Kutatási beszámoló a
PRO PROGRESSZIÓ ALAPÍTVÁNY
számára

Korszerű informatikai módszerek vizsgálata

Ismeretlen malware automatizált detekciója élő rendszereken

Towards the Automated Detection of Unknown Malware on Live Systems

Buttyán Levente
www.crysys.hu

2014. október 2.
Budapest

Abstract

In this report, we propose a new system monitoring framework that can serve as an enabler for automated malware detection on live systems. Our approach takes advantage of the increased availability of hardware assisted virtualization capabilities of modern CPUs, and its basic novelty consists in launching a hypervisor layer on the live system without stopping and restarting it. This hypervisor runs at a higher privilege level than the OS itself, thus, it can be used to observe the behavior of the analyzed system in a transparent manner. For this purpose, we also propose a novel system call tracing method that is designed to be configurable in terms of transparency and granularity.

1 Introduction

The problem of automated detection of unknown malware has been studied before, and it is part of the field called Host Based Intrusion Detection (HIDS). The main idea of HIDS is that successful attacks usually leave a trace of the attacker's activities including modifications to the file system (e.g., installing a keylogger or a backdoor) and to the data structures used by the operating system (e.g., registry entries, interrupt tables, and other kernel objects). In theory, such modifications could be detected in an automated way, and that is what HIDS systems try to do.

However, an important limitation of existing host-based anomaly detection approaches is that they require either to run the system to be analyzed in an isolated (usually virtualized) environment, or to install some analysis tools on the analyzed system itself. In the first case, one needs to create a virtualized copy of the analyzed system *and* its original environment (e.g., other servers in the same network) in order to run both together in the isolated analysis environment. Note that if the copy of the analyzed system runs alone, then the malware may detect the change in its environment and modify its behavior in order to escape detection. Creating a faithful copy of the operating environment of the analyzed system, however, is a major problem that requires a lot of resources and may cause interruptions in the operation of the live system. There is also a high risk that the copy will actually not be sufficiently faithful, which may jeopardize the entire malware detection process. In the second case, when some analysis tools are installed on the analyzed system itself, the problem is that the analysis will not be transparent, meaning that the installed analysis tools may be detected by the malware. In addition, in some environments such as, for instance, in IT systems of critical infrastructures, neither interruption of operation nor installing arbitrary software on the system are allowed.

We propose a new system monitoring approach that enhances either the transparency or performance of existing methods (e.g., Nitro [14] and Ether [3]) . In addition, our approach does not require to create a copy of the system to be analyzed, neither it requires the installation of analysis tools on the analyzed system itself. Thus, it is free from all limitations of prior approaches. Our approach takes advantage of the increased availability of hardware assisted virtualization capabilities of modern CPUs, and its basic idea is to launch a hypervisor layer on the live system on-the-fly, without stopping and restarting it or any of the running applications. This hypervisor runs at a higher privilege level than the OS itself, thus, it can be used to observe the behavior of the analyzed system in a transparent manner,

without installing any analysis tools on the analyzed system itself. At the same time, the live system stays in its original operational environment, so the malware may not detect any suspicious change.

In this report, we describe our design and implementation of such an on-the-fly virtualization platform, and we also propose a novel system call tracing method that can be used to observe the behavior of the analyzed system with configurable tradeoff between transparency and granularity. Our proof-of-concept implementation leverages AMD64 processors with SVM (Secure Virtual Machine) technology and 64-bit Windows Vista/7 operating systems.

2 Related work

Typical AV products try to detect known malware by signature based methods which are clearly not usable for detecting unknown malware. Therefore, many AV companies and researchers are proposing new approaches to detect unknown malware. Here, we summarize the approaches that are the most similar to the approach that we propose in this paper, and highlight the differences with respect to our approach.

