Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation

Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, Yubin Xia
Shanghai Key Laboratory of Scalable Computing and Systems &
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University, Shanghai, China
{ytliu.cc, zhoutianyu007, kalzium, haibochen, xiayubin}@sjtu.edu.cn

ABSTRACT

Exploiting memory disclosure vulnerabilities like the HeartBleed bug may cause arbitrary reading of a victim's memory, leading to leakage of critical secrets such as crypto keys, personal identity and financial information. While isolating code that manipulates critical secrets into an isolated execution environment is a promising countermeasure, existing approaches are either too coarse-grained to prevent intra-domain attacks, or require excessive intervention from low-level software (e.g., hypervisor or OS), or both. Further, few of them are applicable to large-scale software with millions of lines of code.

This paper describes a new approach, namely SeCage, which retrofits commodity hardware virtualization extensions to support efficient isolation of sensitive code manipulating critical secrets from the remaining code. SeCage is designed to work under a strong adversary model where a victim application or even the OS may be controlled by the adversary, while supporting large-scale software with small deployment cost. SeCage combines static and dynamic analysis to decompose monolithic software into several compartments, each of which may contain different secrets and their corresponding code. Following the idea of separating control and data plane, SeCage retrofits the VMFUNC mechanism and nested paging in Intel processors to transparently provide different memory views for different compartments, while allowing low-cost and transparent invocation across domains without hypervisor intervention.

We have implemented SeCage in KVM on a commodity Intel machine. To demonstrate the effectiveness of SeCage, we deploy it to the Nginx and OpenSSH server with the OpenSSL library as well as CryptoLoop with small efforts. Security evaluation shows that SeCage can prevent the disclosure of private keys from HeartBleed attacks and memory scanning from rootkits. The evaluation shows that SeCage only incurs small performance and space overhead.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813690 .

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access control, information flow controls*

General Terms

Security

Keywords

Privacy protection, memory disclosure, virtualization

1. INTRODUCTION

Problem. Cloud servers are increasingly being deployed with services that touch critical *secrets* such as cryptographic keys, personal identity and financial information. Yet, such *secrets* are being continually disclosed due to memory disclosure attacks. For example, HeartBleed (CVE-2014-0160) [3, 23], as one of the most notorious vulnerabilities, can be leveraged by attackers to persistently read up to 64KB memory data. This leads to the leakage of the most confidential *secrets* such as private keys or session keys of all connections.

Actually, memory disclosure vulnerabilities are routinely discovered and pose severe threats to the privacy of *secrets*. Table 1 shows that there are 388 such vulnerabilities from the CVE database according to the triggering mechanisms for both applications and kernels, which can be exploited by local (L column) or remote (R column) attacks. Worse even, leakage of *secrets* can be directly done if an attacker gained control over a privileged application or even the OS kernel through other vulnerabilities such as privilege escalation.

Facing such a strong adversary model, it is notoriously hard to prevent attackers from controlling victim applications or even the OS kernel from reading such memory-resident *secrets*. This is further being exaggerated due to the fact that large-scale software usually has a large code base and thus large attack surfaces. While there have been a number of approaches aiming at protecting an application from various attacks, they still fall short in several aspects. For example, hypervisor-based protection schemes [16, 17, 18, 53, 19] only provide protection at an application level and thus vulnerabilities still exist inside a victim application; approaches targeting at Pieces of Application Logic (PAL) [34] require the PAL being self-contained and thus allowing no interaction with other parts of an application. This makes it hard to be adopted for some real and large

software like OpenSSL[1]. Further, due to the close-coupling of security and functionality, prior approaches usually require frequent intervention from a privileged system (e.g., hypervisor), forcing users to make a tradeoff between security and performance.

Table 1: Sensitive memory disclosure vulnerabilities from CVE (2000-2015)

Error type	Application		Kernel		Summary
	L	R	L	R	
Uninitialized memory	1	34	104	8	**147**
Out-of-bound read	4	37	20	4	**65**
Others	3	28	16	7	**54**
Use-after-free vuln	14	7	6	4	**31**
Permission uncheck	4	14	10	0	**28**
Bad error handling	1	12	8	0	**21**
Plaintext in memory	10	2	0	0	**12**
Bad null terminator	0	3	6	0	**9**
Invalid pointer	1	0	6	0	**7**
Off-by-one vuln	1	5	0	0	**6**
String format vuln	0	4	0	0	**4**
Leak to readable file	3	0	1	0	**4**
Total	**42**	**146**	**177**	**23**	**388**

Our solution. In this paper, we leverage the idea of privilege separation by hybrid analysis to (mostly) automatically decompose a monolithic software system into a set of compartments, with each *secret* compartment containing a set of *secrets* and its corresponding code and a main compartment handling the rest of the application logic. This ensures that only the functions inside a *secret* compartment can access the *secrets*. As static analysis is usually imprecise and may introduce large code base in the *secret* compartment, SeCage further combines dynamic analysis to extract the most commonly used functions. To handle possible coverage issues, SeCage resorts to runtime exception handling to detect if an access is legal or not based on the static analysis result and runtime information.

SeCage leverages hardware virtualization techniques to enforce strong isolation among *secret* compartments and the main compartment, even under a strong adversary model such that an application or even the OS is controlled by an attacker. Specifically, SeCage assigns each compartment a completely isolated address space and leverages hardware-assisted nested paging to enforce strong isolation. To provide secure and efficient communication among components, SeCage follows the idea of separating control plane from data plane by using the VMFUNC feature from Intel's hardware-assisted virtualization support. In particular, SeCage first designates the security policy on which *secret* compartment is invokable by another compartment to the CPU, and allows such an invocation from being done without the hypervisor intervention. This significantly reduces the overhead caused by frequent trapping into the hypervisor.

Evaluation on real-world applications. We have implemented a prototype of SeCage in KVM using Intel's hardware virtualization support; we use CIL [1], an analysis framework of C programming language, to (mostly) automatically decompose a software system into a set of compartments. We further use a dynamic analysis engine to refine the results. To demonstrate the effectiveness of SeCage, we apply SeCage to Nginx and OpenSSH server with the OpenSSL library, as well as the Linux kernel's disk encryption module CryptoLoop to demonstrate its effectiveness. We use HeartBleed, kernel memory disclosure and rootkit defenses as examples to illustrate how SeCage protects user-defined *secrets*. Our evaluation shows that SeCage only introduces negligible performance overhead, since the portion of *secrets* related code is small and there are very few hypervisor interventions at runtime.

Contributions. In summary, this paper makes the following contributions:

- A new scheme to protect *secrets* from extensive attack surfaces by compartmentalizing critical code and data from normal ones.
- A separation of control plane (policy) and data plane (invocation) for cross-compartment communication by leveraging commodity hardware features (VMFUNC).
- A working prototype implemented on KVM and its application to large-scale software such as Nginx and OpenSSH with the OpenSSL library and the CryptoLoop module in Linux kernel, as well as security and performance evaluations that confirm the effectiveness and efficiency of SeCage.

The rest of the paper is organized as follows: the next section first illustrates the architecture overview of SeCage and the threat model. Section 3 and section 4 illustrate the design and detailed implementation of the runtime isolation and application decomposition parts of SeCage, followed by the usage scenarios in section 5. Then the security evaluation of SeCage and its incurred performance overhead are evaluated in section 6 and section 7 accordingly. Finally, we review and compare SeCage with state-of-the-art in section 8, discuss the limitation of SeCage in section 9 and conclude the paper in section 10.

2. OVERVIEW

The primary goal of SeCage is to offer the strong assurance of confidentiality for user-specific *secrets* (e.g., private keys), even facing a vulnerable application or a malicious OS. The secondary goal of SeCage is to make the approach of SeCage practical and thus can be deployed for large software systems with small overhead.

2.1 Approach Overview

Hybrid analysis to extract *secret* closure. As *secrets* may be copied and propagated through its lifecycle, it is far from enough to simply secure the storage of *secrets*. Instead, SeCage must provide a thorough mechanism to prevent *secrets* and their provenance from being disclosed during the whole application execution, while not affecting the normal usage of the *secrets*. Hence, SeCage needs to find a closure of all functions that may manipulate the *secrets*.

