Down to the Bare Metal:
Using Processor Features for Binary Analysis

Ralf Hund, Carsten Willems, Dennis Felsch, Andreas Fobian, Thorsten Holz

Abstract

A detailed understanding of the behavior of exploits and malicious software is necessary to obtain a comprehensive overview of vulnerabilities in operating systems or client applications, and to develop protection techniques and tools. To this end, a lot of research has been done in the last few years on binary analysis techniques to efficiently and precisely analyze code. Most of the common analysis frameworks are based on software emulators since such tools offer a fine-grained control over the execution of a given program. Naturally, this leads to an arms race where the attackers are constantly searching for new methods and techniques to detect such analysis frameworks in order to successfully evade analysis.

In this paper, we focus on two aspects. As a first contribution, we introduce several novel mechanisms by which an attacker can delude an emulator. In contrast to existing detection approaches that perform a dedicated test on the environment and combine the test with an explicit conditional branch, our detection mechanisms introduce code sequences that have an implicitly different behavior on a native machine when compared to an emulator. Such differences in behavior are caused by the side-effects of the particular operations and imperfections in the emulation process that cannot be mitigated easily. Even powerful analysis techniques such as multi-path execution cannot analyze our detection mechanisms since the emulator itself is deluded. Motivated by these findings, we introduce a novel approach to generate execution traces. We propose to utilize the processor itself to generate such traces. More precisely, we propose to use a hardware feature called branch tracing available on commodity x86 processors in which the log of all branches taken during code execution is generated directly by the processor. Effectively, the logging is thus performed at the lowest level possible. We present implementation details for both Intel and AMD x86 CPUs and evaluate the practical viability and effectiveness of this approach.

1 Introduction

During a typical attack against a computer system, an attacker first exploits some kind of (software) vulnerability to gain access to the system. Once she has control over the compromised machine, the next step is to install some kind of malicious software (abbr. malware) that, for example, steals sensitive information or hides the presence of the attacker on the system. Both software vulnerabilities and malware are thus closely linked and we need to have a precise understanding of their semantics to combat this threat. Most importantly, detailed analysis reports about the tools used by an attacker are required to fix vulnerabilities in operating systems or applications, and to develop new protection techniques and tools.

Nowadays, antivirus companies analyze tens of thousands of malware samples on a daily basis [45] with new exploits being released frequently. Thus, there is a clear need for automated approaches to analyze these threats. As a result, a lot of research has been done on efficiently and precisely analyzing malicious code both in academia and industry (e.g., [2,3,7,8,11,12,23,30,44,48]) and many tools and techniques for automated analysis are currently available. The analysis of malicious and vulnerable code can be implemented in several ways and on several semantic levels. Broadly speaking, the methods can be divided into static and dynamic approaches, each having their own (dis-) advantages. For example, static analysis is often complicated with code obfusca-
tion and encryption [25, 35, 43], whereas dynamic analysis is typically only capable of efficiently examining a limited number of execution paths [30].

A very detailed behavioral view of code can be obtained by examining every single instruction, but this approach produces a huge amount of data, which has to be mined for valuable information. On the contrary, monitoring only the system calls performed by the program achieves a smaller analysis data set, but results in a high abstraction level and can be evaded in many different ways [14], leading to incomplete analysis results. From a performance perspective, single stepping a program to perform an instruction-level analysis is much slower than intercepting only system calls.

Furthermore, malware analysis can be implemented on a native machine (also called *bare metal approach*) or in an emulated/virtualized one. When using a native machine, we are faced with several problems. Most importantly, the analysis system may get infected by malicious code and it has to be reverted back to a clean state after the analysis process has finished. Furthermore, most machines only offer rudimentary monitoring facilities and additional mechanisms have to be implemented first. Sophisticated approaches use techniques like dynamic translation (e.g., *Cobra* [48]) or hardware virtualization extensions (e.g., *Ether* [11]) to achieve such monitoring.

In contrast, emulators pose a powerful trade-off between performance and convenience with respect to native machines, but they lack transparency and correctness. Many malware authors have come up with a variety of detection mechanisms that uncover the presence of such artificial environments [14, 15, 34, 36, 39] and several systematic studies on detecting virtual machines or CPU/system emulators have been performed [28, 29, 36]. Once the malware has detected the presence of the analysis environment, it can behave differently leading to incorrect analysis reports. Apart from these explicit detection techniques employed by malware, there are also different CPU instructions [39] and real-life conditions (e.g., timing aspects or specific artifacts like the username of the analysis machine) under which a binary might behave differently when executed inside an emulated environment as opposed to a native system.

In this paper, we continue this line of work and present mechanisms an attacker can use to implement code that behaves differently in the presence of an analysis environment. Our mechanisms are novel as they do not perform any explicit test on the analysis environment. Instead, we use instruction sequences that have different semantics on a real machine when compared to an emulated one. More precisely, such instruction sequences have an implicitly different behavior on a native machine with respect to an emulator due to the side-effects of particular operations and imperfections in the emulation process that cannot be mitigated easily (e.g., self-modifying code or caching effects). Effectively, our techniques delude the emulator and thus we call this approach a *delusion attack*.

We use delusion attacks as our motivation and propose to utilize hardware features of commodity x86 processors to overcome the (accidental or intended) incorrectness of dynamic analysis in an emulated environment. More precisely, we introduce a promising approach to analyze the behavior of binary programs by using a processor feature called *Branch Tracing (BT)*. With this hardware primitive (available on both Intel and AMD CPUs [18]), the processor *itself* keeps track of all branches taken during code execution. The logging is thus performed at the lowest level possible, making our approach robust to attacks. Our performance overhead is also significantly lower in contrast to other approaches that use hardware features such as single stepping [11].

To demonstrate the effectiveness and applicability of our approach, we show how our method is sufficient to analyze malicious PDF documents. In an empirical evaluation, we demonstrate that the branch tracing results can be used to automatically cluster similar vulnerabilities which are exploited within the analyzed documents: a set of 4,869 PDF documents can be clustered into eight different root causes based on the analysis results of our tool. Most notably, our framework can also deal with advanced exploits that use concepts like structured exception handler (SEH) for control flow diversion and even return-oriented programming [41].

