A fistful of red-pills:
How to automatically generate procedures to detect CPU emulators

Roberto Paleari[†] Lorenzo Martignoni[‡] Giampaolo Fresi Roglia[†] Danilo Bruschi[†]

[†] *Dipartimento di Informatica e Comunicazione*
Università degli Studi di Milano
{roberto,gianz,bruschi}@security.dico.unimi.it

[‡] *Dipartimento di Fisica*
Università degli Studi di Udine
lorenzo.martignoni@uniud.it

Abstract

Malware includes several protections to complicate their analysis: the longer it takes to analyze a new malware sample, the longer the sample survives and the larger number of systems it compromises. Nowadays, new malware samples are analyzed dynamically using virtual environments (e.g., emulators, virtual machines, or debuggers). Therefore, malware incorporate a variety of tests to detect whether they are executed through such environments and obfuscate their behavior if they suspect their execution is being monitored. Several simple tests, we indistinctly call *red-pills*, have already been proposed in literature to detect whether the execution of a program is performed in a real or in a virtual environment. In this paper we propose an automatic and systematic technique to generate red-pills, specific for detecting if a program is executed through a CPU emulator. Using this technique we generated *thousands of new red-pills*, involving *hundreds of different opcodes*, for two publicly available emulators, which are widely used for analyzing malware.

1 Introduction

With the development of the underground economy, malicious programs are becoming very profitable products; they are used to spam, to perpetrate web frauds, to steal personal information, and for many other nefarious tasks [20]. The longer a malicious program survives, the larger the diffusion and the number of victims, and the higher the financial income derived from the malicious activities accomplished by the program. Thus, to improve survivability and to maximize the financial gain, malware developers are packaging their software with protections to impede, or at least to make more complex, the analysis (e.g., packing, polymorphism, and anti-debugging techniques).

To face the increasing complexity of malicious software, the research community relays on dynamic behavior-based analysis techniques: the suspicious program is executed and monitored to analyze its behavior. Several architectures have been proposed to perform such kind of analysis. Nevertheless, the most widely adopted solution is based on *emulated* (or virtualized) environments, that offer realistic execution contexts, and provide fine grained monitoring capabilities. Typically, these infrastructures are built on top of off-the-shelf CPU emulators [3, 8], extended with introspection functionality [2, 4, 10, 22, 23].

In order to be employed for malware analysis, a CPU emulator should not only provide a bulletproof separation between the host and guest systems, but it should also operate transparently. That is, a program should not be able to notice at all whether it is run in a native execution environment (i.e., the physical CPU) or in an emulated environment [5]. The majority of victims of malware are normal end-users with very modest computer skills and consequently they do not run emulators or similar tools. Therefore, an analyzed malicious program, able to detect the presence of an emulator, could consider such presence very dangerous for its survivability and consequently would alter its behavior to thwart the analysis.

Developers of malware have built a large arsenal of tools and techniques to complicate the analysis of their software. In this arsenal we find a suite of tests that a program can perform to detect if it is run inside an emulator (or a virtual machine) and thus very likely analyzed by malware analysts. We refer to such tests as *red-pills*. Red-pill was the name of one of the first tests developed to detect the presence of a virtualized environment (e.g., VMWare) [18]. We use this term to indistinctly refer to the entire class of tests that can be used to achieve the same goal. Red-pills are typically based on one or more machine instructions that return particular information about the system (e.g., the address of the interrupt descriptor table) or that behave differently when executed in a real system and when executed in an emulated one.

A sample red-pill. An example of a red-pill, able to detect if a program is run inside QEMU [3], is the byte sequence `08 7c e3 04`, corresponding to the x86 assembly instruction `or %bh,0x04(%ebx)` (our architecture of reference is IA-32 and we adopt the AT&T assembly syntax). The instruction computes the bit-wise `or` of the value in the register `%bh` with the value, in memory, at address `%ebx + 0x04`, and stores the result in the latter. A bug in QEMU causes the instruction to reference the wrong memory address. Therefore, the red-pill consists in executing such instruction and then checking the result of the computation. If a page fault exception occurs, or if the value at address `%ebx + 0x04` does not correspond to the expected result of the logical operation, the attacker can conclude that the instruction is executed within QEMU. Otherwise, he can conclude that the instruction is either executed in the physical CPU or in another emulator.