McAfee DeepSAFE [7] technology provides real-time kernel memory and CPU event protection using hardware-assisted virtualization technology. This is technically similar to our approach, but DeepSAFE must be installed on the system permanently, which leads to constant performance degradation. In contrast to this, our work allows for system monitoring on an on-demand manner. While this will not allow for preventing successful attacks, it can still be used for detection. Another approach, which is being more similar to our work, is called HyperDbg [4]. It allows for the live installation of an analysis framework under a running OS using hardware virtualization. However, the original goal of HyperDbg was not the monitoring of unknown malware, but it was proposed as a general-purpose testing and debugging tool.

While previous solutions offered these monitoring capabilities from the host [14, 3], or from another trusted VM [17], a dedicated environment (e.g., KVM, Xen) had to be installed for them. On the contrary, our approach enables monitoring without any interruption in the OS operation, which is suitable for systems that do not permit downtime. In addition, our approach makes it possible to configure either the granularity or the level of transparency of our monitoring framework which can be very useful for catching unknown malware which is armed with virtualization specific anti-debug

capabilities [13, 10].

As mentioned in the introduction, the problem of detecting unknown malware belongs to the field of Host Based Intrusion Detection (HIDS). HIDS systems try to detect suspicious modifications made by hostile codes in user/kernel space data structures and user/kernel space code, or the unintended use of existing code base. Our work complements these approaches by running these monitoring mechanisms within an on-the-fly installable analysis framework. While it has been shown in [6] that system calls alone are insufficient for malware detection on their own, we believe that our approach can be completed with other building blocks suitable for detecting unknown malware behavior.

3 On-the-fly Virtualization

As mentioned in the introduction, our system monitoring approach is based on on-the-fly virtualization, which allows for launching a thin virtualization layer below the running OS without system downtime. Our solution is based on New Blue Pill [16], which was originally introduced as an offensive approach to take over the control of a system, but we modified it for our purposes. In this section, we first shortly introduce New Blue Pill, and then we describe the modifications we made.

3.1 Introducing New Blue Pill

The original Blue Pill [15] was developed by Joanna Rutkowska for COSEINC. Later, a completely new version called New Blue Pill (NBP) [16] was designed and implemented by Alexander Tereshkin. Here, the author uses the AMD64 Secure Virtual Machine (SVM) extension to virtualize the running OS on the fly. While NBP requires no modification in the BIOS, boot sector or system files to operate correctly, it does not survive OS reboot. For this reason, a kernel mode driver is installed that enables the SVM mode as a first step. From this point on, the required SVM instructions are available for the driver. After that, NBP prepares virtual machine data structure (Virtual Machine Control Block) to be used by the running OS to store its state when context change occurs. Finally, the native OS starts to execute in guest mode when the `VMRUN` instruction is executed by NBP.

