POSTER: PT-DBG: Bypass Anti-debugging with Intel Processor Tracing

Guancheng Li[†], Yongheng Chen[§], Tianyi Li[†], Tongxin Li[†], Xinfeng Wu[†], Chao Zhang[‡*] and Xinhui Han[†*]
[†]Peking University [‡]Tsinghua University [§]Nanjing University
{atum, litianyi, litongxin, wuxinfeng, hanxinhui}@pku.edu.cn,chaoz@tsinghua.edu.cn,changochen@smail.nju.edu.cn

***Abstract*—Debugging is one of most useful techniques used in reverse engineering and diagnosing. However, some softwares, especially commercial and malicious ones, have embedded anti-debugging techniques to protect themselves from being analyzed. Due to the diversity of anti-debugging techniques, evading anti-debugging is challenging work, which relies heavily on expertise. In this poster, we propose a novel approach to bypass anti-debugging with Intel Processor Tracing (PT). It identifies the location of anti-debugging code with the help of PT, and automatically patches the code to bypass it, enabling developers to debug anti-debugging software.**

***Index Terms*—Reverse Engineering, Anti-debugging**

I. Introduction

Debugging is a popular technique used in software diagnosing and reverse engineering, for vulnerability analysis and malware analysis [1] etc. However, sometimes software (e.g., malware) authors would like to protect their software from being analyzed, and thus deploy some countermeasures. Among them, anti-debugging is one of the most widely used ones.

A process deployed with anti-debugging usually detects the presence of debuggers (including virtual machines) in various ways, and behaves differently (e.g., exit or hide the malicious behavior) when a debugger is identified. Alternative anti-debugging techniques could even interfere debuggers' execution, and stop debuggers from attaching to the process.

To bypass anti-debugging, reverse engineers have to (1) hide the presence of debuggers, or (2) find out the locations of anti-debugging code and disable them. However, there are various types of anti-debugging techniques to detect debuggers. It is challenging to hide debuggers from all of them. Moreover, multiple anti-debugging techniques could be deployed in a single software. So, bypassing all of them costs abundant time and efforts of experienced experts in practice.

In this poster, we present a novel solution PT-DBG to automatically identify anti-debugging code and disable them. It compares two runtime execution traces of target applications, with and without the presence of debuggers, and identifies the divergence checks that are caused by anti-debugging code. Then it patches the applications to bypass these checks.

However, it is challenging to collect the traces of applications with anti-debugging techniques. Traditional trace collecting solutions, e.g., those based on dynamic binary instrumentation (DBI) or virtual machine introspection (VMI), could be detected by anti-debugging techniques. Fortunately, Intel Processor Tracing (PT) [2], a new hardware feature in Intel CPU, provides a new way to collect traces of processes. Moreover, user-mode applications could not detect or interfere the execution of PT. Our solution PT-DBG thus utilizes this hardware feature to collect traces, and provides a strong resilience to existing anti-debugging techniques.

[*] Corresponding Author.

II. Anti-Debugging Techniques

Existing anti-debugging techniques could be classified into the following two categories.

A. Debugger Detection (type-1)

This type of anti-debugging techniques relies on detecting the presence of debuggers (or VMs). The most straightforward solution is directly calling a limited set of APIs, e.g., IsDebuggerPresent. However, this is not a reliable way to detect debuggers, since these APIs could be hooked.

In practice, most solutions exploit footprints introduced by debuggers (or VMs) to detect their presence. For example, many debuggers use INT3 interrupts to set breakpoints, and thus will catch and handle INT3 interrupts. So, a try/catch statement, which deliberately throws and handles an INT3 interrupt, could be used to detect debuggers. If a debugger is present, the interrupt will be caught by the debugger and the catch block will not be executed. Since there are so many unintended footprints introduced by debuggers, it is really challenging to hide debuggers from all of them in practice.

B. Debugger Interfering (type-2)

This type of anti-debugging solutions stop debuggers by interfering with their functionalities. For example, overwriting the DebugPort structure, i.e., an object used for communication between debugger and debuggee, could break the debuggers. According to the survey [3] [4], this type of anti-debugging techniques is rare.

III. System Design And Evaluation

Fig.1 shows the architecture of our approach. At first, it collects the runtime trace of target applications without the presence of debuggers using Intel PT (i.e., shadow process). Then, it compares the traces against the patterns of known type-2 solutions. Once a pattern is matched, it patches the binary and traces accordingly. Furthermore, it collects the runtime trace again using a debugger, and compares the traces