Our contribution. Researchers have put a lot of efforts in studying and mitigating the various mechanisms a malicious application could use to detect the presence of an emulated execution environment [14, 15, 17]. The techniques discovered so far have been found manually, either by analyzing the code of the emulator or by accident.

In this paper we present a *fully automated technique for generating red-pills*, specific for detecting when a program is executed inside a CPU emulator. Our approach is based on EmuFuzzer, a testing methodology specific for CPU emulators [9], that is in turn based on differential and fuzz testing [11, 12]. Given a physical CPU P and a CPU emulator E, that emulates P, the red-pills generated using our methodology are based on sequences of bytes (representing valid or invalid instructions) that trigger defects in the implementation of E and that produce behaviors that differ from the behaviors found in P. We identify sequences of bytes that produce different behaviors in P and E by executing the same sequence in both execution environments and by comparing their state at the end of the execution. Any such sequence is a candidate red-pill and is transformed automatically into a program that executes the sequence of bytes, analyzes the state of the environment at the end of the execution, and returns 0 or 1 according to the result of the analysis. Candidate red-pills are finally tested, by running each program in multiple physical CPUs and multiple versions of the emulator, to detect whether they are reliable or not.

It is worth noting that our technique, besides being able to generate red-pills based on bugs in emulators, can also generate red-pills based on peculiar system configurations, necessary to virtualize environments on virtualization-unfriendly CPUs. That is, our technique can also generate red-pills like the original one presented by Rutkowska [18].

Our results. We have implemented a prototype tool able to automatically generate red-pills for detecting when a program is executed in two state-of-the-art IA-32 CPU emulators, namely QEMU and BOCHS [3, 8], which are widely adopted for malware analysis. Using this prototype we discovered a large number of previously unknown red-pills. Overall, we have discovered 20728 red-pills for detecting QEMU and 2973 for detecting and BOCHS. These red-pills involve instructions with 875 and 171 different numerical opcodes respectively.

2 Automatic generation of red-pills

Our technique to generate red-pills consists in two main steps: (I) generation of candidate red-pills and (II) detection and discharge of unreliable red-pills. The details described in this paper are specific for IA-32. Nevertheless, our technique is generic and could be used also with other architectures.

2.1 Finding candidate red-pills

Our technique to generate candidate red-pills is based on EmuFuzzer [9], a testing methodology specific for CPU emulators that aims at detecting improper behaviors in emulators. This methodology allows us to identify CPU states that cause different behaviors in the physical and in the emulated CPU. We exploit CPU states that produce different behaviors in the two execution environments to construct candidate red-pills.

Detecting if a given CPU state is a candidate red-pill. A candidate red-pill is a CPU state (or configuration) that causes P, the physical CPU, and E, the emulated CPU, to behave differently. Let $s = (pc, r, m, e)$ be a CPU state, where pc is the program counter, r is the state of the CPU registers (i.e., the value assigned to each register), m is the state of the memory (i.e., the content of the entire memory), and e is the exception state (i.e., indicates if a CPU exception has occurred during the execution of the last instruction). We say that E emulates *faithfully* P if, given the same initial CPU state $s_P = s_E$, the execution of the code pointed by pc in both P and E results in the same output state $s'_P = s'_E$. Practically speaking that means that the two execution environments behave equivalently and are thus indistinguishable. If instead E does not faithfully emulates P, we expect the execution of the code pointed by pc to result in two different output states $s'_P \neq s'_E$. Therefore, from the output state it is possible to tell which of the two execution environments executed the code. Any input state $s_P = s_E$ is a *candidate red-pill* if, after the execution of the code pointed by