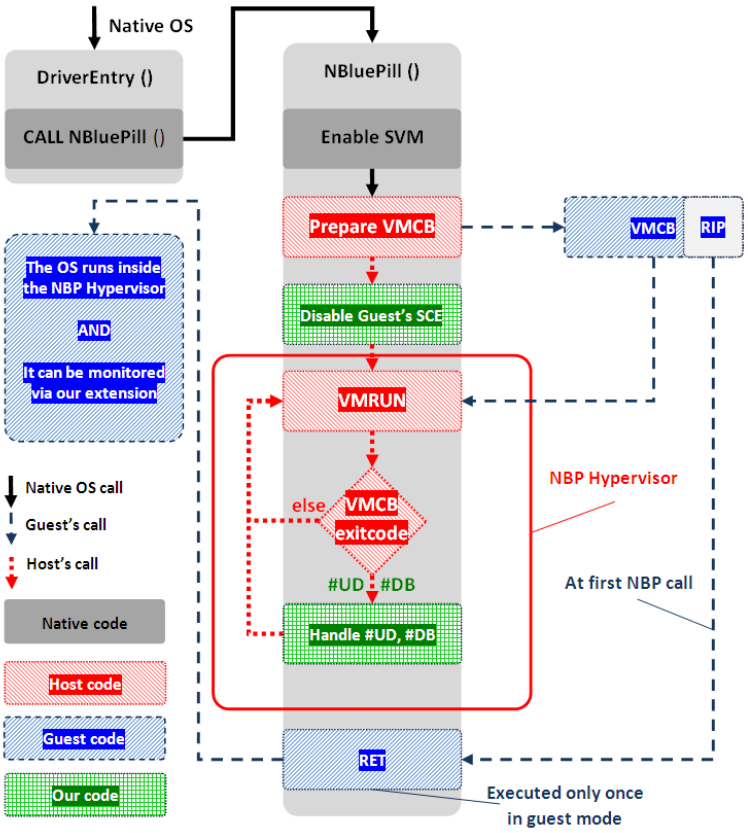

Figure 1: Extending New Blue Pill with monitoring capabilities. Using source [15, 16]

3.2 Extending New Blue Pill with Tracing Capabilities

In order to have a hypervisor-level on-the-fly installable system monitoring framework, we modified the New Blue Pill as Figure 1 depicts it. First of all, we tightened the semantic gap between the hypervisor and the monitored OS by means of virtual machine introspection (VMI) [5]. Secondly, we implemented the proposed system call tracing method for 64-bit systems. To achieve this, we exploit a hardware support available only in 64-bit processors, which makes it possible to enable/disable system calls for a running OS. If system calls are disabled in a processor and a `SYSCALL` instruction is executed an invalid opcode exception #`UD` is raised. Whenever, a sensitive or privileged instruction is executed by the guest mode OS (e.g., `SYSCALL`

in our case) a VMEXIT is generated which can be handled by the hypervisor. To identify the nature of VMEXIT, an exitcode is written into the Virtual Machine Control Block that can be later read by the hypervisor. For this reason, we registered traps for the related VMEXITs that are handled whenever the corresponding exceptions are raised in the guest mode OS. When an #UD is intercepted, we extract the required features from the guest's memory and enable system calls again. In this way, the system call can be successfully re-executed when the handler returned. By re-enabling system calls, we also guarantee a more transparent execution environment in contrast to earlier approaches [14], which disable system calls for the entire analysis process. This seems to be a subtle difference, but it can be exceptionally useful against attacks (e.g., [13, 10]) which generate invalid opcodes to detect virtualization or debuggers. So as to re-enable system call monitoring, we set a hardware breakpoint on instructions chosen by the analyst. The place of breakpoint (user-, or kernel space) influences the type of data structure we can extract as well as the granularity of our monitoring process. Thus, we register a trap for hardware breakpoint related VMEXITs to catch the system calls being executed by the guest mode OS.

This is achieved by the built-in debugging feature of NBP as it sends messages to a kernel-level debug window list, which can be read by adequate applications such as `DebugView`. Note that this kernel-level debug window list can also be reached via a kernel debugger such as `WinDbg`. More details about our system call tracing method is discussed in section 4.

3.3 64-bit CPU modes

The 64-bit mode of various vendors (AMD, Intel) comes with slightly different considerations. First of all, AMD extended the original x86 architecture with 64-bit mode (long mode) and supports previous architectures (e.g., 32-bit protected mode) in legacy mode [1]. On the other hand, the Intel 64 architecture introduces the IA-32e mode which is a new 64-bit mode supporting two submodes: the 32-bit compatibility mode and the 64-bit mode. Note that in the rest of the paper we focus on the AMD long mode, however, the concepts can be easily adapted to IA-32e 64-bit mode as well. The most important difference between the two platforms is that IA-32e still supports the `SYSENTER` instruction in 64-bit mode, however, AMD64 does not and raises an invalid opcode exception #UD if being used. Nevertheless, both of the platforms support the `SYSCALL` instruction in 64-bit mode, which makes our methods general. Furthermore, NBP supports hardware virtualization extension of Intel processors (Intel-VT) as well, so our concept could work

on that architecture as well.

