Article

Fine-Grained Isolation to Protect Data against In-Process Attacks on AArch64

Yeongpil Cho

School of Software, Soongsil University, 06978 Seoul, Korea; ypcho@ssu.ac.kr

Received: 8 December 2019; Accepted: 24 January 2020 ; Published: 1 February 2020

Abstract: In-process attacks are a new class of attacks that circumvent protection schemes centered around inter-process isolation. Against these attacks, researchers have proposed fine-grained data isolation schemes that can protect sensitive data from malicious accesses even during the same process. Their proposals based on salient hardware features, such as ARM[®] processor architecture's domain protection, are quite successful, but it cannot be applied to a specific architecture, namely AArch64, as this does not provide the same hardware features. In this paper, therefore, we present Sealer, a fine-grained data isolation scheme applicable in AArch64. Sealer achieves its objective by brilliantly harmonizing two hardware features of AArch64: The eXecute-no-Read and the cryptographic extension. Sealer provides application developers with a set of application programming interface (API) so that the developers can enjoy the fine-grained data isolation in their own way.

Keywords: isolation; eXecute-no-Read; AArch64

1. Introduction

Many software attacks on a running program aim at exposing or manipulating security sensitive data in the memory, e.g., cryptographic keys, password, and certificates. They achieve their objectives by exploiting bugs or vulnerabilities in a program that enables them to obtain the same access capabilities as the victim program holding sensitive data. Sadly, the perfect defence against those attacks is extremely difficult because even carefully designed programs may contain many defects in proportion to the code size, thus eradicating all exploitable defects from a program is quite difficult [1]. In fact, formal verification methods [2] have been developed for this purpose, but the wide adoption of these methods, particularly in legacy programs written in C/C++, is still unmet. In the context of fighting with these real threats, the principle of least privilege [3] is deemed by software developers as an important design consideration for strengthening the protection of their sensitive data. The principle mandates that in a program, every code section must be allowed to access only the data required for its legitimate purpose. As a rational and effective solution to implement this principle in the field, researchers have proposed various data isolation schemes [4–6] which separate sensitive data physically from all the others in a program, and bind them with a small code section that owns the sole authority to access them. To realize data isolation, a developer first reserves a private region in memory that is by default accessible from the authorized code alone, and then places sensitive data in the region, hence in effect protecting the data against all illegal accesses from the rest of the program.

Early schemes for data isolation relied on the memory isolation mechanisms supported by operating systems (OS) that operate at the process level. In an OS, each process has its own page table that maps the process's code and data regions into a separate address space, wherein it can maintain a private region that has its access from other processes blocked by the OS. Therefore, as long as the OS is trusted, it is guaranteed that accesses to sensitive data in a process are permitted only to the code of the same process. Unfortunately, as software attacks are ever advancing and becoming

sophisticated, they always find their ways to use vulnerabilities inside a victim process and execute its code to access sensitive data belonging to the process under attack [6]. Obviously, to cope with this class of attacks labelled as in-process attacks [7], a developer demands a new scheme for fine-grained data isolation that can grant data access not to the victim's entire code (as done in the process-level isolation) but exclusively to a small section of it. However, in previous work [4,5], researchers opted for schemes that resort to the OS's existing process-level isolation support. At the compiling time, they first partition or split the original program in a way to separate the sensitive data and a segment of code authorized for access to the data. Then at runtime, they isolate the pair of those separated code and data from the rest of the program by creating a new process (or thread) dedicated to them. The problem of utilizing this process-level isolation against in-process attacks is that running a single program would be expensive. To explain this, note that in these schemes, more isolated processes tend to be forked for one running program proportionally as more distinct pairs of sensitive data and code are separated from the program. At runtime, whenever one process needs to access sensitive data from another isolated process, there must be a context-switch to transfer the control between processes, which entails a certain overhead. It is thus obvious that the more processes that are active and running for a program, the higher runtime cost for it. In addition, such frequent context switches call for an incessant intervention of the OS during program execution, which adds to the already high cost.

Most recently, a researcher team proposed a novel fine-grained data isolation scheme, called Shred [6], in order to address the performance problem of previous work in dealing with in-process attacks. By employing Shred, developers are able to set up for their sensitive data a private region whose access is strictly confined only to authorized code of arbitrary sizes specified at their disposition within the process, thereby averting illicit attempts of other parts of the process to access the data. As a result, no matter how many pairs of sensitive data and the associated code they have to isolate in preparation for in-process attacks on their programs, they can enforce efficient fine-grained isolation by establishing strong security boundaries inside one single process, instead of dispersing the pairs to multiple processes as in the aforementioned previous work. To support fine-grained isolation, Shred capitalizes primarily on ARM's salient hardware feature provided for memory isolation, specifically the memory domain [8], which can perform access control on specific memory regions based on IDs assigned to the memory regions. Shred first assigns unique domain IDs to private regions. It then controls the domain access control register that determines the access permissions of the memory regions associated with the domain IDs, thereby delicately allowing or prohibiting access to the private regions depending on whether the authorized code is running or not. Thanks to ARM's memory domain, Shred can isolate with minimal performance cost multiple pairs of private data and code from each other in a process. Unluckily however, such an efficient solution for fine-grained isolation will be no longer viable for those who want to protect their programs running on the latest ARM processors because the memory domain feature, having been available in ARM's 32-bit architecture (a.k.a. AArch32), became obsolete in its 64-bit successor (a.k.a. AArch64). This should be an obvious limitation of Shred, since nowadays AArch32 is rapidly losing its dominant position to AArch64 in the market for both high-end and budget mobile devices. Another limitation is that similarly to previous work, Shred also necessitates the intervention of an OS during program execution, since ARM's memory domain operates under the supervision of the OS kernel. Every time an access to sensitive data is requested, the kernel must be invoked twice each to turn on and off the protection state of the corresponding memory domain, which spends thousands of extra CPU cycles.