One intuitive approach is using static analysis to discover the closure of code. However, the static analysis still has precision issues for large-scale software written in C/C++ due to issues such as pointer aliasing. This may easily lead to a significantly larger closure than necessary, which may enlarge the code base of the *secret* compartment and add

[1]For example, the authors of TrustVisor [34] explicitly acknowledged that "We faced the greatest porting challenge with Apache + OpenSSL ...This proved to be difficult due to OpenSSL's extensive use of function pointers and adaptability to different cryptographic providers and instead resort to PolarSSL" in section 6.4.

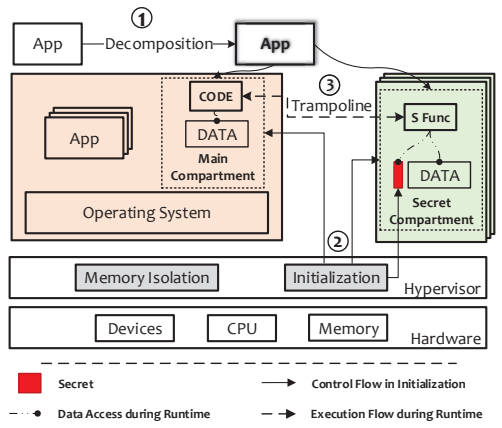


Figure 1: Architecture overview of SeCage

a large number of unnecessary context switches. Another approach would be rewriting the code related to *secrets* and decoupling the operations of *secrets* into a standalone service [13, 34], or even to a trusted third-party node. However, this may involve high manual effort and can be prohibitively difficult for large-scale software such as OpenSSL [34].

SeCage instead combines static and dynamic analysis to extract the closures of functions related to *secrets*. It first uses static analysis to discover potential functions related to *secrets*. To reduce the size of *secret* closure, it then relies on dynamic analysis using a set of training input to derive a compact and precise set of functions related to *secrets*. To handle coverage issue that a function may be legal to touch the *secrets* but is excluded in the *secret* compartment, SeCage adaptively includes this function to the *secret* compartment according to the static analysis result and execution context during runtime exception handling.

Hypervisor-enforced protection. Facing the strong adversary model of a malicious OS, SeCage leverages a trusted hypervisor to protect the privacy of *secrets*. Specifically, SeCage runs the closure of a particular *secret* into a separate compartment and leverages hardware virtualization support to provide strong isolation among different compartments.

Separating control and data plane. As each compartment still needs to communicate with each other, it seems inevitable that the hypervisor will need to frequently intervenes such communications. This, however, will cause frequent VMExits and thus high overhead. To mitigate such overhead, SeCage leverages the idea of separating control and data plane to minimize hypervisor intervention. Specifically, SeCage only requires the hypervisor to define policies on whether a call between two compartments are legal (control plane), while letting the communications between two compartments go as long as they conform to the predefined policies (data plane). In the entry gate of each compartment, it may do further check of the caller to see if the communication should be allowed or not. SeCage achieves such a scheme by leveraging the commodity hardware features called VM functions (section 3.1).

Architecture overview. An overview of SeCage's architecture is shown in Figure 1. The protected application is divided into one main compartment and a set of *secret* compartments. Each *secret* compartment comprises a set of *secrets* and the corresponding sensitive functions manipulating them. We do not assume that the compartment is self-contained, the functions inside are able to interact with the main compartment of the application. However, SeCage guarantees that the *secret* in one compartment cannot be accessed by other compartments of the same application and the underlying software.

Once the compartments are generated (step ①), during the application initialization phase (step ②), the hypervisor is responsible to setup one isolated memory for each compartment, and to guarantee that the *secrets* can only be accessed by the functions in the corresponding compartment. During runtime, *secret* compartments are confined to interact with the main compartment through a trampoline mechanism (step ③) without trapping to the hypervisor. Only when a function outside of a *secret* compartment, e.g., the main compartment, tries to access the *secrets*, the hypervisor will be notified to handle such a violation.

2.2 Threat Model and Assumptions

SeCage aims at protecting critical *secrets* from both vulnerable applications and malicious operating systems (which may also collude).

For vulnerable applications, we consider an adversary with the ability to block, inject, or modify network traffic, so that she can conduct all of the well-known attacks in order to illegally access any data located in memory space of the vulnerable application. Specifically, the adversary can exploit buffer over-read attack as in HeartBleed bug [3], or try to use sophisticated control flow hijacking attacks [40, 11, 12], to invalidate access-control policy or bypass permission check, and read sensitive *secrets* located in the same address space.

The underlying system software (e.g., OS) is untrusted that they can behave in arbitrarily malicious ways to subvert application into disclosing its *secrets*. We share this kind of attacker model with other related systems [18, 17, 28]. Additionally, SeCage also considers the Iago attack [15], where the malicious OS can cause application to harm itself by manipulating return value of system services (e.g., system call), as well as rollback attack [51, 41, 52], where the privileged software can rollback the applications' critical states by forcing memory snapshot rollback.

SeCage assumes that the protected *secrets* should only be used within an application, the functions inside the *secret* compartment won't voluntarily send them out. This is usually true for commodity software like OpenSSL as the software itself is designed to keep such *secrets*. Even if not, this can be detected during the static and dynamic phase of SeCage when generating *secret* compartments. Further, SeCage makes no attempt to prevent against DoS attack which is not aimed at disclosing data. It does not try to protect against side-channel attacks [38, 55], as well as the implicit flow[39] attack which consists in leakage of information through the program control flow, since they are typically hard to deploy and have very limited bandwidth to leak *secrets* in our case. Finally, SeCage does not consider the availability of application in the face of a hostile OS.

3. RUNTIME ISOLATION ENFORCEMENT

In this section, we will introduce how to enforce SeCage protection during application runtime, including memory protection, mechanisms of runtime execution flow and other aspects.

3.1 Memory Protection

In SeCage, compartment isolation is guaranteed by two-dimensional paging[2] mechanism. In general, a guest VM can only see the mapping of guest virtual address (GVA) to guest physical address (GPA), while the hypervisor maintains one lower-level extended page table (EPT) for each guest, the EPT maps GPA to the host physical address (HPA).

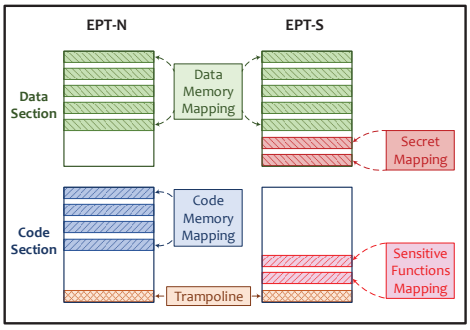


Figure 2: EPT layout of SeCage-enabled application and one of its *secret* compartments

In the initialization phase of SeCage, besides the original EPT called *EPT-N* for the entire guest VM, the hypervisor initializes another EPT, called *EPT-S*, for each protected *secret* compartment. As shown in Figure 2, SeCage classifies the memory into two parts: data and code. For the data section, *EPT-S* maps all data including the *secrets*, while *EPT-N* has data other than the *secrets*. For the code section, the trampoline code is mapped into these two EPTs as read-only. Besides, *EPT-S* only contains the sensitive functions code in the *secret* compartment, while *EPT-N* maps code other than the sensitive functions.

Through the above EPT configuration, SeCage ensures that *secrets* will never exist in *EPT-N*, only code in sensitive functions can access the corresponding *secrets*. These code pages are verified in the setup phase, and the EPT entries are set as executable and read-only. Meanwhile, the data pages in *EPT-S* are set to non-executable so that they cannot be used to inject code by attackers. Therefore, both a vulnerable application and the malicious OS have no means to access the *secrets*. It should be noted that if we only put *secrets* in the *secret* compartment, there may be excessive context switches since the sensitive functions may access other data memory besides *secrets*. For simplicity, since the code pieces of sensitive functions are very small and considered to be trusted in our threat model, SeCage maps the whole data sections into the *secret* compartment.

EPTP switching. SeCage leverages the Intel hardware virtualization extension called VMFUNC, which provides *VM Functions* for non-root guest VMs to directly invoke without VMExit. EPTP switching is one of these VM functions, which allows software (in both kernel and user mode) in guest VM to directly load a new EPT pointer (EPTP), thereby establishing a different EPT paging-structure hierarchy. The EPTP can only be selected from a list of potential EPTP values configured in advance by the hypervisor, which acts as the control plane defining the rules that a guest VM should comply with. During runtime, the hypervisor will not disturb the execution flow within a guest VM.

