Efficient and Stealthy Instruction Tracing and Its Applications in Automated Malware Analysis: Open Problems and Challenges

Endre Bangerter, Stefan Bühlmann, Engin Kirda

Efficient and stealthy instruction tracing and its applications in automated malware analysis: Open problems and challenges

Endre Bangerter[1], Stefan Bühlmann[2], and Engin Kirda[3]

[1] Bern University of Applied Sciences, Switzerland
endre.bangerter@jdiv.org
[2] Bern University of Applied Sciences and
Joe Security, Switzerland
stefan.buehlmann@bfh.ch
[3] Northeastern University, USA
ek@ccs.neu.edu

Abstract. Malware is substantial security threat today and most likely in the foreseeable future. The analysis of malware is a key activity in the fight against the threat. Since manual analysis is time consuming and given the extent of the malware threat, malware analysis needs to be automated. Malware analysis sandboxes offer such automation and play already an important role in practice. Yet, they only uncover certain aspects of malware behavior, and still require manual analysis in many cases. This is not a viable way to go, and thus the automation and quality of automated analysis needs to be pushed further. A promising technique towards this goal is instruction tracing combined with analyzes algorithms that uncover malware behavior from an instruction trace.
In this position paper, we shall argue that instruction tracing is still in its infancy and point out challenges and open problems of instruction tracing in general. In particular, we shall describe Helios, which is our new instruction tracer that offers a better balance of tracing speed and transparency than existing techniques.

1 Introduction

Malware is one of the major security issues threatening current networked devices and IT infrastructures. The root causes of the malware problem (i.e., software vulnerabilities, insufficiently secure operating systems, and human failures) are not likely to be solved anytime soon. Moreover, the proliferation of networked devices (e.g., smart phones, tablets, appliances, smart grids etc.) is ongoing. It is thus likely that the malware threat is going to increase further and stay with us for a long time to come. It thus seems that we need to learn to live with malware in the foreseeable future.

Currently, there are two main scenarios of how malware is deployed by miscreants. One are large scale attacks where the goal is to maximize the number

of infections of a more or less arbitrary user base. The other are targeted attacks, where the goal is to compromise a small set of specific users within an organization. The first type of attack is used to commit fraud, steal money from e-banking systems, etc. The latter is used for intelligence gathering and espionage (industrial, and governmental) or selective attacks on the reputation of an institution.

The *analysis of (potential) malware is a key activity in dealing with the threat.* Analysis is required to update protection mechanisms (e.g., AV, IDS signatures, domain blacklists, etc.), to assess the effects and damages of an attack, to initiate recovery measures, etc. Important aspects of malware analysis are *quality* (i.e., to maximize the understanding of a piece of malware) and *speed* (i.e., to obtain the information as quickly as possible). These are clearly competing goals and we are thus in a trade-off situation.

While manual analysis delivers quality (if carried out by a qualified engineer), it lacks speed. Manual analysis does not scale, and given the shortage of qualified experts it is not a feasible way to handle the malware threat. Today, it is common for anti-malware organizations to receive thousands of new, unknown samples every day. As a result, *automated analysis techniques are indispensable tools in the fight against malware.*

Static analysis techniques potentially deliver the best analysis quality, and in particular, good code coverage. Unfortunately, static analysis techniques are easily defeated. In fact, it is often trivial for malware authors to automatically generate different versions of a specific malware component by using techniques such as encryption, code obfuscation, instruction substitution, self modifying code, etc. Thus a purely static analysis of modern malware is currently out of reach [MKK07].

Because of the limitations of static techniques, various dynamic techniques have been proposed that aim to automatically analyze a malware sample by executing it and by logging the behavior that it exhibits. Dynamic analysis is much harder to evade than static analysis, and it thus currently plays an important role in practical malware analysis.

So called malware analysis sandboxes, such as, Anubis [BKK06], Joebox [BK08], and CWSandbox [WHF07], have received considerable interest in the research community as well as by malware analysts in practice. Analysis sandboxes are one of the principal automated analysis technologies being used in practice. They are widely deployed and used by various CERTs, anti-virus companies, and governmental organizations. Technically, sandboxes mainly track the execution flow of a malware by instrumenting and recording API calls (user-space, and system calls), their arguments and related state information. Based on this information, the malware behavior is analyzed. Sandboxes are adequate for identifying the installation behavior of malware, infection strategies (e.g., code injections) and network communications. Some sandboxes track malware in nearly real-time, and there are systems in use that can process tens of thousands of samples per day.

