CloudImmu: Transparent Protection of Binary Applications in the Cloud

Xinyuan Wang, *Member, IEEE*

***Abstract*—As more organizations are moving their IT infrastructures from on-premises to the cloud, cloud security breaches have just surpassed on-premises breaches. There is a pressing need to develop practical and deployable cyber defense capabilities to protect the enormous amount of potentially vulnerable binary applications in the cloud from previously unseen cyberattacks.**

In this paper, we present *CloudImmu*, a practical cloud cyber defense system that is built upon a novel combination of binary rewriting and instrumentation techniques, virtual machine introspection and hypervisor level anomaly detection techniques. Our immunization tool has successfully "immunized" large real world binary applications such as bash, Snort, and our experiments with real world exploits have shown that CloudImmu can detect and block cyberattacks on properly immunized, otherwise vulnerable binary applications in virtual machines in real-time without using any prior knowledge of the attacks. Our benchmark experiments show that CloudImmu incurs less than 1.06% overall run-time performance overhead on typical applications with typical workloads.

***Index Terms*—Real-time intrusion detection and prevention, cloud securty, hypervisor based cyber defense.**

I. Introduction

Due to its advantages in cost, flexibility and scalability, cloud computing is becoming increasingly popular for businesses and government agencies. Now more and more organizations are moving their IT infrastructures from on-premise to the cloud. Garner Inc. has predicted that worldwide cloud market size will grow from $270 billion in 2020 to $332.3 billion in 2021, $397.5 billion in 2022 [1].

While cloud has brought many benefits to enterprises and organizations, it has also introduced new attack surfaces and security challenges. A recent study by IDC [2] has found that 79% of the 300 companies surveyed had experienced at least one cloud data breach, and nealy 43% had experienced 10 or more cloud data breaches in the past 18 months. Verizon's 2021 annual report on data breaches [3] has found out that 73% of the 79,000 cybersecurity incidents analyzed involve external clouds. It is the first time that cloud security breaches surpass on-premises breaches. Specifically, cloud hosting and software service provider Blackbaud was forced to pay a ransom to the attacker who had stolen data from the cloud and threatened to publish it online [4].

While existing cyber defense systems (e.g., firewall, authentication, IDS/IPS) are very useful, they have been shown to be inadequate in protecting mission critical systems in the cloud from increasingly stealthy and damaging cyberattacks. Specifically, currently deployed cloud security mechanisms lack the intrusion detection capabilities to protect vulnerable binary applications in the cloud from previously unseen application specific exploits in real-time. Our mission critical cloud systems can not be effectively secured before we can reliably detect and block previously unseen, application specific exploits in real-time.

In this paper, we persent *CloudImmu*, a practical cloud cyber defense system that is designed to provide transparent and real-time protection for potentially vulnerable binary applications in the cloud such that those vulnerable binary applications may be immune from various known or unknown cyberattacks. Specifically, CloudImmu is designed to detect and block previously unseen, application specific exploits on potentially vulnerable binary applications in the cloud in true real-time without any prior knowledge of the exploits.

In order to achieve such an objective, we first use *software immunization tool* to transparently "immunize" those binary applications into "immunized" binary applications such that each immunized binary application will 1) keep the original application semantics; and 2) tag each run-time invocation of system calls with some dynamically generated random mark. Since the dynamically generated random mark is unknown to any attacker, no attacker could invoke any system call with the correct mark. This allows CloudImmu to detect and block any illegal system calls from the attacker by simply checking if each invoked system call has the correct random mark at run-time. By using such a anomaly detection technique, CloudImmu is able to detect and block previously unknown cyberattacks without any prior knowledge (e.g., signature) of the cyberattacks.

When the immunized binary application properly tags each run-time invocation of system call with the correct random mark, then any system call without the correct random mark must be invoked by some foreign code (e.g., malware) rather than the proper immunized applications. In other words, any system call invoked with the wrong random mark must be some attack. Therefore, CloudImmu is able to detect and block previously unknown cyberattacks on properly immunized binary applications with no false positive while it can miss detecting certain cyberattacks.

We have successfully implemented a working prototype of CloudImmu in Linux KVM [5], have immunized large real world binary applications (e.g., Bash shell, Snort IDS/IPS), and have empirically validated CloudImmu's effectiveness with real world exploits. Our experiments show that CloudImmu is able to transparently detect and block cyberattacks on properly immunized, otherwise vulnerable binary applications in virtual machines in real-time without using any prior knowledge of the cyberattacks. Our SPEC CPU2006 macro benchmarking tests show that CloudImmu incurs less than 1.06% overall run-time performance overhead on typical applications under typical workloads.