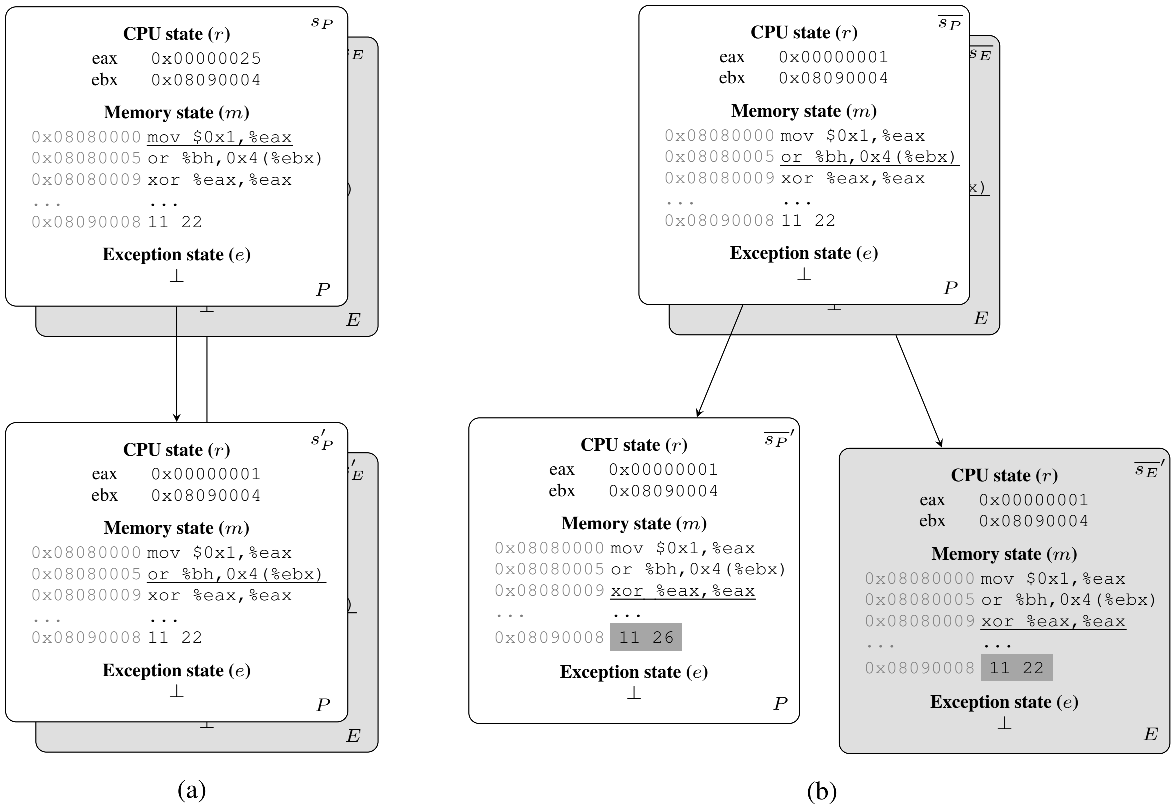

Figure 1: Comparison of the behaviors observed in the physical (P) and in the emulated (E) execution environments for two different input states: (a) no deviation in the behavior is observed, (b) the wrong memory address is referenced and an incorrect result is computed (highlighted in gray).

pc in P and E, $s'_P \neq s'_E$. Note that we do not consider information about the state of the CPU which are not directly observable by a program (e.g., difference in the content of the caches and difference in the time requested to execute a particular instruction). However, such information could give further opportunities to detect when a program is not run in a physical environment.

