AppGuard: A Hardware Virtualization Based Approach on Protecting User Applications from Untrusted Commodity Operating System

Zili Zha*, Min Li*, Wanyu Zang*, Meng Yu* and Songqing Chen†
*Department of Computer Science, Virginia Commonwealth University
Email: {zhaz, lim4, wzang, myu}@vcu.edu
†Department of Computer Science, George Mason University
Email: sqchen@gmu.edu

***Abstract*---The security of user applications largely relies on the proper execution of the underlying operating system. However, existing commodity OSes are inevitably vulnerable due to their enormous code base containing a whole bunch of bugs that can be easily exploited by attackers. In such situations, a proper way of protecting users' data privacy and integrity at runtime is a paramount task that needs efficient solutions. While quite some efforts, such as Overshadow, SP3, InkTag, and AppShield, have been made to deal with this problem, existing solutions either induce non-trivial performance overhead, or demand modifications to the OS, applications, or the underlying hardware architecture.**

In this paper, we present AppGuard that can efficiently and feasibly protect user applications even on a compromised OS. AppGuard utilizes the hardware virtualization extensions to achieve such a goal. Compared to the existing solutions, AppGuard does not require any modifications to the application or the OS. Our evaluation results demonstrate that AppGuard can provide effective protection to user applications with much lower performance overhead.

I. Introduction

Commodity operating systems are always vulnerable to various attacks due to their large code base. This has continuously raised concerns on the user applications's security and privacy running on these OSes. The problem is getting worse today and protecting user applications from untrusted OS has become one of the most crucial tasks.

Quite some efforts have attempted to provide secure protection for user applications [1], [2], [3], [4]. In Overshadow [1] and SP3 [3], the hypervisor presents an encrypted view of the application's memory to the OS, thus preventing the OS from accessing the plaintext of the user memory. However, complex encryption and decryption operations dramatically degrade the performance of the whole system. InkTag [2] not only protects the application data but also verifies the OS behaviors through paraverification technique but it requires to modify the the kernel, making this approach impractical for commodity OSes. Proxos [4] introduces a dedicated trusted virtual machine for the protected applications. However it dramatically increases the TCB size and therefore weakens its security strengths and XOM OS requires hardware modifications which appear impractical for commodity platforms.

In short, these existing solutions fail in one or several of the following aspects: (i) requring changes on either the applications, the operating systems or the hardware architectural support, (ii) heavy computation overhead on frequent encryption and decryption operations, (iii) only checking system call from possible malicious access from untrusted OSes while overlooking the dynamic library which may contain malicious code.

To best address these concerns, we propose a novel architecture AppGuard, we design and implement AppGuard to protect user applications running on untrusted OS. AppGuard does not require any modifications to the application or the underlying OS. It also significantly reduces the performance overhead by utilizing the hardware virtualization support (e.g., Intel VT-x and AMD-V provided by the Intel and AMD). AppGuard protects applications context and address space from the OS kernel and other applications by interposing on the interactions between the user applications and guest kernel, such as faults, interrupts, and systems calls. Moreover, in AppGuard, encryption and decryption operations are only needed to perform on I/O data to ensure the confidentiality and integrity of users' data. To evaluate the performance of AppGuard, we implement a prototype and compare it against existing solutions. The results show that AppGuard provides more protection with much less performance overhead. While details are presented later in the paper, the highlights of AppGuard includes the following:

- AppGuard is easily portable since it does not require any changes on the applications or OS. Thus it preserves the same OS interface to retain backward compatibility for existing applications.
- In AppGuard, the encryption and decryption operations are only performed on I/O data to ensure the confidentiality and integrity of users' data. The minimum usage of such operations dramatically reduces the overhead.
- AppGuard checks both system call and library to prevent malicious access from untrusted OSes.

The rest of this paper is organized as follows. In Section II, we discuss related work on protecting user applications from untrusted OSes. Section III presents the threat model along with a general description of our overall architecture. Section IV devotes to implementation details of AppGuard, consisting of application memory protection, event interception mechanisms, I/O data protection and so on. Some evaluation results are presented in Section V. Section VI discusses the future work and concludes this paper.

II. Related Work

In this section, we discuss in more details on the existing approaches related to application protection in untrusted computing environments.

Hypervisor based protection. A lot of previous efforts protecting the secrecy and integrity of user's application against untrusted operating system take a VMM-based approach. The most related work to AppGuard includes Overshadow [1], SP3 [3], InkTag [2], AppShield [5].