Xinyuan Wang is with the Department of Computer Science, George Mason University, Fairfax, VA 22030, USA. email: xwangc@gmu.edu

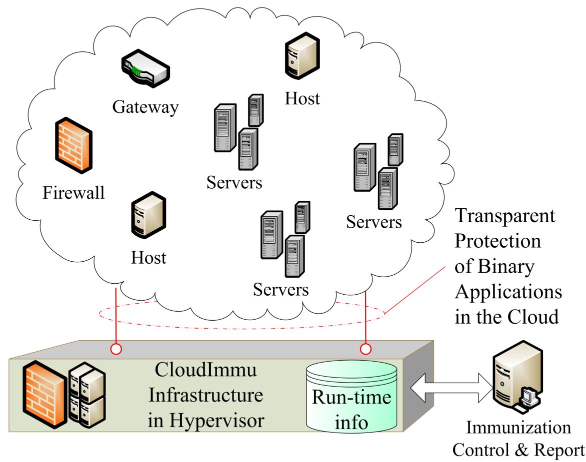

Fig. 1. CloudImmu Run-time Infrastructure inside Hypervisor

II. CloudImmu Functional Overview

A. Threat Model and Assumptions

We assume the Linux KVM hypervisor is trustworthy, and the to-be-protected binary applications in the cloud 1) are not self-modifying and contain no deliberate obfuscation; 2) can be either open source or commercial off-the-shelf (COTS) binaries without symbol information. While the binary applications could have known or unknown security vulnerabilities, we assume that the whole target system (e.g., binary applications, libraries, operating system) is benign in that there is no deliberate backdoor or malware built in the system. Supply chain attacks (e.g., SolarWind breach) that can implant backdoor or malware inside authentic software distribution is beyond the scope of this work.

We assume the adversary can reach the to-be-protected cloud applications via network, and he can exploit security vulnerabilities in some cloud applications, and inject, execute arbitrary code, read/write content from/to those memory locations allowed by the exploited cloud applications and the underlying operating system.

B. CloudImmu Overview

The CloudImmu system consists of offline binary immunization tool and CloudImmu infrastructure. Figure 1 illustrates the CloudImmu *run-time infrastructure* at the hypervisor level and how it interacts with various binary applications in various virtual machines. The hypervisor level infrastructure is the key component of CloudImmu, and it is responsible for 1) generating random mark for each new process/threat in each virtual machine; 2) keeping track of each run-time process/thread in each virtual machine via novel virtual machine introspection (VMI) techniques; 3) checking and handling each invocation of system calls by all monitored processes/threads in monitored virtual machines; and 4) protecting the immunized binary applications at run-time according to user specified configuration.

III. Immunizing Software Binaries

Given that Linux runs 90% of the public cloud workload [6], we choose to instrument and immunize the Linux binaries in the Executable and Linking Format (ELF) format.

ELF is a cross-platform, common standard binary file format for executable files, object files and shared libraries [7] that supports different endianness, different address sizes and different CPU instruction sets (e.g., X86, MIPS, ARM). Each ELF file starts with a mandatory ELF header followed by optional section header table, program header table, and sections. For the purpose of protecting Linux binary applications, we only need to immunize ELF executable file and ELF shared object file.

To immunize a given ELF binary application, we need to 1) keep the original application semantics; and 2) instrument the ELF binary application with our immunization code that tags the invocation of each system call with the random mark dynamically generated by the AppImmu infrastructure at run-time. Given an ELF binary, our software immunization tool automatically 1) creates a new text section and place our immunization code in the new text section; 2) instruments existing code to jump or call our immunization code in the new text section. Such a approach keeps the existing ELF code and data at their original locations as much as possible thus avoids complications caused by indirect function calls (e.g., `call *%eax`) where the address of the called function is determined at run-time.

We use two thread local storage (TLS) variables `c_mark` and `cover` to pass the random mark between the immunized binary applications and the CloudImmu infrastructure, where `c_mark` is the covered random mark and `cover` is the random cover. Since every thread has its own TLS, and the called shared library function (e.g., printf()) can access the TLS of the calling process/thread, our approach supports multiple threads.

