Enhanced privilege separation for commodity sofware on virtualized platform

Mingyuan Xia, Miao Yu, Qian Lin, Zhengwei Qi, Haibing Guan
School of Electronic, Information and Electrical Engineering; School of Software Engineering
Shanghai Jiao Tong University
Shanghai, China
{*kenmark, superymk, linqian, qizhwei, hbguan*}*@sjtu.edu.cn*

***Abstract*—Conventional privilege separation can effectively reduce the TCB by granting privilege to only the privileged process. However, since they rely on process isolation for security assurance, kernel space malware can easily compromise. Meanwhile, the frequent inter-process communications between the privileged and unprivileged process incurs notable and inevitable overhead. To ameliorate these problems, we propose to keep both the privileged and unprivileged portions in the same process and leverage virtualization to construct a secure container for the privileged fraction. The virtual machine monitor intercepts all the boundary switches transparently to ensure the integrity and privacy.**

We have implemented a prototype, named Coir, based on Xen hypervisor. As a case study, a real-world remote control application named TightVNC is partitioned and protected in Coir-enabled unmodified Windows XP. The experiment result illustrate the effectiveness and performance penalty of our system.

***Keywords*-virtualization; privilege separation; security;**

I. INTRODUCTION

Growing functionalities of commodity software incur more complexity and uncertainty, which downgrades the security guarantee of them. Recent researches [1] and practical projects [1] apply privilege separation on the source code of application to maintain least privilege. These approaches partition the original program into two processes (privileged process and unprivileged one) and employ Inter-Process Communication (IPC) for interaction. However, since the security of this setting relies on process isolation, malware residing in the OS kernel can easily exploit against the integrity and privacy. Furthermore, frequent IPC requests impose remarkable and inevitable overhead which is claimed to be about 2x to 8x slowdown for basic communication primitives from existing tools. Facing this situation, we propose to perform the privilege separation without employing the dual process mechanism. Instead, we rely on virtualization to ensure the isolation of security-critical fraction of a target application from the other parts (unprivileged application code and untrusted OS kernel).

With the renewed popularity of system level virtualization, an increasing number of recent researches on security are leveraging virtual machines. Recent research showed that strong isolation of virtualization benefits security tools by separating them from untrusted surroundings. Introspection techniques [2], [3] monitoring the system from trusted VM witness better reliability [4]. Though strong isolation characterizes the virtualization, it always suffers tough efforts to be effectively applied for the security usage. Many former studies of the virtual machine based software protection are rough and of the guest OS dependence [5]. The approaches of passive monitoring [6], [7], [8], [9] only detect malicious instructions without defending the attacks. Although the active monitoring could detect attacks earlier and prevent certain attacks from succeeding [10], [4], the considerable overhead induced by the frequent boundary switching between Virtual Machine Manager (VMM) and virtual machines limits their practical usage. Moreover, fine-grained protection is generally unavailable in these approaches due to the lack of sensitive semantic and protector awareness.

Instead of monitoring the overall guest environment from a higher privilege level, our system targets privilege separated applications which expose the security semantics during privilege partition. We reuse part of the conventional privilege partition to delimitate the security-critical portions with the help of user annotations. During runtime, we add a protection component to the hypervisor and realize the isolation between the privileged code and unprivileged fraction.

The contributions of our work are summarized below:

- We propose a novel mechanism to combine the advantage of virtualization and privilege separation to gain better protection for privileged fraction of applications.
- We employ virtualization isolation instead of process isolation to ensure security assurance. Instead of depending on the whole OS kernel which is prone to compromise, security-critical portions are ultimately protected by the separate execution environment that our system provides.
- By taking a solo process approach, we manage to reduce the overhead caused by IPC methods. We observe a general speedup from our practical case study when compared with conventional privilege partition. Besides, our performance is expected to improve as advanced virtualization support (like hardware-assisted nested paging) matures.

