In-process Memory Isolation Using Hardware Watchpoint

Jinsoo Jang
KAIST
jisjang@kaist.ac.kr

Brent Byunghoon Kang
KAIST
brentkang@kaist.ac.kr

ABSTRACT

Memory disclosure vulnerabilities have been exploited in the leaking of application secret data such as crypto keys (e.g., the Heartbleed Bug). To ameliorate this problem, we propose an in-process memory isolation mechanism by leveraging a common hardware-feature, namely, hardware debugging. Specifically, we utilize a watchpoint to monitor a particular memory region containing secret data. We implemented the PoC of our approach based on the 64-bit ARM architecture, including the kernel patches and user APIs that help developers benefit from isolated memory use. We applied the approach to open-source applications such as OpenSSL and AESCrypt. The results of a performance evaluation show that our approach incurs a small amount of overhead.

1 INTRODUCTION

Memory disclosure vulnerabilities [17] have been remotely or locally exploited by attackers for the leakage of secret data. For instance, the Heartbleed vulnerability (CVE-2014-0160) [4] persistently reads up to 64 KB of process memory, leading to the leakage of a crypto key for the server process. Unfortunately, process-level isolation given by the OS is insufficient to prevent such an attack because the attacker can take advantage of the lack of in-process memory isolation.

Previous studies have proposed several ways to enable in-process memory isolation and thus mitigate the vulnerability to a memory disclosure. Shred [14] provides programming primitives that help developers define and fulfill access control of a critical memory region. SeCage [20] also enables isolating a secret compartment from the remaining process compartments by leveraging a hardware-assisted virtualization technique. Although these approaches effectively isolate critical regions, their adoption is generally difficult because previous works have required specific hardware components, the availability of which is dependent on the hardware architecture. For example, Shreds uses the memory domain and the domain access control register (DACR), which are only supported in the 32-bit ARM architecture. SeCage leverages a VMFUNC instruction, which is part of the virtual-machine extensions (VMX) on x86.

In this paper, we propose an architecture agnostic approach that aims at realizing in-process memory isolation. We leverage the hardware watchpoint to create secure memory on a thread basis. This basically exploits the fact that configuring the watchpoint for a certain memory region with read and write monitoring capability allows access control to that region, and thus readily builds an isolated region. Further, most importantly, the watchpoint can be configured on each processor independently, which facilitates the design of thread-based security mechanisms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317843

Our prototype of watchpoint-based security application is implemented on a Juno development board [7] using Linaro Release 17.10 [9]. Because the watchpoints need to be configured with privileged mode, we patch the Linux kernel to insert watchpoint configuration operations in the kernel exit code. Moreover, we additionally implement a user library and a kernel driver to provide watchpoint configurability to user applications. In the performance evaluation, the proposed approaches are found to impose a small overhead. A maximum overhead of 5% is imposed for OpenSSL applied with in-process secure memory.

2 BACKGROUND AND RELATED WORK

2.1 Self-hosted debugging

Commercial processor architectures, such as ARM and x86, generally support two types of debug mechanisms: (1) external and (2) self-hosted (internal) debugging. External debugging refers to performing debugging using external hardware equipment, such as a JTAG debugger. Self-hosted debugging exploits debug-related exceptions raised by a processor. The exceptions are caught and handled by privileged software, such as the OS kernel. Several debug facilities are provided by processors to support self-hosted debugging. For instance, breakpoint registers allow a breakpoint address to be set for an instruction execution that needs to be trapped by raising a breakpoint exception. By using watchpoint registers, access to a certain memory region can be configured to incur a watchpoint exception.

