. Variables are parameters of atomic predicates and can be quantified by the existential and universal quantifiers. We use regular expressions over the stack alphabet and variables to represent predicates over the stack content. The formal definitions can be found in [21, 19, 22]. For example, the SCTPL formula $\Psi_{sp} = \exists m \mathbf{EF}\big(call(GetModuleFileNameA) \wedge 0m\Gamma^* \wedge \mathbf{EF}(call(CopyFileA) \wedge m\Gamma^*)\big)$ is a malicious behavior of worms, where $0m\Gamma^*$ (resp. $m\Gamma^*$) is a regular expression specifying that the top of the stack is $0m$ (resp. m). Ψ_{sp} states that there exists an address m and a path in which *GetModuleFileNameA* is called when $0m$ is on the top of the stack (i.e., this function is called with 0 and m as parameters, since parameters are passed through the stack in assembly). Later, *CopyFileA* is called when m is on the top of the stack (i.e., *CopyFileA* is called with m as parameter). Worms commonly copy themselves to other locations. To do this, a worm first calls *GetModuleFileNameA* with 0 and an address m as parameters. By calling this function, its own file name is stored in the address m. Later, it calls *CopyFileA* with m as parameter (i.e., its file name) to copy itself to other locations. The above formula describes this malicious behavior.

2.3 The Model-Checking Algorithms

PoMMaDe is based on the model-checking algorithms of [21, 22], where SCTPL/SLTPL model checking for PDSs is reduced to the emptiness problem for Symbolic (Alternating) Büchi Pushdown Systems. We refer the reader to [21, 22] for more details.

2.4 Example: Data-stealing Malware

We show how to use SLTPL to express the malicious behavior of a data-stealing malware. More malicious behaviors are described in Appendix B. Note that there exist malicious behaviors that can be expressed in SLTPL but not in SCTPL, and vice versa.

The main purpose of a data-stealing malware is to steal the user's personal confidential data such as username, password, credit card number, etc and send it to another computer (usually the malware writer). The typical behavior of data-stealing malware can be summarized as follows: the

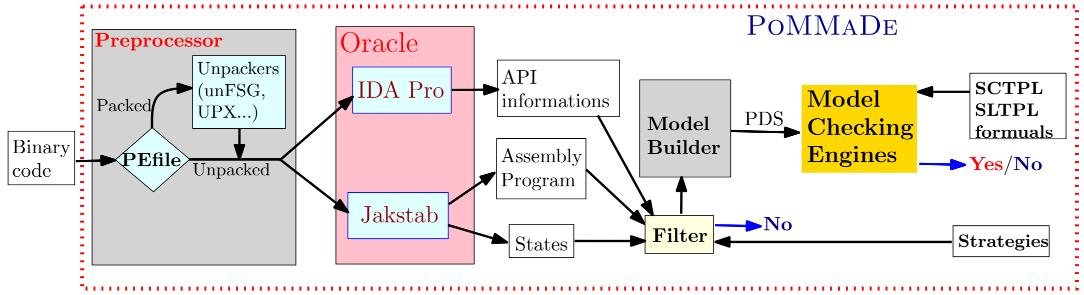

Figure 1: Architecture of PoMMaDe.

malware will first call the API function *ReadFile* in order to load some file of the victim into memory. To do this, it needs to call this function with a file pointer f (i.e., the return value of the calling function *OpenFile*) as the first parameter and a buffer m as the second parameter (m corresponds to the address of a memory location), i.e., with $f\ m$ on the top of the stack since in assembly, function parameters are passed through the stack. Then, the malware will send its file (whose data is pointed by m) to another computer using the function *send*. It needs to call *send* with a connection c (i.e., the return value of the calling function *socket*) as first parameter and the buffer m as the second parameter, i.e., with $c\ m$ on the top of the stack. Figure 2 shows a disassembled fragment of a malware corresponding to this typical behavior. Before calling the function *ReadFile*, it pushes the two parameters m and f onto the stack. Later it calls the function *send* after pushing the two parameters m and c onto the stack. This behavior can be expressed by the following SLTPL formula $\Phi_{ds} = \exists m\ \mathbf{F}\Big(call(ReadFile) \wedge \Gamma\ m\ \Gamma^* \wedge \mathbf{F}\big(call(send) \wedge \Gamma\ m\ \Gamma^*\big)\Big)$, where the expression $\Gamma\ m\ \Gamma^*$ states that the second value of the stack is m (corresponding to the second parameter of the function *ReadFile* and *send*). Φ_{ds} states that there exists an address m which is the second parameter when calling *ReadFile*, and such that later, eventually, *send* will be called with m as its second parameter.

3. ARCHITECTURE AND IMPLEMENTATION

Given a binary program, and a set of malicious behaviors expressed by SCTPL or SLTPL formulas, PoMMaDe outputs **Yes** if the program satisfies one of the formulas. It means that the binary program may be a malware. Otherwise, PoMMaDe outputs **No**, meaning that the binary program is benign. As shown in Figure 1, PoMMaDe consists of five components: **Preprocessor**, **Oracle**, **Filter**, **Model Builder** and **Model-Checking Engines**.