Figure 1 shows how our technique to detect candidate red-pills works in practice. To ease the representation in the figure we report only meaningful state information, we represent the program counter by underlying the instruction it is pointing to, and we overlap the states if they do not differ. The first initial CPU state is $s_E = s_P$ (Figure 1(a)), where the program counter points to the instruction `mov $0x1,%eax`. We initialise P and E and execute in both the aforementioned instruction (the exception state $e = \perp$ denotes that the CPU has not raised any exception). At the end of the execution of the instruction we compare the state of the two execution environments. We observe that s'_E corresponds to s'_P and therefore we conclude that the chosen initial state does not allow to distinguish between the two execution environments. The second initial CPU state is $\overline{s_P} = \overline{s_E}$ (Figure 1(b)), where the program counter points to the instruction `or %bh,0x4(%ebx)`. After having executed the instruction in both execution environments, we observe that $\overline{s_E}'$ differs from $\overline{s_P}'$: the value of the half-word at address `0x4(%ebx)` has been updated as expected in P, while the value of the half-word has remained unvaried in E because the emulator dereferenced a different half-word (not shown in the figure). The analysis of the state resulting from the execution of the code pointed by the program counter in state $\overline{s_P}$ can reveal information about the execution environments in which the code is executed and therefore the state of the example is a candidate red-pill. More details about our technique to detect different behaviors in the physical and in the emulated CPUs can be found in [9].

Generating input CPU states for candidate red-pills detection. We address the problem of generating input CPU states for candidate red-pills detection using a mixed technique: data are generated randomly and the code is generated with the help of the CPU, to include only valid instructions and with specific operands.

We leverage the physical and the emulated CPUs to identify sequences of bytes encoding instructions valid for at least one of the two CPUs and to interpret the format of the instructions found. We treat the CPU as a oracle: if the CPU executes the sequence of bytes we consider the string as a valid instruction. Such approach allows us to discard all input states that would produce the same predictable output states (i.e., invalid instruction exception in both environments). We also use the CPU to decode instructions and to systematically explore the instruction set. The decoding permits us to narrow the state space by considering only few peculiar instances of each instructions, instead of considering all the myriad of possible combination of opcodes and operands. Thus, we can search for candidate red-pills in the entire instruction set of the architecture. For each opcode found, we generate a small number of instruction instances, by choosing empirically only those operands that are more likely to exercise the largest class of behaviors of the instruction (e.g., to cause an overflow or to cause an operation with carry).

The decoding of an instruction is trial-based: we mutate an executable sequence of bytes, we query the oracle to see which mutations are valid and which are not, and from the result of the queries we infer the format of the instruction. Such CPU centric approach guarantees complete results; approaches based on assemblers (or disassemblers) instead might not allow to generate exotic, but valid, instructions. Mutations are generated following specific schemes that reflect the ones used by the CPU to encode operands (e.g., Mod R/M) [7]. The idea is that, if a sequence of bytes contains an operand, we expect all the mutations applied to the bytes of the operand, and conforming with its encoding scheme, to be valid (i.e., the CPU executes only valid mutations). If all the mutations conforming with a particular encoding scheme lead to valid instructions and some mutations generated with the other schemes do not, we conclude that the mutated bytes of the instruction encode the operand and the mutation scheme successfully applied represents the type of the operand. Moreover, the bytes that precede the operand constitute the opcode of the instruction. More details about our technique to explore the instruction set space of the architecture and to generate valid instructions for the testing without redundancy can be found in [9].

Transforming candidate red-pills into programs. Each candidate red-pill is translated into a program RP (or procedure) that executes the code of the pill in a particular CPU state and analyzes the resulting state to guess in which execution environment the execution took place. The program RP takes no input and returns 0 if it is executed on the physical CPU and 1 otherwise. Given the initial state $\overline{s_P} = \overline{s_E}$ and the diverging states observed in P and E, $\overline{s_P}'$ and $\overline{s_E}'$, the RP program operates as follows:

```
// Catch page-fault exceptions
void pf_handler(int i) {
  exit(1);
}

void main() {
  // Initialize the state:
  // - registers
  asm("mov $0x1, %eax");
  asm("mov $0x08090004, %ebx");
  // - memory
  memcpy(0x08090000, "...\x11\x22...", 4096);
  // - setup handlers to catch exceptions
  signal(SIGSEGV, pf_handler);

  // Execute the code: or %bh, 0x4(%ebx)
  asm(".byte 0x08,0x7c,0xe3,0x04");

  // Compare the state (only the potentially
  // differing bytes) and exit accordingly
  if (memcmp(0x08090008, "\x11\x26", 2) == 0)
    exit(0);
  else
    exit(1);
}
```

Figure 2: Program for the candidate red-pill of Figure 1(b)

1. initializes the state of the CPU to $\overline{s_P}$;
2. executes the instruction pointed by pc;
3. compares the state resulting from the execution with $\overline{s_P}'$;
4. returns 0 if the comparison succeeds and 1 otherwise.