A. Immunizing ELF Binary Executables

To immunize an ELF executable, we first instrument the entry point of the executable to call our immunization initialization code in the new text section, which triggers a special system call (e.g., system call #511) that is not used by the operating system running in the virtual machine:

```
push %eax
push %ebx
mov $0x1ff, %eax
mov <auth code>, %ebx
call *%gs:0x10
pop %ebx
pop %eax
```

Such a special system call notifies the CloudImmu infrastructure that some immunized process/threat is starting. The CloudImmu infrastructure intercepts the special system call and checks register `%ebx` for the virtual machine specific authentication code. If matches, it will dynamically generates a random mark X and random number r for the process/threat and secretly store $X \oplus r \oplus$`vm_rand` in TLS variable `c_mark`, r in TLS variable `cover` where `vm_rand` is a virtual machine specific random value. If the value of register `%ebx`

TABLE I
NUMBER OF SYSTEM CALL INVOCATIONS IN GNU C LIBRARIES

Library Name	No. of int $0x80	No. of sysenter
ld-2.12.2.so	56	0
libanl-2.12.2.so	6	0
libc-2.12.2.so	10	501
libpthread-2.12.2.so	10	176
librt-2.12.2.so	48	0

does not match the correct virtual machine specific authentication code, the calling process/thread is marked immunized without generating the random mark. This makes sure every subsequent system calls from unauthorized binary executables will be caught.

B. Immunizing ELF Libraries

The GNU C libraries in Linux use both the soft interrupt instruction `int $0x80` and the fast system call instruction `sysenter` to trigger system calls. As shown in the following snippet from libc-2.12.2, the GNU C library code moves the system call number to register %eax before triggering the `int $0x80` or `sysenter` instruction.

```
98099:   b8 be 00 00 00        mov    $0xbe,%eax
9809e:   cd 80                 int    $0x80

10f1ff:  b8 75 00 00 00        mov    $0x75,%eax
10f204:  65 ff 15 10 00 00 00  call   *%gs:0x10
```

Out of all GNU C shared libraries (.so) used by the 32-bit CentOS 6.4 Linux, we have found only 5 .so files have system call invocations as shown in Table I. For each system call invocation, our immunization simply replaces the 5-byte move instruction (e.g., `mov $0xbe,%eax`) with a 5-byte long unconditional jump instruction to a system call specific stub function in the new code section. The system call stub function recovers the process/thread specific random mark from `c_mark`⊕`cover`⊕`vm_rand` and tags the system call with the random mark via unused register space (e.g., %eax) and jumps back to the next instruction after the move instruction. Our immunization tool has successfully immunized these ELF binary libraries.

IV. REAL-TIME APPLICATION PROTECTION BY THE HYPERVISOR LEVEL CLOUDIMMU INFRASTRUCTURE

Linux KVM (Kernel based Virtual Machine) [5] is a proven open source hypervisor that can deliver near native performance by utilizing the hardware virtualization support (e.g., Intel VT, AMD-V) from modern CPUs. KVM has become part of standard Linux kernel since version 2.6.20 was released on February 2007. It is also the most deployed (90%) hypervisor for OpenStack – an open standard cloud computing platform for both public and private cloud computing. We choose to implement our CloudImmu infrastructure upon Linux KVM hypervisor running Intel CPU with Intel VT virtualization.

A. Intercepting System Calls from Virtual Machines

In order to check system calls invoked by any applications in the cloud, the CloudImmu infrastructure needs to intercept every system call in every virtual machine managed and monitored by the KVM hypervisor. For Intel and AMD CPUs, system calls can be triggered by three instructions: 1) the software interrupt instruction such as `int 0x80` for Linux; 2) the `sysenter` instruction used in 32-bit Intel and AMD CPUs; 3) the `syscall` instruction used in 64-bit Intel and AMD CPUs. Therefore, we need to have the KVM hypervisor to intercept all these instructions and cause VMexit for the virtual machine from which those system calls have been invoked. Unfortunately, neither Intel VT virtualization nor AMD-V virtualization provide native interception of sysenter, syscall instructions. In addition, Intel VT virtualization does not intercept any software interrupt. Therefore, we need to make the execution of these instructions (e.g., `int` n, sysenter, syscall) to cause some other exceptions that will trigger VMExit according to the exception-bitmap in the VMCS structure.

1) Intercepting Software Interrupt from Virtual Machines: To support the software interrupt instruction, Intel CPU maintains the interrupt descriptor table (IDT) which is an array of 256 8-byte descriptors in memory. Each descriptor contains the segment selector, privilege level and pointer to the interrupt handler in memory. When some software interrupt instruction (e.g., `int` n) is executed, the CPU will change to the specified privilege level and jump to the interrupt handler specified in the corresponding descriptor (e.g., descriptor #n).

In order to intercept the software interrupt from a virtual machine, we need to make the execution of the software interrupt causing the virtual machine VMExit so that our CloudImmu infrastructure inside KVM hypervisor can have chance to examine and process the software interrupt triggered before the guest operating system in the virtual machine process the software interrupt. Specifically, we need to make the execution of the software interrupt in the virtual machine cause some exception that Intel VT virtualization will catch.