**Related Work** As discussed previously, there is a large body of published work on malware analysis and detection of different execution environments. Complementary to our work are recent
approaches that compare the behavior of a sample in different (analysis) environments [2, 21, 22]. Such techniques could also be used to detect our delusion attacks, but they incur huge runtime overheads as a single sample has to be executed in at least two analysis environments. Vasudevan et al. introduce a way to use branch tracing on AMD CPUs to record host execution trace to an external, trusted system [47]. In contrast, we also show how BT can be used on Intel CPUs and perform several empirical experiments to demonstrate the practical usefulness of this approach.

Contributions The main contributions of this technical report can be summarized as follows:

• We introduce several delusion attacks for software emulators. These instruction sequences behave differently when executed on a native machine as opposed to an emulator. Delusion attacks work by exploiting some implicit imperfections in the emulation process.

• Motivated by delusion attacks, we introduce an approach to perform behavior analysis that takes advantage of the branch tracing feature of commodity x86 CPUs. Our approach performs the logging of the actual behavior on the lowest level possible since we directly instrument the CPU to generate traces. We illustrate this analysis process and provide technical implementation details for both Intel and AMD CPU architectures.

• We have implemented a fully-working prototype of our approach and show in an empirical evaluation the usefulness of our approach by performing a crash analysis of malicious PDF documents.

2 Background

In this section, we review existing techniques that are used by malware analysis frameworks. We shed light on the advantages and disadvantages of each approach and provide some examples of tools and methods that have been proposed in the literature. This serves as a discussion of related work and motivates our delusion attacks that we introduce in Section 3.

2.1 Debuggers

Every modern operating system offers the possibility to debug a specific process from another process. For example, Windows provides the so called Windows Debugging API and Linux provides the `ptrace()` system call. These debugging facilities can be used to set breakpoints at specific code locations or certain memory read/write accesses. Furthermore, it is possible to set a debuggee into single-stepping mode and thus perform instruction-level tracing. All this functionality is implemented with the help of certain hardware CPU features such as the trap flag or the int 3 breakpoint instruction. This enables the analysis of a program on a very fine-grained level.

However, debugger-based solutions are slow and easy to detect, since the analyzer interacts heavily with the process environment, its memory layout, and the instruction scheduling as well. Furthermore, the scope of such analyzers is typically limited to usermode binaries only. While there are also kernel-debugging interfaces, they are usually even slower and harder to automate. Another drawback is the limited amount of hardware debugging resources, e.g., the number of hardware breakpoints. In contrast, software breakpoints require overwriting certain instructions in memory, which is easy to detect and may cause problems in the presence of self-modifying code. An example of a debugger-based analysis framework is Ether, which uses single-stepping and hardware virtualization extensions to implement instruction level tracing.

2.2 Binary Instrumentation

Another class of malware analysis frameworks uses binary instrumentation to add monitoring capabilities inline into the observed program. To that end, the analyzed code is patched, i.e., other instructions are inserted before or after existing code, or code is redirected and executed in
an entirely new memory region. Normally, the modified code itself takes no notion of the presence of instrumentation and executes identically as in a normal environment. The rewriting can either be done statically (i.e., the executable file is patched before execution) or dynamically (i.e., the code is patched in memory before/while executing). In addition, instrumentation can also be achieved by overwriting only certain function pointers. For example, this is common practice in several function call tracing frameworks to hook library calls. Notable examples of binary instrumentation-based frameworks are PIN [26], Valgrind [31], DynamoRIO [6], CWSandbox [50], and Anubis [3].

The advantage of binary instrumentation over debugger-based solution is that it is typically faster since no time-consuming interrupts and process switches between the debugger and the debuggee are needed. This comes with the disadvantage of the non-trivial task of implementing a sound and complete rewriting engine. Binary rewriters also run on equal terms with the emulated code and can thus be detected and attacked easily. Finally, current frameworks only provide a single-process view and cannot be used to analyze the entire system.

2.3 Software Emulators

Software emulation-based solutions are oftentimes more appealing for malware analysis since, in contrast to the previously described techniques, they provide full control over the emulated system: the analyzer can intervene at any point in the execution of the analyzed code. There are also no restrictions on analyzing privileged code within the guest. Furthermore, emulators provide isolation between the analyzer and the malware.

**BOCHS** [24] is a PC emulator that emulates an x86/x64 processor with a set of common attached devices (graphics card, network card, etc.). It is an emulator in the classical sense in that the emulated code is fetched, decoded, and emulated instruction-wise — implemented in one large loop of the **BOCHS** code. This enables a precise emulation of the guest system, but has the drawback that execution is typically slow in comparison to a real system. **BOCHS** is often used for malware analysis in conjunction with the disassembler IDA Pro [40], which ships with built-in support for that purpose. This combination can efficiently be used to (partially) execute malware within the emulator to analyze it.

Similar to Bochs, **QEMU** [4] is a generic full system emulator. However, by using an intermediate language and a technique called dynamic translation, it achieves support for a variety of host and target platforms along with good performance. **QEMU** is thus considerably faster than other emulators. The dynamic translation engine works as follows: whenever new code is executed in the emulator, **QEMU** translates the corresponding block of instructions (i.e., the instructions until the next branch) into an internal intermediate language. From this representation, the code is optimized to reduce unnecessary overhead (e.g., setting certain flags that are not further evaluated anyway) and then translated into the final, architecture dependent target code. The resulting block of code is called translation block (TB). TBs are cached so that the translation process is ideally only executed once. Guest code memory accesses are translated into safe memory accesses in the target code such that they cannot escape from the isolated, emulated memory space. Self-modifying code is detected with the help of a page fault exception handler. To this end, executable pages of the emulated guest are marked as non-writable. Whenever translated code attempts to overwrite code that corresponds to a TB, an exception within **QEMU** happens and the emulator invalidates all affected TBs.

**QEMU** forms the basis of several malware analysis platforms, such as the dynamic analyzer of BitBlaze [44] (called TEMU) and Anubis [3]. Amongst other things, TEMU and Anubis extend **QEMU** by providing taint propagation tracking [32], a technique that allows to backtrack which input values influenced the value of a certain register, memory location, or similar storage units. In order to do so, every translated write operation has to be instrumented and dependencies between memory values have to be saved in a dedicated internal memory region. Taint propagation tracking thus comes with a significant performance penalty. These frameworks also introduce OS awareness through virtual machine introspection [16]. This provides access to runtime information such as running processes or loaded drivers, and also allows to hook specific events in the emulated system.
such as certain API calls. Since both systems rely on the emulation code of QEMU, they are both prone to design- or implementation-related flaws of QEMU.

