PRECISE PACKER DETECTION USING MODEL CHECKING

Nguyen Minh Hai, Do Duy Phong, Quan Thanh Tho, Le Duc Anh

Ho Chi Minh City University of Technology, Vietnam

hainmmt@cse.hcmut.edu.vn, doduyphongbktphcm@gmail.com, qttho@cse.hcmut.edu.vn, tintinkool@gmail.com

ABSTRACT Over the past decades, malware has been becoming a real threat. It costs more than $10 billion in each year and the damage is still increasing. Most of the modern popular malwares are either packed or obfuscated. The main goal of these obfuscation techniques is to thwart the signature based technique of anti-virus software. It also increases the difficulty of the reverse engineering work since it often takes a very long time for unpacking or decrypting a packed file. As a counter solution, most of anti-virus software tends to detect packer signature for verifying the packed malware. However, since hacker can easily modify signature header of packed file, this solution cannot determine precisely whether a malware is packed or not. This paper proposes a model checking method for packer detection using a combination BE-PUM tool and model checker NUSMV. BE-PUM (Binary Emulator for PUshdown Model generation) is designed for generating a precise control flow graph (CFG), under presence of typical obfuscation techniques of malware e.g. indirect jump, self-modification, overlapping instructions, and structured exception handler (SEH), which are supported in packers. Currently, BE-PUM can cover the patterns of 14 techniques mainly used in 27 packers e.g. UPX, FSG, NPACK, ASPACK, PECOMPACT, PETITE, YODA, TELOCK... Applying the temporal logic formula for that patterns as properties of proposed model checker tool, we can detects totally all the malwares which are packed by these packers. We have implemented our technique for automatically detecting packed malware. The experiment results are encouraging.

1. INTRODUCTION

Most of the modern popular malwares are either packed or obfuscated. The main goal of these techniques is to thwart the signature based technique of anti-virus softwares. It also increases the difficulty of the reverse engineering work since it often takes a very long time for unpacking or decrypting a packed file. As a counter solution, most of anti-virus software tends to detect packer signature for verifying the malware. However, since packer can be utilized in normal software for protecting against hacking and invalid cracking, this solution cannot determine precisely whether a packed target is a malware or not.

According to [1,2] malware is packed by many kinds of packers. Among them, the most popular packers are UPX[1], PECOMPACT[2], TELOCK[3], FSG[4], YODA's Crypter and Protector and ASPACK. The packer transforms the targeted file into another compressed executables which preserves the original functionality. This new packed binary contains a restoration loader stub which decrypts the original file with different algorithms specific to each packer. After unpacking, it then transfers the control flow to original entry point. Moreover, many packers e.g. TELOCK, YODA's Crypter are provided with a armored stub for protecting against straightforward reverse engineering, cracking and tampering with many special techniques of anti-debugging and anti-reversing trick.

In this paper, we introduce a new method of packer detection on 2000 real-world malwares. We combined two tools, BE-PUM for CFG generation and model checker NUSMV for packer detection. The rest of this paper is organized as followed. Section 1 briefly describes the packers and techniques which are used in them. Section 2 introduces the tools BE-PUM and NUSVM which are used for detecting obfuscation techniques. In the next section, section 3 and section 4, we presents our techniques for identifying packers. Section 5 shows our experiments on 2000 malwares taken from Loria. The final section is the conclusion of our paper.

1.1 Overview of packers:

A packer is a software that can mutate a binary file

into another executable. The new executable preserves the original file's functionality, but has a different content on the system. This feature prevents the process of linking between them. Packers are used on executable for mainly two reasons: to reduce the size of binary file, and to evade analysis, reverse engineering, or detection.
For the first reason, packer minimizes targeted file by compressing its content and then uncompressing it on-the-fly during the execution. However, existing real-world packers are used mainly for the second reason, i.e. to protect the original file from being observed, analyzed and tampered with. For achieving this goal, packer combines many obfuscation methods which include anti-debugging, anti-cracking, anti-tracing, anti-reverse engineering, and more for preventing target file from straightforward analysis. These packers are used for protecting the licensed softwares or games from crackers. However, this feature is also exploited in malware for protecting them from detection of anti-virus software. From [1], 79 % of malwares use packing techniques for evading the detection.