Each descriptor in the IDT has a present bit that indicates whether the segment is present or not. If the present bit of a descriptor is not set, then any software interrupt to that descriptor will cause the *Segment Not Present* (#NP) exception. Fortunately the #NP exception is a catchable event by the Intel VT virtualization. To intercept software interrupt #n, we can simply clear the present bit of descriptor #n, and set the corresponding #NP bit in the exception-bitmap in the VMCS structure. Once our CloudImmu infrastructure catches the VMExit, it needs to emulate the hardware actions of the software interrupt before giving the control back to the virtual machine. Upon user's direction, our CloudImmu infrastructure inside the KVM hypervisor can set or clear the present bit of any specified IDT descriptor by directly overwriting the bit at the correct guest virtual address.

2) Intercepting `sysenter` Instruction from Virtual Machines: `sysenter` is a fast system call instruction that enables efficient transition from user code running at privilege level 3 to operating system kernel code running at privilege level 1. Before executing `sysenter` instruction, system software needs to specify the privilege level 0 code segment, the system call service routine address, and the privilege level 0 stack segment and stack pointer in the following MSRs (Model

Specific Registers):

- IA32_SYSENTER_CS
- IA32_SYSENTER_EIP
- IA32_SYSENTER_ESP

When any `sysenter` is executed in protected mode, Intel CPUs will generate the *General Protection* (#GP) exception if IA32_SYSENTER_CS[15:2]=0. To intercept the `sysenter` instruction, we can simply set the IA32_SYSENTER_CS MSR to be all zero, and set the corresponding #GP bit in the exception-bitmap in the VMCS structure. Once the CloudImmu infrastructure intercepts the `sysenter`, it needs to emulate the hardware actions in response to the `sysenter` instruction before giving the control back to the virtual machine to allow the guest operating system kernel to complete the processing of the fast system call triggered by the `sysenter` instruction.

To stop intercepting the `sysenter` instruction, we just need to restore the original value of the MSR IA32_SYSENTER_CS and clear the corresponding #GP bit in the exception-bitmap in the VMCS structure.

3) Intercepting `syscall` *Instruction from Virtual Machines:* `syscall` is another fast system call instruction that can only be executed in 64-bit mode. It invokes the kernel level system call handler by loading register RIP from the IA32_LSTAR MSR after saving the return address to register RCX. One necessary condition to execute the `syscall` instruction is that the System Call Extension (SCE) flag of the Extended Feature Enable Register (EFER) IA32_EFER.SCE must be set. If IA32_EFER.SCE is not set, any execution of the `syscall` instruction will generate the Invalid Opcode exception #UD, which is a catchable event by the Intel VT virtualization.

To intercept the fast system calls invoked by the `syscall` instruction inside the virtual machine, our CloudImmu can simply set IA32_EFER.SCE=0 of the chosen virtual machine, and set the corresponding #UD bit in the exception-bitmap in the VMCS structure. Once the ImmuCloud intrastructure catches the VMExit caused by the `syscall` instruction inside the virtual machine, it needs to emulate the hardware actions of the `syscall` instruction before resuming the virtual machine's execution to allow the guest operating system kernel to complete the processing of the fast system call triggered by the `syscall` instruction.

To stop intercepting the `syscall` instruction, we just need to set IA32_EFER.SCE=1 and clear the corresponding #UD bit in the exception-bitmap in the VMCS structure.

B. Bridging the Semantic Gap via Virtual Machine Introspection

Once the CloudImmu infrastructure has intercepted the VMExit triggered by some system call inside the virtual machine as described in section IV-A, it needs to figure out which process from what virtual machine invoked what system call (i.e., the system call number and other system call related parameters). Unlike the guest operating system kernel which sees and manages such process information directly, the KVM hypervisor does not directly see such high-level semantic rich

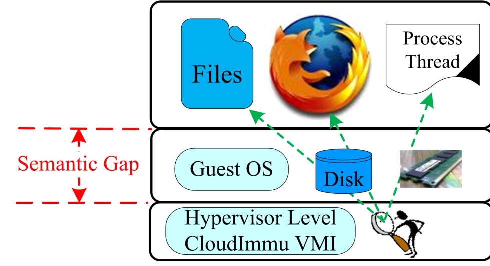

Fig. 2. Bridging the Semantic Gap via CloudImmu Run-time Virtual Machine Introspection

information but only low-level information such as exceptions, instructions, registers, guest physical memory. Therefore, there is a semantic gap between what the hypervisor and the guest operating system kernel can see directly.

