Hardware-Assisted Malware Analysis

Marcus Felipe Botacin[1], **Paulo Lício de Geus**[1] , **André Grégio**[2]

[1] UNICAMP

[2]UFPR – Co-Advisor

{marcus,paulo}@lasca.ic.unicamp.br

gregio@inf.ufpr.br

Abstract. *Malicious software (malware) are persistent threats to modern computer systems and the development of countermeasures to them becomes harder each day due to the emergence of anti-analysis and anti-forensics techniques, able to evade software-based monitoring solutions. In this scenario, hardware-assisted solutions are effective alternatives, but still present development gaps. The presented dissertation surveyed the limits of software-based solutions, pinpointed the existing development gaps on hardware-assisted solutions and introduced a lightweight, hardware-based alternative for malware analysis. The developed framework was released as open-source and is being used on further research developments.*

1. The Malware Problem

Malicious software (malware) are persistent threats to modern computer systems, being responsible for a myriad of attacks, from information leaks to financial losses. To handle malware attacks, analysis procedures are applied, thus enabling vaccine development, incident response and forensic procedures.

Analysis procedures are classified into static and dynamic [Sikorski and Honig 2012]. Static analysis procedures rely on binary inspection without its execution, thus being defeated by obfuscation techniques. Dynamic analysis approaches defeat obfuscation by running the suspicious binary on an isolated, controlled environment named sandbox, thus requiring the development of runtime monitoring solutions.

Monitoring tools can be implemented as pure-software solutions or hardware-based ones. Whereas software-based monitoring solutions are easier to implement—thus, so-far more popular—, these are ineffective against modern malware, which employ anti-analysis techniques to remain stealth. As we identified in a survey [Botacin et al. 2017a], many anti-analysis techniques succeed on software-based environments given the identification of internal monitoring components [Marpaung et al. 2012] and/or because the execution side-effects caused by instruction emulation and instrumentation [Shi et al. 2014]. In a summary, anti-analysis tricks can be classified in three main categories, as presented in the Table 1.

On a 4-year-long, Brazilian malware study [Botacin et al. 2015], we identified the use of anti-debug tricks have grown from 25% in 2012 to 40% in 2015. Similarly, anti-VM tricks have grown from 3% to 8% in the same period. In total, more than 50% of samples

Table 1. Anti-Analysis: Tricks summary. Malware samples may employ multiple techniques to evade distinct analysis procedures.

Technique	Description	Reason	Implementation
Anti Debug	Check if running inside a debugger	Blocks reverse engineering attempts	Fingerprinting
Anti VM	Check if running inside a VM	Analysts use VMs for scalability	Execution Side-effect
Anti Disassembly	Fool disassemblers to generate wrong opcodes	AV signatures may be based on opcodes	Undecidable Constructions

are armored with at least one anti-analysis trick. Given this scenario and its evolution, more transparent monitoring solutions are required to handle current and future threats. A way of achieving more transparent monitoring is to rely in hardware assistance instead of software components.

In the dissertation, I surveyed existing hardware features which may be used for system and binary analysis and identified their strong and weak points as well as existing development gaps. I also contributed to advance the current scenario by developing an analysis framework able to fill the existing gap regarding branch-monitor-based solutions. These developments are presented in the following sections.

2. Hardware-Assisted Solutions

To draw a panorama, I surveyed existing and upcoming hardware features which could be applied for security purposes. The survey was focused on identifying the pros and cons of each technology and the existing development gaps. The results are summarized in Table 2.

Table 2. Hardware features. Distinct approaches present advantages (PROS) and disadvantages (CONS), requiring analysts to identify the best usage scenarios. For some cases, there are existing development gaps.