This paper presents a fine-grained data isolation scheme for AArch64, named Sealer. For lack of the memory domain support, a natural choice for this new architecture would be to employ one of the process-level schemes backed by OS's memory isolation. However, even if it could offer deterministic security guarantees against in-process attacks, this was not acceptable to us, as explained above, due to its inherent high performance overhead. As a more practical alternative, we have relied on a probabilistic approach where we can gain efficiency by avoiding run-time checks and OS

interrupts, both of which contribute to performance degradation. In the approach, the values of sensitive data are encrypted to raise the bar for attacks by stymieing their attempts to access the intended data. To implement Sealer, every pair of sensitive data and authorized code must be first determined and each assigned a unique encryption key. The authorized code exclusively owns one key and during code execution, uses it to store its sensitive data encrypted in memory and decrypt them when loading. The advantages of our approach are twofold. One is that Sealer ensures a probabilistic integrity of sensitive data while the data resides in memory because accesses to these data from any unauthorized code without the proper key will have unpredictable results. The other is that unlike Shred, Sealer does not require OS's intervention because encryption is basically user level operations and thus not depending on OS service.

However, we admit that Sealer only works correctly when all the keys can be safely protected and concealed from attackers. Unfortunately, unlike those sensitive data we like to protect, these keys cannot be protected simply by encrypting them with another new key because otherwise, it will result in an infinite repetitive sequence of encrypting one key with another. To solve this problem, other probabilistic approaches using data randomization [9,10] devised software mechanisms for key protection. The downside of these pure software solutions, as can be expected, is that they consume the CPU time, often to an excessive extent. Such performance overhead of software-based key protection was unacceptable to us since it would increase the overall execution time of our scheme possibly even up to the level of those of the previous process-level schemes, which is surely contrary to our design objective for Sealer, that is, efficiency or low cost. Motivated by this observation, we have investigated AArch64 searching for a hardware feature that may support efficiently our key protection, and found a memory permission primitive, called Execute-no-Read (XnR), that was newly introduced in AArch64. Originally, XnR was intended to prevent security critical code like firmware from being exposed by malicious memory accesses, and have been actively used in previous research that wants to hide their executable binaries from adversaries [11,12]. But in our work, we discovered that it could also be useful for our purpose, that is, efficient protection of the key values in our data isolation scheme, as will be detailed in Section 3. According to the empirical results in Section 5, the efficient key protection mechanism built on top of XnR has been pivotal to attain high performance of our data isolation scheme even when we employ for our data protection a more sophisticated, thus expensive encryption algorithm (advanced encryption standard (AES) in our work) than the naive lightweight ones like eXclusive-OR (XOR) employed in most conventional probabilistic approaches mentioned above. As argued in [13], a probabilistic solution based on strong encryption like AES may attain in effective almost the same security level as a deterministic one. Consequently, we here believe that our data isolation scheme is of practical merit with respect to both performance and security.

The remainder of the paper is organized as follows: In Section 2 we give background information on hardware features that Sealer relies on. Next, in Section 3, we evince the threat model and some assumptions regarding Sealer. In Section 4 we present the design of our data isolation scheme. Then in Section 5 we discuss in details the implementation. In Section 6 we evaluate its performance and in Section 7 we contrast Sealer with the related works. Lastly, we conclude the paper.

2. Architecture Background

This section discusses two major hardware features of AArch64 that we have utilized to efficiently implement Sealer on this architecture. These features play a vital role in enhancing both the security level and performance of a program working with Sealer.

2.1. Execute-Only Access Permission

If the CPU generates a memory access, its memory management unit (MMU) performs a virtual-to-physical address translation. At the same time, the MMU verifies that the memory access complies with the access permission for the target address, and if not, it generates a page fault exception to prevent the access. Conventionally, the ARM architecture determines access permissions

by combining some flag bits specified inside page tables, such as the access permission bits and the execute-never bits. By default, both AArch32 and AArch64 control access permissions in a similar way, but they differ in setting permissions to allow the execution of instructions. In AArch32, the read permission is necessary to enable execution, wherein all code has to be set to readable. On the other hand, AArch64 introduced a new type of permission, called the execute-only, in order to grant the execute permission even without the read permission.

The execute-only permission of AArch64 allows an instruction to be just executed but not read or written, which provides an efficient and deterministic way to prevent code from being exposed or tampered with by unintended users. Therefore, a multitude of studies have depended on this permission in thwarting control-flow hijacking attacks via preventing unintended code disclosure [11,14–16]. In this paper, we like to emphasize that the execute-only permission can be used to protect against disclosure not only code, as designed for the original purpose, but also read-only data, such as secret constants in the code. As an example, consider a simple code that assigns a fixed key value to a variable.

```
unsigned int key = 0x12345678;
```

This code is compiled to the following assembly code that consists of the `mov` and `movk` instructions intended to generate large immediate values.

```
mov x0, #0x5678
movk x0, #0x1234, lsl #16
```

Now, if we enforce the execute-only permission on this code, attackers may not get the key value presented in the code through disclosure attacks because read access to the code is prohibited. Such a data protection capability of the execute-only permission has inspired us to take an encryption-based probabilistic approach in the design and implementation of Sealer, as will be described in Section 3.