Considering that the syntax of the instructions in candidate red-pills is known and that the operands of these instructions are also well known, the CPU state $\overline{s_P}$ can be minimized to include only meaningful state information, necessary to reproduce the environments required to execute each instruction. For example, for the byte sequence `08 7c e3 04` (representing the instruction `or %bh, 0x4(%ebx)`), our instruction decoder is able to decode the two operands: a register and a memory address encoded in Mod R/M encoding [7]. Thus, we can predict which memory pages the instruction will access, but also manipulate the value of the registers and the content of the memory to force the instruction to access a particular and predetermined page if needed. The practical advantage is that when encoding $\overline{s_P}$ and $\overline{s_P}'$ all non-meaningful state information (e.g., the memory pages that will not be accessed by the instruction) can be ignored. This approach allows us to translate red-pills in very small C programs (with in-lined assembly); their current average size is about 94 LOC.

Figure 2 shows a simplified RP program for the candidate red-pill of Figure 1(b). The program shown in figure further optimizes the comparison of the resulting state with the expected state ($\overline{s_P}'$) by comparing only the bytes of the memory where it was found the difference.

2.2 Discarding unreliable red-pills

Some of the candidate red-pills found could be *unreliable*: the output produced is correct only for certain physical CPUs and certain versions of the emulator, but not generally. Unreliable red-pills are typically caused by slightly different versions of the microcode of the CPU and by different sets of extensions supported by the CPU (e.g., SSE2, SSE3, SSE4). Different versions of the microcode can cause different behaviors of the CPU and thus different observable output states for the same input state. For example, for logical instructions the IA-32 specification does not state explicitly the effects on certain status flags (e.g., the value of the flag `AF` is undefined for the instruction `and`). It is reasonable to expect different versions of the microcode to produce different results in such a situation. Moreover, certain CPUs can support instructions that others do not. For example, modern CPUs support different extended instruction sets (e.g., the SIMD instruction set), not supported by older CPUs. Red-pills based on instructions belonging to extended instruction sets are clearly unreliable. On the other hand, different versions of a CPU emulator can present differences in the emulation code of certain instructions (e.g., due to bug fixes or features addition) and thus to give rise to unreliable red-pills.

Given n different physical CPUs, for the same architecture (e.g., IA-32), $P_1, P_2, \ldots, P_n$, we say that a candidate red-pill RP is unreliable if: $\exists\ P_i, i \in \{1, \ldots, n\}$, *such that RP returns 1 (emulated CPU), instead of 0 (physical CPU), when executed in P_i*. Similarly, given m different versions of the same CPU emulator (e.g. QEMU), $E_1, E_2, \ldots, E_m$, we say that a candidate red-pill RP is unreliable if: $\exists\ E_j, j \in \{1, \ldots, m\}$, *such that RP returns 0, instead of 1, when executed in E_j*. Since our goal is to discover red-pills capable of identifying reliably an emulator, we discard those that detect some versions of the emulator, but do not detect other versions. However, if one were interested in detecting the particular version of the emulator used for the analysis (e.g., to mount an attack), red-pills should be selected using different criteria.

In practice, the detection of unreliable red-pills is a straightforward task. It is sufficient to take each candidate red-pill program generated and to run it in different machines and different versions of the emulator. If in any of the environments the results obtained differ from the expected one, the red-pill is considered unreliable and thus discarded.

IA-32	QEMU	BOCHS
P 4 (2.0GHz)	0.8.2 (4)	2.3 (2)
P 4 (3.0GHz)	0.9.1 (6)	2.3.6 (4)
P Mobile (1.3GHz)	0.9.1 (10)	2.3.7 (1)
Celeron (2.66GHz)	0.10.4 (1)	2.3.7 (20090416-1)
Core-Duo (2.1GHz)	0.10.5 (1)	
Xeon (2.8GHz)		

Table 1: Environments used for unreliable red-pills detection. The numbers associated with the two emulators represent the release number of the GNU/Debian packages we used for the evaluation.

