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

Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, and Thorsten Holz
Ruhr-University Bochum, Horst Görtz Institute for IT-Security (HGI)
{firstname.lastname}@rub.de

Amit Vasudevan
Carnegie Mellon University
amitvasudevan@cmu.edu

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 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. Motivated by these findings, we introduce a novel approach to generate execution traces. We propose to utilize the processor itself to generate such traces. Mores 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 evaluate the practical viability 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 [41] 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, 6, 7, 10, 11, 22, 27, 40, 44]) 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 obfuscation and encryption [24, 31, 39], whereas dynamic analysis is typically only capable of efficiently examining a limited number of execution paths [27].

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 [13], 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

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

(e.g., *Cobra* [44]) or hardware virtualization extensions (e.g., *Ether* [10]) 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 [13,14,30,32,35] and several systematic studies on detecting virtual machines or CPU/system emulators have been performed [25,26,32]. 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 [35] 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 [17]), 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 [10].

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 [37].

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, 20, 21]. 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 [43]. 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 paper are 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 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.

Technical Report.

Due to space limitations, we have published an extended version of this paper as an accompanying technical report [47]. In that paper, we introduce more delusion attacks, describe the processor BT facilities in detail, and explain the payload of a practical delusion attack.

2. SOFTWARE EMULATORS

A lot of research has been focused on malware analysis in the past. Accordingly, many different techniques have evolved in this field, e.g., the application of debuggers and recently also hypervisors or binary instrumentation. Nevertheless, software emulation-based solutions are oftentimes more appealing for malware analysis since 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.

In this section we thus will review existing emulation techniques that are used by malware analysis frameworks. This serves as a discussion of related work and motivates our delusion attacks that we introduce in Section 3. A more detailed description of other analysis techniques can be found in the extended version of this paper [47]. There, 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.

BOCHS [23] 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 in 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 together with the disassembler IDA Pro [36], 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* [40] (called *TEMU*) and *Anubis* [3]. Amongst other things, *TEMU* and *Anubis* extend *QEMU* by providing *taint propagation tracking* [28], 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* [15]. 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*.

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 [13, 14, 30, 32, 35] and several systematic studies on detection approaches were performed [25, 26, 32]. 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 [44]. 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 [27] 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 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 steppings 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 *branchless code* as well. Therefore, the boundary between those and our new delusion methods is blurred to a certain extend. 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., `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 `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 `esi` register, the destination location by `edi`, and the amount of copy iterations by the value of `ecx`. This value is decremented with each copy iteration and used as a stop condition once it reaches zero. Accordingly, the value of `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 `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, `ecx` is zero and the `rep movs` instruction, as well as

```
        lea eax, BENIGNCODE
        lea ebx, MALICIOUSCODE
        lea esi, NEW
        lea edi, OLD
        mov ecx, 2
OLD+0x0: rep movsd
OLD+0x2: nop
OLD+0x3: nop
OLD+0x4: call eax   // => BENIGNCODE
OLD+0x6: nop
OLD+0x7: nop
        ret

NEW+0x0: nop
NEW+0x1: nop
NEW+0x2: nop
NEW+0x3: nop
NEW+0x4: call ebx   // => MALICIOUSCODE
NEW+0x6: nop
NEW+0x7: nop
```

Figure 1: Delusion attack with a `rep movs` instruction.

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 `rep movs`, a different behavior arises when compared to a real machine. For example, if the instruction is overwritten with `nop` operations, only one single copy loop iteration is performed: only the `rep movs` instruction is overwritten and the following instructions remain untouched. Furthermore, the `ecx` register is only decremented by one.

This different behavior can be exploited by the delusion code shown in Figure 1. Note that the `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 *NEW* to *OLD*. After finishing, the memory at *OLD +0x4* contains the call to the *MALICIOUSCODE*. Accordingly, the malicious code is executed. Hence, on an emulated machine, the copy operation stops after overwriting the `rep movs` and the call to *BENIGNCODE* is *not* modified and, therefore, the benign code will be executed. *QEMU* and *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 `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 `rep movs` instruction and terminate the loop prematurely. In contrast, most other CPUs interpreted this code sequence in the way discussed above.

3.4 INVD Instruction

Besides SMC, there are other aspects of a system that are hard (if not impossible) to deal with when building an

```
  lea eax, BENIGNCODE
  lea ebx, MALICIOUSCODE
  lea esi, A
  inc esi
  wbinvd
  mov byte ptr [esi], 0xD0
  invd
A:
  call ebx       // FF D3 = call ebx
                 // FF D0 = call eax
```

Figure 2: Delusion with the help of the `invd` instruction.

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, *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 2. 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 `wbinvd` instruction. Then the `call ebx` instruction is modified to `call eax`, which changes the call target from *MALICIOUSCODE* to *BENIGNCODE*. Afterwards, the instruction `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 `wbinvd` is an uncommon instruction, it takes vast effort to perfectly emulate the effects of `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 *BOCHS* and *QEMU*.