2.2. ARMv8 Cryptographic Extension

In probabilistic data isolation, there is a trade-off between security and performance when it comes to a choice of the cryptographic algorithm for data encryption. Naive algorithms like XOR encryption are fast, but they are weak and even vulnerable to plaintext attacks [17]. In contrast, sophisticated encryption algorithms are usually strong but expensive. To raise the security level while minimizing performance loss, modern architectures are equipped with special hardware extensions for accelerating encryption algorithms used in diverse security solutions. For instance, in ARM's latest architecture ARMv8, there has been introduced an extension for the AES (advanced encryption standard) algorithm. As listed in Table 1, the cryptographic extension of ARMv8 supplies four instructions for acceleration of AES. The `aese` and `aesd` instructions perform a single round of AES encryption or decryption, respectively. The `asemc` and `aesimc` instructions perform AES MixColumns and Inverse MixColumns operations, respectively. In addition, these instructions take a 128-bit vector register as an argument, so we can effectively express each key value of AES-128 as a single vector register. A 128-bit AES key expands to eleven 128-bit values, requiring the same number of vector registers to hold the key. AES has three versions: AES-128, AES-192, and AES-256. The difference is their key lengths (128, 192, and 256 bits) and the number of rounds (10, 12, and 14).

Table 1. AES instructions on ARMv8.

Instruction	Description
<code>aese</code>	AES single round encryption
<code>aesd</code>	AES single round decryption
<code>asemc</code>	AES mix columns
<code>aesimc</code>	AES inverse mix columns

As our target is AArch64 which is the ARMv8 64-bit architecture, we have chosen the AES-128 algorithm for Sealer in order to achieve cryptographically strong defence with low runtime overhead by capitalizing on the AES extension. The probabilistic defence based on strong encryption like AES is known to provide virtually the same security strength as its deterministic counterpart [13]. By the same token, we expect that Sealer can be almost as secure against in-process attacks as deterministic ones [4–6].

3. Threat Model and Assumption

We assume a powerful attacker who can conduct an in-process attack by exploiting memory vulnerabilities in the program code and by subverting execution-flow. By launching the attack, the attacker has the capability to perform arbitrary reads/writes to all memory regions belonging to the victim process with the same access permission as the process, which paves the way for the attacker to arbitrarily manipulate a control flow of the victim process.

However, importantly, we trust the OS, which serves as the trusted computing base for Sealer. It means that we do not consider any privilege escalation attacks to abuse the OS. We believe it is reasonable because Sealer only focuses on thwarting to in-process attacks. Since the attackers who already succeeded to take over the OS do not need to carry out in-process attacks, such an assumption has been commonly accepted in previous works [6] with the same goal as Sealer. The trusted OS is responsible for securely running the modules comprising Sealer. The OS also enforces WX (Write XOR eXecute) [18] on user processes, thereby preventing code corruption and injection attacks. Besides the OS, we trust the authorized code protected by Sealer as well. We deem that the authorized code will be small enough to be formally verified [2], not considering security vulnerabilities residing in the authorized code. Also, side channel attacks using shared resources between processes such as cache [19,20] and branch predictor [21] are out of the scope of this paper. Similarly, hardware attacks, such as cold boot attacks [22] and snoop-based attacks [23] are not considered as well.

4. Design

The primary design goal of Sealer is to provide efficient data isolation on AArch64 while overcoming the lack of specialized hardware support for memory isolation. Sealer achieves this by combining salient hardware features existing in AArch64.

4.1. Design Overview

Sealer provides application developers with an ability to specify a private memory that is accessible only by a specially authenticated code in their programs. Sealer allows the developers to protect their sensitive data against in-process attacks at runtime by storing the data in the private memory that is isolated from attackers who may have control over non-authenticated code in the same process. Since in AArch64, we can no longer leverage the memory domain support to create and protect private memory regions for fine-grained data isolation, we cannot efficiently sandbox sensitive data sets in separate address spaces, as done by Shred. As a more efficient alternative solution, we have decided to isolate the data sets in the data space by borrowing the idea of data randomization [9,10]. Now for each set of sensitive data, a unique key is assigned. The entire contents in a set are grouped logically to form a private region in the data space, in the sense that they are isolated from others by always remaining in memory encrypted with their unique key. The key is given exclusively to the authorized code that requires one to access the encrypted contents for legitimate purposes. This code uses the key to decrypt and encrypt the contents, respectively while reading into CPU registers and writing back to memory. Through these operations, Sealer provides probabilistic read and write integrity of the contents in a private region against in-process attacks where attackers exploit vulnerabilities to gain control over the execution of a vulnerable program. Sealer ensures that these attackers cannot force the victim to disclose the private contents because the contents that they read or write without proper encryption/decryption will induce unpredictable execution results. However,

such integrity is only guaranteed when all the keys for private regions in a process are completely protected and concealed from the attackers.

To efficiently protect keys, Sealer relies on the execute-only permission of AArch64, as described in Section 2. The ultimate goal here is to prevent the keys from being accessed via any memory load/store instructions in the process. For this, we first implant each key in the authorized code itself that owns the key, rather than placing it in the data region. Then, we protect the code with the execute-only permission in order to thwart any attempts to disclose the corresponding key. Thanks to this encryption-based isolation technique, Sealer can serve data isolation on AArch64 without requiring specific memory protection features such as the memory domain and MMU. Relying on these features has two disadvantages. First, the availability of these features is dependent on the target architecture, as in the case of the memory domain in AArch32. Second, using these features often requires frequent OS kernel's intervention because their functions are security-related, thus always operating under the control of privileged software. Although Sealer also uses the architecture-dependent hardware for execute-only permission, since we target AArch64 in this work, it does not need OS interrupts to change the permission during program execution.