3 Evaluation

We evaluated our technique for automatic red-pills generation with QEMU and BOCHS, two CPU emulators widely used for malware analysis. However, the same approach could be used to generate redpills to detect CPU virtualizers (e.g., VirtualBox [19] and VMWare [21]). Overall, we detected 20728 reliable red-pills for QEMU and 2973 for BOCHS. The evaluation, in total took about two hours, and required no manual intervention. These numbers testify the effectiveness of the proposed approach and the importance of developing transparent CPU emulators: malware developers have too many opportunities to detect environments used to analyze their software.

For candidate red-pills detection we generated in total more than 2 millions of different input CPU states and then we selected a random subset of 50.000 CPU states. As physical CPU we used an Intel Pentium 4 (3.0GHz) and as emulated CPU we used QEMU vanilla release 0.9.1 and BOCHS vanilla release 2.3.6. For unreliable red-pills detection instead we used the environments reported in Table 1.

Table 2 reports the most important numbers of our evaluation. For each analyzed emulator the table reports the total number of candidate red-pills found, the number of reliable red-pills, and the number of unique pills, counted as the number of distinct numerical opcodes (recall that for each opcode we generated multiple instruction instances using different operands). Opcodes were identified using the algorithm described in Section 2.1. For QEMU, we detected 21713 candidate red-pills, 20728 of which turned out to be reliable. The latter were based on instructions involving 875 different opcodes. For BOCHS, we detected 4671 candidate red-pills, 2973 of which turned out to be reliable and involved 171 different opcodes.

Table 3 reports a breakdown of the unique reliable red-pills discovered, divided into classes according to the dif-

Emulator	**# candidate red-pills**	**# reliable red-pills**	**# unique red-pills**
QEMU	21713	20728	875
BOCHS	4671	2973	171

Table 2: Results of the evaluation

Emulator	**Registers**	**Memory**	**FPU**	**Exceptions**
QEMU	55	24	660	235
BOCHS	1	0	3	171

Table 3: Breakdown of the unique reliable red-pills

ference in the CPU state on which the various red-pills are based on. Note that some red-pills might belong to multiple classes. For example, the sample red-pill of Figure 2 belongs to the "memory" class. The "FPU" class encompasses all red-pills that exploit differences in the registers of the Floating Point Unit, while the "registers" class encompasses those that exploit differences in general purpose registers. Figure 3 instead reports some of the red-pills we found in the two CPU emulators analyzed.

To demonstrate the effectiveness of our red-pills at evading widely used system for malware analysis we tested them against two public malware analysis services, namely Anubis [1] and Renovo [16]. Both services are based on QEMU, although we do not know on which versions of the emulator they are based on. We developed a small set of sample programs, each of which contained two main disjoint program paths, with easily distinguishable behaviors, and a different red-pill procedure, such that the path executed at run-time depended exclusively on the return value of the red-pill procedure. We submitted these programs to the aforementioned services. In all cases the behavior described in the report of the analysis corresponded to that associated to the program path triggered when the return value of the red-pill procedure was 1 (i.e., emulated environment). This experiment demonstrates the efficacy of our red-pills and also highlights their dangerousness.

4 Related work

Several researchers have analyzed state-of-the-art CPU emulators and virtualizers to find unfaithful behaviors and the corresponding red-pill programs [6, 13, 15, 18]. These results were obtained through a manual scrutiny of the source code or rudimentary fuzzers, and thus are largely incomplete. The methodology presented in this paper instead is automatic and allows to systematically explore the entire instruction set of the architecture to detect instructions that can be used to detect the presence of a CPU emulator, and to generate the appropriate procedure.