4 Proposed System Call Tracing Method for x64

In contrast to previous out-of-the-guest system call tracing methods introduced by Ether [3], we designed and implemented a new, and general method that is fully compatible with 64-bit systems. In 32-bit mode the `SYSENTER` instruction is used to prepare fast system calls, which reads the corresponding kernel address from the `SYSENTER_EIP` MSR register and the target code segment from the `SYSENTER_CS` MSR. However, the `SYSENTER` instruction is not supported by AMD64, so we rely on another fast system call instruction (`SYSCALL`) that is compatible with both Intel and AMD processors. Note that in case of AMD64, the `SYSCALL` instruction can be executed in legacy modes (16-bit and 32-bit) as well, however, on Intel processors, it is supported only in 64-bit mode.

In the following we introduce the details of our novel system call tracing method for 64-bit mode. To monitor the system calls of an unmodified target OS from the hypervisor, we have to observe context changes induced by VMEXITs. However, similarly to `SYSENTER`, `SYSCALL` does not generate VMEXIT by default when being executed, thus this problem have to be solved by a specific method. One way to handle this issue is advised in [3] for the `SYSENTER` instruction. By translating this concept to `SYSCALL`, we have to modify the target address of this instruction, which is stored in *`STAR` registers (`STAR, CSTAR, LSTAR`), to an address being paged out. When the processor accesses this address, a page fault is raised (#`PF`) and a VMEXIT context change occurs. By catching the page fault in the hypervisor, the original jump address of `SYSCALL` is reloaded into *`STAR` and it is re-executed. The problem with this solution is that the overwhelming number of page faults generated in an OS during its normal operation induces a large performance degradation which makes the OS drastically slower. Moreover, due to swapping, the number of raised page faults increases significantly on machines with lower memory capacity. Furthermore, a system with more processes generates more page faults as well.

For this reason, we have to figure out another approach, where the number of VMEXITs do not depend on the hardware and the number of executed applications. Our new method, similarly to [14], is based on invalid opcode exceptions instead of page faults to mitigate the performance degradation problem of previous solutions. As a first step of our implementation, we unset the `SCE` (system call enable) bit of the guest's `EFER` register, which

makes SYSCALL instructions unknown for the processor. As a consequence, when being executed, an invalid opcode exception (#UD) is generated that induces a corresponding VMEXIT (VMEXIT_EXCEPTION_UD). At this point, the system call number can be retrieved in the hypervisor from the EAX register and the guest's user-space data structures via the GS selector (see below for more details on data structure extraction). As the #UD exception does not increases the instruction pointer (RIP), we do not have to bother with alignments to re-execute the SYSCALL. In contrast to [14], we enable here the EFER.SCE bit again for higher transparency. However, depending on the chosen transparency-granularity tradeoff, later we disable this bit again. This can be achieved in one of the following ways:

1. Set up a timer which checks the state of the bit periodically and modifies it if necessary
2. Unset the bit when other type of VMEXITs occur
3. Generate another VMEXIT for this reason

Naturally, all of these methods have advantages and disadvantages: A constant timer offered by 1) has a constant performance overhead, and have to be fine-grained enough to catch all the required syscalls. Case 2) promises better performance than 1) and 3) as a result of piggybacking, however, there is no direct control here on VMEXITs to gain a given system call granularity. For this reason, we chose option 3), which puts extra performance overhead to our solution, but makes the transparency of our analysis controllable. Note that in Section 5.2, we demonstrate that the suggested method still offers acceptable performance loss. Thus, when an invalid opcode exception is handled, and the required features are extracted by reading the guest memory, a hardware breakpoint is inserted (by means of debug registers DR0 and DR7) to an address that has to be reached before the next chosen system call depending on the chosen transparency-granularity level. This breakpoint can either be placed on a user-, or kernel-space code, however, it predestinates if user-, or kernel-level objects can be extracted when handling it. See Figure 2 to read the details of the invalid opcode exception handler algorithm.

When the guest's instruction pointer reaches our hardware breakpoint, a VMEXIT_EXCEPTION_DB is raised in the processor that is handled by our registered trap. This handler disables system calls again by disabling EFER.SCE for the guest mode OS and deletes the breakpoint. Finally, OS objects can be extracted here as well. The details of this algorithm is described in Figure 3.