3.5 LEAVE Instruction

Our third method requires *virtual memory* and utilizes the x86/64 machine instruction `leave`, which behaves like the two instructions `mov esp, ebp` and `pop ebp` combined into a single operation. On a real machine, the `leave` instruction is always executed atomically. However, within an emulator it is possible to force a partial execution only. If for instance the `ebp` register initially is set to an inaccessible virtual memory address, this address will be correctly copied into `esp`, but the `pop` operation will trigger a page fault. If the emulator does not take special care of this situation, the `esp` value contains the overwritten value from `ebp` when entering the invoked exception handler. Obviously, on a real machine `esp` has not changed at all, since the `leave` operation could not be executed completely. A detailed description of this attack and appropriate example code to apply it can be found the technical report [47].

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.

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 [10,34]) or by emulation [3,40]. 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, 40, 46], the usage of virtual machines empowers the analyst to perform *virtual machine introspection* (VMI) [3, 10, 15].

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.

To implement 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 [17], we mostly refer to the names and mechanism descriptions in that specification. In addition, we also learned how to use BT on the AMD platform through reverse engineering and empirical experiments.

We note that Intel and AMD both publish public documentation regarding light-weight processor performance monitoring mechanisms [1, 18]. 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, sysenter/sysexit, exceptions, SMM mode etc [1].

A detailed description of the BT facilities of Intel and AMD platforms is given in the extended version of this paper [47]. There, we describe all the involved CPU registers and data structures and modes of operation.

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 *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 [37] 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* [16, 29]. 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 *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.

Trace Generation.

For evaluating this method, we have extended a tool called *CWXDetector* [45] 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 [45] 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 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)
                1601 x     ....................
[ 101] from   0x781804f6   (MSVCR80.strcat+0xa6)
[ 101]   to   0x781804d9   (MSVCR80.strcat+0x89)
[ 100] from   0x80541f57   (ntkrnlpa.KiExceptionExit+0xab)
[ 100]   to   0x7c91e45c   (ntdll.KiUserExceptionDispatcher
      )
[  99] from   0x7c91e465   (ntdll.KiUserEx...Dispatcher+0x9
      )
[  99]   to   0x7c93a950   (ntdll.RtlDispatchException)
```

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

Binning 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) [5,38]. 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 [42], 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 stilled logged and can be analyzed separately.

As clustering algorithm we have used *DBSCAN* [12] and as distance function a modified version of the *Jaro-Winkler distance* [19,48]. 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

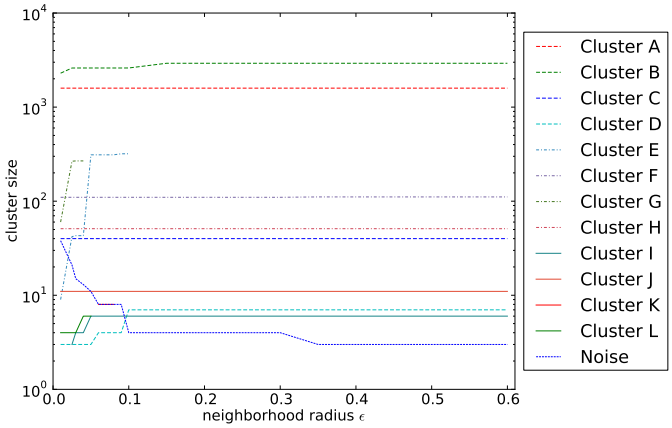

Figure 3: Distribution of Clusters Depending on ϵ.

cluster size k, which discriminates noise from valid cluster objects, and the neighborhood radius ϵ, 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 ϵ has a severe effect on the number of resulting clusters, as can be seen in Figure 3. For very low values we get 12 different clusters and by increasing ϵ 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 ϵ 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* [8]. 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 instead 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 [10].

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 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) [37] 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 [9]. 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*.