As shown in Figure 2, the CloudImmu infrastructure uses virtual machine introspection (VMI) techniques [8], [9] to bridge the semantic gap via reconstructing the guest semantic view from what the hypervisor can see. Since the guest operating system kernel contains all the data structures (e.g., `struct task_struct`) in the guest memory, we can reconstruct the guest semantic view once we have found the proper data structures in the guest memory.

Note, the location and offset of guest data structures depend on the operation system running in the virtual machine. In X86 Linux that uses 8KB kernel stack, the current `thread_info` location can be conveniently obtained by masking the 13 least significant bits of the stack pointer. By reading the content of the current `thread_info`, we can obtain the guest virtual address of the current `task_struct` from field `task_struct *task`, and the process ID and process name which are part of the current `task_struct`. The system call number is always in the eax or rax register.

Based on effective VMI, the CloudImmu infrastructure is able to figure out what process (e.g., process ID, process name) from what virtual machine has triggered what system call (e.g., system call number) once it catches the VMExit due to some system call inside some virtual machine monitored.

C. Managing Virtual Machines and the Guest Processes

To protect selected processes of selected virtual machines in the cloud, the CloudImmu infrastructure needs to maintain per-VM (e.g., whether it is being monitored, protected) and per-process (e.g., the dynamically generated random mark, immunization status) state information outside all virtual machines, and check the maintained per-VM and per-process state information for every intercepted system call from every monitored virtual machine. Given the high frequency (e.g., hundreds of times per second) of system calls inside every virtual machine, we need to have a highly efficient access of the maintained per-VM and per-process state information.

We have used hash tables to keep track of all virtual machines and all processes inside each virtual machine. We use the unique creator process ID of each virtual machine as the VM hash table key. By definition, the process ID within each virtual machine is unique. Thus we can use the process