2.4 Virtualizers

Another class of analysis frameworks runs the analyzed code within virtualizers such as VMware, VirtualBox, or XEN. The code is executed on the native machine, typically by using the recently introduced hardware virtualization features of Intel (Intel-VT) and AMD (AMD-V). While these frameworks provide isolation between the analyzer and the analyzed program, they eventually have to rely on the monitoring techniques already described previously in order to provide their functionality. This stems from the fact that virtualizers are not as powerful as emulators per se in that they cannot intervene at any point in execution without using additional techniques (such as binary rewriting). For example, Ether is based on the hardware virtualizer XEN and uses single-stepping in the guest to provide instruction level tracing [11].

3 Implicit Methods to Delude Software Emulators

Obviously, attackers have an incentive to evade automated malware analysis frameworks. Thus, there is an arms race in this area where the attackers are constantly searching for new methods and techniques to detect such analysis frameworks. To this end, different techniques to detect emulators were introduced in the last few years [14, 15, 34, 36, 39] and several systematic studies on detection approaches were performed [28, 29, 36]. As a result, there is a large body of work on detection approaches and attackers have plenty of ways to detect the presence of a virtual environment. In the following, we introduce another approach in which code implicitly behaves differently on a native machine compared to an emulated one, a technique we call delusion attack.

3.1 Motivation

To motivate the benefit of utilizing hardware features for dynamic program analysis, we propose a new class of emulator detection techniques. Current methods consist of two different steps: first, the existence of a non-native system environment is probed and then, depending on the outcome of the test, different actions are performed. These detection attempts are easy to spot and mitigate during (manual or automated) examination of the performed operations [48]. In contrast, our methods have no explicit check and do not contain a conditional branch that takes one control path on a native machine and another on an emulated one. Even powerful analysis techniques like multi-path execution [30] cannot analyze this kind of code sequences since the emulator itself does not execute the correct code. We call this delusion attacks to emphasize the fact that such code sequences effectively delude emulators in the way code is interpreted.

All our examples that we present follow the same methodology: a sequence of instructions is executed and as an implicit effect, either a malicious or a benign function is called. In a real attack, the malicious instructions are executed directly inline instead of calling a separate function. For the sake of simplicity we use two dedicated subfunctions in our examples: MALICIOUSCODE (that should be executed on a native system) and BENIGNCODE (to be executed in an emulator). The examples were implemented and validated against real hardware as well as the targeted emulators.

There are hundreds of different processor types (including the different stepping of CPUs) available that vary in small implementation details. Due to this fact, we were not able to test all of them in an empirical evaluation. Hence, we focus our analysis on common Intel and AMD CPUs and, as a result, we cannot claim that these techniques are universally usable and provide a way for a guaranteed emulator delusion. The goal of this work is to show that there are still and – most probably – will always be methods to exploit behavior differences of emulators in order to force different execution for the same piece of code. As a result, we argue that it is not safe to trust analysis results that are gathered with the help of emulators since a program might behave differently on a native machine.
Note that traditional detection techniques could be modified in such a way that conditional code branches are transformed into \textit{branchless code} as well. Therefore, the boundary between those and our new delusion methods is blurred to a certain extent. Nevertheless, we are confident that our techniques are significantly harder to detect and mitigate compared to previous approaches.

### 3.2 Basic Principle: Self-Modifying Code and Atomicity

Several of our attacks are based on self-modifying code (SMC). Correct handling of SMC is a non-trivial and complicated task when not done by the CPU itself, but by an emulator. Thus we expect that an attacker can use SMC to detect the presence of an emulator.

On a native system, the modification of data within a code segment has to trigger different actions. Most importantly, the old version of the modified code has to be flushed from the instruction prefetch queue and from the instruction caches. Depending on the underlying cache organization and the number of processors available within the system, also the other CPUs have to be informed and take care of this problem accordingly.

Contemporary systems contain sophisticated measures to detect SMC correctly and there have been many flaws in the past that required the developer to perform specific actions in order to realize properly working code. For example SMC typically only operated correctly if (after modifying the memory) either a jump operation to that modified code or a memory-serializing operation (e.g., \texttt{cpuid} instruction) had been performed. Additional problems occurred with instructions that had already been loaded into the instruction prefetch queue: since only the linear address of a modified memory location was checked, it was possible to use two different linear addresses for code and data access, which both are associated with the same physical memory.

Modern CPUs can handle these older problems with SMC correctly. In an emulator, however, the CPU facilities for SMC detection obviously cannot be utilized. Hence, they have to be emulated and implemented in software as well. One way is to check every memory write operation against a list of addresses that possibly contain instructions or vice versa when an instruction is about to be executed. Apparently, this implies a huge overhead of execution performance and space required for managing the related data structures. Thus, most emulators use page fault handling for SMC detection. To this end, all executable memory pages are marked as read-only. If the emulated code performs a write attempt to such a memory area, the page fault handler is triggered. It performs the following actions if the target memory should be writable, i.e., if it was marked read-only by the emulator and not by the application itself:

1. all other threads are suspended,
2. the memory protection is modified to writable,
3. the faulting write instruction is executed again,
4. the memory protection is changed to read-only, and
5. all other threads are resumed.

This approach is problematic due to the fact that emulated instructions are normally not executed atomically, but they are translated into several sub-instructions. Therefore, the faulting write operations may already be partially executed and then have to be re-executed after the memory is made writable. This behavior can be exploited for delusion purposes as shown in the following subsection.

### 3.3 REP MOVs Instruction

The first delusion method uses the \texttt{rep movs} instruction, which copies a number of bytes, words, or double words within an implicit loop. The source memory location is specified by the \texttt{esi} register, the destination location by \texttt{edi}, and the amount of copy iterations by the value of \texttt{ecx}. This value is decremented with each copy iteration and used as a stop condition once it reaches zero. Accordingly, the value of \texttt{ecx} equals to 0 when the complete loop is finished.

To delude an emulator, the copy destination can be set to the memory address of the \texttt{rep movs} instruction itself. As an effect, this instruction is overwritten by the first copy iteration. On a real machine, the copy loop is performed atomically, so this instruction overwriting has no actual
effect on the loop execution. After the copy operation is completed, \texttt{ecx} is zero and the \texttt{rep movs} instruction, as well as the consecutive ones, are overwritten. In an emulator, however, the situation is different: due to the detection mechanism already the first loop iteration triggers the page fault handler when trying to write to memory that contains code. The emulator makes the destination memory writable and re-executes the memory write operation. Afterwards, the instruction is re-read from memory, in order to not miss any SMC. Since now the re-fetched instruction is no longer \texttt{rep movs}, a different behavior arises when compared to a real machine. For example, if the instruction is overwritten with \texttt{nop} operations, only one single copy loop iteration is performed: only the \texttt{rep movs} instruction is overwritten and the following instructions remain untouched. Furthermore, the \texttt{ecx} register is only decremented by one.