4.2. Security Property

To guarantee secure data isolation, Sealer fulfills two security properties:

- Atomic Execution: Sealer must be able to detect and abort all attempts to tamper with the execution flow from the outside into the authorized code. By doing this, the isolation mechanism of Sealer is guaranteed to be secure.
- Exclusive Access: Sealer must ensure that a private region is only accessible to its associated authorized code. Thereby, the integrity and confidentiality of the private region contents must be protected.

Figure 1 describes step by step the overall data isolation mechanism of Sealer. In a process, there runs a multitude of threads which we assume already have their own thread-level protections mechanisms. As stated earlier, the objective of Sealer is to protect threads (more specifically, their sensitive data) with an additional mechanism against in-process/thread attacks. The execution of a thread will reach the entry gate of an authorized code ①. Two important roles of the entry gate are loading an encryption key and creating an authorization token for satisfying the two security properties of Sealer. The encryption key is used by the authorized code to grant the exclusive right to access the sensitive contents stored encrypted in the private region. Without a valid encryption key, an attempt of any code in a thread to read or write the private contents will run into crash or unpredictable results. The authorization token is used to confirm the atomic execution of the authorized code. The token is only issued when the code is entered through the entry gate so that solely by checking the presence of the token, Sealer can detect any attempts to jump into the code without going through the entry gate. After both the encryption key and the token are loaded in the entry gate, the authorized code can access the sensitive data exclusively and atomically ②. If the authorized code is terminated, the exit gate clears remained security critical states, such as the encryption key and the authorization token created by the entry gate, and continues the thread execution ③.

Figure 1. The data isolation mechanism of Sealer.

4.3. Development Process

Figure 2 describes the development process of applications with Sealer to facilitate enforcement of data isolation. The only requirement for application developers is to use the application programming interfaces (API) of Sealer at the source code level to define their isolation target, i.e., the authorized code to access sensitive data and block of memories to contain sensitive data. The executable augmented with Sealer's APIs will be loaded and executed on the system. At this time, the Sealer's loader is responsible for providing kernel-level support for enforcing the data isolation of Sealer, such as configuring the execute-only permission.

Figure 2. Development process in Sealer for data isolation.

4.4. APIs of Sealer

Before explaining the APIs of Sealer, we first introduce the details of Sealer's management scheme for authorized code and private region. As described in Figure 3, Sealer can support multiple pairs of the authorized code and the private region. Every authorized code and private region is assigned an ID at development-time. Sealer assigns encryption keys at run-time to each authorized code and private region according to the given IDs, thereby each authorized code is paired with the private region having the same ID. Some authorized codes may have the same ID, and in this case, all of them are paired with the same private region and are allowed to share the same sensitive data. However, as each authorized code can have only one ID, it cannot be paired with multiple private regions. Lastly, a private region is composed from a set of sensitive data, and sensitive data are allowed to be added to or removed from the private region dynamically, which can be done by the authorized code associated with the private region.

Figure 3. The management scheme for authorized code and private region.

To support the management scheme introduced so far, Sealer provides developers with four APIs. First, two APIs are given to define the authorized code at development-time. Application developers can use `Sealer_start` and `Sealer_end` to indicate the start and end gates of each authorized code. The number (ID) passed as a parameter of `Sealer_start` is used to identify each authorized code. Second, the `Sealer_encrypt` and `Sealer_decrypt` are offered to developers for encrypting and decrypting of Sealer-protected sensitive data.

4.5. Loader of Sealer

The Sealer's loader is the component of Sealer that operates in the kernel to handle tasks that can only be done at the kernel-level. When a program with Sealer is launched on the system, it intervenes to assign encryption keys and authorization tokens to the authorized code. For this, the Sealer's loader first generates encryption keys and random authorization tokens, and inserts them as a part of the authorized code. After that, it enables the execute-only permission on the memory regions of the entry gate, the exit gate and the authorized code, thereby protecting the encryption keys from attackers at run-time without noticeable performance penalty.

Another role of the Sealer's loader is to protect the secrets stored in the registers such as encryption keys from being exposed to the outside of the authorized code. As stated in Section 3, we trust the authorized code by default, so that we do not consider the secrets leaking due to the internal vulnerabilities of the authorized code. However, attackers who run other parts of the victim program besides the authorized code may be able to expose the authorized code's secrets in the registers by raising arbitrary signals on purpose. More specifically, when the control is forcibly transferred from the authorized code to the signal handlers possessed by the attackers, the secrets could be exposed by remaining in the registers or by being spilled to the handlers' stack. Sealer fends off such potential exposures of the secrets as follows. If the authorized code is running when a signal is delivered, the Sealer's loader backs up the registers in the kernel and nullifies them to prevent them from being exposed to the attackers during signal handling. When the signal handling is done and the control is returned to the authorized code, the Sealer restore the old values of the registers to continue the execution of the authorized code correctly.

5. Implementation

This section describes an implementation of Sealer presented in Section 3, which are directly related to the Sealer's APIs.