Preprocessor uses PEfile [17] to check whether the binary program is packed or not. If this is the case, it uses the corresponding unpacker (if it exists) to unpack the binary code and feed the resulting binary program to **Oracle**. Otherwise, it directly passes the binary code to **Oracle**. So far, PoMMaDe supports dozens of popular packers for Windows, and hundreds of packers for Linux. Moreover, users can easily add a new unpacking tool by modifying the database file. **Oracle** takes as input a (unpacked) binary program and outputs the assembly program, and the informations of API functions and the states (values of the registers and memory addresses at each control point). **Oracle** relies on Jakstab [15] and IDA Pro [12]. Jakstab performs static analysis of the binary program, provides an assembly program and the states. However, it does not allow to extract the informations of API functions and some indirect calls to the API functions. PoMMaDe uses IDA Pro to get these informations. The outputs of **Oracle** are used by **Filter** to filter out benign programs according to the given optimization strategy by "syntactically" checking the assembly program. Indeed, suppose we would like to check whether a program P satisfies the formula Φ_{ds} described above. Then if P does not contain any call to the functions *ReadFile* or *send*, we can deduce that P does not contain this malicious behavior Φ_{ds} only by performing a syntactical check over it. We don't need to apply model-checking to reach this conclusion. PoMMaDe provides three strategies: (1) *keywords strategy*, (2) *sequence strategy* and (3) *direct strategy*. When "keywords strategy" is chosen, the user has to provide a set of instructions to PoMMaDe (in our example, the user can provide $\{call(ReadFile), call(send)\}$). **Filter** will syntactically check whether or not the assembly program contains these instructions. If this is not the case, PoMMaDe outputs **No** (we know that P does not contain this malicious behavior, no need to apply model-checking). Otherwise, **Model Builder** is called (we need to apply model-checking to decide whether P is a malware or not). When "sequence strategy" is chosen, the user has to provide a sequence of instructions to PoMMaDe (in our example, the user can provide the sequence $call(ReadFile); call(send)$). **Filter** will "syntactically" check in the control flow graph of the assembly program whether or not these instructions occur in the same order as in the sequence. If this is not the case, PoMMaDe outputs **No** (no need to apply model-checking, P is benign). Otherwise, **Model Builder** is called. If "direct strategy" is chosen, **Model Builder** is directly called. **Model Builder** outputs a PDS modeling the assembly program. **Model-Checking Engines** takes as input the PDS from **Model Builder** and performs model-checking of the PDS against all the formulas given by the user. PoMMaDe outputs **Yes** if there is one formula satisfied by the PDS, **No** otherwise.

4. EXPERIMENTS

All the experiments were run on a Linux platform (Fedora 13) with a 2.4GHz CPU, 2GB of memory. The time limit is fixed to 20 minutes.

Detecting Real Malwares: PoMMaDe is applied to check more than 600 real malwares and 400 benign programs tak-

Table 1: Detection rates of new malwares generated by NGVCK and VCL32.

Generator	No. of Variants	PoMMaDe	Avira	Kaspersky	Avast	Qihoo 360	McAfee	AVG	BitDefender	Eset Nod32	F-Secure	Norton	Panda	Trend Micro
NGVCK	100	**100**%	0%	23%	18%	68%	100%	11%	97%	81%	0%	46%	0%	0%
VCL32	100	**100**%	0%	2%	100%	99%	0%	100%	100%	76%	0%	30%	0%	0%

en from Microsoft Windows System. PoMMaDe can detect all the real malwares and prove benign program are benign with only three false positives. PoMMaDe was able to detect several families of malwares using only one formula. In particular, our tool can detect the malware *Flame*, the most complex attack toolkit which can record audio conversations, intercept the keyboard, etc. Flame has been active for more that 5 years and was not detected by any antivirus.

The average time (resp. memory) of checking one malware against the satisfiable formula is 93.57 seconds (resp. 225.15 MB). The average time (resp. memory) of checking one benign program against all the malicious behaviors we considered is 24.73 seconds (resp. 21.86 MB) (Thanks to **Filter**, PoMMaDe does not need to apply model-checking for many benign programs).

PoMMaDe **vs Existing Antiviruses:** To compare our tool with the well-known existing anti-viruses, and show its robustness, we automatically created 200 new malwares using the generators NGVCK and VCL32. [24] showed that these systems are the best malware generators. These programs use very sophisticated features such as anti-disassembly, anti-debugging, anti-emulation, and anti-behavior blocking and come equipped with code morphing ability which allows them to produce different-looking viruses. Our results are reported in Table 1. PoMMaDe was able to detect all these 200 malwares, whereas several well-known and widely used anti-viruses such as Avira, Avast, Kaspersky, McAfee, AVG, BitDefender, Eset Nod32, F-Secure, Norton, Panda, Trend Micro and Qihoo 360 were not able to detect several of them.

5. REFERENCES

[1] G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In *CAV*, 2005.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[3] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *RV*, pages 168–182, 2010.

[4] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Abstraction-based malware analysis using rewriting and model checking. In *ESORICS*, pages 806–823, 2012.

[5] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *SREIS*, 2001.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[7] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. 2008.

[8] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *DIMVA*, 2006.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, 2003.

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

[11] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, 2007.

[12] Hex-Rays. IDAPro, 2011.

[13] A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *EUROCAST*, 2007.

[14] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *DIMVA*, 2005.

[15] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *CAV*, 2008.

[16] A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Software Eng.*, 31(11), 2005.

[17] PEfile. `http://code.google.com/p/pefile`, 2012.

[18] P. K. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *IAW*, 2003.

[19] F. Song and T. Touili. Efficient malware detection using model-checking. In *FM*, 2012.

[20] F. Song and T. Touili. PuMoC: A CTL Model-Checker for Sequential Programs. In *ASE*, 2012.

[21] F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS*, 2012.

[22] F. Song and T. Touili. LTL Model-Checking for Malware Detection. TACAS, 2013.

[23] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps. Directed proof generation for machine code. In *CAV*, pages 288–305, 2010.

[24] W. Wong. Analysis and detection of metamorphic computer viruses. Master's thesis, San Jose State University, 2006.