[2]called *EPT* in Intel, and *NPT* in AMD; we use EPT in this paper.

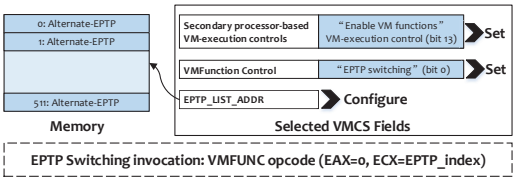


Figure 3: Description of EPTP switching VMFUNC

Figure 3 shows an example configuration that a hypervisor needs to set in order to use EPTP switching VM function: besides some function-enable bits, the hypervisor needs to set bit 0 (EPTP switching bit) in the *VM Function Control* VMCS field, and store the configured EPT pointers to the memory pointed by the *EPTP_LIST_ADDR* VMCS field. During runtime, the non-root software invokes the VMFUNC instruction with *EAX* setting to 0 to trigger the EPTP switching VM function and *ECX* to select an entry from the EPTP list. Currently, EPTP switching supports at most 512 EPTP entries, which means SeCage can support up to 512 compartments for each guest VM.

3.2 Securing Execution Flow

SeCage divides the logic into sensitive functions, trampoline and other code (including application code and system software code). Only sensitive functions in the *secret* compartment can access *secrets*, and the trampoline code is used to switch between the *secret* and the main compartment. During runtime, functions in the main compartment may invoke sensitive functions, while sensitive functions may also call functions outside of the *secret* compartment.

Figure 4 shows possible execution flow during runtime. For clarity, we classify trampoline code into *trampoline* and *springboard*, according to the calling direction. *trampoline* invocation is illustrated in the top half of Figure 4: when code in the main compartment invokes a function in a *secret* compartment, instead of directly calling the sensitive function, it calls into the corresponding trampoline code, which at first executes the VMFUNC instruction to load memory of the *secret* compartment, then the stack pointer is modified to point to the secure stack page. If the number of parameters is larger than six, which is the maximum number of parameters passing supported using register, the rest of parameters should be copied to the secure stack. When everything is ready, it calls the real sensitive function. Once this function returns, the *trampoline* code wipes out the content of secure stack, restores the ESP to the previous stack frame location, reversely executes the VMFUNC instruction and returns the result back to the caller.

During the execution of sensitive functions, it may call functions in the main compartment, e.g., library calls, system calls, and other non-sensitive functions in the application. SeCage classifies these calls into two categories: the calls without *secrets* involved, and the calls which may access the *secrets*. SeCage instruments the first category of calls with *springboard* code, as shown in the bottom half of Figure 4, which just reverses the operations as *trampoline* code. For the second kind of calls, if the callee function does not exist in sensitive functions like some library calls (e.g., memcpy, strlen, etc.), SeCage creates its own version

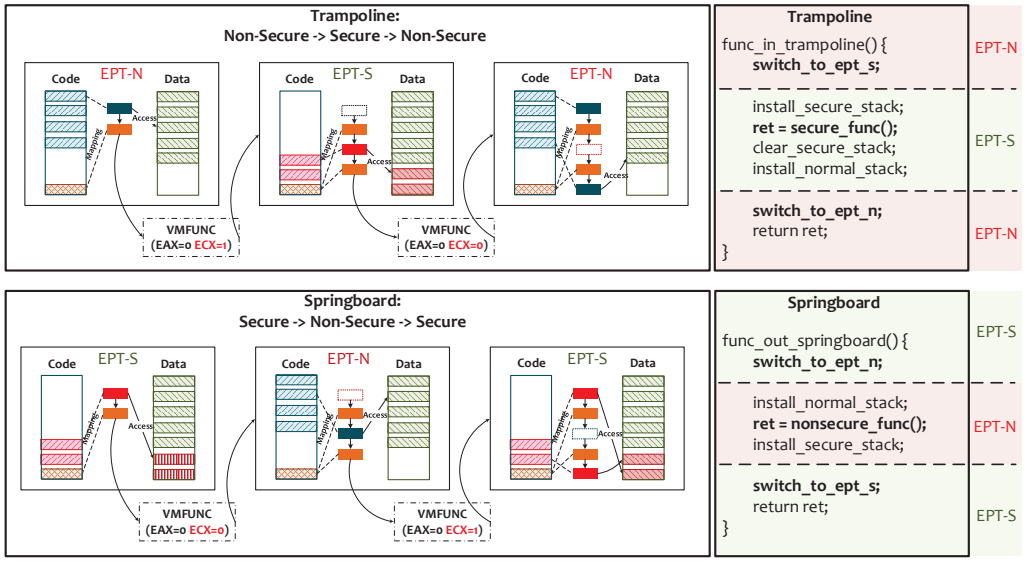


Figure 4: Execution flow through trampoline and springboard mechanisms

of these functions during creating a compartment. There is another situation that the reference to the *secrets* may be passed to the main compartment. However, since the callee function is not able to dereference the reference (otherwise, it should be added to the *secret* compartment), the *secrets* will not be leaked out.

3.3 Miscellaneous

Storage. In the initialization phase, *secrets* may be read from a standalone configuration file, executable binary, or DB schema. Such storages can always be accessed by system software, that there is no effective way to protect them. SeCage solves this problem through another approach by ensuring no *secret* in these storages. The *secrets* in the storage are replaced with some dummy data, during application launching, the dummy data will be restored to the real *secrets*. In the runtime, SeCage ensures no I/O write may happen in the sensitive functions so that *secrets* will not be leaked to the storages.

Interrupt handling. During execution in a *secret* compartment, there is no operating system support within the *EPT-S* context, thus no interrupt is allowed to be injected to the guest VM. When a non-root guest VM traps to the hypervisor due to an interrupt, the corresponding handler in SeCage checks whether it is in the context of *EPT-S*, and what kind of interrupt it is. If the interrupt happens during sensitive functions execution, it simply drops some kinds of interrupts (e.g., timer interrupt), and delays others (e.g, NMI, IPI) until *EPT-N* context.

Multi-threading. SeCage supports multi-threading programs. If there is only one VCPU running all the threads, since we drop timer interrupts to the *EPT-S* VCPU, the *EPT-S* context will not be preempted by other threads until it returning back to the *EPT-N* environment. If there are more than one VCPUs, since every VCPU has its own EPT, if one VCPU is in *EPT-S* context, other VCPUs can still run in *EPT-N* context and they are not allowed to read *secrets* in *EPT-S*.

3.4 Lifecycle Protection of *secret* Compartment

Figure 5 shows the lifecycle protection of a *secret* compartment. SeCage adds three hypercalls [3] as shown in Table 2.

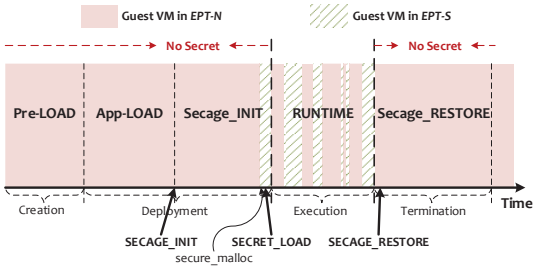


Figure 5: Life-cycle protection of *secrets*

Table 2: Descriptions of SeCage related hypercalls

Hypercall	Description
SECAGE_INIT	Called in *EPT-N*, triggers initializing.
SECRET_LOAD	Called in *EPT-S*, triggers secret loading.
SECAGE_RESTORE	Called in *EPT-N*, triggers restoration

Creation. Before an application is loaded into guest VM, SeCage utilizes the application decomposition framework to analyze the application and decompose it into a main compartment and several *secret* compartments. According to how the *secrets* are loaded, the *secrets* are replaced with dummy data in persistent storages like configuration file, executable binary or database. For example, if the *secrets* are loaded from the files (e.g., OpenSSL) or database during runtime, the *secrets* in the storage are replaced. Otherwise, the application is compiled after replacing the *secrets* in the source code with dummy data. Meanwhile, the developer is required to provide the mapping of *secrets* and the dummy

[3]Hypercall (or vmcall) is similar to syscall but is used to request the hypervisor's services

data (e.g., <*key, length*> → *secret* binding), to the hypervisor through predefined secure offline channels. By this means, the hypervisor can load the real *secrets* into secure memory in the deployment phase.