5.1. Entry Gate

The main objective of `Sealer_start` is to initiate the entry gate of each authorized code. In the entry gate, the authorized token must be securely created and should be hidden to prevent it from being tampered or leaked. First, the value of the token is stored in the code region enforced execute-only permission like the encryption key. That is, the value is expressed in the immediate field of the move instruction, and then it is loaded into the general purpose register by executing this instruction as shown in Figure 4. Next, since the authorization token is used to verify the atomic operation when the authorized code is executed, the value of the authorization token must be hidden securely. To do this, Sealer keeps the value of authorization in a debug register called `dbgctr_e10` and prevents access to the token from unauthorized code via a code instrumentation to prohibit the use of this register. This technique of protecting the secret information in the debug register is already used in the existing studies [24,25]. Similarly, Sealer protects the Authorization Token by keeping the Authorization Token value in the debug register.


```

1  mov    x2, AUTH_TOKEN          ; Set an authorization token
2  msr    DBGDTR_EL0, x2
3  lsb
4
5  mov    x0, KEY1_1              ; Set an advanced encryption
6  movk   x0, KEY1_2, lsl #16     standard (AES) round key #n
7  movk   x0, KEY1_3, lsl #32
8  movk   x0, KEY1_4, lsl #48
9  mov    x1, KEY1_5
10 movk   x1, KEY1_6, lsl #16
11 movk   x1, KEY1_7, lsl #32
12 movk   x1, KEY1_8, lsl #48
13 mov    v10.2d[0], x0
14 mov    v10.2d[1], x1
...

```

Figure 4. Entry gate of Sealer. Setting an AES round key (from Line 5 to Line 14) is repeated for all 11 AES round keys.

As shown in the previous design, the entry gate must be able to securely load the encryption key. In Sealer, AES encryption algorithm with 128-bit key length (AES-128) is used as a cryptographic algorithm, which uses 11 round keys each for encryption and decryption operations. The values of these AES round keys is represented in the immediate field of the move instruction and stored securely in the code like the value of Authorization Token. As a immediate-to-register move instruction in AArch64 has a 16-bit immediate field, eight move instructions are required to represent one round key. However, there is a problem that all AES round keys up to 352 bytes in total cannot be loaded in system registers such as debug registers or general purpose registers due to the limit number of them. To tackle this problem, Sealer uses vector registers to efficiently load these AES round keys in entry. The vector registers supported by AArch64 are 32 registers from v0 to v31, each of which is a 128-bit register. Since the size of the round key of the AES-128 algorithm is also 128 bits, Sealer can efficiently load the round keys by using vector register v10 to v32.

5.2. Authorized Access to Private Memory

After loading the encryption keys and authorization token into the registers at the entry gate, the authorized code accesses the private memory with these values. When a sensitive data is stored in a private memory, it must be stored in an encrypted format. Therefore, as shown in Figure 5, the authorized code firstly encrypts the sensitive data using a AES-128 encryption operation, and then stores the encrypted sensitive data in the private memory. The code from lines 4 to 13 of Figure 5 shows the ten encryption rounds of AES-128 utilizing cryptographic extension of AArch64. Cryptographic extension provides four instructions for the AES algorithm acceleration: *aese*, *aesmc*, *aesd* and *aesimc*. Each round encrypts the plain text block contained in v0 using each AES round keys (v10 to v20). Lastly, the authorized code checks the value of the authorization token stored in the debug register to verify that the execution at this point originates from a normal execution flow starting from the entry gate. As a result, Sealer can detect any attempts to divert execution-flow from the unauthorized code to authorized code directly. When decrypting, instead of *aese* and *aesmc* instructions, *aesd* and *aesimc* instructions are used with the other AES round keys (v21 to v31).

```

1  mov    x0, [x1]                ; Store the source value (x0)
                                   to the target address (X1)

```

(a) Original store operation

```

1  mov    v0.2d[0], x0            ; Encrypt the source value (x0)
2  aese   v0.16b, v10.16b
3  aesmc  v0.16b, v0.16b
4  aese   v0.16b, v11.16b
5  aesmc  v0.16b, v0.16b
6  ...
7  aese   v0.16b, v18.16b
8  aesmc  v0.16b, v0.16b
9  aese   v0.16b, v19.16b
10 xor    v0.16b, v0.16b, v20.16b
11
12 st1    {v0.16b}, [x1]         ; Store to the target address (x1)
13
14 mov    x2, AUTH_TOKEN         ; Check the authorization token
15 mrs    x3, DBGDTR_ELO
16 ror    x3, x3, #32
17 cmp    x2, x3
18 b.ne   error_handler

```

(b) Sealer's store operation

Figure 5. Store operation in Sealer.

5.3. Exit Gate

The Sealer's exit gate, shown in Figure 6, is responsible for clearing the encryption key and authorization token. Unlike the entry gate, each AES round register loaded into the vector registers can be initialized to zero by an immediate-to-vector move instruction. In the case of the authorization token, it is necessary to check the value of it after initialization. Because, by subverting the control flow, the attacker can manipulate the value of the token to an arbitrary value through an instruction on line 9.

```

1  movi    v10.2d, #0x0          ; Clear all the AES round keys
2  movi    v10.2d, #0x0
3  movi    v10.2d, #0x0
4  ...
5  movi    v10.2d, #0x0
6  movi    v10.2d, #0x0
7
8  mov     x0, #0x0              ; Clear the authorization token
9  msr     DBGDTR_ELO, x0
10
11 mrs     x1, DBGDTR_ELO        ; Check the authorization token
12 cmp     x0, x1
13 b.ne    error_handler

```

Figure 6. Exit gate of Sealer.

6. Evaluation

In this section, we evaluate Sealer in terms of performance. Experiments have been conducted on the HiKey ARM development board (LeMaker version) which ships with HiSilicon Kirin 620 ARM Cortex-A53 octa-core 1.2 GHz processor and 2GB of RAM. Linux 3.18.0 was used as an OS running on the board.

Table 2 illustrates the performance of Sealer at a glance. We measured the four main APIs of Sealer, *Sealer_start*, *Sealer_end*, *Sealer_encrypt*, and *Sealer_decrypt*, that are introduced in Section 3.