Technique	PROS	CONS	Gaps
HVM	Ring -1	Hypervisor development	High overhead
SMM	Ring -2	BIOS development	High implementation cost
AMT	Ring -3	Chipset code change	No malware analysis solution
HPCs	Lightweight	Context-limited information	No malware analysis solution
GPU	Easy to program	No register data	No introspection procedures
TSX	Commit-based	Store only few KB	Overcome the KB barrier
SGX	Isolates goodware	Also isolates malware	No enclave inspection
SOCs	Tamper-proof	Passive components	Raise alarms

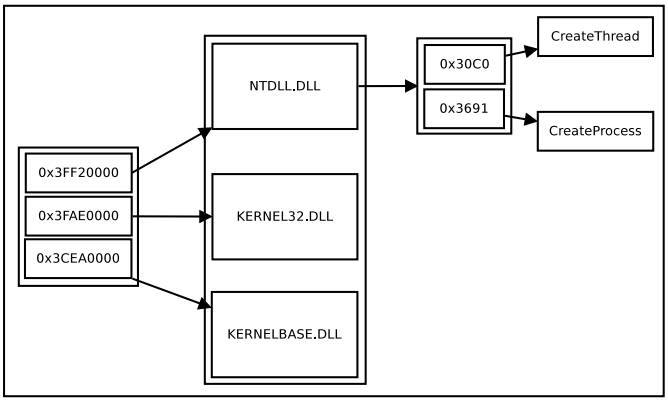

Figure 1. Introspection. The developed procedure allows bridging the semantic gap by converting raw branch addresses into high level function information.

The survey pinpointed many technologies as effective alternatives for security solutions implementations, such as BIOS rewriting and Hypervisor instrumentation. These solutions, however, are very expensive in many aspects—noticeably, in the required development cost to implement a hypervisor or to rewrite the BIOS. In addition, I discovered branch monitors as interesting features, since they are lightweight alternatives both regarding the imposed performance impact as well as the required development cost. I observed branch monitors had not been used in the context of malware analysis sandboxes before. Therefore, the research was guided to present the first branch-based framework able to perform malware analysis, debugging and Return-Oriented-Programming (ROP) attack detection, thus contributing to advance the current scenario.

3. The Branch-Monitoring framework

The Branch Trace Store (BTS) is a processor feature which logs source and target addresses of branch instructions on O.S. pages, raising an interrupt when the collection buffer is full [Intel 2011]. It operates on a system-wide way, having no process filtering capabilities. As a hardware feature, small performance overheads are imposed. To develop a BTS-based monitoring solution, some challenges have to be overcome, namely: i) How to isolate processes despite having no hardware support?; ii) How to interpret low-level branch addresses as human-meaningful information?; iii) How to reconstruct a given program complete execution flow from the collected branch data?

The first challenge derives from the fact that the processor is not aware of the process concept (an O.S. abstraction), thus a way of associating processor execution with O.S. scheduling was required. This issue was approached by setting the BTS interrupt threshold to one, thus triggering an interrupt at every executed branch instruction, and so having the system to operate on a debugger-like, **step-by-step** way. As the execution is immediately interrupted, one can immediately query the O.S. about the last scheduled process, thus allowing **individual process tracking**.

The second challenge derives from the distinct representations data have on distinct system layers, which is known as the **semantic gap** problem. To bridge such semantic gap, an **introspection procedure** was developed, allowing the association of instruction addresses to the loaded modules and their function names, as shown in Figure 1.

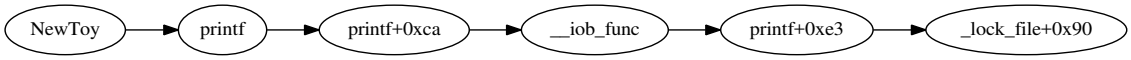

Figure 2. `Printf`'s step-into call graph. All intermediate calls represented.

The developed introspection procedure allows digging into programs' internals to understand their behavior. For instance, Figure 2 shows an excerpt of a `printf` call. We notice it internally calls `_lock_file`, which assures I/O ordering, since `printf` is a non-reentrant function.

The third challenge derives from the branch-based collection, which skips non-branch instructions. However, as a block of non-branch instructions surrounded by branch instructions is a lax **basic block** definition, we demonstrated we could retrieve all executed instructions by disassembling the code portion between two consecutive branches—i.e. from target of a previous branch until the source of the current one. Listing 1 exemplifies the block identification procedure.

Listing 1. Block identification—from `0x48ff5ab8` to `0x48ff5ac0`

```
1 PID: 4876 FROM: 0x48ff5ab0 TO: 0x48ff5ab8
2 PID: 4876 FROM: 0x48ff5ac0 TO: 0x48ff5ad0
3 Disassembly from: 0x48ff5ab8 to 0x48ff5ac0
```

By repeating the procedure for all branches, one is able to rebuild the whole Control Flow Graph (CFG) for any running binary, as shown in Figure 3.