This different behavior can be exploited by the delusion code shown in Figure 1. Note that the \texttt{movsd} instruction copies one double word per iteration. On a real machine, two copy iterations are performed and, therefore, two double words are copied from \texttt{NEW} to \texttt{OLD}. After finishing, the memory at \texttt{OLD} +0x4 contains the call to the \texttt{MALICIOUSCODE}. Accordingly, the malicious code is executed. Hence, on an emulated machine, the copy operation stops after overwriting the \texttt{rep movs} and the call to \texttt{BENIGNCODE} is not modified and, therefore, the benign code will be executed. \texttt{QEMU} and \texttt{BOCHS} can be successfully deluded with this technique.

We would like to stress that the deviating behavior can be fixed in the emulators. However, this would require special handling for a variety of instructions that can be used in conjunction with \texttt{rep}. This not only takes considerable effort to implement, but would reduce the performance of string copy operations in general. As an implementation detail note that not all CPU types behave similar when executing the code shown above. We found that some versions of the latest Intel i7 CPUs react on the modification of the \texttt{rep movs} instruction and terminate the loop prematurely. In contrast, most other CPUs interpreted this code sequence in the way discussed above.

### 3.4 LEAVE Instruction

Our second method also exploits the SMC detection mechanism of emulators based on page fault handling. It requires virtual memory and utilizes the x86/64 machine instruction \texttt{leave}, which behaves like the two instructions \texttt{mov esp, ebp} and \texttt{pop ebp} combined into a single operation. On a real machine, the \texttt{leave} instruction is always executed atomically. However, within an emulator it is possible to force an only partial execution. If for instance the \texttt{ebp} register initially is set to an inaccessible virtual memory address, this address will be correctly copied into \texttt{esp}, but the \texttt{pop} operation will trigger a page fault. If the emulator does not take special care of this situation, the \texttt{esp} value contains the overwritten value from \texttt{ebp} when entering the invoked exception handler.
Obviously, on a real machine \texttt{esp} has not changed at all, since the \texttt{leave} operation could not be executed completely.

Figure 2 shows a small code snippet that demonstrates how to utilize this different behavior. The code makes the following assumptions:

1. \texttt{exc_handler} is set as the \texttt{exception handler},
2. \texttt{invalid_addr} points to an unallocated memory page, i.e., reading from this address causes a protection fault, and,
3. the page located before \texttt{invalid_addr} is allocated.

As described above, the code exploits the fact that under special circumstances the \texttt{esp} register is modified on an emulator, while it is untouched on a real machine. The idea now is to set up two different stack locations and store differing function pointers within each location. The proper preparation of the stacks is performed at the beginning of the \texttt{start} function. For the emulator, it prepares the memory immediately before \texttt{invalid_addr} and for the native system the current stack is used. Since the stored function address is 4 bytes in size and the involved exception handler uses 0x10 bytes function parameters, an offset of \(-0x10-0x4\) is used.

Afterwards, the \texttt{ebp} register is loaded with \texttt{invalid_addr} and the \texttt{leave} instruction is executed. Since the memory pointed to by \texttt{ebp} is invalid, a protection fault is triggered in any case. Within an emulator, the \texttt{esp} register is already updated with the \texttt{ebp} value as an effect of the partial \texttt{leave} execution. On the contrary, on a real system this register is not modified at all. Before the exception handler is executed, the system pushes four parameters onto the current stack position, which contain further exception information. As depicted in Figure 3, different stack locations are used when the exception handler is called, depending on the current \texttt{esp} value. As a result, the function pointer stored immediately before the pushed parameters is either that of \texttt{BENIGNCODE} (within an emulator) or \texttt{MALICIOUSCODE} (on a real system). The exception handler now simply returns to that specific function and thus leads to a different behavior.

While we have found \texttt{QEMU} to be prone to this delusion attempt, \texttt{BOCHS} surprisingly handles this situation correctly and cannot be deluded by this method. An attacker can thus use this technique only on \texttt{QEMU} to detect the presence of an emulated environment.

### 3.5 INVD Instruction

Besides SMC, there are other aspects of a system that are hard (if not impossible) to deal with when building an emulator. One example are the many different kinds of caches available on a contemporary computer system. Some of these caches only contain data, others only instructions, and there are also combined caches that store both data and instructions. Furthermore, some caches are integrated into the CPU itself and others are placed outside (i.e., somewhere between the processor and the main memory).

Emulators cannot use these hardware facilities explicitly, since they need total control over the accessed data and the executed instructions. More precisely, emulators implicitly share the cache with the host operating system, since they also use the RAM for storing data and instructions. Nevertheless, \textit{inside} the emulated system there is no explicit cache support and all cache-related instructions have no effect when being executed. Disabling all cache-related functionalities inside...
the emulated machine is the only reasonable way, since the simulation of caching facilities would degrade the performance of memory accesses even more and that would be counterproductive to the reason for using caches at all.

This missing ability to emulate cache is exploited by our third delusion technique. It works by utilizing write-back cache, which has no effect on an emulated machine. First of all, the instructions residing in a cached memory location are modified. On a real machine, the modification only affects the cache and the propagation to RAM is delayed for a while. On an emulated machine – and on each machine without caching – the modification is written directly to RAM. Now, immediately after modifying the memory, the cache is invalidated. On a real machine, this undoes the previous modification, while on an emulated one it (again) has no effect. Finally, the instructions within that memory buffer are executed and a different behavior between native and emulated machines is achieved. The actual code for this method is shown in Figure 4. For sake of simplicity, the instructions for enabling the caching for memory region \( A \) are left out. The code starts with writing back all potentially pending cache modifications to the RAM via the \texttt{wbinvd} instruction. Then the \texttt{call ebx} instruction is modified to \texttt{call eax}, which changes the call target from \texttt{MALICIOUSCODE} to \texttt{BENIGNCODE}. Afterwards, the instruction \texttt{invd} undoes this modification on a real machine, but not within an emulator. Finally, the call is executed to the resulting call target.