The aforementioned debugging facilities have been leveraged in various ways. GDB [3] is a well-known user-level software debugger that can set hardware breakpoints (and watchpoints) by invoking a `ptrace` system call that configures breakpoint registers. Ninja [22] provides a stealthy malware analysis framework by combining the processor debug features and security extensions (i.e., ARM TrustZone). Ether [16], a hypervisor-based malware analyzer, exploits the hardware trap flag to monitor every single instruction execution. Kivati [15] and RaceHound [10] adopt hardware watchpoints to detect atomicity violations. Previous works have mainly used debug facilities for offline analysis and tracing of applications. By contrast, here, we show how to exploit debug facilities, particularly hardware watchpoints, for runtime attack prevention.

2.2 In-process memory protection

Various ways have been proposed to protect applications. As hypervisor-based approaches, TrustVisor [21] and Inktag [18] enable developers to isolate critical logic in the hypervisor-protected memory.

Similarly, ARM TrustZone [2] and Intel SGX [6] provide security hardware primitives that can create a trusted execution environment in which the developers can deploy critical resources, such as crypto key and services.

However, despite the strict isolation between the critical and non-critical domains, such techniques remain vulnerable to attacks that are performed inside the protected region owing to the coarse protection granularity (e.g., heartbleed attack [4]). To address this problem, SeCage [20] was proposed as a hypervisor-based approach that separates an application into several compartments, and isolates and protects them by creating exclusive extended page table (EPT) mappings for each compartment. Further, VMFUNC [5] has been leveraged to minimize the performance overhead during EPT changes. On ARM, Shreds [14] exploits the memory domain [1] to provide fine-grained in-process private memory. Although SeCage and Shreds address important problems, it is difficult to generally apply them to high-end devices for the following reasons: (1) VMFUNC is only available on x86 and (2) the memory domain is obsolete on 64-bit ARM architecture.

3 WATCHPOINT ON ARM

Because our prototype is implemented on ARM, we discuss ARM architecture-based watchpoint. To generate the watchpoint exception, the relevant watchpoint registers need to be configured along with the setting of the monitor debug events (MDE) flag in the monitor debug system control register (MDSCR_EL1). With ARM, there are two types of watchpoint registers: the debug watchpoint value register (DBGWVR(n)_EL1, n = 0-15) and the debug watchpoint control register (DGBWCR(n)_EL1, n = 0-15). The watchpoint value register (DBGWVR(n)_EL1, n = 0-15) sets up the starting address of the monitoring. The watchpoint control register (DGBWCR(n)_EL1, n = 0-15) comprises important attributes such as the monitoring size and activation control. The monitoring size is determined by referencing the BAS and MASK flags. For instance, the MASK flag is leveraged to monitor the address range, the size of which is a power of 2; the minimum and maximum sizes are 8 bytes and 2 GB, respectively. The two types of registers with the same index (n) should be configured together to properly set up the watchpoint. According to ARMv8 [1], the number of watchpoint pairs can be as high as 16. However, in our development board [7], we have four available watchpoint register pairs, i.e., from (DBGWVR0_EL1, DBGWCR0_EL1) to (DBGWVR3_EL1, DBGWCR3_EL1). Most importantly, the watchpoint registers are banked for each processor. This enables us to design watchpoint-based solutions on a thread basis.

Configuration requirement. The watchpoint configuration should strictly comply with the monitoring size and the address-alignment requirements. The starting address of the monitoring, the size of which is less than or equal to 8 bytes, needs to be aligned with a word or doubleword. For a size of larger than 8 bytes, the monitoring size should be a power of 2, and the starting address of the monitoring should be aligned with that size. If this requirement is not satisfied, the watchpoints will not be activated.

4 ATTACK MODEL

We assume that an application can encompass a vulnerability in which arbitrary process memory is leaked. The attacker's goal is exfiltrating secret data by exploiting the vulnerability. However, we do not assume a kernel-privileged attacker. *If the kernel is already compromised, exploiting the memory disclosure vulnerability of an application will not be necessary.* Therefore, we trust the OS kernel as the trusted computing base (TCB) of our proposed mechanism.