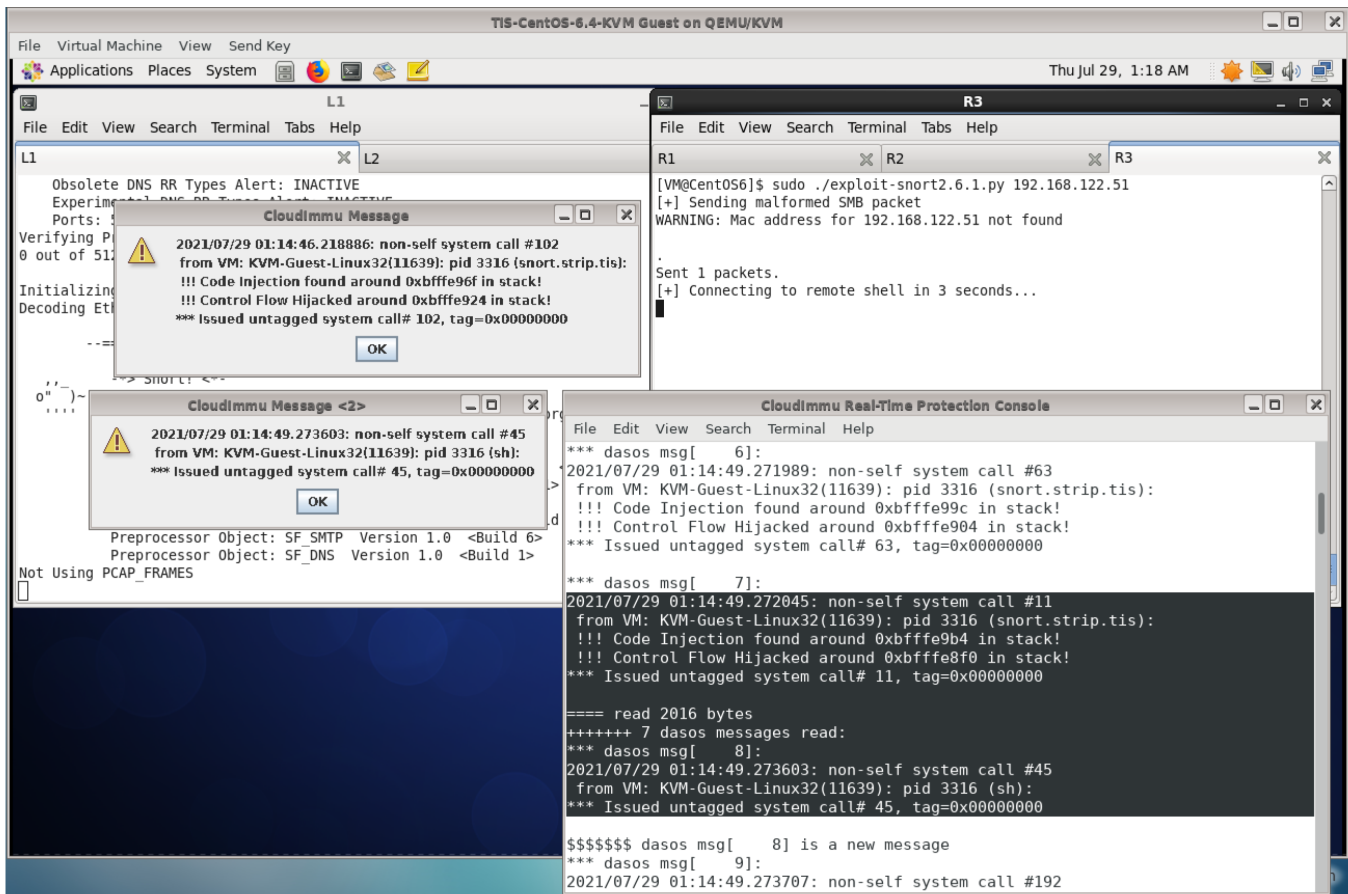

Fig. 3. CloudImmu Run-Time Detection of Exploit of Vulnerable Snort 2.6.1 inside Linux Virtual Machine

ID as the key of the process hash table specific to some virtual machine.

When the cloudImmu infrastructure starts to monitor some virtual machine, it does not have any per-process state information of any process running inside the virtual machine. It will, however, collect the information about each process when it intercepts some system call invoked by the process. Furthermore, the CloudImmu infrastructure intercepts any process termination system call (e.g., `exit`) so that it can remove the terminated processes from its process hash table.

V. Empirical Evaluations

We have implemented a working prototype of CloudImmu infrastructure upon KVM [5] in the standard 64-bit Linux kernel 4.4.50, and the generic ELF binary immunization tool, and we have used real world applications and real world exploits, the SPEC CPU 2006 benchmarks to evaluate the effectiveness and efficiency of our CloudImmu prototype.

A. Real-Time Detection and Blocking of Exploits of Vulnerable Binary Applications inside KVM Virtual Machine

To validate the CloudImmu's capability in detecting and blocking previously unknown exploit on vulnerable binary applications in virtual machines, we have used the DCE/RPC Preprocessor Remote Buffer Overflow exploit of Snort 2.6.1 – the top free network intrusion detection system [10].

We have immunized the binary executable of Snort 2.6.1 have set the CloudImmu to detection mode only. Figure 3 shows that the CloudImmu is able to detect the exploit on Snort 2.6.1 in real-time without using any prior knowledge of the exploit. Specifically, the CloudImmu has detected the non-self first system call (#112 that does not have the expected tag) invoked by process `snort.strip.tis` of pid 3316 from KVM virtual machine: KVM-Guest-Linux32. It also has captured all the rest non-self system calls invoked by the exploited snort process. As shown in the highlighted area of the CloudImmu Real-Time Protection Console, after invoking system call #11, the exploit has successfully turned the process 3316 from snort into a shell (sh). When the CloudImmu is set to blocking mode, it will stop the exploit upon detecting the first non-self system call that does not have the expected tag in real-time.

B. Run-Time Performance Overhead of CloudImmu

To understand how much run-time performance overhead CloudImmu would introduce while it actively intercepts and checks system calls from specified processes of specified virtual machines, we have used the CPU2006 macro benchmarks to measure the performance of typical applications under typical workloads in the following two scenarios on a computer with a quad-core 2.3 GHz Intel Core i7 CPU, 8GB RAM: 1) Host running the original 4.4.50 kernel, KVM

TABLE II
CLOUDIMMU RUN-TIME OVERHEAD BY SPEC CPU 2006 BENCHMARKS

SPEC CPU 2006	CloudImmu On	Baseline	Overhead
400.perlbench	22.10	22.50	1.7778%
401.bzip2	14.80	14.60	-1.3699%
403.gcc	20.30	22.00	7.7273%
429.mcf	15.90	15.60	-1.9231%
445.gobmk	18.60	18.70	0.5348%
456.hmmer	17.20	17.20	0.0000%
458.sjeng	17.90	17.80	-0.5618%
462.libquantum	34.80	35.40	1.6949%
464.h264ref	29.10	29.10	0.0000%
471.omnetpp	13.10	13.30	1.5038%
473.astar	11.60	11.70	0.8547%
483.xalancbmk	20.40	20.40	0.0000%
SPECint(R)_base2006	**18.70**	**18.90**	**1.0582%**

VM running the original 2.6.32.20 kernel with the original libraries; 2) Host running the CloudImmu 4.4.50 kernel with active CloudImmu checking turned on, KVM VM running the original 2.6.32.30 kernel with the immunized libraries.