[1]Privilege separated OpenSSh: http://www.citi.umich.edu/u/provos/ssh/privsep.html

The rest of this paper is organized as follows. Section II provides the assumptions and design goals for our work. Section III explains the design of the architecture and the implementation of current prototype. Section IV evaluates our system with a practical case study which includes the effectiveness test and overhead analysis. Section V summarizes related work and Section VI concludes.

II. Assumptions and Design goals

In this section we examine the requirements for security sensitive portions and present the pre-requisite for our protection approach.

A. Security Requirements

After privilege partition, the sensitive portions are extracted and the security requirements can be concluded as follow:

- (R1) **Launch Time Integrity Verification:** Before the secure execution environment builds up, any attempt to tamper with the integrity sensitive code and data before launch time should be detected and blocked.
- (R2) **Legitimate Runtime Access:** Only privileged fraction owns the permission to manipulate the sensitive data or change the sensitive code (self-modifying code). The unprivileged part can by no means corrupt the integrity of the sensitive code or have access to the sensitive data.
- (R3) **Controllable Sensitive Boundary Transition:** The switches between the untrusted and sensitive zone should be known, i.e., the execution flow can transfer between the untrusted part (OS or unprivileged fraction) but the protection system should be aware and provide the correct view of private resources.
- (R4) **Trusted Services:** Sensitive code should be able to request system services by issuing system call but the OS should not be able to corrupt the code integriy or leak the private data to others.

B. Assumptions and Threat Model

We make the standard assumptions seen in most other virtualization security architectures [10], [11], [12], [2], [13]. *Coir* is expected to run on the bare metal environment, i.e. directly on hardware rather than within the host operating system, excluding the recursive virtualization situation. The VMM ensures the isolation among guest VMs, and owns the highest privilege level to monitor VMs' behavior. Note that attacks such as *Blue Pill*[2] [14] are not possible because a VMM using virtualization extensions is already installed as part of our architecture [10].

This paper is concerned with privilege-aware application that can be separated by conventional privilege partition tools. The vulnerabilities of the privileged code are not within the protection and in extreme cases, the proactive leakage of private information by the privilege code will not be protected.

Our threat model is realistic and assumes that an attacker can do any behaviors including diverse attacking tricks in the guest VM. This includes malicious code injection, rootkit, buffer overflow, etc. Besides, the untrusted guest VM environment involving OS kernel, drivers and services may also expose security holes.

[2]Blue Pill: http://invisiblethings.org/

III. Design

A. Overview

The overall composition of our system is shown in Figure 1. Coir reuses the front end of conventional privilege separation tools to identify and extract the sensitive-critical portion from the original source code. The back end is redesigned to cater to Coir runtime protection component. Instead of constructing another process for security-critical portions extracted, Coir utilizes compiler directives to gather sensitive portions into certain section of the executables. During runtime, Coir-enabled hypervisor will create a separate execution environment for the privileged part, which is aided by memory virtualization as well as the hooks plugged by the back end to inform Coir of possible boundary switch.

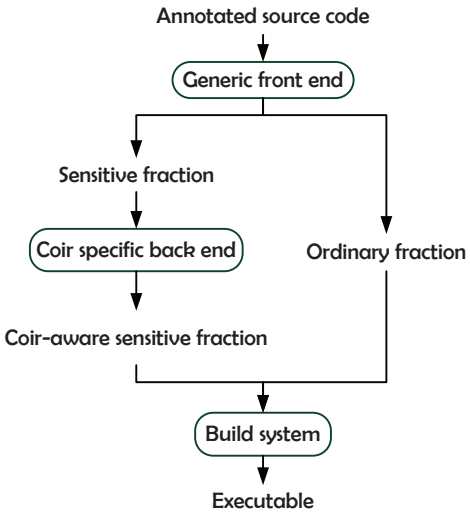

Figure 1. The work flow of Coir. The back end of traditional privilege separation tools is modified to cater to Coir.