QEMU	
`bsf -0xc(%esp),%edi`	Produces differences in some status flags
`\xb9\xff\xff\x00\x00\x0f\x22`	Raises a general protection fault exception instead of an illegal operation exception

BOCHS	
`mulps -0x4(%esp),%xmm1`	Does not raise a general protection fault exception if the source operand is not aligned
`push $0x407d`	Alters inexplicably the state of the FPU

Figure 3: Examples of some of the red-pills discovered

The complementary problem, that is developing a completely transparent environment to analyze malicious programs, has been addressed by Dinaburg *et al.* [5]. The environment they proposed and developed, called Ether, is based on a CPU virtualizer and adopts the virtualization facilities offered by modern CPUs. In such an environment, instructions are executed directly on the physical CPU and thus the state resulting from the execution of each instruction necessarily corresponds to the state that would result from the execution of the same instruction without the virtualizer. Furthermore, by adopting the virtualization facilities available in modern CPUs, non-virtualizable instructions do not open opportunities for detecting the analysis environment because they do not need manual handling [17]. Nevertheless, CPU emulator facilitates fine-grained analysis (e.g., hooking of all machine instructions) because the semantics of each machine instructions can be easily extended to incorporate other functionality. Analyzers based on virtualization instead require the use of debugging exceptions, which could introduce a high overhead.

5 Conclusions

CPU emulators are very powerful tools and are widely used to analyze malicious programs. In this paper, we presented an automatic technique for generating red-pills, that is, programs (or procedures) capable to detect whether they are executed in a CPU emulator or directly in a physical CPU. The proposed technique has been implemented in a prototype, which we used to discover new red-pills for detecting two state-of-the-art IA-32 CPU emulators, namely QEMU and BOCHS. We discovered thousands of new red-pills, involving hundreds of different opcodes.

References

[1] Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org/`.

[2] Bayer, U., Kruegel, C., and Kirda, E. TTAnalyze: A Tool for Analyzing Malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)* (2006).

[3] Bellard, F. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (Berkeley, CA, USA, 2005), USENIX Association.

[4] Böhne, L. Pandora's Bochs: Automatic Unpacking of Malware. Master's thesis, University of Mannheim, Jan. 2008.

[5] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008).

[6] Ferrie, P. Attacks on Virtual Machine Emulators. Tech. rep., Symantec Advanced Threat Research, 2006.

[7] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2008. Instruction Set Reference.

[8] Lawton, K. P. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal* (Sept. 1996).

[9] Martignoni, L., Paleari, R., Fresi Roglia, G., and Bruschi, D. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA), Chicago, Illinois, U.S.A.* (July 2009), ACM. To appear.

[10] Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., and Mitchell, J. C. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (Sept. 2008), Lecture Notes in Computer Science, Springer.

[11] McKeeman, W. M. Differential Testing for Software. *Digital Technical Journal 10*, 1 (1998).

[12] Miller, B. P., Fredrikson, L., and So, B. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM 33*, 12 (December 1990).

[13] Ormandy, T. An Empirical Study into the Security Exposure to Host of Hostile Virtualized Environments. In *Proceedings of CanSecWest Applied Security Conference* (2007).

[14] Quist, D., and Smith, V. Detecting the Presence of Virtual Machines Using the Local Data Table. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.

[15] Raffetseder, T., Kruegel, C., and Kirda, E. Detecting System Emulators. In *Proceedings of Information Security Conference (ISC 2007)* (2007), Springer-Verlag.

[16] Renovo: Hidden Code Extraction for Packed PE files. `https://aerie.cs.berkeley.edu/`.

[17] Robin, J. S., and Irvine, C. E. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th conference on USENIX Security Symposium (USENIX'00)* (Berkeley, CA, USA, 2000), USENIX Association.

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

[19] Sun Microsystems, Inc. Virtualbox. `http://www.virtualbox.org/`.

[20] Symantec Inc. Symantec internet security threat report: Volume XIV. Tech. rep., Symantec Inc., Apr. 2009.

[21] VMware, Inc. Vmware. `http://www.vmware.com/`.

[22] Willems, C., Holz, T., and Freiling, F. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy 5*, 2 (2007), 32–39.

[23] Yin, H., Song, D., Egele, M., Kirda, E., and Kruegel, C. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS, Alexandria, VA, USA* (2007), ACM.