1.2 Packer obfuscation techniques:

Packer contains many obfuscation techniques, which make binary code difficult to explore. Packer techniques are investigated in [3], and with our observation on malware, we categorize them into 6 groups such that each group consists of element techniques below. We will briefly explain each element technique

- Entry/code placing obfuscation which is Code layout consists of overlapping functions, overlapping blocks, code chunking and Dynamic code consists of overwriting and packing/unpacking.
- Self-modification which is Dynamic code and overlapping blocks.
- Instruction obfuscation which is also called as indirect jump.
- Anti-tracing consists of SEH (structural exception handling) and 2 APIs (the use of two special APIs comprises of LoadLibrary function which base module is kernel32.dll and GetProcAddress function which base module is kernel32.dll.
- Arithmetic operation which is obfuscated constants and checksumming.
- Anti-tampering which is checksumming, anti-debugging, anti-rewriting. Anti debugging comprises of timing check and hardware breakpoints.
- Anti rewriting further consists of stolen bytes and checksumming.

2. OVERVIEW OF BE-PUM AND NUSMV

2.1 BE-PUM

2.1.1 Contributions:

Nowadays, malware detection is not simply signature based detection. The malware detection focuses on constructing the precise control flow graph of malware which are obtained by disassembly. This statiscial method is used widely in commercial as profiency technique to disassembly the malware and generate the control flow graph as model of malware, e.g. some commercial dissamblers tool like IDA Pro, HOPPER, as well as non-commercial using likes METASM, Capstone, Unicorn or Jackstab. However, it is easily cheated by typical obfuscation techniques or anti-reversing techniques. As mentioned above, malware uses packers to obfuscate them for preventing disassembling.

BE-PUM (Binary Emulator for Pushdown Model Generation) is the framework which can handle all of obfuscation techniques. BE-PUM can also unpack completely the packer and generate the precise model for packer which other popular disassembler tool i.e. IDA Pro, Jackstab fails. BE-PUM has implemented many techniques to bypass all of the obfuscation, especially anti-reversing techniques. The techniques implemented in BE-PUM are the combination of on-the-fly control flow graph generation, dynamic symbolic execution (concolic testing), formal x86 instruction and API calling as the special stub.

2.1.2 BE-PUM Architecture:

BE-PUM implements the CFG reconstruction based on concolic testing with SMT Z3 as a backend engine to generate a test instance for concolic testing. Core of BE-PUM is framework Jackstab, it also a preprocessor to compute a single-step disassembly. The Fig. 1 shows the architecture of BE-PUM, which consists of three

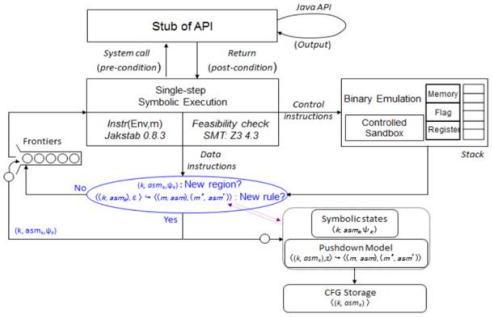

Fig. 1 BE-PUM architecture.

components: symbolic execution, binary emulation, and CFG storage. The symbolic execution picks up one from the frontiers (symbolic states at the ends of explored execution paths), and it tries to extend one step. If the instruction is a data instruction, it will simpy disassemble the next instruction. If the instruction is a control instruction, the concolic testing is applied to decide the next location. Note that some variable does not appear in the path-condition, the SMT will not return its value. If the concolic testing needs this value, BE-PUM terminates. When either a new CFG node or a CFG edge is found, they are stored in CFG storage and a configuration is added to the frontiers. This procedure continues until either the exploration has converged, or reaching to unknown instruction, system calls, and/or

addresses.

2.1.3 Limitation:

There are several limitations in BE-PUM. First, the number of X86 instructions are about 1000 and Windows API are more than 4000. Current BE-PUM covers only 200 x86 instructions and 400 APIs. They are selected by the frequency appearing in malware from VX Heaven. Second, BE-PUM needs to support methods of handling loop invariant. These are the future works.

2.2 NuSMV

NuSMV is an open source tool for the model checking on finite state systems. NuSMV only accepts NuSMV model which described by SMV language. With the correct NuSMV model and the specification which expressed in temporal logics formula, NuSMV supports CTL model checking and LTL model checking. As the big advantage, SMV syntax is clear and it can be easily applied for expressing the model of binary file.