Yet, since the tracking granularity of sandboxes is coarse grained (i.e., only API calls are recorded), sandboxes miss valuable information, which inherently limits their analysis quality. For instance, they are not adequate for detection of code similarity and vulnerabilities, data flow and execution flow - analysis. More generally, they give little information about purely algorithmic aspects of malware, like triggering logics, domain name generation, etc.

In summary, sandboxes excel through high analysis speed, but feature only a moderate analysis quality. They are thus appropriate for first cut and mass analysis. Yet, when it comes to understanding the details of a malware, which is typically required in targeted attacks, then sandboxes are of limited value only. In these cases one still needs to resort to manual analysis. This is a major bottleneck and challenge in current malware analysis. To be able to handle the malware threat, we need to push the boundary of automation further and develop novel techniques that improve the analysis quality.

2 Instruction traces – challenges and open problems

Instruction tracers (tracers, for short) record every single instruction and related state during execution of a piece of code. The resulting traces are then processed by a *trace analyzer* that extracts information on the behavior of the process being analyzed. By providing instruction level granularity, tracers clearly have the potential to offer better analysis quality than sandboxes. The major inherent short coming of instruction tracers, is that they only observe one code path at a time and thus lack code coverage. However, in practice, this is often not an issue since malware does show relevant behavior shortly after execution. It is thus not astonishing that instruction tracers and their applications have received considerable interest in the malware research community recently. Various tracers specifically geared towards malware analysis have been proposed. These are Ether [DRSL08], Cobra [VY06], TEMU [SBY+08], Pin [SDC+10] and MmmBop [Ban09]. Additionally, there are also hardware based tracers, such as ICE debuggers. Examples of applications of tracers are: Identification of keyloggers (by taint tracking keyboard inputs to a network interface) [EKK+07], detection of exploits [NS05] and self-modifying code [DRSL08], and automated protocol reverse engineering [LJXZ08,KKCW08,CYLS07].

Despite these promising results, tracers are still in their infancy and play little role in practice. In fact, there are substantial challenges that remain to be resolved:

- Current tracers are not ready for practical use, since they are either too inefficient or easy to detect and evade. Also, there are no viable solutions to improve the code coverage of tracers.
- The degree of automated analysis needs to be pushed further. To this end more advanced trace analyzers need to be explored, which can automatically uncover algorithmic aspects of malware, and "not just" interactions of malware with its run-time environment.

The first of the above challenges is more an engineering type of challenge, while the latter is a probably hard research challenge. In the following we describe these challenges, the underlying state of the art, and potential solutions in more detail (§3 discusses tracing technologies and §4 trace analysis).

3 Tracing

We believe that a practically viable tracer for malware analysis should be *efficient* (to facilitate live and ideally large scale analysis), *transparent* (to avoid detection and evasion by malware), and *complete* (i.e., record the instructions being traced, but also additional state information like register and memory values).

None of the current tracers achieves these goals simultaneously. In fact we have roughly the following situation: Emulation based tracers are detectable and rather slow. Those using dynamic binary instrumentation are fast but easily detectable. In fact, system emulation and dynamic binary instrumentation, are both easily detectable by malware [MPRB09]. Techniques using trap flags are transparent but rather slow. Finally, hardware based techniques, like branch tracing featured by certain Intel CPUs, are very promising, since they are fast and transparent. Yet, these techniques are not yet supported by virtual machines, which play an important role in malware analysis. Also, they do not feature completeness, since tracing is performed at basic block granularity. The completeness of such tracers operating at basic block granularity could be improved by reconstructing the state within the basic block in some cases. To our knowledge, this question has not yet been considered.

Let us briefly consider existing technologies. Ether[DRSL08] is highly transparent due to the fact that it is running in the hypervisor, it is quite slow since tracing is realized using single stepping. PIN[LCM+05] and the tracer proposed by Bania [Ban09]is fast, but both are detectable; PIN seems to have issues with self modifying code. On the other hand, TEMU [SBY+08,YS10] is very powerful but rather slow and detectable. Given these limitation, it is thus not astonishing that currently tracers play a little role in practical malware analysis.