5 IN-PROCESS SECURE MEMORY

To address in-process abuse, such as a heartbleed attack, we designed a method to provide in-process secure memory using hardware watchpoints.

5.1 Secure memory creation

As discussed in Section 3, the watchpoints are configurable on a per CPU basis. By exploiting this feature, we can create the in-process secure memory by enabling watchpoint read/write monitoring for a certain memory area. Once the monitoring is enabled, any access to the area causes the watchpoint exception, which can be caught and verified by the kernel. We exploit this property to conduct access control to the area. The monitoring will be disabled only when the legitimate (allowed) code accesses the area (we discuss access control to the area in the following section).

As shown in Figure 1, a simple approach whereby each secure memory slot is dynamically and thus sparsely allocated might limit the number of creatable slots to be equal to the number of watchpoints, which is SoC-dependent and four in our case (note that we assume that the minimum slot size is 8 bytes here). Further, Figure 2 shows that more than one watchpoint is required to protect a slot if the size or starting address of the slot is not aligned with a power of two. Thus, to maximize the number of possible slots, we allocate the secure memory slots from a reserved linear memory area. In addition, we ensure that the size and address of the slots are aligned with a power of two and that the sizes of all slots are equal.

With this allocation strategy, the maximum number of slots (N_{slot}) should be carefully chosen to ensure that the following conditions are satisfied: (1) the entire slot size should be covered by the available watchpoints and (2) when a certain slot is used (i.e., not monitored by the watchpoint), the remaining slots should be monitored by configuring the available watchpoints. To satisfy (1), the size of the reserved area should be within N_{wp} (number of supported watchpoints) * 2 GB (maximum monitoring size of each watchpoint), which is 8 GB in our system. Further, the size of certain area (a group of continuous slots) that is monitored by each watchpoint should be a power of two. As each slot size is a power of two, the number of slots to be monitored by each watchpoint should also be a power of two. This implies that N_{slot} is a sum of powers of two.

Satisfying (2) is more restrictive than satisfying (1) because of the watchpoint configuration requirement and the fact that size of the remaining slots under a certain watchpoint monitoring is not a power of two when an open slot exists. As can be seen in Figure 3, finding the target open slot can be regarded as continuing to bisect the memory area that includes the slot until the size of

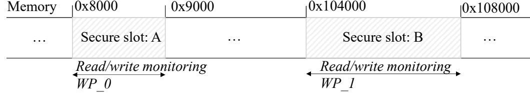

Figure 1: Sparse slot allocation limits the number of creatable secure slots to be equal to the number of available watchpoints.

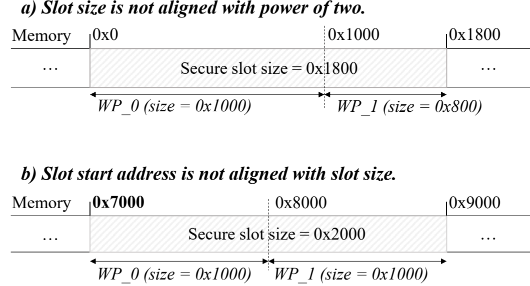

Figure 2: More than two watchpoints are required to protect a secure slot (a) if the size of the slot is not aligned with a power of two or (b) the starting address of the slot is not aligned with the size of the slot.

the bisected area equals the slot size, which is similar to binary search. The bisection can be conducted as many times as the number of watchpoints (N_{wp}), as one of the bisected memory areas that does not include the target slot in each step needs to be monitored by a watchpoint. This implies that N_{slot} should be $2^{N_{wp}}$ (N_{wp}: number of supported watchpoints), which is 16 in our development environment supporting four watchpoints. Besides, each bisected area should be monitored by a single watchpoint. Thus, the size of the first bisected area, which is the largest one, should be within 2 GB. As a result, the size of each slot (S_{size}) can be as large as 2 GB/(half of N_{slot}), which is 256 MB (2 GB/8) in our case.