In Overshadow [1] and SP3 [3], to protect the user applications' security and privacy, the hypervisor presents an encrypted view of the applications' memory to the OS. But encryption and decryption are time consuming, which dramatically degrades the performance of the entire system. AppGuard does not suffer from this drawback by utilizing EPT (Extend Page Table) provided by hardware virtualization extensions. In addition, marshalling code at the user level in Overshadow could be exploited by the attacker. In AppGuard, all code is inside the hypervisor, which provides more strict protection. InkTag [2] verifies the OS behaviors through a new paraverification technique. It effectively secures the application from Iago attacks. However, it requires kernel modifications which is not practical to commodity operating systems. Unlike InkTag, AppGuard requires no changes to the OS or applications.

AppShield [5], the most similar work to ours, claims that it does not need any cryptographic operation by EPT reconfiguration. However, it does not explain how OS performs disk I/O operations, such as memory page swap in/out, without accessing the application. Just relying on EPT configuration, OS will crush if it cannot access the application. Moveover, Appshield requires a trusted component in the guest kernel. AppGuard differs from Appshield in (1) although both AppGuard and Appshield use EPT configurations to isolate the application from the OS, AppGuard allows the OS to access the encrypted applications' memory page in order to support normal disk I/O operations; (2) AppGuard does not require any modifications to the application or the OS, while Appshield needs a trusted component in the guest kernel; 3) AppGuard directly traps the events that need to be intercepted without going through the guest kernel as in AppShield, thus not relying on any trusted components of the guest OS. This makes AppGuard backward compatiable.

Secure processor based protection. A secure process is a dedicated process with hardware implementation for carrying out cryptographic operations. It can directly protect individual process bypassing the whole or most of OS. XOM OS [6] provides compartments to isolate one application from the others. However, XOM OS requires hardware modifications and heavy compiler/assembler support that appears impractical for commodity platforms. SecureME [7] and Bastion [8] both use secure processors to deal with untrusted OS and hardware. Bastion is a new hardware-software architecture for protecting security-critical software modules. It is composed of enhanced microprocessor hardware and enhanced hypervisor software. SecureME is also a hardware-software approach to provide protection on application code and data. It requires modifications to the OS and applications. AppGuard is different from above as it does not require any modification to hardware, OS or application.

Some other related work, e.g., Flicker [9], TrustVisor [10], and Memoir [11] protects a small piece of code and data rather than a whole application or device drivers that requires OS functionality. The negative impact of TCB size on system security has been widely recognized.

AppGuard differs from these existing work because AppGuard does not require any modifications to the application or the guest OS. Furthermore, AppGuard not only protects the user application by interposing on the control transfer between the user process and the guest kernel upon system call invocation and interrupt handling, but also considers the potential privacy leakage during shared library calls. Finally, AppGuard takes a cryptographic approach only in system call based I/O and paging related I/O to ensure both the privacy and the integrity of the data, and the performance.

III. Overview

A. Threat Model

In our work, the OS is assumed to be untrustworthy since it is vulnerable to attacks due to its large code base, which consists of not only the kernel but also device drivers and system services. The large body of code exposes broad attack surfaces to attackers. The sensitive data of users' application will be easily tampered with once any part of the OS gets compromised. We don't consider the protection of its availability in this paper.

Compared to a commodity OS, the underlying hypervisor normally has a much smaller and simpler code base. In our work, we assume that the hypervisor is trustworthy and its integrity can be ensured by utilizing TPM (Trusted Platform Module) from the Trusted Computing Group (TCG). Besides, System Management Mode based attacks are out of the scope of our consideration.

We also assume that the underlying hardware, i.e, CPU, cache, memory are all trustworthy except for peripheral I/O devices since they are vulnerable to exposure. To prevent the privacy leakage from I/O devices, we adopt crypto to ensure the privacy and integrity of all I/O data.

B. Design Goal

The aim of AppGuard is to provide a secure execution environment which protects the integrity and data privacy against a compromised OS for critical user applications. This goal is the same as some previous work, such as Overshadow and AppShield..

Address space isolation. To fully protect the privacy and integrity for code and data in the application's address space, we must impose strict isolation between the address space of the application and the guest kernel. This is achieved by utilizing hardware assisted virtualization support for memory virtualization instead of repetitively encrypting and decrypting the pages.. Any access to the application address space from the guest kernel will be intercepted and handled by AppGuard.

Control flow integrity. Even though the address space of applications is strictly isolated, an application still needs to interact with the OS for system call handling, fault and interrupt handling. To ensure control flow integrity, AppGuard interposes on every context switch to prevent the application from being tampered with. The kernel cannot make any changes of the application code, thus, the control flow integrity of the application will be assured.