Although it is easy to detect such a scenario since \texttt{wbinvd} is an uncommon instruction, it takes vast effort to perfectly emulate the effects of \texttt{wbinvd}. This would require to provide a write history buffer that holds the recently written values to roll back the invalidation, which results in a significant performance degradation of the entire emulator. The instruction also needs elevated privileges (ring-0). However, this poses no problem since a full-system emulator can also be used to analyze privileged code. We verified that this kind of delusion attack works against both \texttt{BOCHS} and \texttt{QEMU}.

![Figure 3: Memory Map for the \texttt{leave} example.](image3)

![Figure 4: Delusion with the help of the \texttt{invd} instruction.](image4)
4 Binary Analysis with Branch Tracing

Motivated by the examples of detecting emulated environments, we now introduce an approach to observe the actual operations performed by a CPU. We then do not need to take into account the effects of SMC, caching effects, or other kinds of delusion and/or detection attacks. Effectively, this is the lowest level an analysis framework can be based on since we directly observe the behavior of the code when running on a CPU, i.e., we obtain a precise trace of the runtime behavior of the code.

4.1 General Approach

To implement such a tracing framework, we take advantage of the branch tracing (BT) facilities available on x86/64 architectures from Intel and AMD. Though the implementation details for both platforms differ, the general approach is the same. Since only the Intel specifications contains detailed information on this topic [18], we mostly refer to the names and mechanism descriptions in that specification. In addition, we also learned how to employ BT on the AMD platform through reverse engineering and empirical experiments, and thus we are able to present implementation details for this platform as well.

We note that Intel and AMD both publish public documentation regarding light-weight processor performance monitoring mechanisms [1, 19]. While these mechanisms allow tracing of retired branch instructions, they are severely restricted in the information that can be captured. For example, Intel CPUs only log the last 4 to 16 branches. While AMD CPUs do not place any restrictions in terms of the number of branch instructions that can be profiled, they can only capture branches in user-mode (ring-3) and cannot capture far jumps, returns, syscall/sysexit, exceptions, SMM mode etc [1].

As discussed before, the traditional way to trace a binary program operates on the instruction level. This can be achieved by either utilizing specific hardware features (e.g., single-stepping the CPU or virtualization features [11, 38]) or by emulation [3, 44]. For the single-stepping case, some specific debug control registers are set such that the CPU stops execution after each performed instruction and invokes an exception handler. This handler then can be used to examine or modify the processor registers or the memory, e.g., the heap or stack memory. Before returning control to the interrupted piece of code, the tracing can be re-enabled, since it is normally deactivated automatically after each handler invocation. Virtualization features of modern CPUs can be used to also generate single stepping traces, but this approach has a severe performance penalty in practice.

A more coarse-grained tracing granularity has been employed in the last years: when tracing on the function-/system-call level, execution is not stopped after each single operation, but only when specific functions are called. Often the set of monitored function calls is restricted to a subset of critical system calls. On interception, several actions can be taken. Typically, the call parameters are first examined and optionally modified. Then, the originally called function is executed or simulated. Finally, the result values are examined and/or modified appropriately. There exist several techniques for function level tracing. While native systems mostly apply API hooking [3, 44, 50], the usage of virtual machines empowers the analyst to perform virtual machine introspection (VMI) [3, 11, 16].

The granularity of tracing on the branch level is located between those of instructions and function calls. The interception happens on each taken branch, i.e., on each conditional and unconditional jump, call, interrupt, and exception. In the preceding the term tracing was used to describe a technique in which execution of a process is actually stopped after each instruction, branch, or, function call. However, with BT logging the execution is actually not interrupted, but only log information is stored at each interception point which results in a significantly smaller overhead. In practice, the performance overhead of BT logging is smaller than interrupting the actual execution (either via single-stepping the CPU or using virtualization features). Branch tracing provides a rather coarse overview of the behavior exposed by a given binary: the data which is collectable by BT logging is less comprehensive: the trace only contains the addresses of
the source and target instructions of the branches. Nevertheless, even by viewing only addresses, it is possible to completely reconstruct the execution/decision path that was taken during execution as we will show in Section 5. In the following, we describe the BT mechanism in detail for both Intel and AMD platforms and discuss how this feature can be used on these two different x86/x64 hardware platforms.

4.2 Branch Tracing on Intel

Depending on the actual CPU, Intel offers several different modes for branch tracing, which can be combined in several ways. The Intel reference manuals [18] state that branch trace stepping was already introduced with the P6 family and branch trace logging was added with newer CPUs like the Intel Xeon or Core Solo/Duo technology. Accordingly, all commodity processor types support the methods described in the following but it seems like the potential of this obscure feature was overlooked in the past. All branch tracing modes are enabled and configured via several control bits of the IA32_DEBUGCTL MSR (see Figure 6 in the Appendix). The different modes of operation are described in the following paragraphs.

When using Last Branch Recording (LBR), the CPU maintains an internal MSR stack of Last Branch Records that are organized as a ring buffer: when all locations of the stack have been written, the oldest position is overwritten next and so on. Depending on the CPU type, a different number of LBR stack elements are available, e.g., 4 on a Core2Duo CPU and 16 on the Nehalem architecture. To activate the recording, the LBR bit of the IA32_DEBUGCTL has to be set. Several branch source and target addresses are then written into the MSR_LASTBRANCH_i_FROM_IP and MSR_LASTBRANCH_i_TO_IP registers respectively, where i denotes the i'th stack entry. The current stack pointer is stored in MSR_LASTBRANCH_TOS, i.e., it contains the index of the most recently written stack location. LBR is very fast since it only accesses CPU registers, but it is restricted due to the limited stack size.

Instead of recording branch information into MSRs, they can also be logged to some monitoring device or the system memory. This can be enabled via the trace message enable (TR) flag of IA32_DEBUGCTL. As an effect, branch trace messages (BTMs) are generated and placed onto the system bus. If further the branch trace store (BTS) flag is set, the BTMs are also written to a dedicated buffer in memory. Like the LBR stack, this BTS buffer can also be used as a ring buffer or it can be configured to raise an interrupt each time the last slot is written to. The contents of a BTM (as shown in Figure 7 in the Appendix) are rather similar to those of the LBRs: there is a branch-from address, a branch-to address, and one additional control word that currently only contains one valid bit that determines if the branch has been predicted by the processor logic or not. Obviously, logging of each branch within a system produces a huge amount of data and storing this information to memory imposes a dramatic performance penalty. In order to reduce the amount of data, logging can be restricted to only cover branches that were executed in user mode (ring-3) or kernel mode (ring-0). The branch trace off in privileged/user code control bits BTS_OFF_OS and BTS_OFF_USR can be used to achieve this.