Algorithm 1: setupWP() is recursively invoked to configure the watchpoints with the aim of monitoring every slot other than the target slot.

```
1  setupWP (start, end, targetSlot, slotSize);
2  /* start, end, targetSlot are virtual addresses */
3  mid = (end − start) / 2;
4  if mid <= targetSlot then
5      configWPRegister(start, mid);
6      if (end − mid)! = slotSize then
7          setupWP(mid, end, targetSlot, slotSize);
8      end
9  else
10     configWPRegister(mid, end);
11     if (mid − start)! = slotSize then
12         setupWP(start, mid, targetSlot, slotSize);
13     end
14 end
```

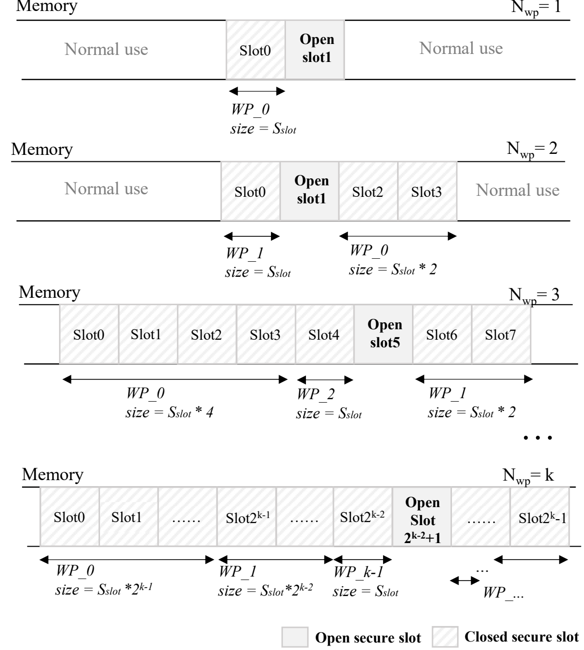

Figure 3: Total number of slots can be as high as $2^{N_{wp}}$ when we assume that the size of each slot is aligned with a power of two and all slot sizes are equal.

5.2 Access control

For access control of the secure slot, the watchpoint should properly be configured on the basis of access to the slot. We assume that the critical code region that is allowed to access the slot is verified in advance and thus the possibility of this region containing a vulnerability is minimized. Note that this assumption is generally found in previous works that aim to provide an isolated and a secure memory region [14, 18, 20, 21]. Before executing the critical code, we open the slot so that it is accessible. This is achieved by configuring the watchpoints such that every slot other than the slot to be accessed is monitored. We configure the watchpoints by recursively searching for a group of slots that is monitored by each watchpoint, as shown in Algorithm 1. First, we bisect the entire reserved memory. One of the bisected areas that does not contain the target slot is monitored by configuring the watchpoint. For the other area of the bisection that contains the target slot, we recursively run the algorithm until the last one remaining is the target slot. As discussed in Section 5.1, because the total number of slots is $2^{N_{wp}}$, N_{wp} recursions are required to configure all the watchpoints.

Exit from a certain critical region is required to protect (monitor) the secure slot associated with the region by reconfiguring the watchpoint. Compared to the entry, this procedure is quite simple. We enable watchpoint read/write monitoring for the entire reserved memory, the size of which is $S_{slot} \times N_{slot}$. In our prototype, the maximum size of the reserved memory can be as large as 4 GB (256 MB * 16) because S_{size} is limited to 256 MB. Therefore, a

Table 1: User library for in-process secure memory creation.

API	Description
initSlotAll (int slotSize)	Reserves a memory with size = $S_{slot} \times N_{slot}$
enterCriticalRegion (int slotNum)	Makes the specified slot accessible
exitCriticalRegion (int slotNum)	Makes all slots not accessible
wp_malloc (int size, int slotNum)	Allocates a heap memory in the specified slot
wp_free (void *p, int slotNum)	Frees an allocation in the specified slot

configuration of at most two watchpoints each monitoring 2 GB is sufficient for this purpose.