Deployment. The process of application deployment includes following steps:

1. When launching an application, the instrumented code issues the *SECAGE_INIT* hypercall, which passes the start virtual addresses and the number of pages of sensitive functions and trampoline code as parameters. The hypervisor first checks the integrity of sensitive functions and trampoline code, and setups the *EPT-N* and *EPT-S* as described in section 3.1. It should be noted that *EPT-S* maps several reserved pages that are invisible from *EPT-N*, which will be used as secure heap and stack later.

2. The hypervisor invokes *VMENTER* to restore untrusted application execution. When the untrusted code invokes memory allocation function for *secrets* dummy counterpart, it is redirected to the *secure_malloc* in sensitive functions to allocate pages from the secure heap.

3. After the dummy *secrets* being copied to the secure heap in sensitive functions, the *SECRET_LOAD* hypercall is issued. The hypervisor then scans the secure heap memory, and replaces the dummy *secrets* with the real ones, according to the dummy to *secrets* mapping provided by the user.

Through the above protocol of application deployment, SeCage ensures that before *SECRET_LOAD*, there is no *secret* existing in the memory, thus even the execution environment is untrusted, no *secret* will be exposed. After *SECRET_LOAD*, *secrets* can only be accessed in the *secret* compartment, and thus code in the main compartment can never disclose it. Although the untrusted code may violate the protocol, by either skipping the invoking of hypercall, or not obeying rules of *secure_malloc*, the hypervisor can detect such a violation and the *secrets* will not be loaded into memory in such cases.

Runtime. At runtime, code in the main and the *secret* compartments execute concurrently. The SeCage mechanisms ensure that: (1) the *secrets* and their copies only exist in *EPT-S* mapping, (2) the *secrets* and their copies can only be used during sensitive functions execution. If the code in the main compartment attempts to access the *secrets* memory, a VMExit happens due to EPT violation. Hypervisor is then notified to checks the access pattern. If the access request is originated from the main compartment code to the *secrets*, and the corresponding function is not included in the extracted functions from static analysis, an attack attempt may happen. In this case, the hypervisor should stop the application execution and inform the user of the abnormal access request. If this access request is complied with the predefined policies according to the result of static analysis and the execution context, the hypervisor will then include the corresponding function to the sensitive functions closure in the *secret* compartment.

Termination. When the application is terminated, *secrets* should also be wiped out. If the application exits normally, it issues the *SECAGE_RESTORE* hypercall, so that hypervisor helps to remove the *EPT-S* of *secret* compartment. Even if the application exits abnormally or the application or OS refuses to inform the hypervisor, the *secrets* still only exist in *EPT-S* and thus will not be disclosed.

4. APPLICATION DECOMPOSITION

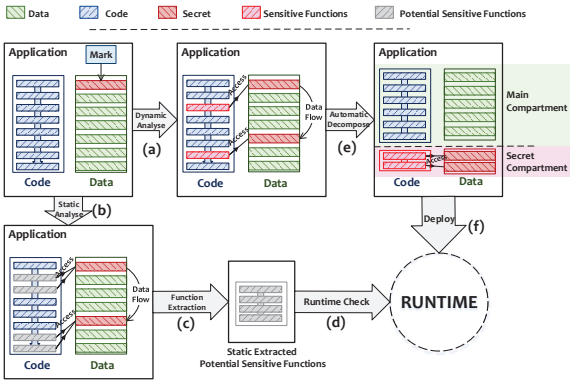


Figure 6: The general process of application analysis and decomposition

Figure 6 shows the process of the application analysis and decomposition. Given an application and the user-defined *secrets*, we need to analyze the data flow of the *secrets*, as well as the sensitive functions that are possible to access the *secret* data. While the static analysis can give a comprehensive analysis on all possible execution paths of the program, it has precision issues and may lead to larger TCB and overhead. We observe that in most cases, the execution flow of *secrets* manipulation is relatively fixed.

Based on this observation, We use a hybrid approach to extracting the *secret* closure. Specifically, we adopt the dynamic approach to achieving a flexible information flow control (IFC) analysis to get the most common but possibly incomplete *secret* closure (step (a)), and we also rely on the comprehensive results of static analysis to avoid these corner cases during runtime (step (b)(c)(d)). On the other hand, it provides a series of mechanisms to automatically decompose application during compilation time. Then, SeCage decouples these *secrets* and sensitive functions into an isolated *secret* compartment, which can be protected separately by the hypervisor (step (e)(f)).

4.1 Hybrid Secret Closure Extraction

Static taint analysis. We leverage CIL to carry out static taint analysis on the intermediate representation of the application. We denote the set of *secret* data as $\{s\}$, and the set of references to the *secrets* is donated as $\{s_{ref}\}$. Our target is to find all possible instructions, which is denoted as *sink*, that dereference variable $x \in \{s_{ref}\}$. We define the taint rules as follows:

$$\frac{n \rightarrow m, y := x, x \in \{s_{ref}\}}{\{s_{ref}\} := y :: \{s_{ref}\}} \quad (1)$$

$$\frac{n \rightarrow m, f(y_1, \ ... \ y_i, \ ... \ y_n), y_i \in \{s_{ref}\}}{\{s_{ref}\} := arg_i :: \{s_{ref}\}} \quad (2)$$

$$\frac{n \rightarrow m, y := f(y_1, ...y_i, ...y_n), ret_f \in \{s_{ref}\}}{\{s_{ref}\} := y :: \{s_{ref}\}} \quad (3)$$

$$\frac{n \to m, y := \textbf{sink}(x), x \in \{s_{ref}\}}{\{s\} := y :: \{s\}, \{s_{ref}\} := y_{ref} :: \{s_{ref}\}} \quad (4)$$

When a program transmits from n to m, the data flow is tracked. Rule (1) says that the propagation of references to *secrets* should be tracked. Rule (2) and rule (3) define the rules of function calls, which mean that the propagation of references to *secrets* through function parameters and return values should also be tracked. Finally, rule (4) indicates that, upon any dereference of references to *secrets*, the *secret* and its reference need to be tracked, and the sink instruction should be also recorded. According to polyvariant analysis [7], the functions are analyzed multiple times. In our approach, the $\{s\}$ and $\{s_{ref}\}$ keep changing during the iterations and we stop the analysis when the program comes to a fixed point, where the *secret* and its reference set does not change anymore. Through this taint analysis, we can finally get a large number of potential sensitive functions.

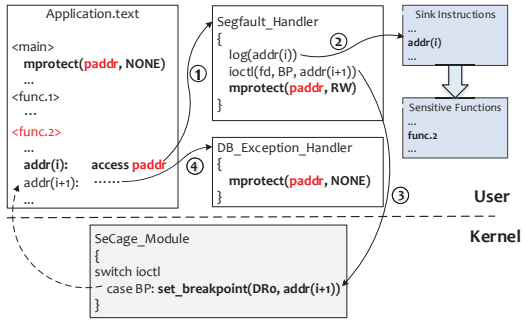


Figure 7: Combining *mprotect* and *debug exception* to dynamically extract sink instructions

Dynamic closure extraction. To get the compact *secret* closure, we adopt a simple but precise dynamic analysis by an innovative combination of *mprotect* and *debug exception* techniques. Figure 7 shows the process of dynamic sensitive functions extraction. At the very beginning, the application is instrumented to use *secure_malloc* to allocate memory for *secret* data. Then we use *mprotect* system call to protect the secure memory (*paddr*), and register a user mode handler to handle the corresponding segmentation fault upon violation. Anytime a sink instruction accesses the protected memory, it traps into the segfault handler (①). This handler records the fault address of the sink instruction (②), and issues one *IOCTL* system call to the kernel module (③), which set up the breakpoint to the next instruction in debug register 0 (DR0). After that, the handler revokes the *mprotect* to the *paddr* to make forward progress. Then, the sink instruction can successfully access the memory after the segfault handler returns; but the program traps to the predefined debug exception handler immediately in the next instruction (④), and in that case, the exception handler can setup the *mprotect* to the *paddr* again. We run the application several times with different workloads. For example, for Nginx server case, we can send different kinds of requests, until the set of sink instructions is fixed. Then we can get most of the sink instructions, as well as their corresponding sensitive functions.