The last available operation mode is last branch stepping. When the single-step on branches (BTF) control bit of IA32_DEBUGCTL is enabled, an enabled trace flag TF in the EFLAGS register forces the CPU to invoke a debug exception at each branch. In the general case (i.e., when the BTF bit is not set), the TF flag invokes an exception after each single instruction.

4.3 Branch Tracing on AMD

The general approach of branch tracing on AMD machines is similar to BT on Intel systems, but there are several technical implementation differences. Since BT on the AMD platform is mostly an undocumented feature with only a few published documents talking about its architecture [27,47], the information presented in the following is a result of system-level exploration and reverse engineering. We are mostly interested in the logging of BTMs and thus we only present this mode of operation. All analysis and experiments have been performed on a Barcelona Quad-Core B3
stepping CPU and thus some technical details of our results might need to be adjusted when applying them to another AMD CPU due to differences in their specific implementation.

Similar to the Intel architecture, everything is controlled via specific MSRs. Firstly, a memory region has to be specified that is used to store the BTMs. This is done by writing the physical address of the first byte of the buffer to the MSR \texttt{AMD MSR BT BASE} (0xC0011007). The end address of this buffer has to be specified via \texttt{AMD MSR BT LIMIT} (0xC0011009). The logging of BTMs is then enabled by setting an appropriate control value into the \texttt{AMD MSR BT CONTROL} (0xC0011010) register. For the system we used during our analysis phase (\textit{AMD Barcelona Stepping B3 Quad-Core}), a value of 0x20230340 enables the logging. After activation, each record is written to the next free slot of the specified buffer, which is specified by an index pointer in \texttt{AMD MSR BT PTR} (0xC0011008).

Accordingly, the value of this register should be initialized with 0 and then is automatically incremented by one with each generated BTM. Once the end of the buffer is reached, an interrupt is raised. The interrupt handler is then responsible to reset the \texttt{AMD MSR BT PTR} back to the position of the next to be written BTM.

The format of each BTM differs from the records created on an Intel system. Each message consists of 96 bits, which contain a 64 bit address value, a 16 bit segment selector, and additional 16 reserved bits. There are two basic message categories: (i) branch destination message, and (ii) branch bitmap message. The branch destination message records the destination branch address for a given conditional or unconditional branch, while the branch bitmap message records the conditional branches that have been taken before a branch destination message. Together, they help in identifying the source and destination addresses for any branch taken by the CPU.

5 Application of Branch Tracing

In this section we describe several experiments that demonstrate the effectiveness of BT over traditional analysis approaches and show its wide applicability. We first show that the addresses contained in the BTMs are sufficient to reconstruct precise code paths taken during execution in the context of a practical and challenging application called \textit{binning}. Here, we are interested in automatically grouping crash reports resulting from malware exploiting a vulnerability into different representative classes. After that we present how BTs can be enriched with additional information to obtain a deeper insight into executing code and demonstrate how complex return-oriented programming attacks [41] can be detected and analyzed with this approach. Finally, we describe a practical delusion attack that underlines the necessity of a robust tracing facility such as BT since traditional analysis approaches are unable to produce correct results.

5.1 Experiment 1: Binning of Malicious PDF Documents

One powerful application of BT is the grouping of crash reports gained by fuzzing [17, 33]. This kind of automated vulnerability analysis very often produces a large number of application crashes that are ultimately caused by the same software vulnerability. Since the post-verification of each single crash is very time consuming, an analyst wants to reduce the amount of reports to be examined as good as possible. For that purpose a technique called \textit{binning} is used, in which the crash reports are automatically grouped into different classes, each one consisting of crashes that are a result of the same root cause. This saves a lot of time, since an analyst only has to manually analyze one instance of each resulting bin. One efficient way to realize binning is to compare the execution paths that have led to the crashes. Obviously, the specific part of a BT log that was protocolled just before an error has occurred contains all the necessary information for reconstructing the control path leading to the fault. The same technique can also be used to group a set of (possibly unknown) exploits by the root cause vulnerability they exploit.

\textbf{Trace Generation:} For evaluating this method, we have extended a tool called \textit{CWXDetector} [49] that is capable of detecting exploitation attempts and extracting shellcode used during the exploit. The tool uses a detection approach that is generic in the sense that it captures the
fact that unauthorized code is being executed and is thus capable of detecting shellcode that is embedded and invoked from arbitrary types of data or programs. For example, the tool can be used to extract shellcode from malicious Microsoft Office documents or shellcode that is contained in network traffic. One of its limitations is that it does not become active before the execution of the first shellcode instruction. As an effect, no information can be gained about the exploited vulnerability that led to the execution of malicious code. One way to gain more insight about the actual root cause of the exploitation is to utilize BT in combination with this tool. This enables to virtually “look into the past” once the shellcode execution is detected: the concrete execution path that led to the malicious instruction can be reconstructed and examined in detail.

For the evaluation, we have added BT support to CWXDetector and examined a set of 4,869 malicious PDF documents. This malware corpus was originally collected by a well-known AV vendor in January 2011 from different sources [49] and it is known that each file in this corpus exploits some kind of vulnerability in Acrobat Reader 9.00. Hence, we could be sure that opening each file within that particular Acrobat Reader version would lead to a successful exploit. What we got as result of this experiment is a set of 4,869 different exploit reports with BT logs that cover the last 10,000 branches taken before the first shellcode instruction was executed. Note that we could also generate similar reports with other kinds of analysis tools (e.g., Ether or single stepping), but the performance overhead of BT is significantly smaller.