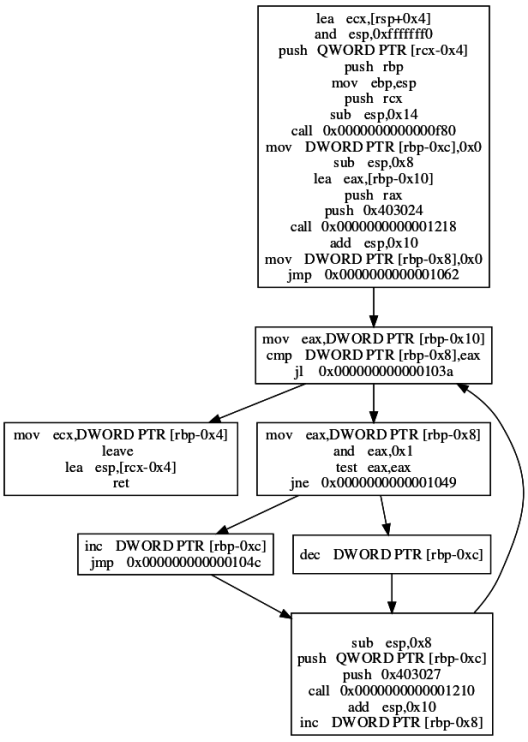

Figure 3. Control Flow Graph. By repeatedly querying consecutive branches, it is possible to reconstruct the whole execution flow.

The aforementioned developments were compiled in a modular monitoring framework which allows security policies to be built on top of it. During the master period, three

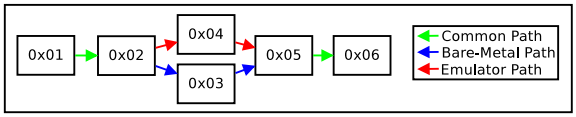

Figure 4. Divergent Behavior Identification. By aligning branches, distinct execution paths can be compared and evasion attempts identified.

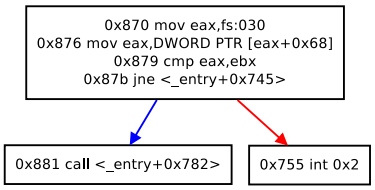

Figure 5. Identified evasion-based divergence case. Distinct paths are taken on an emulator and on a real machine.

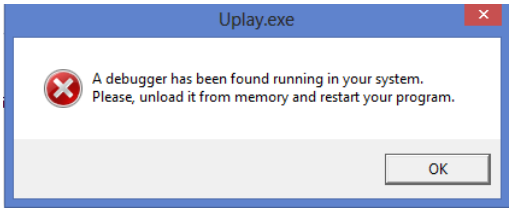

Figure 6. Ordinary Debugger. The monitored application refused to run.

solutions were developed: i) a malware analysis tool; ii) a debugger; iii) a ROP detector. They are below detailed.

Analyzing malware using branch monitors has many advantages: i) no side effects are observed, as the code runs on a native processor; ii) no code injection is required, as the processor stores taken branches natively; iii) no significant overhead is imposed, as BTS is a hardware feature.

I explored such features to transparently analyze anti-analysis-armored samples. A major analysis capability is to identify evasion attempts. More specifically, one can leverage the solution to identify execution deviation between the transparent monitor and software-based solutions. By applying a sequence alignment algorithm to the obtained traces, as shown in Figure 4, I was able to identify true divergent behaviors, such as the one shown in Figure 5.

In addition to trace-oriented malware analysis, I developed a complete debugger, able to perform **real time inspection**. To implement this capability, an **inverted I/O** mechanism was implemented from within the interrupt handler, thus allowing the framework to call the debugger client **on-demand**.

The main debugger capability is to inspect **even protected applications**. To demonstrate this, I inspected some game launcher software, applications which are well-know for applying anti-cheat mechanisms. As an example, the Ubisoft launcher refused to run under an ordinary debugger (Fig. 6) but was successfully analyzed by the developed solution (Fig. 7).

To reduce the analyst learning curve, the debugger is **integrated to GDB**, thus allowing the use of ordinary GDB commands to control debugger steps (Fig. 8).