For the OpenSSL case, we totally get 242 sensitive functions from static analysis and 20 from dynamic analysis. In our experiment, we select the dynamically extracted ones as the sensitive functions set $\{f_s\}$, and we do not find any EPT violation during the runtime. Nevertheless, we still decompose the potential sensitive functions which are not included in the dynamically extracted set to several separate memory sections, so that if any corner case happens, hypervisor can dynamically add them to the *secret* compartment in the runtime according to the predefined policies. When the hypervisor is required to add a potential sensitive function to the *secret* compartment, SeCage will collect as much information as possible to distinguish between legal and illegal accesses. For example, it uses the trapped instruction to check whether this is the sink instruction found in the static analysis; it is also required to check whether this potential sensitive function is modified, and whether the call-trace is forged, etc. If there is any predefined policy violated, SeCage aborts the execution and sends the abnormal information to the user to make decision.

4.2 Automatic Application Decomposition

Since the sensitive functions in $\{f_s\}$ are not self-contained, SeCage uses the trampoline mechanism to enable communications between $\{f_s\}$ and the functions in the main compartment, which is defined as $\{f_n\}$. For each function call f, we notate f_{caller} and f_{callee} as its caller and callee functions. If and only if one of the f_{caller} and f_{callee} belongs to $\{f_s\}$, it is required to define a corresponding trampoline function $t(f)$, which is used to replace f in the next phase. The formal definition is as follows:

$$P_s(func) = true \iff func \in \{f_s\}$$

$$\forall f, \exists t(f) = \begin{cases} f_{in} & P_s(f_{callee}) \wedge \neg P_s(f_{caller}) \\ f_{out} & P_s(f_{caller}) \wedge \neg P_s(f_{callee}) \\ f & \text{else} \end{cases}$$

If there is any function call from $\{f_n\}$ to $\{f_s\}$, we define a trampoline function f_{in}. Similarly, a springboard function f_{out} is defined for each function call from $\{f_s\}$ to $\{f_n\}$. We notate $\{f_t\}$ as the union of f_{in} and f_{out} sets.

Then we decompose an application to *secret* and main compartments for SeCage protection. There are totally three steps involved. First, 3 hypercalls are added to the application accordingly (Section 3.4). Second, an automated script is used to generate a file with definition and declaration of trampoline functions $\{f_t\}$, and modify the definition of sensitive functions *sfunc* in $\{f_s\}$ and trampoline functions *tfunc* in $\{f_t\}$ with GCC *section* attribute:

```
__attribute__ ((section (.se))) sfunc;
__attribute__ ((section (.tr))) tfunc;
```

Normally, the GCC compiler organizes the code into the *.text* section. The additional *section* attributes specify that *sfunc* and *tfunc* live in two particular *.se* and *.tr* sections' memory region which are isolated from memory of $\{f_n\}$. Thus, SeCage can protect them in the page granularity. Finally, during the compilation phase, the CIL parses the whole application, and replaces the $\{f_s\}$ involved function calls with their respective trampoline function calls in $\{f_t\}$.

SeCage also needs to modify the linker, to link the newly created *.se* and *.tr* sections to the predefined memory location, so that the *SECAGE_INIT* hypercall can pass the appropriate memory addresses as the *secret* compartment memory for the hypervisor to protect.

5. USAGE SCENARIOS

We have implemented the compartment isolation part of SeCage based on KVM. The hypervisor is mainly responsible for compartment memory initialization, EPT violation and interrupt handling. In total, SeCage adds 1,019 lines of C code to the KVM, and the application decomposition framework consists of 167 lines of C code, 391 Bash code and 1,293 OCaml code to the CIL framework [1].

In this section, we present our case studies by applying SeCage to three representative examples: protecting keys of Nginx server with OpenSSL support from infamous HeartBleed attack, protecting keys of OpenSSH server from a malicious OS, and protecting CryptoLoop from kernel-level memory disclosure.

5.1 Protecting Nginx from HeartBleed

The HeartBleed attack allows attackers to persistently over-read 64KB memory data, which can lead to leakage of private keys [4]. We use one of the PoCs [6] to reproduce RSA private keys disclosure attack targeted on the Nginx server with a vulnerable OpenSSL version *1.0.1f*.

In our case, OpenSSL uses RSA as its cryptographic scheme. In short, two large prime numbers (p and q) are chosen as *secrets* in RSA, while their production N ($N = p \times q$) is referred to as the public key. When a client connects to the server, the server uses p and q as private keys to encrypt data (e.g., certificate), so that the client can use the corresponding public key N to decrypt them. To steal the private keys, an attacker can search the memory data returned in HeartBleed messages for prime numbers. Since the N and the length of the prime numbers (in bits) are known, it is pretty simple to find p or q by iteratively searching the exact (e.g., 1024) number of bits and checking if they can be divided by N as a prime. Once the attacker gets either of p or q, the other is just the result of dividing N by the known prime. In addition, there is a private key exponent called d which is another prime number we need to protect.

To defend against HeartBleed attack using SeCage, we first mark the *secrets* to protect. In the OpenSSL case, the exact prime numbers are stored in the memory pointed by d field in BIGNUM structure p, q and d. During the application decomposition phase, we get all of sensitive and trampoline functions as illustrated in section 4. The *secure_malloc* function is added to replace *OPENSSL_malloc* in some cases. OpenSSL uses BIGNUM memory pools to manage BIGNUM allocation, SeCage adds another secure pool which is used to allocate BIGNUM when it is protected one. The CILLY engine then delivers the modified intermediate code to GCC, which compiles it to the final executable binary. Besides, the private keys stored in the configuration file should be replaced by the same length of dummy data. After that, the hypervisor pushes the generated binary and configuration file to the guest VM, and starts the Nginx.

5.2 Protecting OpenSSH from Rootkit

In this scenario, we assume that the kernel is untrusted, e.g., there is a malicious rootkit[4] acting as part of operating system to arbitrarily access system memory, so that the *secret* is exposed to the light of day. We simulate this scenario by manually installing a rootkit that simply scans the system's memory and tries to find out *secret* in applications.

[4]Rootkits are mostly written as loadable kernel module that can do whatever kernel can do.

We run the OpenSSH server in this untrusted environment. During the authentication of OpenSSH server, the client uses the host key and server key as server's public keys to encrypt the session key, and the private keys are used by OpenSSH server for session key decryption. The server is also required to send the acknowledge encrypted using the session key to the client for authentication. Thus the private keys are of vital importance such that the disclosure of them can lead to the leakage of all network traffic, and the attacker can pretend as server to deceive clients. Similar to the Nginx server protection, we leverage the application decomposition framework to analyze and decompose OpenSSH, since OpenSSH uses OpenSSL with RSA cryptographic schema as well, the process is quite similar with the previous example.

5.3 Protecting CryptoLoop from Kernel Memory Disclosure

As shown in Table 1, about half of the vulnerabilities are kernel-level memory disclosure. Different from application-level memory disclosure, one successful exploit of these vulnerabilities can put the whole kernel memory data at risk of being disclosed. In this scenario, we choose to enhance the Linux kernel's disk encryption module, CryptoLoop, which is used to create and manipulate encrypted file systems by making use of loop devices. CryptoLoop relies on the Crypto API in kernel, which provides multiple transformation algorithms. In our case, it uses the CBC-AES cipher algorithm to do the cryptographic operations.

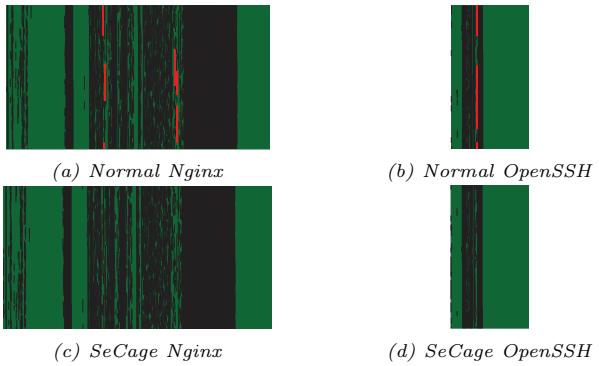
In this case, we define the AES cryptographic keys as *secrets*. The sensitive functions extraction framework is a little different, but the overall principle is the same: we allocate a new page for the AES keys, and set the corresponding page table entry as non-present. After that, we combine the page fault with the debug exception to track and record the *secrets* related functions. Different from the cases of Nginx and OpenSSH, the AES keys are originated from user-provided password with hash transformation, thus the user has to provide a dummy password, and calculate the real hash value offline, so that SeCage can replace the dummy AES keys with the real ones.