Disk I/O and paging related memory protection. Different from memory isolation, disk I/O data and swap pages need to be encrypted since the disk is shared among processes and vulnerable to attacks. This is achieved by encrypting the user pages upon EPT violation when the kernel attempts to access the application's memory.

Unmodified application and OS. In this paper, we aim to provide a novel system that doesn't require any modification to the user application or the OS, which makes the key difference from the previous work. This novelty simplifies the system to the largest extent but still manages to enforce full protection of all the security sensitive applications.

C. *Overall Architecture*

Figure 1 shows the overall architecture of AppGuard. It consists of an untrusted OS, user application, a bare-metal hypervisor and a shared buffer in the user space.

To avoid any change to the application or OS, we modify the hypervisor by adding three components for the following functions: (a) *Context protection*: protect the application context during the switch between the user and the kernel mode; (b) *Parameter marshalling*: assemble the arguments that need to be passed to the kernel. (c) *EPT reconfiguration*: configure EPT entries in order to isolate the address space between the application and the guest kernel. Therefore, the application address space will be inaccessible to the kernel. The shared buffer, which is also accessible to OS, stores all the marshalled arguments that need to be passed to the kernel for normal event handling. AppGuard follows the following four steps to handle the interrupt and system call.

1) Any flow control from the user application to the untrusted kernel traps to the hypervisor due to different interception mechanism based on what type of event is occuring, e. g, system call, interrupt, shared library function invocation.
2) The hypervisor saves the user context, marshals the required parameters, marks the corresponding EPT entries of the user process as inaccessible and injects the interrupt into the kernel or transfers control to the shared library.
3) After the guest OS handles the forwarded system call or interrupt, it transfers controls to the untrusted code block that again issues another hypercall into the hypervisor or triggers an EPT violation after execution of the shared library function.

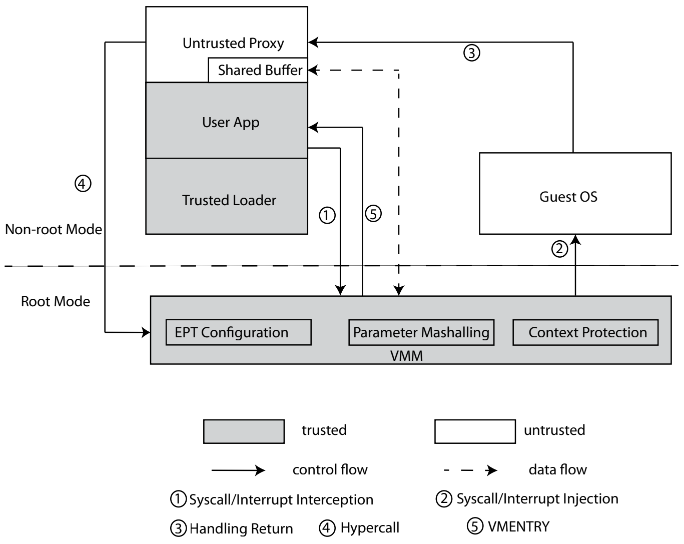

Fig. 1: The architecture of AppGuard for handling system calls/interrupts.

4) The hypervisor restores the original contexts of the interrupted process and returns control to the user application.

At first glance, the architecture in Figure 1 looks similar to that of AppShield. However, AppGuard differs from AppShield significantly. The largest difference lies in the fact that the system call or interrupt is intercepted directly by the hypervisor. Importantly, AppGuard does not require any modifications to the commodity OS which makes it easily portable.

IV. Implementation

In this section, we describe the detailed design of AppGuard. We mainly focus on how it isolates the address space of user applications from the OS, and the interception mechanisms for system calls, interrupts and library function calls in order to protect the privacy and integrity of users data.

A. *Application memory protection*

Figure 2 depicts the basic state transition diagram for maintaining the secrecy and integrity of a single protected memory page. In the figure, RA/WA represents read/write operation on the accessible plaintext, and RK/WK represents read/write operation on encrypted data. The detailed procedures of the state transition are as follows:

- Transition 1: When a system call is invoked or an interrupt occurs, the hypervisor intercepts the event and marks the EPT entries of the corresponding application as inaccessible and forwards the events to the kernel.
- Transition 2: On return to the user process, the hypervisor restores the bits in the EPT entries to their original state and the application can perform read/write (RA/WA) operations.
- Transition 3: When the kernel tries to read or write the application pages for disk I/O, the hypervisor intercepts this operation based on EPT violation, encrypts the I/O data, sets those entries as inaccessible.