B. Privilege separation

Coir does not invent a new method to analyze the legacy source code. Instead, we utilize the majority of existing privilege partition techniques [1], [15] and take full advantage of the static analysis capacity of these tools. These previous work leverage static analysis like data flow analysis [1] to delimitate the privileged code that handles the resources

indicated by the programmer annotation. These tools follow two phases to process the source code. The front end takes annotated source code as input and export the partition result (in forms like a list of sensitive functions or the like). Then the back end is cascaded to revise the extracted sensitive portions. The modification done to the sensitive portions is on demand. Traditional privilege partition needs to examine all the function call in the sensitive portions and replace some of them with Remote Procedure Call (RPC) to functions within the unprivileged process, or slave process. Finally the build system will generate the executables from the modified source code.

Coir rewrites the back end and manages to plugs hooks on each function call in the sensitive portions. These hooks are necessary to inform the runtime protection component of the upcoming function call, which will be discussed in the next section. Sensitive functions identified by the front end of privilege partition are adorned by certain compiler directives to gather them in fixed section of the final executables. Though these directives differ from one compiler to another, we manage to wrap them with macros and eliminate the difference of compiler and system. In our current prototype, Microsoft Visual C++ and GNU C/C++ Compiler that generate Portable Executable (PE) for Windows are supported and Executable and Linkable Format (ELF) for Linux is also acceptable since both executable format employ the segment (section) design and modern linkers are capable of customizing the layout with specific directives. Finally, the sensitive section of the executables is encrypted and hashed to enable checksumming at launch time, which verifies that the static integrity is not corrupted (R1).

C. Sealed runtime execution environment

Once the sensitive area is validated by R1, Coir will shepherd the runtime integrity to ensure that the access to sensitive area is well limited (R2), the control flow transfers between boundaries are verified by VMM(R3) and the system services for sensitive code are trusted(R4). Coir virtualizes a trusted execution environment called *sealed execution environment* for sensitive code while unprivileged code still inhabits the primitive environment supplied by the guest operating system. The sealed execution environment (SEE), creates an isolated and limited scope for the sensitive resources to be accessed. These resources (code, data, runtime stack, etc.) will be automatically cloaked at the very point that the border transition happens (R2) and will by no means be leaked to the outside world.

We abstract the communication between the sealed execution environment and the primitive one into three patterns:

- *Sensitive entry* refers to the invocation of a sensitive function from the outside world.
- *Ordinary entry* refers to calling functions outside the privileged area during the sealed execution. Common manifestations are calling unprivileged functions or making system call.
- *Interrupt* refers to the interruption of sealed execution by asynchronous events like external interrupts and program exceptions.

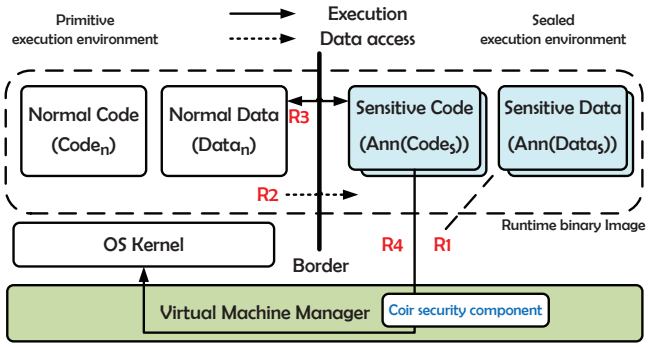

Figure 2. Coir relies on a runtime protection component added to hypervisor to protect the Coir-aware applications. Four requirements for the separated model are examined in the architecture.

Coir employs multiple strategies to intercept all these transition events, including hardware-assisted virtualization (HAV) and Coir-aware configuration done via privilege partition. These strategies take the advantage of advanced virtualization technology and privilege partition to achieve a balance of flexibility and efficiency.

Nested paging.: The fundamental mechanism in use benefits from the memory virtualization technology. *Nested paging* enables another level of address mapping which translates guest physical address to machine address. The additional mapping can be performed either by hardware (Intel Extended Paga Table (EPT) [16], AMD Nested Page Table (NPT) [17]) or by software (Shadow Page Table [18]) transparently and unnoticeably to the guest OS. Similar to conventional paging scheme, nested paging allows for setting access bit for a certain guest physical page. When HAP access violations occur (also named as *nested page fault*), a VM Exit causes control flow to be passed from the guest to the VMM. The analogue relationship between conventional paging facility on x86 architecture and HAP enabled by hardware-assisted virtualization is demonstrated in Figure 3.