Example: An example of a BT log excerpt is shown in Listing 1. The trace shows an excerpt of the behavior observed during the analysis phase based on the recorded branches. As discussed in Section 4, for each branch we obtain a log message that contains the branch source code location and the branch target. In between these branches, we do not obtain any direct insights into what code was executed within a basic block. Nevertheless, this coarse overview of the branches taken by a program already contains enough information for our purposes as we demonstrate in the following.

| 1704 | from 0x781804d7 (MSVCR80. strcat+0x87) | 1704 | to 0x781804de (MSVCR80. strcat+0x8e) |
| 1703 | from 0x781804f6 (MSVCR80. strcat+0xa6) | 1703 | to 0x781804d9 (MSVCR80. strcat+0x89) |
| 101  | from 0x781804f6 (MSVCR80. strcat+0xa6) | 101  | to 0x781804f6 (MSVCR80. strcat+0xa6) |
| 100  | from 0x80541f57 (ntkrlpa.KiExceptionExit+0xab) | 100  | to 0x791e45c (ntdll.KiUserExceptionDispatcher) |
| 99   | from 0x791e455 (ntdll.KiUserExceptionDispatcher+0xa9) | 99   | to 0x791e450 (ntdll.KiUserExceptionDispatcher+0xa9) |

Listing 1: Excerpt of a Branch Trace generated for a malicious PDF file

Binig Approach: Based on the collected data, we then clustered the BT reports into distinct bins, one for each exploited vulnerability. In order to achieve reasonable results for the binning, the logged data first has to be normalized in several ways. First, we used the relative addresses instead of the absolute ones, since the base address of modules could change over time due to techniques such as Address Space Layout Randomization (ASLR) [5,42]. Second, we collapsed loops, since we did not want to assign different bins to files that only differ in the number of loop iterations (e.g., due to differences in the size of the input data). To this end, we implemented an altered version of the approach from Tubella and Gonzalez [46], which main concept is that every backward jump forms a loop. Third, we also removed those parts from the traces that are related to the internal exception handling routines of the Windows system. Exceptions are frequently used by exploits in the last stage before control is transferred to the actual shellcode. By removing these parts from the traces, we prevent different exploits to be mistakenly put in the same bin because of this effect. Finally, we ignored those BTs of the actual executed shellcode since for binning these are not related to the vulnerability that was exploited. The shellcode is still logged and can be analyzed separately.

As clustering algorithm we have used DBSCAN [13] and as distance function a modified version of the Jaro-Winkler distance [20,51]. This function originally is used to measure the difference between two strings and calculates a similarity score based on several conditions. Mainly it is
influenced by the amount of common characters and the amount of transpositions between them. Additionally, it prioritizes the prefixes of the compared strings, i.e., strings with a similar prefix get a higher score than those with only a similar suffix. This reflects our observation that branch traces (that are always considered backwards) require a common prefix if they reflect the same vulnerability. Experiments have shown that best results could be achieved if we prioritize the last 50 branches. For performance reasons we have further limited the overall amount of considered branches to 80. Preliminary tests have shown that nearly all characteristic variations of BT logs happen within these first 80 branches.

The DBSCAN algorithm has two configuration parameters that influence its behavior and quality: the minimum cluster size $k$, which discriminates noise from valid cluster objects, and the neighborhood radius $\epsilon$, that specifies the maximum distance of two objects to belong to the same cluster. The choice of $k$ is merely a matter of taste (as long as it is greater than 1) and can be used to control the size of the resulting noise-cluster. Experiments have shown that $k = 3$ produces best results. In contrast, the value of $\epsilon$ has a severe effect on the number of resulting clusters, as can be seen in Figure 5. For very low values we get 12 different clusters and by increasing $\epsilon$ some of them merge into combined ones. From a value of 0.1 on we are left with 8 clusters and further increasing the neighborhood radius has no more effect on its number. However, for higher values the amount of the noise objects still decreases, because some of them fall into existing clusters. Note that the logarithmic scaling of the figure conceals that growing element size of these clusters.

We have manually analyzed randomly picked objects from the merging clusters and in all cases determined that they are based on the same root cause that is simply exploited in a different manner. For example there was a buffer overflow in a stack variable and one exploit diverts the control flow when the `memcpy` function is called with that buffer and another when `strcat` is executed. It is debatable if this is the same vulnerability or not. Nevertheless, by tweaking the $\epsilon$ radius one can determine how these cases are treated by the clustering algorithm. For the following comparison with other tools we have chosen $\epsilon = 0.1$, resulting in 8 different clusters and less than 10 outliers in the noise group.

**Comparison With Other Approaches:** For further evaluation of our results, we compared them against those from the PDF analysis framework Wepawet [9]. This tool combines machine learning techniques with emulation and uses signatures of known CVEs to classify malicious documents and labels them appropriately. Note that many PDF documents do not only exploit one single vulnerability. Instead, they trigger different exploits, depending on the used PDF viewer application. For those samples Wepawet may not only generate one single label, but in-
stead output a list of several CVEs. Furthermore, sometimes no known exploit can be detected at all and no label is generated. We have analyzed each sample with Wepawet, which resulted in seven different detected vulnerability signatures (CVE-2007-5659, CVE-2010-2883, SA33901, CVE-2009-0927, CVE-2009-4324, CVE-2008-2992, CVE-2010-0188). After removing those CVEs from the resulting list, which only address exploits of Acrobat Reader versions other than 9.00, we are left with only five vulnerabilities (CVE-2010-2883, SA33901, CVE-2009-0927, CVE-2010-0188, CVE-2009-4324).

While most of our clusters have been consistent with the Wepawet results, we found two general differences. First, there was a small number of samples for which Wepawet did not return any CVE number (at least after removing the CVEs that do not affect the used Acrobat version). In contrast, our BT approach was able to successfully cluster those samples into six different clusters. We have manually verified a subset of those samples and learned that other samples from the same cluster seem to exploit the same vulnerability. Second, there have been some outliers that Wepawet detected as being malicious but labeled incorrectly. Again, we were able to manually verify that our clustering has grouped them correctly with other samples that exploited the same vulnerability.

**Performance Evaluation:** Obviously, there is a performance impact when using BT. Nevertheless, all of the analyzed PDF documents actually executed their shellcode within a reasonable time: we set an upper limit for each analysis run of ten minutes and all runs finished in this time. More specifically, the fastest analysis took 11 seconds with BT (only 2s without BT), the slowest took 406s (117s without BT), and the average time was 129s (11s without BT). These measurements show that we have encountered a performance degradation factor of around 12 compared to the same system without BT. Apparently, this is magnitudes faster than performing single stepping on a native machine or with the help of hardware-assisted virtualization [11].

**Discussion:** Our clustering approach and its evaluation have some limitations that we need to discuss. First, we cannot preclude that two different vulnerabilities are merged due to the fact that some function pointer is preliminary overwritten by different means, but then later called from the same calling site. If the number of executed branches between modifying the pointer and calling it exceeds a certain amount, our approach is blinded. Second, though we are using a sophisticated loop detection mechanism, that is also able to collapse nested loops with varying loop iterations, it is not perfect and may fail to successfully collapse in certain situations. Finally, the biggest problem arises from the missing known truth about our samples. We are not able to manually analyze thousands of malicious PDF documents and there are no sources capable of delivering trustworthy information about the contained exploit, especially not AV products and other heuristic-based scanners. Therefore, we can only provide accurately generated evaluation data and reason about their validity. However, by comparing our results to those from Wepawet and further manually analyzing selected samples from our set, we are confident that our approach works properly.