Algorithm for Handling Invalid Opcode Exceptions (#UD)

1. Read the guest's `EFER` value from VMCB (Virtual Machine Control Block)
2. If `EFER.SCE` is set, then RETURN.
3. Retrieve the candidate syscall number from `VMCB.EAX`.
4. If the syscall number is valid, then
   (a) Set the `EFER.SCE` bit. /* Enable syscalls again for the guest */
   (b) Set `DR7.G0` and `DR7.GE` bits /* Enable global exceptions for `DR0` */
   (c) Set `DR0` to an appropriate address. /* Depends on data extraction */
   (d) Extract user-space data objects via VMI.

Figure 2: Algorithm for handling invalid opcode exceptions.

Algorithm for Handling Debug Exceptions (#DB)

1. Read the guest's `EFER` value from VMCB
2. Unset the `EFER.SCE` bit.
3. Unset `DR7.G0` and `DR7.GE` bits /* Disable global exceptions for `DR0` */
4. Extract data structures. /* Depends on CPL */

Figure 3: Algorithm for handling debug exceptions.

In order to extract the required features for a usable system call tracer, we revealed OS-level data structures by means of virtual machine introspection [5]. In contrast to previous solutions, where mainly Windows XP operating systems were monitored (e.g., [3, 12]), the current process was extracted from fragile user-space data structures such as Thread Information Block (NT_TIB) [17], or via low-level paging information [14], we show how the more robust, kernel-level data structures can be extracted from a 64-bit Windows 7 OS.

The first main difference is that 64-bit operating systems use flat memory model, so different segment registers (e.g., stack) are treated with the same base: 0 [1]. Another difference is that user and kernel-space data structures can be accessed via the `GS` register instead of `FS` as in the x86 Windows NT family. Furthermore, the well known offset values between kernel objects also vary in different Windows versions. On top of this, data structures reside at varying addresses due to Address Space Layout (Load) Randomization (ASLR) in Windows Vista and later versions. It means, that only consecutive memory reads allow us to get sufficient information. For example, in Figure 4, we show how the current process name (`EPROCESS.ImageFileName`) can be extracted via the `GS` segment register at kernel mode in case of a 64-bit Windows 7. Furthermore, many other kernel structures (e.g., KPCR (Processor Control Region)) can be extracted

relatively easily via the kernel mode `GS`. In order to extract user-level data structures, the processor has to be in user mode. Thus, when a `SYSCALL` is invoked, the processors checks if system calls are enabled, and if they are not, an #`UD` exception is generated before a context change could have occurred to CPL=0. That is, our #`UD` handler is an ideal place to extract user-level data structures as the guest mode OS is still in user mode.

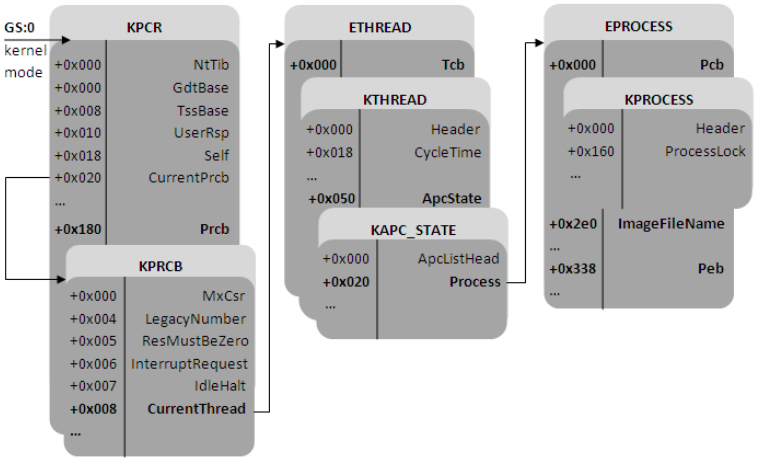

Figure 4: 64-bit Windows 7 data structures.

5 Evaluation

In this section, we evaluate our system on an AMD Athlon 64 X2 Dual Core 6000+ processor with 6 GB of RAM and 64-bit Windows 7 operating system. We use relatively large RAM capacity so as to demonstrate that the OS raises too many page faults even in this case, which is unacceptable on mission critical systems.