Finally, we also leveraged our framework to implement a ROP attack detector. The Return-Oriented-Programming is a technique which chains `RET` instructions to change a program control flow and thus perform a code injection attack without the restrictions

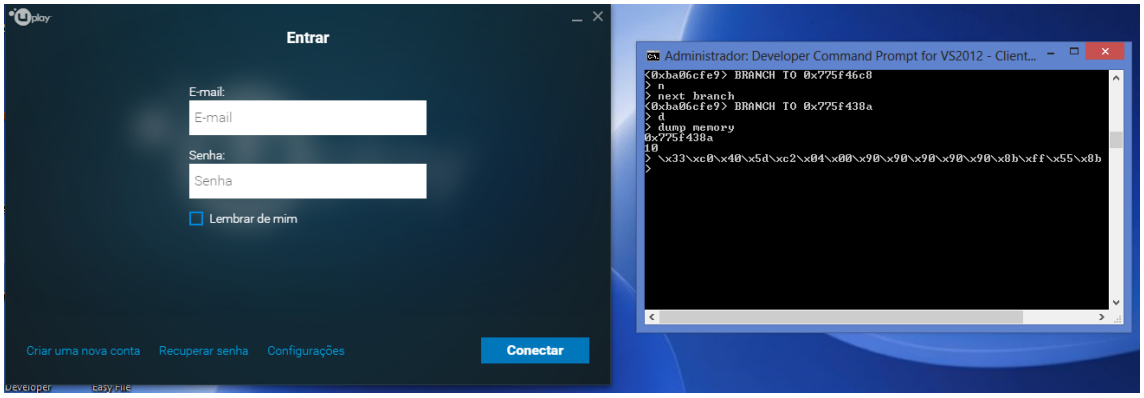

Figure 7. Developed Debugger. The monitored application executed and was inspected.

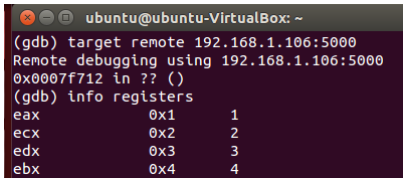

Figure 8. GDB integration. The remote stub allows a Linux client to monitor a Windows instance.

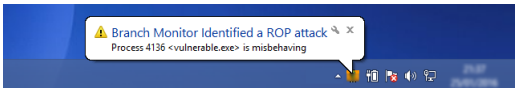

Figure 9. Daemon Alert. A warning is raised by the solution when a ROP attack is detected.

of current buffer overflow protections. ROP attacks against applications can be remotely launched through the internet or an application can be locally exploited by a malware sample. Our solution is well suited for the detection task since, as the attack is based on a branch (the `RET` instruction), these are naturally logged by the BTS mechanism. In addition, a significant advantage of our solutions is that, as the taken branches are extracted in hardware, the solution is not vulnerable to alignment issues (Listing 2 and 3), a frequent code construction employed by the attackers. Our solution can perform **real-time** monitoring and operates as a daemon (Figure 9).

Listing 2. Static disassembly of the `MSVCR71.dll` library. This code is executed on non-corrupted flows.

```
1 c08: f2 0f 58 c3 addsd %xmm3,%xmm0
2 c0c: 66 0f 13 44 24 04 movlpd %xmm0,0x4(%esp)
```

Listing. 3. ROP Payload. The gadget is build based on a misaligned instruction block.

```
1 c0a: 58 pop rax
2 c0b: c3 ret
```

4. Current Results and Future Work

During the Master course, project results were published in many ways, as presented below:

A survey on malware anti-analysis was published in the XVII SBSEG, in the form of the article "*Analysis, Anti-Analysis, Anti-Anti-Analysis: An Overview of the Evasive Malware Scenario*" [Botacin et al. 2017a].

A survey on hardware-assisted security solutions, titled "*Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools and methods for systems and binary analysis on modern platforms*", has preliminary acceptance for publication on the ACM Computing Surveys [Botacin et al. 2018b] (**A1**).

Preliminary framework results were published in the XVI SBSEG, in the form of the following articles: VoiDbg: Projeto e Implementação de um Debugger Transparente para Inspeção de Aplicações Protegidas [Botacin et al. 2016c]; Detecção de ataques por ROP em tempo real assistida por hardware [Botacin et al. 2016b]; Análise Transparente de Malware com Suporte por Hardware [Botacin et al. 2016a].