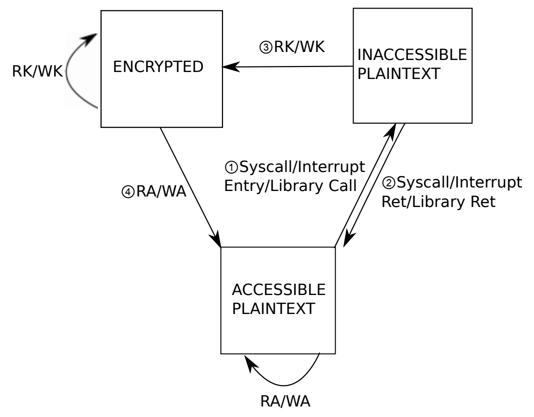

Fig. 2: State transition diagram for maintaining the secrecy and integrity of single memory page of user application.

- Transition 4: When the encrypted page is subsequently accessed by the application, the hypervisor decrypts it and sets the corresponding EPT entry as accessible.

B. System Call Handling

The general procedure for handling a system call consists of several steps as described below:

1) Any system call, interrupt or shared library function call causes VMEXIT and is intercepted by the hypervisor.
2) The hypervisor will (1) marshal the parameters required by the system call handler into the shared buffer; (2) save the contexts of the trapped process and prepare dummy context in VMCS; (3) set the EPT entries corresponding to the guest memory as inaccessible except for the shared buffer and inject the event into the guest kernel. The shared buffer is used for communication between the application and the kernel.
3) Return control to the untrusted code block in the user address space which again issues a hypercall into the hypervisor.
4) The hypervisor restores the original EPT configuration and the original context (GP registers, flag registers, control registers).

C. Library call handling

Another potential security issue is from the shared libraries that are linked to the running program at runtime, which may contain malicious code intending to tamper with users' privacy [12]. Some of the library functions are implemented by invoking the system calls provided by the kernel. Therefore, to prevent them from accessing the sensitive information of the application, we need to intercept the user space library calls in the hypervisor before the system calls are invoked. The difference from the system call handling is that we don't need to re-exit for system calls embedded inside library functions. For performance, we need to design a mechanism that prevents re-entry for system calls in library functions. The detailed mechanism for intercepting and handling shared library function calls will be discussed in our future work.

D. Parameters Mashalling

One critical component of AppGuard is parameters marshalling that exports the data needed by the system call routine into a shared buffer. The arguments inside the shared buffer will be used to maintain the normal execution of the system call.

On VMEXIT, in order to prevent the kernel from accessing registers since they may contain sensitive application data, the hypervisor saves these registers in a private section in its own address space and creates some dummy context. However, there are still some system calls that place their parameters in some of these general purpose registers. To ensure the correct execution of the system call service, AppGuard keeps those parameters in the corresponding registers.

E. I/O protection through encryption

As we assume that the disk and the OS are not trustworthy, we use a cryptographic approach which supports system call based I/O and paging related I/O to ensure both the privacy and the integrity of the data.

Disk I/O protection: As mentioned earlier, all system calls, including system call based I/O, such as read() and write(), are intercepted by the hypervisor. It will encrypt the I/O data that needs to be passed to the untrusted kernel.

Paging related memory protection: For paging related I/O, since before VMENTRY, the EPT entries corresponding to the target application have already been marked as inaccessible to the guest kernel, it triggers an EPT violation when the guest tries to access the user's memory. The hypervisor encrypts the required memory and marks those entries as accessible and returns control to the guest VM and resumes the operation. For instance, if a page is to be swapped out, the hypervisor intercepts this operation based on the EPT violation, encrypts the page and swaps it out. When the page is swapped in from the disk later on, the hypervisor decrypts this page and hands it over to the application.

Overshadow [1] encrypts/decrypts the application memory at the occurrence of every context switch between the user mode and the kernel mode. This dramatically degrades the performance of the entire system due to the costly encryption/decryption operations. However, AppGuard just needs to encrypt/decrypt the pages for I/O operations. For normal interrupts or system calls, there is no need to encrypt these pages since we already set these EPT entries as inaccessible previously.

V. Evaluation

A. Comparison with Existing Approaches

Our design differs from other existing approaches in a number of different ways, which are summerized in Table I. Y means that encryption is needed for that particular event while N means that encryption/decryption is not needed during the process. From this table, we can find that Overshadow requires encryption or decryption operations at every context switch between the application and the guest OS while AppGuard only needs to reconfigure the corresponding EPT entries utilizing hardware virtualization extensions. Encryption/decryption

TABLE I: Comparison of different approaches.