5.1 Tracing Malware

To evaluate our system we downloaded all the in-the-wild 64-bit samples from Offensive Computing [9], which were available at the time we did the evaluation. We searched this database for both the "win64" and "w64" strings, but only some samples were provided with the following labels: Win64.Rugrat.A, Trojan.Win64.A, Win64.Shruggle.1318, W64/BackdoorW.(C and D) and w64/Gael.A. Note that AV labels can be confusing as different

vendors can identify the same malware differently. The small number of 64-bit malware samples indicates that we are at the very beginning of a new era in the field of malware research and defense.

In order to successfully install our system, we had to disable the driver signing feature in Windows 7, however, this is an unnecessary step if we signed our driver with a trusted CA. Note that driver signing was built into the Windows operating system from Vista, in order to eliminate drivers with unknown origin. On the other hand, driver signing turns to come with problems due to the appearance of malware signed by legitime private keys (see e.g., Stuxnet and Duqu [2]). After downloading these samples, we started them in our isolated test environment on a sanitized physical machine. Then, we installed our tool and verified the execution of samples by observing known system call traces.

5.2 Performance

5.2.1 #UD versus #PF

Another important contribution of this paper is to demonstrate that our system call tracing method guarantees better performance than the well-accepted method [3, 17] applied on x86 systems. First of all, we registered traps for `#UD` and `#PF` in the hypervisor to measure the number of invalid opcode exceptions and page faults that occur in a system by default. To demonstrate the performance differences between the two methods, we first counted the number of page faults and invalid opcode exceptions generated by the same system under various conditions. To achieve this, we registered a trap handler for `#UD` in NBP, and we used `TraceView`, available as a part of the Windows Driver Kit [8], to count the number of page faults at the same time. Note that we could not measure the number of `#PF` with NBP due to the low response time of the monitored system. Our measurements demonstrate that the number of page faults highly depends on the configuration of the machine such as the processes being launched.

To verify that extra processes increase the number of page faults, we measured the number of exceptions right after the reboot of the target system and after starting the following applications: Wireshark, Foxit Reader, Internet Explorer 8, Process Explorer, CPUID CPU-Z, CPUID Hardware Monitor and MobaSSH Server. We summarized our results in Table 1. The most interesting part of our measurement is that we did not observe any invalid opcode exceptions (even with lengthened time window of 30 minutes) under the normal operation of the OS. On the other hand, as we can see the

Table 1: Counting the number of page faults and invalid opcode exceptions being raised under native operation on two CPU cores. The examined time interval is 2.5 minutes, and the statistics are calculated from samples with 12 elements.

Configuration	#UD		#PF	
	Mean	Std	Mean	Std
1. No extra process	0	0	16935	15839
2. Extra processes	0	0	43547	24155

number of page faults highly depend on the active processes.

5.2.2 Benchmarking Syscall Tracing

Table 2: Results of Passmark CPU performance test for the unmonitored, blue-pilled and monitored system. The first column lists the operations executed by the test, while the numbers show the mean and standard deviation of execution times.

Operations	Native		Blue-Pilled		Monitored	
	Mean	Std	Mean	Std	Mean	Std
1. Integer Math (MOps/Sec)	344.5	2.40	346.0	1.13	228.0	82.35
2. Floating Point Math (MOps/Sec)	1072.3	8.30	1072.4	5.63	742.3	226.88
3. Find Primes (Thousand Primes/Sec)	322.9	3.22	323.7	1.14	220.1	74.44
4. SSE (Mill. Matrices/Sec)	9.4	0.06	9.4	0.10	6.3	2.17
5. Compression (KBytes/Sec)	2063.1	42.84	2072.6	6.99	1333.1	430.76
6. Encryption (MBytes/Sec)	9.7	0.03	9.7	0.03	6.1	2.08
7. Pyhsics (Frames/Sec)	95.9	1.98	95.6	1.99	63.7	17.57
8. String Sorting (Thousand Strings/Sec)	1243.6	27.86	1254.0	11.46	843.1	235.83