1) Sensitive entry.: Coir manipulates two sets of sensitive pages for different execution environment and maintains the correct mappings between switches. Only the sensitive pages for sealed execution contain decrypted sensitive content and no access control. The other set, however, has useless data with no execution permission. During the primitive execution, when a certain function call refers to the address within sensitive pages, a nested page fault will happen. The VMM intercepts this fault and informs Coir to change the execution state. Once Coir has done all the preparation work and enters the sealed execution environment, sensitive pages

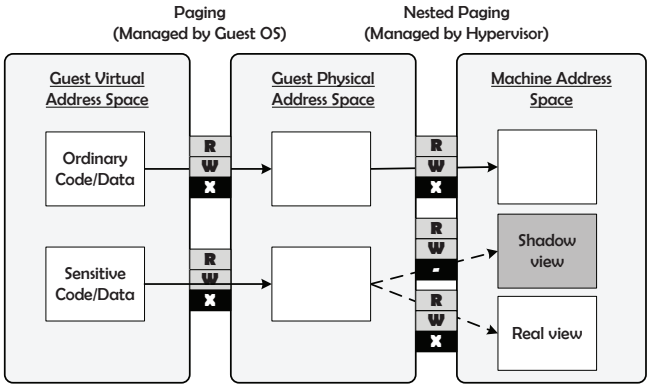

Figure 3. Nested paging compared with traditional paging scheme. The extra mapping enables Coir to manipulate different view of memory according to the context while keeps unnoticeable to the guest environment.

with real content and full access are mapped and the return address is marked non-executable to trap the exit (R3). The VMM resumes the execution of the guest virtual machine and unless another two events that transfer the control back to the outside are encountered, sensitive code can execute without any interference from the VMM. On exit, CPU will be redirected to the return address and trapped again into Coir to finish necessary tasks before resuming to primitive execution environment.

2) Ordinary entry.: Although hardware-assisted memory virtualization can intercept most transfers, yet this method cannot settle all the problems engaged. Suppose a function call to the unprivileged code is needed during the sealed execution, it is essential to cloak sensitive area at the very moment when the control is transferred (R2). Under this circumstance, Coir cannot intercept the crucial transfer, unless all the pages other than the sensitive ones are marked non-executable. This method, as discussed in SecVisor[19], tortures unbearable overhead for applications. Besides, before the very instruction that causes border transition, it is necessary to pass some parameters for the function call. These parameters, however, cannot be encrypted along with the sensitive stack frame when the boundary switch happens (the preservation of private stack will be discussed later). Before the branch instruction and parameter stacking, the VMM should be notified to stop preserving private stack. To ameliorate this situation, Coir utilizes the hooks previously plugged to each function calls within sensitive portions accomplished by the privilege partition to be informed of the ordinary entry. Before making a function call inside the sensitive portion, a hypercall with the destination address is invoked first, right before it begins to stack the parameters. Coir is informed to stop the preservation of private stack and temporarily marks the destination page non-executable. Then, when the branch instruction triggers the nested page fault, Coir is able to preserve the correct range of private stack and switch execution state timely. Finally, the execution guard on the destination page is evacuated and the VMM resumes the guest machine running in normal code. On exit, similarly to sensitive entry, Coir verified the return address before resuming to sealed execution. The ordinary entry also applies to system calls since modern programming language wraps system calls with function calls.

3) Interrupt and exception.: Interrupts are asynchronous events that will lead to the transfer of control flow from anywhere to handler function given by Interrupt Descriptor Table (IDT). Modern virtualization supports can intercept the interrupts and exceptions, especially with hardware-assisted virtualization supports. Coir handles the interrupt during sealed execution by emulating a burst normal entry. In our current prototype, the trap for interrupts is only used to maintain the page remapping. Supporting interrupts has no conflict with the overall architecture but requires additional engineering effort and is reserved for future work.