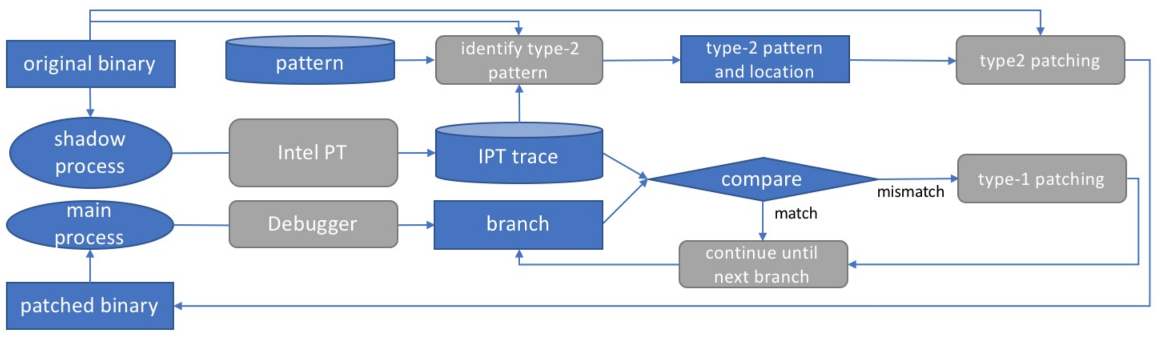

Fig. 1. Architecture of PT-DBG

at each branch, to detect type-1 anti-debugging code. Once a divergence is found, the debugger patches the code at runtime and continues debugging.

A. *Bypass type-2 anti-debugging*

If a program uses the type-2 anti-debugging technique, which interferences with debuggers, we could not collect traces using a debugger. Thanks to the fact that known ways to interfere debuggers are limited, we are able to analyze them case by case and summarize their patterns.

We could thus use Intel PT to collect the traces of target applications (in shadow process), including the branch taken or not taken decision, indirect call/jmp target IP, and exception handling target IP. Then, we will match the traces with known type-2 patterns. As a result, we could get a list of type-2 anti-debugging locations, and then apply the associated binary patching algorithms accordingly. Then, we could further analyze type-1 anti-debugging in the patched binary.

B. *Bypass type-1 anti-debugging*

1) Identify type-1 anti-debugging code: We will reuse the traces collected by Intel PT in the previous step. Then, we will use the debugger to start the main process to analyze the target application. The main process is fed with the same input as the shadow process. Moreover, all randomness sources (e.g., the random function) are patched. So, the control flow of the main process and the one of the shadow process are identical until the anti-debugging code is triggered.

The debugger then inspects the main process' control flow, including targets of each conditional branch instruction, indirect call/call instruction and exception handling, with the one collected by Intel PT. If the control flow diverges at some point, then an anti-debugging code is identifies.

It is worth noting that, we ignore control flow inside library functions that are not shipped with the target application, since they are not related to anti-debugging.

2) Patch type-1 anti-debugging code: After identifying the anti-debugging code, we will patch the execution and force it to bypass the anti-debugging checks and continuously execute.

Depending on the type of the control flow divergences, different algorithms will be used to patch the binary. If it diverges at a conditional branch, we could simply overwrite this instruction with an unconditional branch that goes to the correct target. If it diverges when handing exceptions, the debugger will pass the exceptions caught by the debugger back to the target application again. In a rare case, the control flow diverges at indirect call/jmp instructions. In this case, we will patch the target to the right place.

C. *Preliminary Evaluation*

We have analyzed several anti-debugging challenges from famous international CTF competitions. The result shows that all the anti-debugging techniques in these challenges can be evaded by our approach effectively. We will further evaluated our solution on real world software in the future.

IV. Related-Work

There are some new anti-debugging techniques to protect software from being debugged [5]. But bypassing anti-debugging is rarely studied in the community [6] [7]. Few solutions aim at bypassing anti-debugging automatically. To our best knowledge, our work is the first generic hardware-based solution to bypass anti-debugging automatically.

V. Conclusion

In this poster, we proposed a novel approach based on Intel-PT to deal with the anti-debugging challenge. It can bypass anti-debugging by automatically identifying the locations of anti-debugging code and patching the binaries accordingly. We have made a preliminary evaluation to demonstrate the effectiveness of our approach. We will study deeper in this topic and evaluate it on real world software in the future.

References

[1] A. Mylonas and D. Gritzalis, "Book review: Practical malware analysis: The hands-on guide to dissecting malicious software," *Computers & Security*, vol. 31, no. 6, pp. 802–803, 2012.
[2] "Intel processor tracing," https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, accessed: 2013-09-18.
[3] P. Chen, C. Huygens, and L. e. Desmet, *Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware*. Springer International Publishing, 2016.
[4] R. Rubira Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti- vm technologies - white paper," 2012.
[5] H. Cho, J. Lim, H. Kim, and J. H. Yi, "Anti-debugging scheme for protecting mobile apps on android platform," *Journal of Supercomputing*, vol. 72, no. 1, pp. 232–246, 2016.
[6] P. Xie, X. Lu, Y. Wang, J. Su, and M. Li, "An automatic approach to detect anti-debugging in malware analysis," *Communications in Computer & Information Science*, vol. 320, pp. 436–442, 2013.
[7] J. K. Lee, B. J. Kang, and E. G. Im, "Rule-based anti-anti-debugging system," in *Research in Adaptive and Convergent Systems*, 2013.