We also measured the performance loss of our system call tracing method by means of the Passmark Performance Test [11]. We executed the tests 12 times in each configuration, and calculated the corresponding statistical

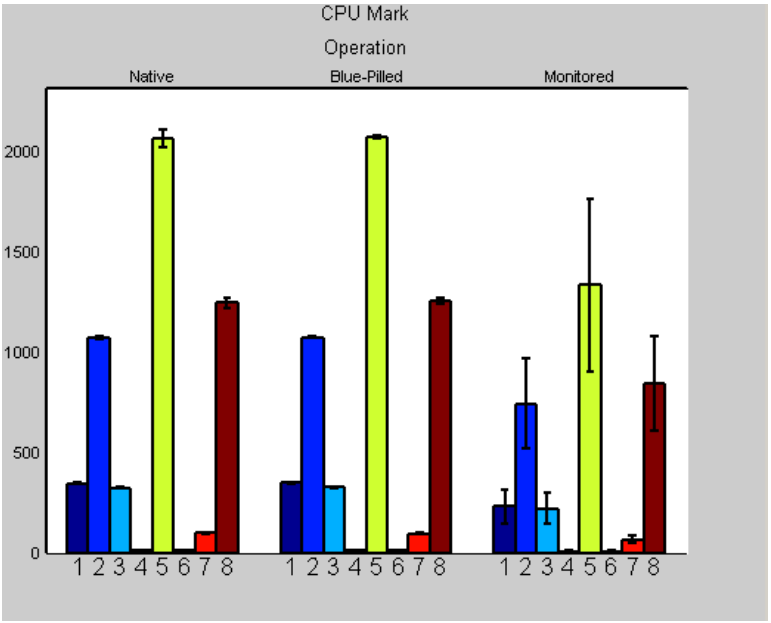

Figure 5: Statistics of CPU performance test for the unmonitored and monitored system. The bars show the mean value for the corresponding CPU operations in collaboration with their standard deviation.

metrics (mean and standard deviation) for them. We evaluated the performance of the CPU in native operation, when it is Blue-Pilled and when it is monitored by our extension.

As Table 2 and Figure 5 demonstrate, NBP itself does not cause performance overhead to the system, so it offers ideal preconditions for our monitoring extension. When monitoring the system, the observed performance loss ranges between 30.7% (Floating Point) and 35.4% (Compression), which is still acceptable in practice as the OS actually remains usable. Note that we configured our system-call tracer to perform maximally, thus we re-enabled system call monitoring right after the previously evaluated `SYSCALL` instruction. However, the performance degradation of the monitored system was still influenced by the executed debug print operations to verify the result of traces. Furthermore, we benchmarked the memory overhead of the system as well under the same conditions. As Table 3 and Figure 6 show, our monitoring extension induces negligible memory overhead in the range of 0.1% (Read Cached) and 8.5% (Allocate Small Blocks).

Table 3: Results of Passmark Memory performance test for the Unmonitored, Blue-Pilled and Monitored system. The first column lists the operations executed by the test, while the numbers show the mean and standard deviation of execution times.

Operations	Native		Blue-Pilled		Monitored	
	Mean	Std	Mean	Std	Mean	Std
1. Allocate Small Block (MBytes/Sec)	3348.4	33.0	3322.8	26.21	3067.1	217.8874
2. Read Cached (Mbytes/Sec)	1351.5	2.44	1351.7	2.22	1350.6	2.9738
3. Read Uncached (Mbytes/Sec)	1290.5	3.03	1286.7	2.92	1283.9	21.3572
4. Write (MBytes/Sec)	1256.8	8.67	1240.4	21.55	1243.9	43.5518
5. Large RAM (Operations/Sec)	2212.7	11.71	2193.4	27.83	2242.9	52.1760