5.2 Experiment 2: Enriching BT Logs

The aforementioned approach utilizes the BT log data in a straightforward way (i.e., by comparing the instruction addresses of different crashes). The derivable amount of information can be highly enriched by disassembling the particular instructions that are located at the branch sources and destinations. This enriched data can for example be used to detect code related to return-oriented programming (ROP) [41] and reconstruct the performed instruction sequences. Listing 2 shows an example of a BT log enriched with this kind of information where all RET and CALL branches are marked. If a RET without a corresponding CALL is detected, it is labeled as ROP-RET and this heuristic enables us to generically detect ROP code [10]. In the example, we can easily spot how the ROP code abuses existing, legitimate code chunks to prepare and actually perform calls to the API function `CreateFileMappingA` and `MapViewOfSection`.
As Listing 2 shows, the BT log is not only enriched by the CALL and RET sites, but our tool for branch tracing also takes advantage of the publicly available debug symbols from Microsoft. Obviously, for files from other vendors, these symbols are typically not available and, hence, it is more complicated to understand the semantics of the called functions and to isolate the root cause of a given vulnerability. Nevertheless, if such a tool is assisted by the (private) debug symbols, it is very easy to identify the actual code locations.

Though not explicitly mentioned, we already applied this kind of ROP detection during our first experiment described above. Since we did not want to take the branches of the actual shellcode into account, we had thus removed all branches that occurred before the first ROP call. In total, we found 1,721 samples in our corpus that utilized an initial ROP stage and 3,148 which did not.

5.3 Experiment 3: Practical Delusion Attack with a PDF File

Finally we demonstrate a delusion attack with the help of a malicious PDF document. For that purpose, we have generated a specially crafted file that utilizes one of the delusion techniques introduced in Section 3. This document shows a different behavior when executed on a native machine compared to execution in a virtual environment like QEMU (and as a result also in malware analysis frameworks such as BitBlaze [44] or Anubis [3]).

We used the Metasploit Framework [37] to create the PDF document. For exploiting the CVE 2010-0188 vulnerability of the Adobe Acrobat Reader 9.00, we chose the exploit/windows/fileformat/adobe/libtiff module. We then created a modified version of the existing payload module windows/messagebox, in which we utilized the rep movs technique introduced in Section 3.3. The modifications of the payload module are listed in the Appendix B and the PDF document itself can be downloaded from http://bit.ly/xtgRBE.

We have analyzed this PDF document with the help of Anubis (which is based on QEMU) as well as with our BT approach. As expected, the malicious functionality (in our case just a simple message dialog) was only triggered when the document was opened on a real machine. Within the emulator, the PDF viewer simply closed and did not show any suspicious behavior. More precisely, it is not possible to achieve any insights into the malicious functionality since no such functionality is triggered at all. Note that even powerful analysis approaches like multi-path execution [30] cannot spot suspicious code regions since there is no explicit branch that is generally not taken during emulation. By using branch tracing, we were able to successfully observe and analyze the PDF shellcode.
6 Limitations

In this section we discuss the limitations of BT in general and of our specific prototype. First, the data obtainable by BT is rather coarse. Only the source and destination addresses of program branches and no information about runtime memory or register values can be gathered. Nevertheless, this is sufficient to reconstruct the complete execution path of an application. Section 5.1 demonstrates how this kind of information can be used to generate reasonable and useful analysis results. One way to increase the quality of the BT logs is to enrich them with disassembly information as we have shown in Section 5.2.

Another concern is the robustness of our approach against detection and evasion. Our current prototype could be detected by applying timing measurements to observe the introduced performance penalty. If the attacker is operating in ring-0, she is further able to deactivate or manipulate the BT settings directly. Besides deactivating the tracing feature there is no way to circumvent it, since the logging is done by the CPU itself. Nevertheless, these are drawbacks of our current implementation and could be addressed by incorporating a hardware-assisted hypervisor. With that help each read or write access of the related MSRs or the time stamp counter could be detected and simulated. However, incorporating external timing sources still allows to reveal the existence of BT, a general limitation that we share with all other automated analysis frameworks [3,11,44].

7 Conclusion

To obtain a detailed understanding of the behavior of exploits and malicious software, many different analysis techniques and frameworks have been developed in the past few years. A huge fraction of these systems is based on the utilization of software emulators since emulators enable a fine-grained control over the sample. As a result, attackers have constantly developed new methods and techniques to detect such analysis frameworks and armored their malicious programs appropriately. In this paper, we have presented new ways how an attacker can delude an emulator. Unlike other contemporary detection techniques our methods do not combine an explicit environment check with a conditional branch. Instead they constitute implicitly different behavior on a native compared to an emulated machine caused by drawbacks of the particular operations and imperfections in the emulation process that cannot be mitigated easily.

This kind of delusion attacks motivates a new approach for dynamic code analysis: CPU-assisted branch tracing. This technique offers a granularity between instruction- and function-level monitoring and can be realized with reasonable performance overhead. In our view, the greatest advantage is the fact that the logging is performed by the processor itself and, hence, cannot be deluded since we obtain information about the actual executions performed by the CPU. In several practical experiments we showed that the obtained BT traces contain enough information to assist different tasks in malware analysis and vulnerability research.

Acknowledgement This work was supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009) and the German Federal Ministry of Education and Research (BMBF grant 01BY1205A – JSAgents and BMBF grant 16BY1207B – iTES).

References


Figure 6: IA32_DEBUGCTL Model Specific Registers of the Intel Core architecture [18].

Figure 6 shows the individual bits of the IA32_DEBUGCTL Model Specific Register. The functionality (single-stepping on branches, last branch recording, etc.) can be enabled by setting the corresponding bit in this register. Figure 7 shows the format of a Branch Trace Message that is generated and stored each time a branch is taken.

B Modified Payload Module

Listing 3 shows the modifications of the windows/messagebox Metasploit module implemented for our delusion attack with a malicious PDF file. The extended payload now calls a new function `differ` that either directly returns (on a real machine) or terminates the running process (on an emulated one). This is achieved by using the `rep movs` technique introduced in Section 3.3 to overwrite the saved return address on the stack when running within an emulator.
Figure 7: Format of a Branch Trace Message (BTM) for x64 systems [18].

Listing 3: Modified Metasploit Payload Module