3.1 Helios - novel techniques

To remedy the situation, we have developed a novel tracer termed *Helios*, which reaches the desirable properties mentioned above to a better overall level than existing ones. Helios essentially is a collection of optimization techniques designed to speed up malware tracing, e.g., by automatically skipping computationally costly, but non-relevant code parts like unpacking loops. While Helios improves over the state of the art, many challenges, like further improving performance and code coverage, remain.

The basic techniques underlying Helios are relatively straightforward: We interrupt the execution flow whenever a control transfer instruction (CTI) occurs, record the instructions between two such CTIs, and then continue to follow the execution from the destination address of the CTI. We implement this approach

by using solely hardware breakpoints and the trap flag. Helios is currently implemented as kernel driver for the Windows OS, but it is likely to be portable to other operating systems running on Intel CPUs.

While the approach is simple on a high-level, there are series of challenges that need to be resolved. In fact, Helios combines novel and existing techniques to optimize performance and to achieve transparency. For instance, Helios is using a heuristic, that temporarily disables tracing in small code parts that make excessive use of loops, which are often found in packers; this technique alone yields a speed up of a factor of 40 when analyzing packed code. We also use techniques to be able to handle self-modifying code, which is quite common in malware. Finally, for achieving transparency, we use techniques to virtualize hardware debug registers.

Different experiments confirms the viability of Helios in practice. In fact, we were able to successfully trace code that was packed with a large number of commonly used packers used by malware. Since packer code typically contains detection and evasion techniques, these results hint that Helios is indeed reaches a high degree of transparency. In summary, Helios reaches transparency and efficiency in a more balanced way than existing tracers, and is thus a further step towards bringing tracing to practice. In the following we give an overview of the Helios design.

Helios design overview To control the execution of a program our tracing algorithm uses hardware breakpoints and the trap flag (both debugging facilities of the CPU). Hardware breakpoints are controlled by the debug register on the x86 architecture. Four debug registers (Dr0-Dr3) and one control register (Dr7) exist. The four debug registers contain the breakpoint memory address and the control register defines the functionally of the breakpoints. Hardware breakpoints can be used for many different applications. Their main usage is detecting the execution at a memory address.

Figure 1 shows how Windows handles hardware breakpoints. If execution reaches a previously set breakpoint address (14B5A3) a debug interrupt (int1) is thrown by the CPU. If an interrupt is thrown, the CPU calls the appropriate interrupt service routine (ISR, TrapHandler) in kernelmode. It finds the correct trap handler by using the interrupt descriptor table (IDT). The Windows trap handler for interrupt 1 then calls KiDispatchException which is used to launch the exception handler in usermode (KiUserExceptionDispatcher) of the application which has caused the interrupt (program.exe).

The trap flag works in a similar way: if it is set the processor will execute only one instruction at time and then throw a debug interrupt.

Helios is implemented as a Windows kernel mode driver which intercepts the int1 trap handler and the thread creation system call. Figure 2 shows how the trap flag and hardware breakpoints are used to trace (control the execution flow) an application.