D. Session management

For each border transition, Coir creates a new *session* to trace the execution. The session usually contains the call stack status related to certain execution status. Sessions are also organized like a stack similar to the call stack frame each owns. When a new ordinary session is triggered, Coir encrypts and hashes the stack frame of previous sealed execution to prevent integrity corruption or privacy leakage. The session design of Coir enables a track for individual stack frame and protects the runtime stack regions.

IV. Evaluation

In order to illustrate the practical use of Coir, we have picked up a popular open source project for remote control, named TightVNC for a case study. This study includes the partition efforts required, the sensitive part extraction, the effectiveness of sealed execution environment and the overhead analysis based on microbenchmark. The implementation of Coir is based on hardware-assisted virtualization, including Intel Extended Page Table support for nested paging.

A. Privilege separation

To begin with, we download the up-to-dated source code (version 2.0 beta 1) from the website[3]. The authentication password is marked as the sensitive data and we apply the privilege partition tools and receive the result of a list of two functions involved in the operation of the sensitive data. We then manually analyze the source code to examine the correctness of privilege partition. The composition of the source code and the modification done by Coir is shown in Table I.

The practical application of privilege partition shows that sensitive code in the overall application is comparatively isolated from the other part. Thus it is much easier to

[3]http://www.tightvnc.com/

Aspect	Module	Files	LOC	Percentage
Raw I/O	Network	23	1560	1.83%
	File	14	1073	1.26%
Utility	zlib	25	7206	8.46%
	libjpeg	57	23809	27.97%
	omnithread	3	1398	1.64%
	X11	7	3302	3.88%
Protocol	RFB	34	5305	6.23%
	File share	61	6883	8.09%
Misc	Wrapper	45	5652	6.64%
	OS related	64	5083	5.97%
	Other	34	5002	12.41%
UI	GUI	93	9261	10.88%
	controller	2	4028	4.73%
Security	Authentication	3(2)	224(24)	0.26% (0.03%)
	DES	2(1)	493(8)	0.58% (0.01%)
Coir	Coir Headers	2	138	0.16%

Table I
THE SOURCE CODE COMPOSITION OF TIGHTVNC VERSION 2.0 BETA 1. THE VALUE IN PARENTHESES MEANS THE MODIFICATION WE HAVE MADE. THE GENERATED EXECUTABLES WITNESS A SIZE INFLATION OF LESS THAN 0.32% DUE TO THE INCLUSION OF AUXILIARY HOOKS. RPB IS A SIMPLE PROTOCOL FOR REMOTE ACCESS TO GRAPHICAL USER INTERFACES.

extract this small fraction from the whole program. Besides, protection from Coir is not necessarily related with growing expansion in code size. This is also in accord with Coir's goal to constraint the protection domain. And more important, Coir loses no competitive power compared with existing protection methods even with a limited intervention into the original code. This will be further illustrated in the following investigation.

B. *Effectiveness investigation*

To illustrate the protection requirements compliance, we carry out several common attacks on both the original version and the Coir enabled version of TightVNC. These attacks include debugging and hooking to exploit the sensitive data. We discuss the mechanisms employed by general attackers and show how Coir strangles these attempts at the very beginning.

In the code fragment (Figure 4 extracted from the original code tree of TightVNC, the sensitive data are generated from line 3(for file input) and line 9(for UI input) and stored in plain text until cleared by line 16. Any pause of the execution within this period can leak the sensitive data to the attackers. Besides, given the file input situation, the operating of sensitive data is not limited to sensitive code in line 4, 5 and 16. Any hooks in these public functions can spy the first hand decrypted sensitive data. The using of public functions on sensitive data is the security hole of the software design.