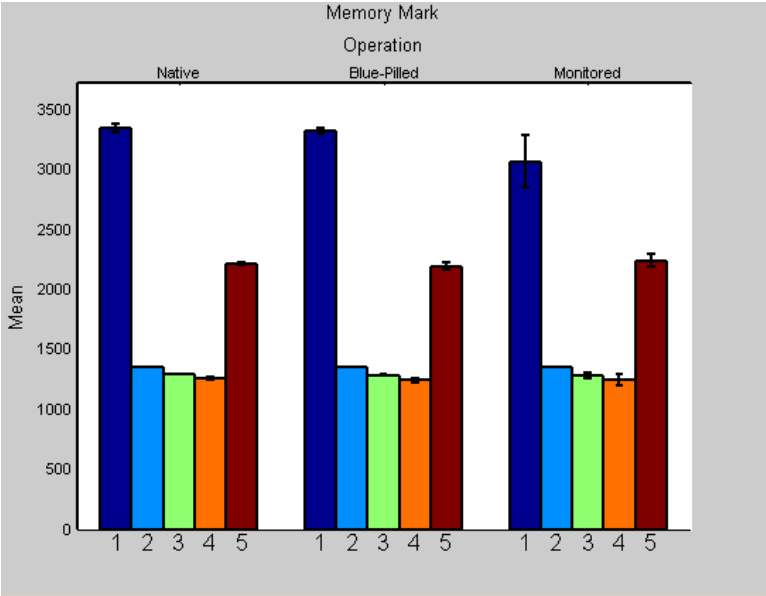

Figure 6: Statistics of Memory performance test for the Unmonitored, Blue-pilled and Monitored system. The bars show the mean value for the corresponding CPU operations in collaboration with their standard deviation.

6 Conclusion

In this paper, we proposed an on-the-fly installable system monitoring framework by extending the New Blue Pill HVM rootkit to meet the require-

ments of live systems that do not tolerate downtime. We also designed and implemented a novel system call tracing method that promises long-term compatibility with current 64-bit systems as well as allows for configurable granularity and transparency for catching system calls. In contrast to previous methods that mainly used page faults, our approach is based on system call invalidation which offers more acceptable performance. Moreover, we demonstrated how user-, and kernel level data structures could be extracted from a 64-bit Windows 7 system, which is essential to tighten the gap between the hypervisor and the guest OS. Our results can be used as building blocks for automated malware detection on live systems, for example, by extending the retrieved information via VMI to be suitable for tools such as AccessMiner [6].

As for future work, we are about to extend the capabilities of our tool to monitor full-kernel malware by dissecting native API calls. Another interesting research direction could be the automatic detection of nefarious malware behavior with host-only sensors. That would allow to identify code injection, data structure modification, and ROP based malware on live systems without monitoring the network behavior.

Acknowledgements

The work reported here was joint work with Gábor Pék, and the author is thankful for his contribution to this research.

References

[1] AMD. AMD64 Architecture Programmer's Manual, December 2011.

[2] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Mark Felegyhazi. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet 2012, 4(4), doi:10.3390/fi4040971*, pages 971–1003, 2012.

[3] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[4] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production

systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 417–426, September 2010.

[5] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[6] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. AccessMiner: Using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, New York, NY, USA, 2010. ACM.

[7] McAfee. Mcafee deepsafe. `http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx`.

[8] Microsoft. Windwos Driver Kit. `http://msdn.microsoft.com/en-us/windows/hardware/gg487428`, Last accessed, February 02, 2014.

[9] Offensive Computing. `http://www.offensivecomputing.net/`, Last accessed, March 26, 2012.

[10] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 2–2, Berkeley, CA, USA, 2009. USENIX Association.

[11] Passmark Software. Passmark Performance Test. `http://www.passmark.com/download/pt_download.htm`, Last accessed, February 12, 2014.

[12] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society.

[13] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[14] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, November 2011.

[15] Joanna Rutkowska. Subvertin Vista Kernel for Fun and Profit. Aug 2006.

[16] Joanna Rutkowska and Alexander Tereshkin. IsGameOver(), Anyone? 2007.

[17] Abhinav Srivastava and Jonathon T. Giffin. Automatic discovery of parasitic malware. In *RAID*, pages 97–117, 2010.