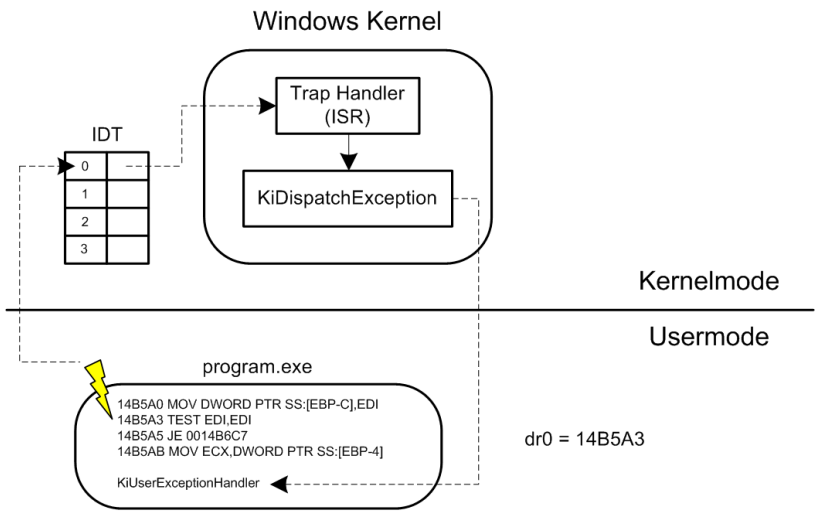

Fig. 1. Windows exception handling

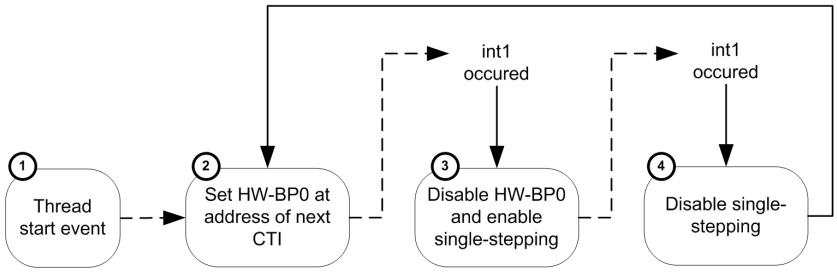

Fig. 2. Basic block tracing by using hardware breakpoints and the trap flag

The tracing algorithm is now explained in detail. If a new thread is detected by Helios ((1) in Figure 2) it will scan (disassemble) the binary code from the beginning of the thread start address until it finds a control transfer instruction (CTI) [Int10]. Then it finishes its first task by setting a hardware breakpoint (Dr0) at the address of the CTI (2). Next the applications thread begins its execution and reaches the address of the CTI where an interrupt is thrown (due to the hardware breakpoint). Helios will catch and handle this interrupt (3). Inside the interrupt handling routine of Helios the hardware breakpoint (Dr0) is cleared and the trap flag (single-stepping) is set. The interrupt handling routine finishes and finally calls the instruction IRETD. The IRETD instruction returns program control from the ISR to the program that was interrupted. Thus the Windows trap handler, KiDispatchException and KiUserException are not called, instead the application thread continues its execution where it has been interrupted before. The CTI is executed and right after the execution a next

interrupt (due to the trap flag) is thrown. Again Helios catches and handles it (4). The trap flag is cleared and the next CTI is searched from the beginning of the current instruction pointer, which points to the address of the CTI target (where execution has been transferred to). Finally the algorithm jumps to its start and continues as described (1).

The algorithm described interrupts a program in a basic block granularity. A basic block is a sequence of instructions with a CTI at the end. Within the basic block, except at the end, no CTI is located. The start of a basic block begins with a target address of a CTI. Thus a basic block has exactly one entry and one exit point. Basic block tracing is much faster than single step tracing because the program is less frequently interrupted and thus the overhead introduced with the interruption and resumption process is smaller. However basic block tracing is not as precise as single step tracing. The CPU state can only be extracted when the program is interrupted and not for every instruction. However the granularity of basic block tracing is good enough for the application of instruction scanning, which is needed to find instructions which read sensitive memory locations.

The algorithm proposed above is portable because it uses hardware breakpoints and the trap flag which are available on all CPUs based on the x86 architecture. Furthermore, it is minimally invasive, meaning that beside debug registers no other register or memory contents are changed in the context of the application being traced. In addition our tests have shown that both mechanisms work inside virtual machines based on VMware and VirtualBox. These systems are mainly used in large scale malware analysis systems and thus it is necessary that the tracer works on them.

4 Advancing trace analysis

Current malware reverse engineering is performed essentially manually. That is, often a pseudo-code or C representation of relevant code parts is reconstructed from assembly. This decompilation step can also be automated by using tools such as Hex-Rays [HR]. From the resulting code the reverse engineer then recovers the semantic of the code under consideration. While this approach will certainly remain adequate and valuable for many years to come, we believe that research should focus on recovering code semantics directly and (semi-) automatically rather than "just" decompiling the code.