We eliminate these security holes and extract the three sensitive functions (shown in red color in Figure 4 and the

```
 1: challenge = ReadSocket(...);
 2: if (m_passwdSet) {
 3:    char *pw = vncDecryptPasswd(m_encPasswd);
 4:    strcpy(passwd, pw);
 5:    free(pw);
 6: } else {
 7:    LoginAuthDialog ad(...);
 8:    ad.DoDialog();
 9:    if (strlen(passwd) == 0)
10:       ...
11:    vncEncryptPasswd(m_encPasswd, passwd);
12:    m_passwdSet = true;
13: }
14: vncEncryptBytes(challenge, passwd);
15: /* Lose the plain-text password from memory */
16: memset(passwd, 0, strlen(passwd));
```

Figure 4. Code fragment from original TightVNC, which contains a potential security hole.

```
1: challenge = ReadSocket(...);
2: if (m_passwdSet) {
3:     passwd = vncDecryptPasswd(m_encPasswd);
4: } else {
5:     ...
6: }
7: vncEncryptBytes(challenge, passwd);
8: clearPasswd(passwd);
```

Figure 5. The partitioned version of TightVNC with potential security holes fixed.

sensitive data into the sealed execution environment and Figure 5 shows the modified version (only the part taking input from file). The invalid operating of sensitive data by normal functions is wiped and `vncDecryptPasswd` is reorganized to decrypt the password, store the plain text in the sensitive data and return only a reference. With the partition done and protection component installed in VMM, we again examine the former attack points. Two common attacks that formerly take effect are eliminated.

Disabling debugging.: We use OllyDbg[4] to emulate the possible debugging procedures taken by attackers to exploit the sensitive data. In the case of TightVNC, we suppose that an attacker attempts to decode the encrypted password inputted from the file to obtain the plain text. First, by constructing an arbitrary authentication and monitoring the access to user input (eg. setting breakpoints on `GetDlgItemText` for Windows), the attacker is able to trace the exact routine of the authentication process and discover the period when plain text appears in memory. Then, when the encrypted password is loaded from file and starts another authentication, the attacker can exploit the plain text view of password by pausing on line 4 or 5 in

[4]http://www.ollydbg.de/

Figure 4.

The Coir-aware version of TightVNC separates the sensitive part from the normal code where precise protection is granted. Then only sensitive code manages to operate the real content of sensitive part. Since the debugger resides the primitive execution environment, it can by no means access the sensitive part (eg. setting software breakpoints on sensitive code or reading sensitive content). We follow the same procedure taken to exploit the original version and the sensitive part keeps a "blackout zone" from the debugger. Breakpoint on line 7 in Figure 5 exhibits only a reference to sensitive data but no real content can be leaked.

Against hooking.: The sensitive data should be only accessible to sensitive code and not be operated by public functions. However, the original version of TightVNC authentication process uses several *stdlibc* functions before the plain text of sensitive data is cleared. Any means of hooking on the public functions `strcpy, free, and strlen` enable the attacker to access the sensitive data in Figure 4. Coir eliminates the potential security hole by restricting the margin for the access of sensitive data. Unnecessary dependence on *stdlibc* functions in the sensitive code is eliminated to satisfy R3 and will not leak the sensitive data. Moreover, even if the attacker intercepts and replaces the return value given by line 3 in Figure 5, the authentication message composed by `vncEncryptBytes` will in no ways be authenticated by the server. Finally, `clearPasswd` is used to replace the invalid use of `memset` and `strlen` to clear the sensitive data (potential dumping before clearing) in the former version. However, since the plain text will seldom be revealed to the primitive execution environment, this function is not necessary at all.

C. Overhead analysis

We measure the influence introduced by Coir to the application being protected by performing microbenchmark on different layers of the whole program. It is shown that through the instruction-level witnesses a notable overhead, when we measure its influence to the whole authentication process the impact is greatly reduced.

All experiments are conducted on the platform configured with 3.20GHz Intel Core i7-965 processor, Intel DX58SO motherboard and 6GB DDR3 memory. The VMM is Xen 3.4.2 with 2.6.18 XenLinux as domain0. The guest virtual runs Windows XP sp3 and is configured with 2GB memory and single processor.