6. SECURITY EVALUATION

6.1 Empirical Evaluation

***Secrets* exposure elimination.** We evaluate to what extent can SeCage achieve the goal of exposing *secrets* to adversaries. To achieve this, we write a tool to scan the whole VM memory and find targeted *secrets* in it when running Nginx and OpenSSH server. Figure 8 shows the heap memory [5] of related processes, for the software without SeCage protection ((a) and (b)), there are several targeted *secrets* found in the heap. In contrast, for the SeCage protected ones ((c) and (d)), we cannot find any fragment of the *secrets*. For the CryptoLoop case, we run the *fio* benchmark to constantly do I/O operations in the encrypted file system, and traverse the kernel page tables to dump the valid kernel memory pages in a kernel module. With SeCage protection, we find no AES keys within the dumped kernel memory. While without SeCage protection, we can find 6 AES keys.

[5]we also dumped the stack memory, and found no *secret* at all



(a) Normal Nginx (b) Normal OpenSSH

(c) SeCage Nginx (d) SeCage OpenSSH

Figure 8: Heap memory layout for Nginx and OpenSSH server processes. (Green blocks are memory in use, black areas are unused and the red bars are targeted *secrets* found in the memory.)

Security protection for Nginx+OpenSSL. Before applying SeCage to Nginx server, we conduct HeartBleed attacks on it, it is relatively simple to retrieve the RSA private keys, we get private keys after sending 69 HeartBleed requests. After decomposing the OpenSSL and deploying it to SeCage framework, we mount the attack again. At this time, no fragment of private keys is leaked no matter how many HeartBleed requests are sent by the attacker.

Security protection for OpenSSH+OpenSSL. From figure 8 (b), we can see that 3 fragments of private keys exist in the OpenSSH process's dumped memory, which are exactly what the kernel rootkit can read and expose. While leveraging the SeCage, no one exists in the malicious rootkit's view, which means no *secret* will be leaked.

Security protection for CryptoLoop. AES cryptographic keys are stored in kernel memory space, any *Out-of-bound read* kernel memory disclosure vulnerabilities (e.g., CVE-2013-5666) and rootkit can read arbitrary memory from kernel space, leading to kernel *secrets* disclosure. With the Crypto API being enhanced by SeCage, no cryptographic key is stored in the normal kernel memory, thus no *secrets* can be leaked during CryptoLoop execution.

6.2 Security Analysis

Reduced attack surfaces. The attack surfaces of SeCage are quite related to the code size of the sensitive functions. The larger code base the *secret* compartment contains, the more vulnerabilities may be exploited by the attackers. After applying the proposed dynamic analysis, we get only 1350 LoCs for OpenSSL case, and 430 LoCs for CryptoLoop case. What is more, SeCage adds some restrictions for the sensitive functions, e.g., there should be no printf-like functions thus no format attack can be conducted, etc. Therefore, it is reasonable to assume no vulnerabilities in the code of *secret* compartment.

Iago attack and ROP attack. Iago attack [15] presents a complete attack example that the malicious kernel can cause s_server in OpenSSL to disclose its secret key by manipulating the return values of *brk* and *mmap2* system calls, and conduct a return-oriented programming (ROP) attack to write the contents of the secret key to *stderr*. With the protection by SeCage, if there is any system call invocation in the *secret* compartment, the return value from the *springboard* will be checked to avoid the invalid one.

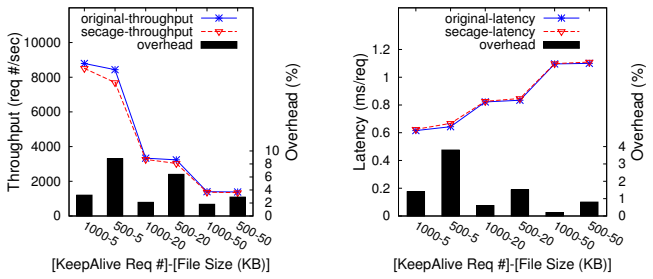
Meanwhile, trampoline code is marked as read-only thus not injectable thanks to the protection from EPT. On the other hand, there is only very small code base in sensitive functions for an attacker to exploit ROP attack. The SeCage's mechanism ensures that the trampoline can only enter *EPT-S* through designated function entries. This means gadgets can only be at the function granularity and thus attackers may not have sufficient gadgets, which further reduces the probability of ROP attacks. What is more, as the execution in the *secret* compartment will use the secure stack, an adversary has no chance to fake a stack on behalf of the *secret* compartment. This further nullifies the chance of performing an ROP attack. If an ROP attack succeeds in the main compartment, the payload still cannot access the *secrets* because of the compartment isolation.

Rollback attack. An attacker may collude with a malicious OS to rollback the password states using tricks like memory snapshot rollback, and launch a brute-force attack to guess the login password. SeCage is resistant to such an attack, due to the fact that the *secret* states are maintained by the hypervisor rather than the OS, it can refuse to rollback *secrets* to old states by well-designed protocols.

Coppersmith's attack. According to [4], when processing requests in Nginx, some fragments of private keys may exist in memory. This makes it vulnerable to Coppersmith's Attack [2], a complex mathematical approach that is used to split N into p and q with the knowledge of just part of a prime number. SeCage is designed to prevent such attacks since fragments of *secrets* can only be copied to secure heap and stack, and no fragment copies can be leaked to the memory of the main compartment.

Guest page mapping attack. A malicious guest OS can conduct attacks by changing the guest page mapping of the secure heap and the stack pages after SeCage initialization. Hence, during invoking trampoline and copying *secrets*, *secrets* may be leaked to unprotected memory space. SeCage prevents such attacks by tracking the page tables modification of these protected data pages. According to section 7, only 3 pages are required to protect, and at most 12 PTEs need to be tracked.

VMFUNC faking attack. Another attack the malicious OS can conduct is to fake the *EPT-S* entry point by invoking self-prepared VMFUNC instructions. However, in *EPT-S*, the executable memory is predefined and fixed, the memory pages which contain the faked VMFUNC instructions are non-executable when switching to the *EPT-S* context, thus the execution will be trapped into the hypervisor. A more sophisticated attack is to carefully construct a guest memory mapping that maps GVA to GPA, so that the GVA of the memory page is the prior one page before the sensitive functions pages, and put the VMFUNC instruction into the last bytes of this page. Hence, when this VMFUNC is invoked, the next program counter locates in the sensitive functions, which is executable. This complicated attack can bypass the restriction that the calls to the *secret* compartment can only be done with several fixed entry gates in the trampoline. Nevertheless, it just adds a few more harmless entry gates (the first byte of the sensitive functions pages). To further eliminate such attack surfaces, we can adapt the approach used by [48] to putting a single byte *INT3* instruction at the beginning of each page of sensitive functions, to prevent this carefully constructed VMFUNC faking attack.



(a) Throughput *(b) Latency*

Figure 9: Nginx performance degradation

7. PERFORMANCE EVALUATION

To evaluate the efficiency of SeCage, we measure the performance slowdown for applications under protection by SeCage and collect some statistics regarding the performance and resource consumption to conduct a detailed analysis.

All experiments are done on a machine with 4 Intel Xeon cores (8 hardware threads using hyper-threading) running at 3.4 GHz and with 32 GB memory. The host kernel for KVM is Linux 3.13.7 and the guest kernel is Linux 3.16.1. Each guest VM is configured with 2 virtual cores and 4 GB memory.

7.1 Application Benchmarks

Nginx throughput and latency. Figure 9 shows the performance degradation of Nginx server with SeCage protection. We use the *ab* benchmark to simulate 20 concurrent clients constantly sending N KeepAlive [6] requests, each request asks the server to transfer X bytes file. To decide what are the common values N and X should be, we scan through the Facebook pages, and find that within one session, there are normally hundreds to thousands of requests, each one ranges from 1 to 100 KB. Since the private keys are only used during connection establishment, the overhead can be largely amortized as the number of KeepAlive requests and the size of file grow. When N is 1000 and X is 50K, SeCage only adds about 0.002 ms latency for each request in average, and the throughput overhead is only about 1.8%. We also run the SeCage in a problematic configuration where all requests require new TLS establishment with 0.5KB small file, to test the performance slowdown in the worst case. The result shows that the overhead is about 40%. While in real-world scenarios, most HTTP-connections are persistent and keep-alive headers are used [5].