Events	AppGuard	AppShield	Overshadow	InkTag
Interrupt	N	N	Y	N
Exception	N	N	Y	N
System Call	N	N	Y	N
I/O Protection	Y	N	Y	Y
OS modification	N	Y	N	Y

operations are only needed for I/O data since the external disk is considered to be untrusted which dramatically reduces the performance overhead compared to Overshadow. Hence, AppGuard outperforms Overshadow in terms of time overhead, which is especially important for time critical applications. As we have mentioned, AppShield is the most similar work to AppGuard. However, it requires a trusted component called transit module to be inserted into the guest OS. To some extent, this contradicts with its threat model that assumes the guest OS is totally untrusted. In our assumption, we assume that every component of the guest OS is considered untrusted. Furthermore, AppGuard doesn't consider the threat of shared library calls that may contain malicious code while AppGuard effectively solves this problem.

To evaluate the AppGuard performance, we focus on measuring the overhead of the most time consuming part of our design, which is the encryption and decryption operations on disk I/O data. Here, the first step we take is to calculate the time spent on encrypting or decrypting a single memory page. The algorithm that we use to encrypt the data is AES-256 and it gives an approximate time of 0.018s for a single page. As the second step, we collect some statistical data that indicates the number of page faults during the execution of a common user application. Table II shows us the approximate amount of page in and page out operations for "make". Each row represents the average pagein/pageout/pagefault per second for a period of 10 minutes.

TABLE II: Statistics of memory performance during Make.

index	pgpgin/s	pgpgout/s	fault/s
1	0.53	1126.84	55261.73
2	0.00	1113.98	54656.57
3	0.00	1117.48	54539.20
4	0.00	1120.52	54585.74
5	0.00	1119.51	55062.79
6	0.00	1122.01	54855.17
7	373.20	1227.23	55372.35
8	0.01	1140.76	56154.34
9	0.01	1134.49	56647.55
10	0.01	1133.95	56578.40

In this table, pgpgin/s represents the total number of kilobytes the system paged in from the disk per second while pgpgout/s represents the total number of kilobytes the system paged out to the disk per second. As shown in the table, in our case, the number of pgpgout/s is 1135.7 KB/s on average. Also, we use strace to keep track of the system calls that are called during the "make" process. The total number we get is 3193 which is much larger than the number of swapping between the memory and the disk. Overshadow needs to encrypt/decrypt the application memory every time a system call is invoked while AppGuard only needs to do this for I/O data which reduces the time overhead of the entire system.

VI. Conclusion

In this paper, we present a novel system, named AppGuard, which can effectively isolate the user applications from untrusted OSes while allowing applications to make full use of the OS services. AppGuard utilizes hardware virtualization support provided by Intel VT and AMD-V and meanwhile, without requiring any modifications to the user applications or the OS. This makes AppGuard backward compatible. In our future work, we will implement a prototype of AppGuard and evaluate our system by testing out a suite of micro and macro benchmarks and analyze the performance of AppGuard in more details.

References

[1] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, ``Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *In ASPLOS*, May 2008.

[2] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, ``Inktag: secure applications on an untrusted operating system," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating system*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 265--278.

[3] J. Yang and K. G. Shin, ``Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 71--80.

[4] R. Ta-Min, L. Litty, and D. Lie, ``Splitting interfaces: making trust between applications and operating system configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 279--292.

[5] Y. Cheng, X. Ding, and R. H. Deng, ``Appshield: Protecting applications against untrusted operating system," in *Singaport Management University Technical Report*, ser. smu-sis-13-101, 2013.

[6] D. Lie, C. A. Thekkath, and M. Horowitz, ``Implementing an untrusted operating system on trusted hardware," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, 2003, pp. 178--192.

[7] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, ``Secureme: A hardware-software approach to full system security," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11, 2011, pp. 108--119.

[8] D. Champagne and R. B. Lee, ``Scalable architectural support for trusted software," in *HPCA'10*, 2010, pp. 1--12.

[9] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, ``Flicker: An execution infrastructure for tcb minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08, 2008, pp. 315--328.

[10] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, ``Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*, May 2010, pp. 143--158.

[11] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, ``Memoir: Practical state continuity for protected modules," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 379--394.

[12] Z. Deng, D. Xu, X. Zhang, and X. Jiang, ``Introlib: Efficient and transparent library call introspection for malware forensics," in *Digital Investigation*, no. 0, 2012, pp. 96--112.