5.3 Components and usage

In this section, we present the core user library and kernel component designed for realizing the creation of in-process secure memory. In addition, a simple usage example is presented.

User library. We created a user library that provides five APIs to support watchpoint-based in-process secure memory creation. *initSlotAll* calculates the entire memory size required for allocating all slots (S_{slot} x N_{slot}) and reserves the memory. Before the calculation, the input parameter slotSize is rounded up to a power of two. The reserved memory is also size-aligned by internally using a `memalign` API. Note that the API can be amended to enable the configuration of the reserved slot number (based on the power of 2) in case the total number of required slots is fewer than N_{slot}. *enterCriticalRegion* and *exitCriticalRegion* generate the value of each watchpoint control and value register for opening and closing the corresponding slot that is indicated by the slotNum parameter. The generated values are delivered to the kernel driver for the actual watchpoint register settings. In addition, *enterCriticalRegion* randomizes the current stack address by subtracting a random value prior to the execution of the function prologue. *exitCriticalRegion* restores the original stack address after the function epilogue is completed. *wp_malloc* and *wp_free* can be used in the critical code region, which allocates and frees the memory in the corresponding slot, respectively.

Kernel patch & driver. Because the watchpoint-related registers are only configurable with kernel privilege, we patch part of the kernel and create a kernel driver. The kernel driver communicates with user APIs, such as *enterCriticalRegion* and *exitCriticalRegion*, through an `ioctl` system call. The driver just obtains the values of watchpoint-related registers from the user space and copies them into *sec_thread_mem* data structure, which is inserted in the `thread_info` Linux data structure as part of our implementation. The copied values are referenced and set in the watchpoint registers later when the mode switches from kernel to user. For adding *sec_thread_mem* and setting the watchpoint register on user mode entry, the Linux kernel source is patched.

Listing 1: Example of API usage in AESCrypt.

```
1  int main(int argc, char *argv[]){
2  /* Variable initialization */
3  ...
4  unsigned char *pass_input;
5  unsigned char *pass;
6  ...
7
8  initSlotAll(0x2000);    // Memory reservation. Each
       slot size is 0x2000.
9  enterCriticalRegion(0);   // Open the slot #0.
10 pass_input = (unsigned char *) wp_malloc(MAX_PASSWD_BUF
       , 0); // Malloc in the slot #0.
11 pass = (unsigned char *) wp_malloc(MAX_PASSWD_BUF, 0);
12 exitCriticalRegion(0);    // Close the slot #0.
13 ...
14
15 /* Processing an input parameter */
16 if (passlen == 0) {
17 ...
18 enterCriticalRegion(0);
19 passlen = passwd_to_utf16((unsigned char*) optarg,
       strlen((char *)optarg), MAX_PASSWD_LEN, pass);
20 exitCriticalRegion(0);
21 ...
22 }
23
24 /* Encrypting the input stream */
25 if (mode == ENC) {
26 ...
27 enterCriticalRegion(0);
28 rc = encrypt_stream(infp, outfp, pass, passlen);
29 exitCriticalRegion(0);
30 ...
31 }
32
33 /* Cleanups */
34 ...
35 enterCriticalRegion(0);
36 memset(pass, 0, MAX_PASSWD_BUF);
37 wp_free(pass, 0);   // Free in the slot #0.
38 wp_free(pass_input, 0);
39 exitCriticalRegion(0);
40 return rc;     // End of main.
41 }
```