OpenSSH latency. To evaluate the slowdown of OpenSSH operation, we run a script to constantly use SSH to login the server and execute common Linux commands such as *ls* and *netstat*, and utilize *time* command to show the latency. We find that on average, SeCage introduces about 6 ms (%3 overhead) for each request.

CryptoLoop I/O bandwidth. We evaluate disk I/O throughput when using CryptoLoop encrypted file system. We leverage *fio* benchmark with sequential read/write configurations to show the overhead introduced by SeCage. We find that the average slowdown of I/O operations is about 4% when using SeCage protection.

[6]KeepAlive header asks the server to not shut down the connection after each request is done.

7.2 Performance Analysis

EPTP switching times and cost. The overhead introduced by SeCage is mainly from context switches between *EPT-N* and *EPT-S*. We instrument the application to log the times of trampoline invocation for different operations; each trampline invocation introduces 2 vmfunc calls (section 3.2). Table 3 shows the statistics after the Nginx server processed 1000 requests, OpenSSH handled one login request, and CryptoLoop operated 10M I/O writes. N→S means the number of trampoline calls from the main to the *secret* compartment, and S→N means that from the *secret* to the main compartment. We can see that there are limited numbers of context switches compared to the long execution path. We then do a study of how much time each vmfunc call takes, and compare it with normal function call, syscall and vmcall (hypercall). As shown in Table 4, the cost of a vmfunc is similar with a syscall, while a vmcall takes much longer time. Thus, SeCage provides the hypervisor-level protection at the cost of system calls.

Table 3: Statistics of trampoline invocation times

Nginx/1000 req		OpenSSH/login		CryptoLoop/10M	
N→S	S→N	N→S	S→N	N→S	S→N
64693	8550	20480	40960	98230	11658

Table 4: Overhead for call, syscall, vmcall and vmfunc

call	syscall	vmcall	vmfunc
3.37 ns	64.54 ns	544.4 ns	63.22 ns

Memory consumption. Since SeCage needs to setup a shadow *EPT-S* for each protected *secret* compartment, we calculate how much additional memory is needed. It depends on how many sensitive functions and trampoline code are extracted, and how much secure heap and stack is reserved. The results are shown in Table 5, in the OpenSSL case, we extracted about 20 sensitive functions, and declare the corresponding 41 trampoline functions. For the *secret* memory, we only reserve 1 page for the secure heap and 2 pages for stack. Thus the total amount of shadow memory is only 40KB. For the CryptoLoop case, since it is much simpler than OpenSSL, it only takes about 7 pages (28KB).

Table 5: Statistics regarding memory consumption

	$\{f_s\}$	$\{f_t\}$	*sec-heap*	*sec-stack*	total
Nginx	20 (5)	41 (2)	1	2	10 pages
OpenSSH	21 (5)	41 (2)	1	2	10 pages
CryptoLoop	12 (3)	14 (1)	1	2	7 pages

8. RELATED WORK

Application-level secret protection. There are many systems aiming at protecting secret data (e.g., cryptographic keys) from application-level memory disclosure attacks by leveraging library or OS kernel support [36, 8, 22, 37, 26]. For example, DieHarder [36] presents a security-oriented memory allocator to defend heap-based memory attacks. SecureHeap [8] is a patch for OpenSSL from Akamai Technology, which allocates specific area to store private keys to prevent private keys from disclosing. CRYPTON [22] designs

a data abstraction and browser primitive to isolate sensitive data within the same origin. However, they all rely on the trustworthiness of the entire library of OS kernel, and thus still suffer from larger attack surfaces compared to SeCage.

Other approaches try to keep cryptographic keys solely inside CPU [37, 26]. For example, Safekeeping [37] uses x86 SSE XMM registers to ensure no cryptographic key appear in its entirety in the RAM, while allowing efficient cryptographic computations. Similarly, Copker [26] implements asymmetric cryptosystems entirely within the CPU, without leaking the plaintext of private keys to memory. However, code running in cache needs to be specially crafted and it needs to rely on the trustworthiness of OS kernel for security.

Hypervisor-based application protection. Systems like CHAOS [17, 16], OverShadow [18] and others [19, 28, 53] leverage virtualization-based approaches to providing isolated execution environment to protect an application from a compromised OS. Specifically, they leverage hypervisor provided memory isolation, and intercept transition between a protected application and the OS to enforce isolation. InkTag [28] makes a step further to use paraverification to ease verifying of OS to defend again Iago attacks [15]. Compared to these systems, SeCage aims to provide fine-grained intra-domain protection instead of whole application protection. Further, the separation of control and data planes significantly reduces hypervisor intervention and thus reduces performance overhead.

Some other virtualization-based systems [29, 25, 45, 42] run protected applications in a trusted component with restricted functionalities (e.g., an application-specific OS), and use hypervisor to enforce the interactions between trusted and untrusted components. Compared with SeCage, such approaches have larger TCB since they need to trust all secure components like the OS. Further, they all need to retrofit the whole OS and thus require significant porting effort to run real, large software atop.

There are various systems aiming at providing isolated environment for the whole virtual machine [54, 50, 44], though they are efficient in guest VM protection, they cannot defend against intra-domain attacks.

Protecting pieces of application logic. Flicker [35] leverages TPM and takes a bottom-up approach to provide a hardware-support isolation of security-sensitive code. With the help of TPM, securely running Pieces of Application Logic (PAL) can rely only on a few hundred lines of code as its TCB. However, since Flicker heavily uses the hardware support for a dynamic root of trust for measurement (DRTM), it introduces very high overhead. TrustVisor [34] proposes a special-purpose hypervisor to isolate the execution of the PAL. Compared with Flicker, it introduces the software-based TPM called *micro-TPM (uTPM)* for each PAL, to realize a more efficient PAL protection. Flicker and TrustVisor show a kind of effectively isolated code execution with extremely small TCB, they both assume that the protected code is self-contained with predefined inputs and outputs, and heavily depend on programmers to specify sensitive functions, as well as (un)marshal parameters between trusted and untrusted mode, which kind of design makes them very difficult to adapt to existing systems. MiniBox [31] focus on providing a two-way sandbox for both PAL and OS: besides protecting PAL from malicious OS as TrustVisor does, it also prevents untrusted applications compromising the underlying OS. Since MiniBox uses TrustVisor to do the first part, they share the same problems.

Hardware-assisted protection. There is also much work on designing new hardware features to isolate critical code and data [33, 43, 24, 14, 20, 10, 47, 49]. For example, XOMOS [33] is built upon XOM [32] to support tamper-resistant software. The concept of compartment is borrowed from XOMOS, but is used to run a closure of code operating specific secret data. Further, SeCage leverages existing hardware virtualization features to enforce strong protection. Haven [10] introduces the notion of shielded execution, which utilizes Intel's recent SGX security proposal to protect the code and data of the unmodified application from divulging and tampering from privileged code and physical attack, but only targets whole application protection instead of intra-application protection. SeCage illustrates how to decompose large-scale software and adapt it to different hardware-assisted isolation environment.

Mimosa [27] uses hardware transactional memory (HTM) to protect private keys from memory disclosure attacks. It leverages the strong atomicity guarantee provided by HTM to prevent malicious concurrent access to the memory of sensitive data. Though Mimosa shares almost the same threat model with SeCage, it has some limitations, e.g., in order to prevent attackers from accessing debug registers (which is used to store the AES master key of the private keys), it needs to patch the ptrace system call, disable loadable kernel modules (LKMs) and kmem, and remove JTAG ports. Compared with SeCage, Mimosa can only adapt much simplified PolarSSL instead of OpenSSL to the HTM protection.

CODOMs [47] enforces instruction-pointer capabilities to provide a code-centric memory domain protection. CHERI [49] implements a capability coprocessor and tagged memory to enable capability-based addressing for intra-program protection. Both designs promise very small TCB (e.g., microprocessor and up to limited software component like trusted kernel), and can provide strong isolation enforced by hardware. However, since they all need newly specific designed hardware which are currently not available, they are all implemented on emulators. Further, they usually require rewriting of the whole software stack, which lead to non-trivial engineering work and make them not ready for deployment.