Table II shows and compares the the SPEC CPU 2006 measurements of the baseline and the CloudImmu active checking. The 403.gcc benchmark has the highest overhead of 7.7273% and all other benchmarks have very small overhead. The overall SPEC CPU 2006 integer benchmark overhead is 1.0582% – indicating that CloudImmu infrastructure has negligible performance impact to typical applications.

C. Security Analysis

The security of CloudImmu is built upon the secrecy of the random mark that tags each legitimate system call at the run-time. CloudImmu uses multiple levels of obfuscation to protect the secrecy of the dynamically generated random mark. First, when the immunized binary software starts, it triggers the special system call with a VM-specific and application-specific authentication code (as shown in section III-A). Such a VM-specific and application-specific authentication code is randomly generated and statically encoded to the specified binary application when it is immunized for the specified VM. Therefore, a binary application immunized for a specific VM can not be used in other VM.

Second, the CloudImmu infrastructure in the KVM hypervisor does not reveal the dynamically generated random mark X to the properly immunized binary application but secretly store the "covered" version of X ($X \oplus r \oplus \texttt{vm_rand}$) and the random cover r in secret TLS locations `c_mark` and `cover` respectively in the virtual machine. Here r is randomly generated for each execution of any properly immunized program, and `vm_rand` is a virtual machine specific random value that prevents the adversary from obtaining the random mark X even if he has somehow obtained values of `c_mark` and `cover`.

Finally, the TLS locations for `c_mark` and `cover` are randomly chosen for each VM. To increase the randomness, we can divide the multi-byte `c_mark` and `cover` into multiple parts of single byte (e.g., $\texttt{c_mark}_1$, ... $\texttt{c_mark}_4$, $\texttt{cover}_1$, ... $\texttt{cover}_4$) each of which can be allocated randomly. This would force the adversary to figure out the exact location and the exact order of those 8 parts before he could read the value of `c_mark` and `cover`. As allocating 8 bytes from a 20-byte TLS storage has $\frac{20!}{12!} = 5,079,110,400$ permutations, the chance for the adversary to happen to figure out the values of `c_mark` and `cover` is extremely low ($\approx 2 \times 10^{-10}$).

VI. RELATED WORKS

System call based defense Ever since Forrest et al. [11] first proposed using short sequences of system calls to build intrusion anomaly detection, a number of followup works [12], [13], [14] have improved the efficiency and effectiveness of the system call sequence based anomaly detection. Based on system call tagging rather than sequence of system call, Wang et al. [15] proposed the software immunization and anomaly detection approach that has eliminated run-time training required by previous system call sequence based approaches [11], [12], [13], [14]. However, their approach can not immunize binary applications in that it requires the access of the source code of the software to be protected.

Control flow integrity based defense Control flow integrity (CFI) [16] aims to ensure the indirect control flow transfers can only reach legitimate destinations in the control flow graph (CFG) of the software at run-time. A large number of CFI based approaches [17], [18], [19], [20], [21], [22], [23], [24], [25] have been proposed to prevent, detect control flow hijacking and code-reuse attacks under various assumptions and countermeasures [26], [27], [28], [29].

Unlike CFI based cyber defense approaches, CloudImmu does not seek to ensure every indirect control flow transfer is within the CFG, but focuses on detecting and blocking illegal system calls based on dynamically assigned secret mark. This not only makes CloudImmu more efficient at run-time, but also enables it to catch those illegal control flow transfers within the CFG (e.g., control-flow bending [30]) that will be missed by existing CFI approaches.

Hypervisor based defense Hypervisor has been used to build various cyber defense capabilities such as 1) monitoring and collecting activities inside the guest operating system [8]; 2) detecting rootkits in the guest operating system [9], [31]; 3) protecting the guest memory data and human-machine interaction data from untrusted operating system kernels in the virtual machine [32]. CloudImmu differs from existing hypervisor based cyber defense in that it can protect immunized binary applications inside the virtual machines from many cyberattacks (e.g., control flow hijacking attack) effectively and efficiently in real-time.

VII. CONCLUSIONS

In this paper, we have presented the CloudImmu cyber defense system that can transparently immunize and protect ELF binary applications in the cloud. Our experiments with real world binary applications and real world exploits demonstrate that (1) CloudImmu is able to detect and block previously unknown cyberattacks in real-time; (2) the combination of binary rewriting, virtual machine introspection and hypervisor level anomaly detection techniques is a promising and practical approach to protect vulnerable real world binary

applications in the cloud. SPEC CPU 2006 benchmarks show that CloudImmu incurs less than 1.06% overall performance overhead to typical applications under typical workloads.