3. COMBINATION OF BE-PUM AND NuSMV

Recall that BE-PUM is a powerful dissasembler tool, which can generate the precise model as CFG for Portable Executable (PE) file. Especially, BE-PUM can handle all the malware techniques. Moreover, BE-PUM is open source and it can be supported by any model checking tool by integrating the model checking tool into BE-PUM. Consequently, the model checker NuSMV is applied for detecting packers based on the observed packer's behavior and the precise packer model generated from BE-PUM. By collecting the patterns of packing techniques and expressed them in the temporal logic formulas which can be used by NuSMV for model checking, we can conclude that whether the file is packed or not.

3.1 BE-PUM model to JSON data:

JavaScript Object Notation abbreviated by JSON is a syntax for storing and expressing the data. BE-PUM convert the model into the JSON data which is more readable and portable. Then JSON data can be used as input of our model checking tool for generating an precise SMV model as general input of NuSMV model checker.

3.2 BE-PUM model to SMV model:

NuSMV takes the input of MV model for the checking process. This section introduces about the SMV model, how it is related to the BE-PUM model, and we propose a method to convert the JSON model to SMV model.

3.2.1 SMV Model:

SMV model basically is same as BE-PUM model which consists of nodes and edges. A node in NuSMV model is a state defined by *addr* is the location of an instruction, *mnem* is instruction's identifier and *op* is operand of instruction. An edge in SMV model is the connection between two nodes of SMV model defined by *src* and *dest*. Fig. 2 is the example of SMV model (b) related to BE-PUM model (a)

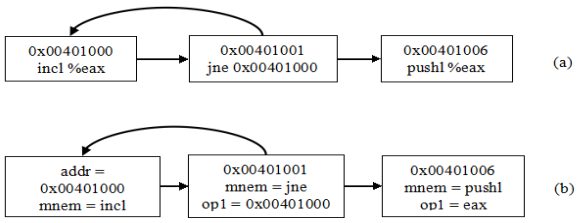

Fig. 2 SMV model generated by BE-PUM model.

3.2.1 SMV Model construction:

Before we can apply the model checking on NuSMV, it is very important to make the SMV model more general. By abstracting the register, segment register, immediate value and API calling, as well as abstracting the type of instruction, the problem can be solved. The main idea is to categorize and abstract the instruction identifier and instruction's operands. Mnemonic of instruction is categorized by type of instruction. The mnemonic will be categorized based on the x86 instruction handler of BE-PUM. Operand of instruction is also categorized by name of operand. The type of register comprises of 16 registers which supported in BE-PUM, in which the type of segment register consists of 6 segment registers; the immediate value is the hexadecimal values, and API calling is any of calling API in the execution. Fig. 3 is the example of applying to SMV model after categorizing the instruction and operands of instruction.

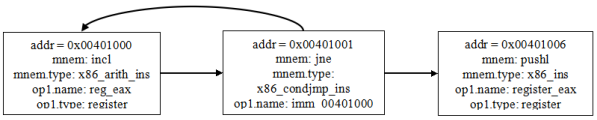

Fig. 3 SMV Model with categorized instruction.

4. PACKER DETECTION WITH NuSMV and BE-PUM:

Structured Exception Handling (SEH) technique can be detected by three sequence instructions. Assumming that statement A is push immediate value to stack, statement B is push fs:[0] value to stack and statement C is move esp value to fs:[0]. The CTL formula to specify the SEH technique can be described is $EF(A \wedge EF(B \wedge EF(C)))$, *and LTL one is* $F(A \wedge F(B \wedge F(C)))$. Indirect Jump technique can be detected by indirect call and indirect jump. Assumming that statement A is call to near address stored in register, statement A' is jump to near address stored in register also. The CTL formula to specify the Inidrect jump technique can be described is $EF(A) \vee EF(A')$, and *LTL one is* $F(A) \vee F(A')$. Anti-debugging technique can be detected by detecting any of calling API IsDebuggerPresent. Assumming that