```
[54] from 0x20c9ba54  AcroForm.DllUnregisterServer+0
     x485fa7
          (ROP-)RET  ####################
[54]   to 0x4a801f90  icucnv36.ubidi_getDirection_3_6+0x18
[53] from 0x80541f57  ntkrnlpa.KiExceptionExit+0xab
[53]   to 0x4a801f90  icucnv36.ubidi_getDirection_3_6+0x18
[52] from 0x4a801f91  icucnv36.ubidi_getDirection_3_6+0x19
          (ROP-)RET  ####################
[52]   to 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
[50] from 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
              CALL  --------------------
[50]   to 0x7c8094ee  kernel32.CreateFileMappingA
....
[26] from 0x7c809545  kernel32.CreateFileMappingA+0x70
               RET  --------------------
[26]   to 0x4a807e7f  icucnv36.uhash_deleteUVector_3_6+0xe
[25] from 0x4a807e7f  icucnv36.uhash_deleteUVector_3_6+0xe
          (ROP-)RET  ####################
[25]   to 0x4a801063  icucnv36+0x1063
[24] from 0x4a801064  icucnv36+0x1064
               RET  --------------------
[24]   to 0x4a8013df  icucnv36.ubidi_getReord...3_6+0x2aa
[23] from 0x4a8013e0  icucnv36.ubidi_getReord...3_6+0x2ab
          (ROP-)RET  ####################
[23]   to 0x4a8063a5  icucnv36.uenum_count_3_6+0x1d
[22] from 0x4a8063a6  icucnv36.uenum_count_3_6+0x1e
....
[20]   to 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
[19] from 0x4a807e7d  icucnv36.uhash_deleteUVector_3_6+0xc
              CALL  --------------------
[19]   to 0x7c80b995  kernel32.MapViewOfFile
```

Listing 2: BT log excerpt for ROP shellcode

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 occured 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 malicous 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* [40] or *Anubis* [3]).

We used *Metasploit* [33] 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 extended paper version 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 [27] 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, 10, 40].

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 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 01BY-1205A – JSAgents).

8. REFERENCES

[1] Advanced Micro Devices. AMD Lightweight Profiling Specification. Specification, AMD, 2010.

[2] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[3] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[4] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*, 2003.

[6] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.

[7] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying Dormant Functionality in Malware Programs. In *IEEE Symposium on Security and Privacy*, 2010.

[8] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *World Wide Web Conference (WWW)*, 2010.

[9] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[10] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis. In *USENIX Annual Technical Conference*, 2007.

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Conference on Knowledge Discovery and Data Mining (KDD)*, 1996.

[13] Peter Ferrie. Attacks on virtual machine emulators. `http://pferrie.tripod.com/papers/attacks.pdf`, 2007.

[14] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Workshop on Hot Topics in Operating Systems (HotOS-XI)*, 2007.

[15] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[16] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[17] Intel Corporation. Intel: 64 and IA-32 Architectures Software Developer's Manual. Specification, Intel, 2007. `http://www.intel.com/products/processor/manuals/index.htm`.

[18] Intel Corporation. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide (Nehalem Core PMU). Specification, Intel, 2010.

[19] Matthew A. Jaro. Unimatch: A record linkage system. `http://books.google.de/books?id=was9AAAAIAAJ`, 1978.

[20] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating Emulation-resistant Malware. In *ACM Workshop on Virtual Machine Security (VMSec)*, 2009.

[21] Martina Lindorfer Clemens Kolbitsch and Paolo Milani Comparetti. Detecting Environment-Sensitive Malware. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[22] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2004.

[23] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996, September 1996.

[24] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Conference on Computer and Communications Security (CCS)*, 2003.

[25] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[26] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing System Virtual Machines. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[27] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[28] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[29] Peter Oehlert. Violating Assumptions With Fuzzing. *Security Privacy, IEEE*, 3(2):58 – 62, 2005.

[30] Travis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. `http://taviso.decsystem.org/virtsec.pdf`.

[31] Igor Popov, Saumya Debray, and Gregory Andrews. Binary Obfuscation Using Signals. In *USENIX Security Symposium*, 2007.

[32] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting System Emulators. In *Information Security Conference (ISC)*, 2007.

[33] Rapid7. The metasploit framework. `http://metasploit.com/`.

[34] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[35] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[36] Hex Rays SA. IDA Pro Disassembler and Debugger. `http://www.hex-rays.com/idapro/`.

[37] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[38] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[39] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[40] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newsome James, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security (ICISS)*, 2008.

[41] Symantec. Internet Security Threat Report, 2010.

[42] Jordi Tubella and Antonio Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *International Symposium on High-Performance Computer Architecture*, 1998.

[43] Amit Vasudevan, Ning Qu, and Adrian Perrig. XTRec: Secure Real-Time Execution Trace Recording on Commodity Platforms. In *Hawaii International Conference on System Sciences (HICSS)*, 2011.

[44] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained Malware Analysis Using Stealth Localized-executions. In *IEEE Symposium on Security and Privacy, Oakland*, 2006.

[45] Carsten Willems. Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis. Technical Report TR-2011-002, University of Mannheim, 2011.

[46] Carsten Willems, Thorsten Holz, and Felix C. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007.

[47] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Amit Vasudevan, and Thorsten Holz. Down to the bare metal: Using processor features for binary analysis. Technical Report TR-HGI-2012-001, Horst Görtz Institute for IT-Security (HGI), 2012.

[48] William E. Winkler. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In *Proceedings of the Survey Research Methods Section*, pages 354–359, 1990.