These APIs are used to achieve Sealer's goal, implementing fine-grained data isolation scheme, with cryptographic mechanism using AES-128.

Table 2. Performance numbers of Sealer. First and tenth measurements display the performance difference due to cache-effects.

1st Measurement	CPU Cycles
mprotect	14363
Sealer_start	1325
Sealer_end	65
Sealer_encrypt	46
Sealer_decrypt	60
10th Measurement	CPU Cycles
mprotect	673
Sealer_start	304
Sealer_end	39
Sealer_encrypt	37
Sealer_decrypt	37

Since the data isolation scheme can also be fulfilled using page-table based memory protection mechanism, we tried to compare Sealer with the page-table based data isolation. As explained in Section 1, even if Shred built on the memory domain feature of AArch32 is the representative implementation of page-table based data isolation, it is not available on AArch64, in which Sealer is realized. Therefore, we chose an alternative implementation based on a `mprotect` system call that can dynamically enforce memory protection by setting access permissions bits in page tables, for comparison with Sealer. We believe this is acceptable because the `mprotect`-based implementation works in a quite similar way to Shred. To be specific, when entering to and exiting from the authorized code, both implementations need the intervention of the kernel to update a configuration of the memory domain (in Shred) or of the access permission bits (in `mprotect`).

A row, `mprotect`, in Table 2 describes the CPU cycles that are consumed by an invocation of a `mprotect` call. As mentioned before, in the `mprotect`-based implementation of data isolation, a `mprotect` call has to be invoked twice at the entry and exit gate of the authorized code. Note that the two calls correspond to the `Sealer_start` and `Sealer_end` of the Sealer. Therefore, Sealer has a shorter entry and exit time for the authorized code, so that Sealer will impose a lower burden on the system when the authorized code is frequently executed. However, the data isolation mechanism of Sealer that maintains sensitive data as an encrypted form necessitates encryption (`Sealer_encrypt`) and decryption (`Sealer_decrypt`) of every access to the sensitive data. Given such overheads are not incurred in the `mprotect`-based data isolation, Sealer will be unfavorable when there are plenty of accesses to sensitive data during each execution of the authorized code. From the observations, in summary, we can conclude that Sealer is quite appropriate for protecting frequently-used but relatively small sensitive data.

Table 3 shows the overall performance of some programs with Sealer. The experiment was conducted following the scenario Shred used for its experiment. Specifically, we modified the target programs in a way that protected their small sensitive data, such as password and crypto key, using Sealer. For this, we created and used wrapper functions to store and load the sensitive data using Sealer. We admit that such an application of Sealer does not protect the sensitive data over their full lifetime, but we believe that it can significantly reduce the likelihood of them being exposed to in-process attacks. The numbers in Table 3 demonstrate the efficiency of Sealer, which is comparable to Shred (As stated earlier, Shred cannot be implemented on AArch64 in which the memory domain protection feature is absent. Therefore, we brought the numbers reported in the literature of Shred.).

Table 3. Slowdown of programs with Sealer.

Program	Protected Sensitive Data	Shred	Sealer
curl	password	5.80%	3.25%
minizip	password	7.90%	3.83%
openssh	credential	3.20%	3.41%
openssl	crypto key	1.75%	1.70%
lighttpd	credential	4.70%	4.50%

7. Related Work

In this section, we classify the related works into several groups, and discuss with our work in terms of purpose and mechanism.

7.1. Data Isolation

Over the past decade, various data isolation schemes sharing the same goal with Sealer have been proposed that allow application developers to protect sensitive data by isolating the data and the code authorized to access them from the other parts of a program. Early works [4,26,27] have suggested process-level data isolation schemes that isolate each pair of sensitive data and authorized code by partitioning them into separate processes.

However, considering that a process, the OS's abstraction for independent execution of each program, entails a large management and context-switch costs, such process-level isolation schemes are not light-weight and scalable enough to be used to isolate many (and small) portions of code and data within a program. To address this problem, many efforts [5,28,29] have been undertaken to enforce the data isolation in a single process via special memory isolation mechanisms. However, their memory isolation mechanisms still rely on a page table similar to a process, incurring context-switch costs in traversing isolation boundaries of the sensitive data and code pairs as seen in Figure 7b. Therefore, their isolation schemes are not efficient enough to enforce fine-grained data isolation against in-process attacks. To the best of our knowledge, Shred [6] is the first work that enables efficient fine-grained data isolation by devising a memory isolation mechanism based on the memory domain of AArch32 instead of the expensive page table switch. Sadly, the memory domain is deprecated on AArch64 so that Shred is not available on recent ARM-based devices, such as smartphones and tablets, adopting AArch64. In comparison, Sealer is designed to support AArch64 by the synergistic use of the XnR feature of AArch64 and an encryption algorithm. Another notable point of Sealer is that, unlike the previous works including Shred, it does not require the intervention of the OS, enabling more efficient fine-grained data isolation as compared to Figure 7. Shred reports in its paper that it consumes about 16.5 ms for context switching, which is comparable to the time for thread switching and superior to that for process switching by about 17.7 ms. The number roughly corresponds to around 18 million CPU cycles (at its experimental environment of 900 MHz CPU), which is worse than that of Sealer (refer to Table 2).

Formally verified microkernels such as seL4 [2] enable efficient data isolation between processes by supporting lightweight context switching. Plus, such microkernels provide more enhanced security than would-be vulnerable commodity OSes like Linux thanks to their verified functional correctness. It is, however, premature yet for such microkernels to totally supersede the commodity OSes in terms of compatibility and functionality. For this reason, these new class of OSes tend to be used in building TEEs from scratch [30].