statement A is IsDebuggerPresent API calling. The CTL formula to specify the anti-debugging technique can be described is EF(A), and LTL one is F(A). Stolen bytes techniques which is VirtualAlloc API using and Timing check which is GetTickCount API, are also specified by CTL, LTL formula same as above. Obfuscated constants technique can be detected by any of instruction whose operand's value is a constant value. Assuming that statement A, A', A'' is the instruction whose first operand, second operand and third operand is immediate value. The CTL formula to specify the Obfuscated constants technique can be described is $EF(A) \vee EF(A') \vee EF(A'')$, and LTL one is $F(A) \vee F(A') \vee F(A'')$. Two Special APIs technique can be detected by detecting any of calling two APIs LoadLibray and GetProcAddress. Assuming that statement A is LoadLibrary API calling and statement B is GetProcAddress API calling. The CTL formula to specify the Two special APIs technique can be described is $EF(A \wedge EF(B))$, and LTL one is $F(A \wedge F(B))$.

5. Experiments:

We perform experiments of packer analysis on 27 packers and packer detection based on model checking method by combining the BE-PUM and model checker NuSMV on 2000 real-world malware from LORIA collection. Our experiments are performed on Windows XP with Intel Core i5 – 2450M 2.5GHz, 2GB RAM. Fig. 4 shows the statiscial of 14 techniques widely used in 27 packers. Fig. 5 shows show the statiscial of packer detection on 2000 real-malwares. In general, our method produces the good result.

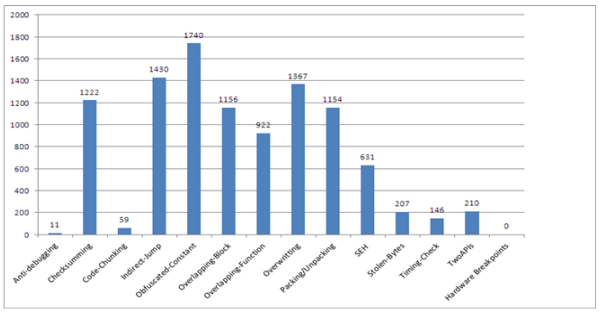

Fig. 4 Statiscial for 14 techniques of 2000 real-malware.

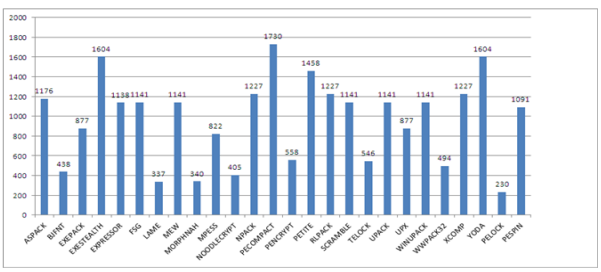

Fig. 5 Statiscial for packer detection of 2000 malwares.

CONCLUSION

In this paper, we have presented a new method for packer detection using a combination BE-PUM tool and model checker NUSMV. BE-PUM applies concolic testing and on-the-fly model generation for handling obfuscation techniques i.e. indirect jump and self-modifying codes which can cover the patterns for 14 techniques mainly used in 27 packers e.g. UPX, FSG, NPACK, ASPACK, PECOMPACT, PETITE, YODA and TELOCK. We have performed the experiments for 2000 real-world malwares. The experiment results show that our approach is very encouraging.

REFERENCES

[1] Anti-virus technology whitepaper. *Technical report*, BitDefender, 2007.

[2] Maik Morgenstern and Andreas Marx. Runtime packer testing experiences. In *2nd International CARO Workshop*, pages 288–305, 2008. LNCS 6174.

[3] K.A. Roundy and B.P. Miller. Binary-code obfuscations in prevalent packer tools. In *ACM Comput. Surv*, volume 46, pages 4:1–4:32, 2013.

Dr. Quan Thanh Tho is an Associate Professor in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam. He received his B.Eng. degree in Information Technology from HCMUT in 1998 and received Ph.D degree in 2006 from Nanyang Technological University, Singapore.

Nguyen Minh Hai is a Phd student at Ho Chi Minh University of Technology (HCMUT). He is also a lecturer at Ho Chi Minh University of Industry.

Anh Duc Le received B.Eng. and M.Sc. degrees in computer science from Ho Chi Minh City University of Technology and Tokyo University of Agriculture and Technology 2011 and 2014, respectively. Since April 2014, he has been a Ph.D. student in the Department of Electronic and Information Engineering at Tokyo University of Agriculture and Technology.

Do Duy Phong is a senior student from Ho Chi Minh City University of Technology.