Usage example. We used the APIs to secure an open source file encryption application, namely AESCrypt (Listing 1). We protect password inputs by locating them in the in-process secure memory. `pass_input` and `pass` variables are allocated in secure slot #0. Note that an arbitrary slot number (within 15) can be used depending on the application design (e.g., allocating an exclusive slot for each individual thread). These variables are stack variables in the original source code but we change them to heap variables by using the *wp_malloc* API. We invoke the *enterCriticalRegion* and *exitCriticalRegion* APIs before and after calling subroutines such as `passwd_to_utf16` and `encrypt_stream` because these functions need to access the protected variables. Finally, before exiting from the main function, we free the allocations in the secure slot by using the *wp_free()* API. Note that our aim is to show the feasibility of creating the in-process secure memory using watchpoints. Thus, we simply instrumented code with a coarse-grained definition of the critical region, i.e., function granularity. However, we expect that the granularity can be made finer by adopting previous approaches for privilege separation [12]. In addition, the security of critical code can be enhanced by compiler-based code instrumentation techniques (e.g., secret leakage prevention) [14].

5.4 Compatibility

The `ptrace` system call supports user interfaces for configuring debug-related registers including watchpoints. Hence, an attacker can abuse a system call to corrupt the watchpoint configuration for in-process secure memory protection. We give higher priority to our security solution in the watchpoint configuration. Thus, if the watchpoints are already used for user-level security, the `ptrace` system call is ignored. We argue that this is a reasonable approach

Table 2: Performance and LoC of open source applications hardened with in-process secure memory (in μs).

Application	Original	Hardened	LoC
OpenSSL	706.2	742.1 (1.05×)	35
Minizip	1708.3	1745.8 (1.02×)	22
AESCrypt	113516.4	113973.9 (1.00×)	33

because debuggers such as GDB rarely contain secret data, and are thus not expected to need such an applied security scheme (e.g., in-process secure memory).

6 IMPLEMENTATION

Our example applications were implemented on a Juno ARM reference board equipped with Cortex-A57 and Cortex-A53 multicore processors and 8 GB of DDR3 memory. It supports ARMv8 (64-bit architecture) and provides four hardware watchpoints.

The four watchpoints are fully utilized to create up to 16 secure slots per thread. The kernel driver is built as a loadable kernel module (LKM), which creates a device file, /dev/secthreadmem. In addition, the `kernel_exit` macro in `entry.S` is patched by inserting watchpoint setup operations to enable protection when switching to user mode. Finally, the user library is built as a shared library (.so); thus, it can be dynamically linked to an application. The library maintains an array that stores 16 individual memory allocation statuses to support *wp_malloc* in each slot. For the stack base address randomization conducted by *enterCriticalRegion*, we use the 14 least-significant bits of the processor cycle counter value as the randomization entropy. Because our system is enabled with a stack alignment check that forces the stack address to be quadword aligned, we shift the stack-based address by a multiple of 16 bytes.

7 SECURITY ANALYSIS

The robustness of our solution depends on the integrity of the watchpoint configuration. Because the watchpoint configuration is a privileged operation, any user privilege attack that attempts to directly manipulate the configuration will fail. Any indirect attempt to disable the configuration such as abusing our APIs (e.g., *enterCriticalRegion*) can be defeated by checking the call sites of the APIs in a runtime. The metadata of legitimate call sites can be created during compile time. The stack used by the critical region is also obfuscated by the address randomization approach, the entropy of which is 2^{14}, making it difficult for an attacker to find the footprint of the critical region (this can also be enhanced by zero masking the stack used). Any vulnerabilities existing in the critical region might neutralize the benefit of the secure memory. For example, a memory corruption vulnerability encompassed by the critical code might exfiltrate the secret data to non-protected memory. Several mitigations such as privilege separation [13] and critical code instrumentation [14] have been suggested to address this problem. Adopting similar approaches will improve the efficacy of our approach.