Figure 7. Comparison of mechanisms for data isolation. Isolation mechanisms based on the memory domain or page table require at least four privilege level changes to run the authorized code. In addition, those based on trusted execution environments (TEE) need not only four privilege level changes but also two secure state changes in order to run the authorized code in the TEE. On the contrary, Sealer allows the authorized code to run without specific changes in a privilege level or a secure state, enabling rapid execution of the code.

7.2. Fault Isolation

For a long time, isolating distrusted modules, e.g., from dynamic libraries and plug-ins in applications to loadable kernel modules and drivers in the OS, through hardware or software mechanisms to prevent their faults from spreading, has been a popular research topic. The most prominent approaches proposed by researchers are the software-based fault isolation (SFI) approaches [31–33] that employs the address masking technique to create certain isolation boundaries in address space. These approaches can be used to enforce data isolation, but they have critical limitations compared to Sealer as they are not suitable for isolating a number of scattered data because they are specialized for bisecting the address space due to the nature of the address masking technique, which is specialized for bisecting the address space. Such limitations are overcome in hardware-based fault isolation (HFI) approaches [34,35] that rely on a memory access control capability provided by hardware components, such as MMU. These approaches can isolate potentially faulty modules by enforcing restricted memory access permission while the modules are running. Note that this mechanism is very similar to the mechanisms used in the data isolation solutions [6,29]. This means that HFI surely shares the same drawbacks as the data isolation solutions compared to Sealer.

7.3. Trusted Execution Environment

Numerous research works have aimed to provide developers with a trusted execution environment (TEE) where ensuring reliable execution of security critical code against not only malicious programs but also against compromised OS. The strong adversary model assumed in these works implies the TEE must be separated with the OS. In this context, they have established TEEs relying on a software or hardware layer that has a higher privilege than the OS, such as DRTM [36], ARM TrustZone [37,38], INTEL SGX [39,40] and Hypervisor [41–43]. The protection provided by the TEE solutions is secure and reliable, but in comparison with Sealer it is less efficient for protecting sensitive data and the associated code against in-process attacks. For example, when the TEE solutions based on the ARM TrustZone are used to enforce data isolation on the code tightly coupled with the host program, the frequent crossing of the isolation boundary surrounding the TEE incurs high costs due to the overall change in the secure state of the system as illustrated in Figure 7c.

8. Conclusions

In this paper, we presented Sealer, a fine-grained data isolation scheme that operates in AArch64. Sealer capitalizes on ARM's eXecute-no-Read and cryptographic extension, thereby accomplishing fine-grained data isolation while minimizing mandatory interventions of the kernel. Application

developers can employ data isolation protection by invoking Sealer's four APIs. Notably, they can effectively protect sensitive data that are accessed frequently but are relatively small using Sealer.

Funding: This work was supported by the Soongsil University Research Fund of 2019.

Conflicts of Interest: The author declares no conflict of interest.

References

- Misra, S.C.; Bhavsar, V.C. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *International Conference on Computational Science and Its Applications*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 724–732.
- Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, MT, USA, 11–14 October 2009; ACM: New York, NY, USA, 2009; pp. 207–220.
- Saltzer, J.H.; Schroeder, M.D. The protection of information in computer systems. *Proc. IEEE* **1975**, *63*, 1278–1308. [[CrossRef](#)]
- Brumley, D.; Song, D. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, USA, 9–13 August 2004; pp. 57–72.
- Bittau, A.; Marchenko, P.; Handley, M.; Karp, B. *Wedge: Splitting Applications into Reduced-Privilege Compartments*; USENIX Association: Berkeley, CA, USA, 2008.
- Chen, Y.; Raymondjohnson, S.; Sun, Z.; Lu, L. Shreds: Fine-grained execution units with private memory. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 22–26 May 2016; pp. 56–71.
- Agten, P.; Nikiforakis, N.; Strackx, R.; De Groef, W.; Piessens, F. Recent developments in low-level software security. In *IFIP International Workshop on Information Security Theory and Practice*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 1–16.
- ARM. Architecture reference manual. ARMv7-A and ARMv7-R edition. *ARM DDI 0406C* **2012**, 1–2736.
- Cadar, C.; Akritidis, P.; Costa, M.; Martin, J.P.; Castro, M. *Data Randomization*; Technical Report TR-2008-120; Microsoft Research: Cambridge, UK, 2008.
- Bhatkar, S.; Sekar, R. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 1–22.
- Crane, S.; Liebchen, C.; Homescu, A.; Davi, L.; Larsen, P.; Sadeghi, A.R.; Brunthaler, S.; Franz, M. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 17–21 May 2015; pp. 763–780.
- Werner, J.; Baltas, G.; Dallara, R.; Otterness, N.; Snow, K.Z.; Monrose, F.; Polychronakis, M. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, Xi'an, China, 30 May–3 June 2016; ACM: New York, NY, USA, 2016; pp. 35–46.
- Koning, K.; Chen, X.; Bos, H.; Giuffrida, C.; Athanasopoulos, E. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, Belgrade, Serbia, 23–26 April 2017; ACM: New York, NY, USA, 2017; pp. 437–452.
- Braden, K.; Davi, L.; Liebchen, C.; Sadeghi, A.R.; Crane, S.; Franz, M.; Larsen, P. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the NDSS*, San Diego, CA, USA, 26 February–1 March 2017.
- Backes, M.; Holz, T.; Kollenda, B.; Koppe, P.; Nürnberger, S.; Pewny, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale, AZ, USA, 3–7 November 2014; ACM: New York, NY, USA, 2014; pp. 1342–1353.
- Gionta, J.; Enck, W.; Ning, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, San Antonio, TX, USA, 2–4 March 2015; ACM: New York, NY, USA, 2015, pp. 325–336.