Privilege separation. PrivTrans [13] is a privilege separation tool that can automatically decouple a program into monitor and slave parts provided a few programmer annotations. Compared with the mechanism of SeCage, PrivTrans approach partitions code in instruction-level, which introduces a large number of expensive calls between the two parts. Meanwhile, it trusts the whole operating system, whose threat model is quite different with SeCage. Virtual Ghost [21] is the first one to combines compilation technique and OS code runtime check to provide critical application with some secure services, and protect it from compromised OS without higher privilege level than the kernel. However, it needs to retrofit OS and runs it in a secure virtual architecture (SVA), which requires non-trivial efforts to deploy it in current infrastructures.

VMFUNC utilization. SeCage is not the first system to use the Intel's VMFUNC hardware feature. Following the philosophy of separating authentication from authorization, CrossOver [30] extends Intel's VMFUNC mechanism to provide a flexible cross-world call scheme that allows calls not only across VMs, but across different privilege levels

and address spaces. We believe such a mechanism can be applied to SeCage to provide more flexible protection. An effort from the open-source community [9] takes advantages of VMFUNC, and proposes an efficient NFV on Xen. As far as we know, SeCage is the first system making uses of VMFUNC to protect pieces of application logic and prevent from both user and kernel level memory disclosure attacks.

9. DISCUSSION AND LIMITATION

Deployment effort. The deployment of SeCage is not fully automated, the minor manual effort is required. For OpenSSL and CryptoLoop, we respectively add 25 and 15 LoCs. They are mainly attributed to the dynamic analysis, *secret* memory allocation and hypercall used for initialization. Comparing it with other work, TrustVisor [34], Mimosa [27] and many others did not support OpenSSL so they resort to the much more simplified PolarSSL. PrivTrans [13] only adds 2 annotations, however it puts all cryptographic operations in the monitor, while SeCage only allows very limited code in the *secret* compartment.

SeCage TCB. For one *secret* compartment, the TCB contains hardware, hypervisor and the sensitive functions. Currently, our implementation uses KVM as the trusted hypervisor, which arguably has a large TCB. However, as SeCage only involves minor functionalities in hypervisor, it should be easy to port SeCage to a security-enhanced hypervisor such as TrustVisor [34] or XMHF [46] to further reduce TCB, which will be our future work.

The scope of *secrets*. SeCage is supposed to protect critical *secrets* like private keys, whose related sensitive functions are limited to a small piece of code. Since the small code base of a *secret* compartment is one of the most important assurance to *secrets*' security, SeCage is not designed for protecting data whose closure involves large fraction of the code. Fortunately, the three cases we studied, which should be representative of large-scale software, fit very well with our scope.

Static library vs. dynamic library. Currently, SeCage compiles OpenSSL as a static library, and slightly modifies the linker to link the sensitive and trampoline functions to specific memory locations. We can also adopt dynamic libraries, by modifying system's loader. We do not assume the loader as trusted; even if it behaves maliciously by refusing to load sensitive functions or loading them to wrong memory locations, the *secrets* will still not be disclosed. This is because the hypervisor can reject to load the *secrets* during the integrity checking phase (Section 3.4).

10. CONCLUSION AND FUTURE WORK

We presented SeCage, a novel approach that leverages virtualization to protect user-defined *secrets* from application vulnerabilities and malicious OS. SeCage prevents sensitive information disclosure by completely shadowing the *secret* compartment from the main compartment, and utilizes Intel hardware extension to enormously reduce hypervisor intervention during application runtime. Meanwhile, SeCage provides a practical application analysis and decomposition framework to (mostly) automatically deploy applications. SeCage was shown to be useful by protecting real and large-scale software from HeartBleed attack, kernel memory disclosure and rootkit memory scanning effectively, and incurring negligible performance overhead to applications.

We plan to extend our work in several directions. First, we plan to deploy more applications with SeCage to prevent sensitive memory disclosure based on the CVEs. Second, we plan to adopt binary rewriting techniques to avoid the requirement of source code rewriting. Third, we plan to implement SeCage in other virtualization platform, e.g., Xen.

11. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful comments. This work is supported in part by a research grant from Huawei Technologies, Inc., National Natural Science Foundation (61303011), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), Program for New Century Excellent Talents in University of Ministry of Education of China (ZXZY037003), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and the Singapore NRF (CREATE E2S2).

12. REFERENCES

[1] CIL. http://kerneis.github.io/cil/.
[2] Coppersmith's attack. http://en.wikipedia.org/wiki/Coppersmith's_Attack.
[3] The heartbleed bug. http://heartbleed.com/.
[4] How heartbleed leaked private keys. http://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-leaked-private-keys/.
[5] Http persistent connection wiki. https://en.wikipedia.org/wiki/HTTP_persistent_connection.
[6] Poc of private key leakage using heartbleed. https://github.com/einaros/heartbleed-tools.
[7] Polyvariance. http://en.wikipedia.org/wiki/Polyvariance.
[8] Secure heap patch for heartbleed. http://www.mail-archive.com/openssl-users@openssl.org/msg73503.html.
[9] Xen as high-performance nfv platform. http://events.linuxfoundation.org/sites/events/files/slides/XenAsHighPerformanceNFVPlatform.pdf.
[10] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.
[11] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *S&P*, 2014.
[12] E. Bosman and H. Bos. Framing signals: A return to portable shellcode. In *S&P*, 2014.
[13] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Usenix Security*, 2004.
[14] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.
[15] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, 2013.
[16] H. Chen, J. Chen, W. Mao, and F. Yan. Daonity–grid security from two levels of virtualization. *Information Security Technical Report*, 12(3):123–138, 2007.
[17] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, and W. Mao. Tamper-resistant execution in

an untrusted operating system using a virtual machine monitor. *Technical Report, FDUPPITR-2007-0801*.

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

[19] Y. Cheng, X. Ding, and R. Deng. Appshield: Protecting applications against untrusted operating system. *Technical Report, SMU-SIS-13*, 2013.

[20] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. Secureme: a hardware-software approach to full system security. In *ICS*, 2011.

[21] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: protecting applications from hostile operating systems. In *ASPLOS*, 2014.

[22] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting sensitive web content from client-side vulnerabilities with cryptons. In *CCS*, 2013.

[23] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, et al. The matter of heartbleed. In *IMC*, 2014.

[24] J. S. Dwoskin and R. B. Lee. Hardware-rooted trust for secure key management and transient trust. In *CCS*, 2007.

[25] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[26] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with private keys without ram. In *NDSS*, 2014.

[27] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*, 2015.

[28] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. 2013.

[29] P. C. Kwan and G. Durfee. Practical uses of virtual machines for protection of sensitive user data. In *ISPEC*. 2007.

[30] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. Reducing world switches in virtualized environment with flexible cross-world calls. In *ISCA*, 2015.

[31] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *ATC*, 2014.

[32] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. 2000.

[33] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, 2003.

[34] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *S&P*, 2010.

[35] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *EuroSys*, 2008.

[36] G. Novark and E. D. Berger. Dieharder: securing the heap. In *CCS*, 2010.

[37] T. P. Parker and S. Xu. A method for safekeeping cryptographic keys from memory disclosure attacks. In *Trusted Systems*. 2010.

[38] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DDSN-W*, 2011.

[39] G. Smith, C. E. Irvine, D. Volpano, et al. A sound type system for secure flow analysis. Journal of Computer Security, 1996.

[40] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *S&P*, 2013.

[41] R. Strackx, B. Jacobs, and F. Piessens. Ice: a passive, high-speed, state-continuity scheme. In *ACSAC*, 2014.

[42] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS*, 2012.

[43] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS*, 2003.

[44] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *ASPLOS*, 2012.

[45] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[46] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *S&P*, 2013.

[47] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: protecting software with code-centric memory domains. In *ISCA*, 2014.

[48] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *CCS*, 2012.

[49] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, et al. The cheri capability model: Revisiting risc in an age of risk. In *ISCA*, 2014.

[50] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *HPCA*, 2013.

[51] Y. Xia, Y. Liu, H. Chen, and B. Zang. Defending against vm rollback attack. In *DCDV*, 2012.

[52] Y. Xia, Y. Liu, H. Guan, Y. Chen, T. Chen, B. Zang, and H. Chen. Secure outsourcing of virtual appliance. *IEEE Transactions on Cloud Computing*, 2015.

[53] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[54] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

[55] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, 2012.