8 PERFORMANCE EVALUATION

We first measure the performance of APIs for the in-process secure memory creation. We then apply the APIs to three open source

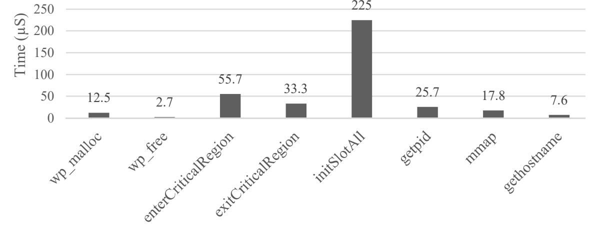

Figure 4: Performance of in-process secure memory APIs and libc functions (in μs).

applications and assess the overhead incurred by adopting a new security feature.

API performance. We compare each API execution time with generic libc functions, such as `getpid` and `mmap`. The results are shown in Figure 4. Among our APIs, *initSlotAll* requires the longest execution time, which is approximately 9× that of `getpid`. This is because the operations conducted by *initSlotAll* are more complex than those of other APIs; it rounds up the requested slot size to a power of 2, invokes `memalign` to reserve memory for all slots, and initializes the flags for managing the memory allocation in each slot. The elapsed times for *enterCriticalRegion* and *exitCriticalRegion* are much shorter than that for *initSlotAll* but longer than those for the reference libc functions. They both invoke `ioctl` system calls to communicate with our kernel driver to update the *sec_thread_mem*. In addition, *enterCriticalRegion* generates each watchpoint configuration value on the basis of the algorithm shown in Algorithm 1. *wp_malloc* and *wp_free* are faster than other APIs. They internally handle the memory allocation and freeing in each slot. Because the allowed memory operation size in each slot is fixed and is limited to the size of the initialized slot, additional system calls such as `sbrk` are not required.

End-to-end test. We applied our solution to three open source applications: OpenSSL, Minizip, and AESCrypt. Using the in-process secure memory, we protect the password usage in each program. Minizip allows a user to encrypt a compressed file with an input password. Similarly, AESCrypt uses an input password to proceed with file encryption and decryption. OpenSSL also provides a functionality that encrypts the input (e.g., private key) using a password. We slightly change the applications so that the password is isolated in the secure slot and is accessible only when the legitimate code associated with the slot is being executed. The lines of changed code are shown in Table 2.

To evaluate the overhead incurred by activating the secure memory, we compare the applications with and without protection. For Minizip and AESCrypt, we measure the entire execution time for the completion of each task, i.e., file compression and encryption, respectively. For OpenSSL, we measure in particular the time for encrypting the private key using a user's password with the 3DES algorithm, which can be conducted as part of a private key creation task. The key creation time can be varied depending on a random seed value; we simply exempt the time in our comparison (note that the *initSlotAll* API is invoked in the key creation part and thus the API execution time is not included in the OpenSSL evaluation). Table 2 shows the result. A maximum overhead of 5% was imposed for

OpenSSL with secure memory. However, the overhead decreased to zero for AESCrypt, which runs longer than any other applications. Considering this tendency, we expect the overhead for OpenSSL to decrease if we include the key creation time.

Note that our implementation does not include the instrumentation of a legitimate (critical) code that accesses the secure memory. Thus, applying additional protection techniques such as control flow integrity (CFI) to the part of the application could degrade the performance to a certain extent. In addition, the granularity of the critical region that affects the frequency of API invocation will influence the performance of a hardened application. The relationship between the granularity and performance will be explored in our future work.

9 DISCUSSION

Limited number of watchpoints. The feasibility of the proposed security applications depends on the availability of the watchpoints. According to our investigation into ARM processor reference manuals, modern ARM 64-bit processors for high-end devices (e.g., Cortex-A series) provide four watchpoints. Because the semantics of the watchpoints in terms of their configuration and operation are the same for all processor versions, our analysis of the watchpoints and the proposed design are scalable for devices equipped with such processors.

The limited number of watchpoints can also influence the effectiveness of our solutions. For the in-process secure memory, we can provide only 16 secure slots that are smaller than those provided in previous works [14, 20]. However, more watchpoints may be required depending on the application or OS type. When the watchpoints are insufficiently provided, we might need to hybridize other hardware or software primitives. For instance, the secondary page table can be used together with watchpoints to provide more than 16 secure slots, although it is expected to incur a much higher overhead than that incurred when the watchpoints are used.