17. Lu, K.; Song, C.; Lee, B.; Chung, S.P.; Kim, T.; Lee, W. ASLR-Guard: Stopping address space leakage for code reuse attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; ACM: New York, NY, USA, 2015; pp. 280–291.
18. Seshadri, A.; Luk, M.; Qu, N.; Perrig, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM SIGOPS Oper. Syst. Rev.* **2007**, *41*, 335–350. [CrossRef]
19. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-level cache side-channel attacks are practical. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 605–622.
20. Lipp, M.; Gruss, D.; Spreitzer, R.; Maurice, C.; Mangard, S. Armageddon: Cache attacks on mobile devices. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 549–564.
21. Evtvushkin, D.; Ponomarev, D.; Abu-Ghazaleh, N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, Taipei, Taiwan, 15–19 October 2016; p. 40.
22. Halderman, J.A.; Schoen, S.D.; Heninger, N.; Clarkson, W.; Paul, W.; Calandrino, J.A.; Feldman, A.J.; Appelbaum, J.; Felten, E.W. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM* **2009**, *52*, 91–98. [CrossRef]
23. Solutions, E. Analysis Tools for DDR1, DDR2, DDR3, Embedded DDR and Fully Buffered DIMM Modules Available online: <http://www.epnsolutions.net/ddr.html> (accessed on 11 January 2020).
24. Müller, T.; Freiling, F.C.; Dewald, A. TRESOR Runs Encryption Securely Outside RAM. In Proceedings of the USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011; Volume 17.
25. Davi, L.; Gens, D.; Liebchen, C.; Sadeghi, A.R. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017.
26. Kilpatrick, D. Privman: A Library for Partitioning Applications. In Proceedings of the USENIX Annual Technical Conference, FREENIX Track, San Antonio, TX, USA, 9–14 June 2003; pp. 273–284.
27. Provos, N.; Friedl, M.; Honeyman, P. Preventing Privilege Escalation. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, 4–8 August 2003.
28. Wang, J.; Xiong, X.; Liu, P. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15), Santa Clara, CA, USA, 8–10 July 2015; pp. 361–373.
29. Litton, J.; Vahldiek-Oberwagner, A.; Elnikety, E.; Garg, D.; Bhattacharjee, B.; Druschel, P. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 49–64.
30. Hua, Z.; Gu, J.; Xia, Y.; Chen, H.; Zang, B.; Guan, H. vTZ: Virtualizing ARM TrustZone. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 541–556.
31. Wahbe, R.; Lucco, S.; Anderson, T.E.; Graham, S.L. Efficient software-based fault isolation. *ACM SIGOPS Oper. Syst. Rev.* **1994**, *27*, 203–216. [CrossRef]
32. Yee, B.; Sehr, D.; Dardyk, G.; Chen, J.B.; Muth, R.; Ormandy, T.; Okasaka, S.; Narula, N.; Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 17–20 May 2009; pp. 79–93.
33. Sehr, D.; Muth, R.; Biffle, C.L.; Khimenko, V.; Pasko, E.; Yee, B.; Schimpf, K.; Chen, B. Adapting Software Fault Isolation to Contemporary CPU Architectures. In the Proceedings of the 19th Conference on USENIX Security Symposium, Washington, DC, USA, 11–13 August 2010; pp. 1–11.
34. Swift, M.M.; Bershad, B.N.; Levy, H.M. Improving the reliability of commodity operating systems. *ACM SIGOPS Oper. Syst. Rev.* **2003**, *37*, 207–222. [CrossRef]
35. Zhou, Y.; Wang, X.; Chen, Y.; Wang, Z. Armlock: Hardware-based fault isolation for arm. In Proceedings of the 2014 ACM SIGSAC Conference on Computer And Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; ACM: New York, NY, USA, 2014; pp. 558–569.
36. McCune, J.M.; Parno, B.J.; Perrig, A.; Reiter, M.K.; Isozaki, H. Flicker: An execution infrastructure for TCB minimization. *ACM SIGOPS Oper. Syst. Rev.* **2008**, *42*, 315–328. [CrossRef]
37. Santos, N.; Raj, H.; Saroiu, S.; Wolman, A. Using ARM TrustZone to build a trusted language runtime for mobile applications. *ACM SIGARCH Comput. Archit. News* **2014**, *42*, 67–80.

38. Kostiainen, K.; Ekberg, J.E.; Asokan, N.; Rantala, A. On-board credentials with open provisioning. In Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, Sydney, Australia, 10–12 March 2009; ACM: New York, NY, USA, 2009; pp. 104–115.
39. Baumann, A.; Peinado, M.; Hunt, G. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.* **2015**, *33*, 8. [[CrossRef](#)]
40. Shinde, S.; Le Tien, D.; Tople, S.; Saxena, P. Panoply: Low-TCB Linux Applications With SGX Enclaves. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017.
41. McCune, J.M.; Li, Y.; Qu, N.; Zhou, Z.; Datta, A.; Gligor, V.; Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, 16–19 May 2010; pp. 143–158.
42. Cho, Y.; Shin, J.; Kwon, D.; Ham, M.; Kim, Y.; Paek, Y. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, USA, 20–24 June 2016; pp. 565–578.
43. Hofmann, O.S.; Kim, S.; Dunn, A.M.; Lee, M.Z.; Witchel, E. *Inktag: Secure Applications on an Untrusted Operating System*; ACM SIGPLAN Notices; ACM: New York, NY, USA, 2013; Volume 48, pp. 265–278.

© 2020 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).