Instruction level: We measure the machine cycles taken for the one sensitive call and obtain a total value of 35,000. Table II shows the detailed overhead composition. The cost for nested page fault and operations on nested page table is hardware specific. We attribute the overhead to hardware-assisted paging and our performance will improve as the technique matures.

Situation	Reason	Cycles
Sensitive entry	Nested page fault	5,000
	Acceptance check	20,000
	Remap the real content	5,000
	Evacuate guard	5,000
Sensitive exit	Nested page fault	5,000
	Remap the shadow content	5,000
	Set guard	5,000

Table II
THE DETAILED CYCLE LATENCY INTRODUCED BY COIR FOR THE BORDER TRANSITION. THE VALUE HAS BEEN ROUNDED.

Sensitive function level: Take an abstraction up to the sensitive function, we measure the runtime of original function and calculate the time inflation induced by Coir. The average time taken for a single encryption and decryption is about 30,000 machine cycles as measured. According to the microbenchmark applied to evaluate the border transition cycles, the sensitive entry requires on average 35,000 cycles and the exit takes 15,000. The total time inflation ratio is 266.67% for each sensitive call. This inflation is insignificant when compared with the network latency of the authentication process although in the sensitive function level the border transition cost surpasses the actual work.

Authentication process level: The upper caller for sensitive functions is the authentication module. It implements the authentication contact between the VNC client with server. Five phases compose the overall process and the cost for each phase is shown in Table III. It is observed that the security sensitive code is limited in both time and space. And when the higher layer is concerned, this portion accounts for only a small share of the total execution time. Thus the limited time inflation has a insignificant (approximately 0.5%) impact on the overall authentication process. The overhead incurred for single socket send (phase 3 and 4) shown in the table is about 18.93%. Compared with 8.83 performance penalty for a socket call exhibited in traditional privilege partition approach like privtrans [1], Coir has a

Phase	Cost(μs)	Percentage	Description
1	1263.25	20.94%	Receive key from the server
2	24.05 (15.02)	0.40% (0.25%)	Call sensitive function to encrypt password
3	24.30 (15.25)	0.40% (0.25%)	Call Sensitive function to compose authentication message
4	56.26	0.93%	Send authentication message
5	1247.78	20.69%	Receive the authentication result

Table III
AUTHENTICATION PROCESS COST. THE VALUE IN PARENTHESES DEMONSTRATES THE OVERHEAD CAUSED BY COIR.

notable improvement.

User interactive level: In this level, the frequent windows message from the user and the refreshing cost overruns the cost of the authentication process. Though the accurate time is not measurable in this level, yet the estimated time delay caused by this level will the larger by several orders of magnitude. And the overhead caused by Coir in the very bottom level is minor to the overall efficiency bottleneck.

V. Related Work

We demonstrate a comparison between Coir and several prestigious software protection models which are also based on the virtual execution environment. While providing no less security guarantees than those former models, Coir owns some unique advantages in several aspects.

Prior approaches focus on supplying secure execution environment, including dividing system call interfaces into trusted and untrusted classifications [20] as well as reconstructing a new private guest OS in separate VM [11]. With the goal of ensuring the target application's dynamic integrity, Overshadow [5] proposed a method of providing different guest process memory views to different execution contexts. However, Overshadow's black-box protection ignores the semantics of target application, forcing it to protect the whole address space and rewrite syscall interfaces when migrating on to a different OS. Coir is similar to Overshadow in providing shadow view of the same guest memory. However, in our approach only the target application's sensitive code and data are protected and the protection is guided by security semantics exposed during privilege partition. By reducing the overall complexity, Coir greatly reduces the chance of being attacked due to the application's own bugs and gain performance improvement. Meanwhile, aided by privilege partition done to the source code, Coir is much easier to port to different guest OSes and practical applications.