Compromised kernel. Although we trust the kernel in our prototype design, the mechanism can be readily enhanced to protect a secure region even in the presence of an untrusted kernel. To realize this, we can adopt a virtualization technique to trap and emulate the privileged operations, such as the watchpoint configuration. Modern mobile devices are already utilizing this technique (i.e., instruction trap and emulation) to protect the OS kernel [11, 23]. Therefore, we expect the coordination between the watchpoint and virtualization technique to require minimal engineering effort.

In-process memory isolation on x86. Our approach is compatible with x86, which also supports the hardware debug features (i.e., DRx registers [19]). However, because of the monitoring range constraint, which is maximal 8 bytes per watchpoint, only secrets with a small size (e.g., crypto key) could be protected. To protect a larger amount of memory, we expect the memory protection key (MPK) [8, 14] to be a convincing hardware primitive to realize in-process memory isolation.

10 CONCLUSION

We presented an in-process memory isolation mechanism using hardware watchpoints to protect critical data from the vulnerability of a memory disclosure. The isolated memory region can be created by configuring the watchpoints for read and write accesses to a particular memory region. Because the watchpoints are banked for each core, the access control to the protected region can be conducted on a thread basis. The performance evaluation demonstrated that the overhead in adopting the watchpoints is insignificant.

ACKNOWLEDGMENTS

This work was supported by the NRF (NRF-2017R1A2B3006360), IITP (IITP-2017-0-01889), and ONR (N00014-18-1-2661) grants.

REFERENCES

[1] 2018. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. (May 2018). https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[2] 2018. ARM Security Technology - Building a Secure System using TrustZone Technology. (May 2018). http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[3] 2018. GDB: The GNU Project Debugger. (May 2018). https://www.gnu.org/software/gdb/

[4] 2018. The Heartbleed Bug. (May 2018). http://heartbleed.com/

[5] 2018. Intel 64 and IA-32 Architectures Software Developer's Manual. (May 2018). https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf

[6] 2018. Intel Software Guard Extensions (Intel SGX). (May 2018). https://software.intel.com/en-us/sgx

[7] 2018. Juno ARM Development Platform SoC. (May 2018). https://www.arm.com/files/pdf/DDI0515D1a_juno_arm_development_platform_soc_trm.pdf

[8] 2018. Memory protection keys. (May 2018). https://lwn.net/Articles/643797/

[9] 2018. Old release notes. (May 2018). https://community.arm.com/dev-platforms/w/docs/226/old-release-notes

[10] 2018. Racehound. (May 2018). https://github.com/kmrov/racehound

[11] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: real-time kernel protection from the ARM trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.

[12] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments.. In *NSDI*, Vol. 8. 309–322.

[13] David Brumley and Dawn Song. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*. 57–72.

[14] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained Execution Units with Private Memory. In *Security and Privacy, 2016. SP 2016. IEEE Symposium on*. IEEE.

[15] Lee Chew and David Lie. 2010. Kivati: fast detection and prevention of atomicity violations. In *Proceedings of the 5th European conference on Computer systems*. ACM, 307–320.

[16] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 51–62.

[17] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. ACM, 475–488.

[18] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. 2013. Inktag: secure applications on an untrusted operating system. *ACM SIGPLAN Notices* 48, 4 (2013), 265–278.

[19] Prasad Krishnan. 2009. Hardware Breakpoint (or watchpoint) usage in Linux Kernel. In *PROCEEDINGS OF THE LINUX SYMPOSIUM*. Citeseer, 149–158.

[20] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1607–1619.

[21] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 143–158.

[22] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards transparent tracing and debugging on arm. In *26th USENIX Security Symposium (USENIX Security 17)*.

[23] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 335–350.