The final framework version was published in the ACM Transactions on Privacy and Security journal (**A2**), under the title "*Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging*" [Botacin et al. 2018a].

To enable reproducibility and allow community participation, the framework's source code was opened and is available on Github[1], being—at this moment—starred by more than 65 people and forked 13 times. Branch data is also available in the project website[2]. In addition, media is available on Youtube[3] and community has created a Reddit topic about the project[4].

In addition to the previously mentioned papers, some other, indirectly-related papers were published during the master course, as presented below:

A conventional, software-based sandbox solution was developed and used as ground-truth for malware evasion, being published in the Journal of Computer Virology and Hacking techniques (**B1**), under the title "The other guys: automated analysis of marginalized malware" [Botacin et al. 2017b].

All referred statistics about malware impact and anti-analysis trends derives from a 4-year-long study published in the XV SBSEG, under the title "Uma Visão Geral do Malware Ativo no Espaço Nacional da Internet entre 2012 e 2015" [Botacin et al. 2015].

As future work, my current PhD research will directly benefit from the developed framework as basis for further developments. More specifically, the following developments regarding branch monitors are research in progress: i) a branch-monitor extension to handle multi-core malware; ii) a branch-monitor extension to handle kernel-based malware; iii) a branch-trace-based solution for application misbehavior detection; iv) a hardware-assisted anti-virus engine based on hardware counters; v) a polymorphic malware classification engine based on transparent traces.

---

[1] `https://github.com/marcusbotacin/BranchMonitoringProject`

[2] `https://sites.google.com/site/branchmonitoringproject/`

[3] `https://www.youtube.com/watch?v=BguVzqMt_j0&list=PLVYZ2jULLUDvqFVpU3pCZGlY9gCzYoyXP`

[4] `https://www.reddit.com/r/ReverseEngineering/comments/5ycg08/code_tracing_framework_based_on_intel_branch/`

References

Botacin, Falcão, Geus, and Grégio (2017a). Analysis, anti-analysis, anti-anti-analysis: An overview of the evasive malware scenario. https://sbseg2017.redes.unb.br/wp-content/uploads/2017/04/20171109_ANAIS_SBSEG_2017_FINAL_E-BOOK.pdf.

Botacin, Geus, and Grégio (2015). Uma visão geral do malware ativo no espaço nacional da internet entre 2012 e 2015. http://siaiap34.univali.br/sbseg2015/anais/WFC/artigoWFC02.pdf.

Botacin, Geus, and Grégio (2016a). Análise transparente de malware com suporte por hardware. http://sbseg2016.ic.uff.br/pt/files/anais/completos/ST8-3.pdf.

Botacin, Geus, and Grégio (2016b). Detecção de ataques por rop em tempo real assistida por hardware. http://sbseg2016.ic.uff.br/pt/files/anais/completos/ST6-4.pdf.

Botacin, Geus, and Grégio (2016c). Voidbg: Projeto e implementação de um debugger transparente para inspeção de aplicações protegidas. http://sbseg2016.ic.uff.br/pt/files/anais/completos/ST6-1.pdf.

Botacin, M., Geus, P. L. D., and Grégio, A. (2018a). Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Trans. Priv. Secur.*, 21(1):4:1–4:30.

Botacin, M., Geus, P. L. D., and Grégio, A. (2018b). Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools and methods for systems and binary analysis on modern platforms. To be published.

Botacin, M. F., de Geus, P. L., and Grégio, A. R. A. (2017b). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*.

Intel (2011). Intel 64 and ia-32 architectures software developer's manual. http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf. Access Date: July/2016.

Marpaung, J., Sain, M., and Lee, H.-J. (2012). Survey on malware evasion techniques: State of the art and challenges. In *IEEE ICACT, 14th Intl. Conf. Advanced Comm. Technology*, pages 744–749.

Shi, H., Alwabel, A., and Mirkovic, J. (2014). Cardinal pill testing of system virtual machines. In *23rd USENIX Security Symp. (USENIX Security 14)*, pages 271–285, San Diego, CA. USENIX Association.

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.