An even more recent research [21] also extracts and protects selected portions of the security-sensitive applications. However, this approach confines the functionality of secure execution environment and work for only self-contained sensitive code. Meanwhile, since no formal ways is provided to extract the self-contained code, user has to manually extract and total rewrite the fraction to cater to the protection system, which incurs unbearable efforts for redevelopment.

Apart from software protection, several approaches have then been proposed to use system level for providing security in production systems [11], [2], [10], [9]. Most of these approaches were implemented as the passive monitoring, which can only detect an already successful attack without defending. Bryan Payne et al [10] proposed an active monitoring architecture that isolated the security tools in a trusted virtual machine. However, it suffered high performance overhead due to the frequent boundary switching between untrusted and secure virtual machines. Secure In-VM Monitoring (SIM) [4] also played an active monitoring role but put the security tool in the untrusted virtual machine while VMM only handled specific events. SIM featured distinguished performance improvement compared with the previous active monitoring approaches. But SIM lacked the security semantic when doing the protection. Coir depends on the security semantics exposed by the privilege partition to offer fine-grained protection. In the meantime, Coir meets the performance requirement in that VMM only intercepts sensitive related event which is usually a small portion of the whole program.

Additionally, unlike most other virtualization security architectures, Coir needs no special booting requirement such as clean or secure booting [10], [22] in that Coir does not depend on any components in the guest OS.

VI. Conclusion

In this paper, we have presented a novel mechanism to combine the benefits of privilege partition and strong isolation that features virtualization. Our system can effectively enforce privilege protection with little user intervention. As an illustration, we have designed and implemented Coir, a virtualization-based prototype that uses privilege partition to protect the sensitive code and data of software. Coir utilizes the security semantics exposed during privilege partition to focus the protection to the security-critical portions of the application. Coir reserves the privileged and unprivileged in the same process, which eliminates the inevitable IPC overhead caused by previous privilege separation approaches.

To evaluate the effectiveness and performance of our approach, we have migrated TightVNC Viewer, an open source remote control application, to Coir and carried out series of evaluation. We succeed in exacting the sensitive part by modifying 32 lines of code and adding 125 lines out of the total 84.9 thousand lines of code. Our experiment results show that common attacks that previously succeed are effectively prevented by Coir while incurring an overhead of about 18.93% to communication primitives. We believe that Coir demonstrates a promising approach in safeguard software on virtualized platforms.

References

[1] D. Brumley and D. X. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, 2004, pp. 57–72.

[2] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *NDSS*, 2003.

[3] B. D. Payne and W. Lee, "Secure and flexible monitoring of virtual machines," in *ACSAC*, 2007, pp. 385–397.

[4] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *ACM Conference on Computer and Communications Security*, 2009, pp. 477–487.

[5] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. S. Dwoskin, and D. R. K. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ASPLOS*, 2008, pp. 2–13.

[6] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *ACM Conference on Computer and Communications Security*, 2007, pp. 128–138.

[7] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *ACM Conference on Computer and Communications Security*, 2008, pp. 51–62.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *USENIX Annual Technical Conference, General Track*, 2006, pp. 1–14.

[9] N. L. P. Jr. and M. W. Hicks, "Automated detection of persistent kernel control-flow attacks," in *ACM Conference on Computer and Communications Security*, 2007, pp. 103–115.

[10] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy*, 2008, pp. 233–247.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *SOSP*, 2003, pp. 193–206.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," in *OSDI*, 2002.

[13] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *SOSP*, 2005, pp. 91–104.

[14] J. Rutkowska, "Subverting vista kernel for fun and profit," in *Black Hat Japan*, 2006.

[15] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16.

[16] *Intel 64 and IA-32 Architectures Software Developer's Manuals*. Intel Corporation, 2007.

[17] *AMD64 Architecture Programmers Manual*. Advanced Micro Devices, 2006.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP*, 2003, pp. 164–177.

[19] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *SOSP*, 2007, pp. 335–350.

[20] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *OSDI*, 2006, pp. 279–292.

[21] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation." in *IEEE Symposium on Security and Privacy*, 2010.

[22] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy*, 1997, pp. 65–71.