ACKNOWLEDGMENT

This work was supported in part by the U.S. Army STTR grants W56KGU-16-C-0064, W56KGU-17-C-0077.

REFERENCES

[1] Gartner, "Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 23% in 2021," Apirl 2021, https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021.

[2] Ermetic, "Nearly 80% of Companies Experienced a Cloud Data Breach in Past 18 Months," June 2020, https://www.securitymagazine.com/articles/92533-nearly-80-of-companies-experienced-a-cloud-data-breach-in-past-18-months.

[3] M. Korolov, "Report: Cloud Security Breaches Surpass On-Prem Ones for the First Time," May 2021, https://www.datacenterknowledge.com/security/report-cloud-security-breaches-surpass-prem-ones-first-time.

[4] C. Cimpanu, "Cloud provider stopped ransomware attack but had to pay ransom demand anyway," July 2020, https://www.zdnet.com/article/cloud-provider-stopped-ransomware-attack-but-had-to-pay-ransom-demand-anyway/.

[5] "Linux Kernel based Virtual Machine," www.linux-kvm.org.

[6] T. Nuggets, "Why Linux runs 90 percent of the public cloud workload," August 2018, https://www.cbtnuggets.com/blog/certifications/open-source/why-linux-runs-90-percent-of-the-public-cloud-workload.

[7] T. Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification," May 1995, https://refspecs.linuxbase.org/elf/elf.pdf.

[8] X. Jiang and X. Wang, "'Out-of-the-box' Monitoring of VM-based High-Interaction Honeypots," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007)*, September 2007.

[9] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-Based 'Out-of-the-Box' Semantic View Reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, November 2007.

[10] U. Team, "Top 6 Free Network Intrusion Detection Systems (NIDS) Software in 2020," August 2020, https://www.upguard.com/blog/top-free-network-based-intrusion-detection-systems-ids-for-the-enterprise.

[11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy (S&P 1996)*. IEEE, May 1996.

[12] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P 1999)*. IEEE, May 1999, pp. 133–145.

[13] R. Sekar, M. Bendre, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P 2001)*. IEEE, May 2001.

[14] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P 2003)*. IEEE, May 2003.

[15] X. Wang and X. Jiang, "Artificial Malware Immunization based on Dynamically Assigned Sense of Self," in *Proceedings of the 13th Information Security Conference (ISC 2010)*, October 2010.

[16] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity: Principles, Implementations, and Applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*. ACM, November 2005, pp. 340–353.

[17] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. A. Mccamant, D. Song, and W. Zou, "Practical Control Flow Integrity & Randomization for Binary Executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P 2013)*. IEEE, May 2013.

[18] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22nd USENIX Security Symposium*. USENIX, August 2013.

[19] R. Gawlik and T. Holz, "Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*, 2014, pp. 396–405.

[20] M. Zhang, Q. Qiao, N. Hasabnis, and R. Sekar, "A Platform for Secure Static Binary Instrumentation," in *Proceedings of the 10th ACM International Conference on Virtual Execution Environments (VEE 20014)*, July 2014, pp. 129–140.

[21] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries," in *Proceedings of the 22th Network and Distributed System Security Symposium (NDSS 2015)*, February 2015.

[22] C. Zhang, K. Z. C. Chengyu Song, Z. Chen, and D. Song, "VTint: Protecting Virtual Function Tables? Integrity," in *Proceedings of the 22th Network and Distributed System Security Symposium (NDSS 2015)*, February 2015.

[23] M. Payer, A. Barresi, and T. R. Gross, "Fine-Grained Control-Flow Integrity Through Binary Hardening," in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2015)*, July 2015, pp. 143–163.

[24] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque Control-Flow Integrity," in *Proceedings of the 22th Network and Distributed System Security Symposium (NDSS 2015)*, February 2015.

[25] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P 2016)*. IEEE, May 2016.

[26] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE, May 2014.

[27] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P 2015)*. IEEE, May 2015.

[28] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*. ACM, October 2015, pp. 901–913.

[29] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*. ACM, October 2015, pp. 243–255.

[30] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium*. USENIX, August 2015.

[31] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor Support for Identifying Covertly Executing Binaries," in *Proceedings of the 17th USENIX Security Symposium*. USENIX, August 2008, pp. 243 – 258.

[32] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi, "AppSec: A Safe Execution Environment for Security Sensitive Applicationst," in *Proceedings of the 11th ACM International Conference on Virtual Execution Environments (VEE 20015)*, March 2015, pp. 187–199.