As mentioned above, so called sandboxes are a first step towards automating malware analysis and they are able to recover many aspects of the interaction of a malware with the operating system. While these are important malware features, there are many properties of malware which cannot yet be recovered automatically, and thus require manual analysis. These are typically purely algorithmic properties, running in user mode. On important example of such properties are domain name generation algorithms, which compute domain names

for rendez-vous points of malware and command control servers [LW09,PSY09]. Another example is the use of cryptographic techniques used by malware, e.g., for authenticating and hiding the communication of malware with a command and control server.

Recovering algorithmic aspects is clearly a challenging goal, and hard in general. However, it might be somewhat easier in the malware domain, because many possible malware behaviors and techniques are known in advance, and can be specifically sought after. Ideally, these techniques would be realized using static analysis. Yet, static analysis is easily defeated by obfuscation, which is thus another formidable challenge to be overcome. Trace based analysis is considerably more resilient to obfuscation and also features state information, which makes analysis much easier. In fact, there has been some interesting initial work on automated analysis using traces.

One direction of research is concerned with recovering protocol specifications from code [Lut08,ZWCW08,CYLS07,CPSK09]. Other research has considered the identification of crypto code in malware using traces [GWH11]. These tools are for instance able to identify reliably certain crypto primitives like AES and RSA. Yet, they cannot handle the composition and higher-level use of cryptography. Recovering crypto code from traces is an example of the case mentioned above, i.e., of recovering a priori known constructs from code. Finally, VERA is an interesting piece of work on trace visualization [QL09,QLN09]. The goal of VERA is to visualize malware behavior, allowing the reverse engineer to identify malware features without resorting to code analysis.

Yet, we believe that the potential of traced based analysis is currently far form being fully utilized, and that there is substantial potential to push automation further. Additionally, there are further little explored related fields like the combination of dynamic analysis with static techniques.

References

[Ban09] Piotr Bania. Generic unpacking of self-modifying, aggressive, packed binary programs. 2009.

[BK08] Stefan Buehlmann and Martin Kropp. Extending joebox - a scriptable malware analysis system. In *University of Applied Science Northwestern of Switzerland, Bachelor Thesis*, 2008.

[BKK06] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Ttanalyze: A tool for analyzing malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006)*, 2006.

[CPSK09] Juan Caballero, Pongsin Poosankam, Dawn Song, and Christian Kreibich. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *In CCS09: of the 16th ACM conference on Computer and communications security*, pages 621–634. ACM, 2009.

[CYLS07] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, 2007.

[DRSL08] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, 2008.

[EKK+07] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, 2007.

[GWH11] Felix Groebert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *The 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[HR] Hex-Rays. Hex-rays decompiler. http://www.hex-rays.com/decompiler.shtml.

[Int10] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 1: Basic architecture. pages 142 – 143, Chapter 5, 5.1.7, 2010.

[KKCW08] Christopher Kruegel, Engin Kirda, Paolo Milani Comparetti, and Gilbert Wondracek. Automatic network protocol analysis. In *15th Annual Network and Distributed System Security Symposium (NDSS 2008)*, 2008.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1065010.1065034.

[LJXZ08] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through conectect-aware monitored execution. In *In 15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[Lut08] N. Lutz. Towards revealing attackers intent by automatically decrypting network traffic. In *Masters thesis, ETH Zuerich*, 2008.

[LW09] F. Leder and T. Werner. Know your enemy: Containing conficker - to tame a malware. In *Know Your Enemy Series of the Honeynet Project*, 2009.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[MPRB09] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *ISSTA*, 2009.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[PSY09] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. A foray into conficker's logic and rendezvous points. In *LEET'09 Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats*, 2009.

[QL09] D.A. Quist and L.M. Liebrock. Visualizing compiled executables for malware analysis. In *6th International Workshop on Visualization for Cyber Security (VizSec 2009)*, 2009.

[QLN09] Daniel Quist, Lorie Liebrock, and Joshua Neil. Visualizing compiled executables for malware analysis. In *Journal in Computer Virology*, 2009.

[SBY+08] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin

Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.

[SDC+10] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert S. Cohn, Kim M. Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *ISPASS*, 2010.

[VY06] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *IEEE Symposium on Security and Privacy*, 2006.

[WHF07] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. 2007.

[YS10] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. 2010.

[ZWCW08] X. Jiang Z. Wang, W. Cui, and X. Wang. Reformat: Automatic reverse engineering of encrypted messages. In *Technical report, NC State University*, 2008.