Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

RAM Encryption Mechanism without Hardware Support

Tsutomu Matsumoto[2] Ryo Miyachi[2] Junichi Sakamoto[2] Manami Suzuki[1] Dai Watanabe[1,a)]
Naoki Yoshida[2]

Received: December 4, 2019, Accepted: June 1, 2020

Abstract: The RAM encryption encrypts the data on memory to prevent data leakage from an adversary to eavesdrop the memory space of the target program. In this paper, we discuss the feasibility of software based RAM encryption and clarify that it is possible to be secure against so-called semi-honest adversaries under some additional and reasonable constraints. In addition, we tentatively embed our RAM encryption mechanism to SHA-256 hash function. The performance evaluation results are also reported in this paper.

Keywords: cryptography, software protection, RAM encryption

1. Introduction

1.1 Background

Most important public services have been built on dedicated hardware for a long time. For example, the financial service consists of ATM for a user interface, mainframe for a server, and they are connected by a hot-line. Nowadays the trend of computation platforms for various services is moving from such dedicated hardware to more publicly available hardware such as mobile-phone, public/private cloud system and the Internet. Of course, this change indicates the increase of security risks: vulnerability of software, malware infection, theft of terminal devices, and so on. In particular malware infection is a serious threat. For example, McAfee reports that about 10% of smartphones were infected by malware in 2014 and this trend has never improved for years. This fact induces measures which cannot completely prevent malware infection, but can reduce the damage of the infection. Mobile payment services and governmental critical services are important examples which are facing these threats. The systems are accessible via the Internet and these systems deal with the personal information of the users. It is also likely that many criminal organizations are tempted to develop powerful malwares to seize the important data. Therefore, user authentication is essential technology to protect them and it is quite important to keep secret the information used for user authentication such as passwords, biometric information.

1.2 Known Researches

This paper discusses the technology about the secure execution of programs in malware infected devices. There are several approaches to tackle this problem and the basic approach is to isolate the execution environment of the program to be protected from those of other programs.

For example, Android OS generates a virtual machine (VM) for each program to logically isolate their execution environments. This is efficient for malwares having non-root privilege but not for one having root privilege; the latter is freely accessible to the isolated execution environment of other programs.

Trusted Execution Environment (TEE) takes another approach. It isolates the TEE from a rich execution environment (REE) and access from the REE to the TEE is limited. In some implementations of TEE, special hardware provides the access control mechanism, which is independent of one provided by the operation system (OS). Therefore, it is claimed that codes and data in the TEE are secure even if there is a vulnerability in the OS of the REE. These hardware supported solutions are practical and are welcomed by engineers and some cloud systems have started to provide services to use the TEE functions.

So far, there are several different TEE implementations. ARM TrustZone [19] provides a dedicated OS for the TEE and only the programs which are stored in the TEE by default can run in the TEE. AMD Secure Memory Encryption (SME) [20] provides the function to encrypt whole memory space which is equivalent to a disk encryption function. AMD Secure Encrypted Virtualization is more sophisticated; each VM is allocated to isolated memory space which is encrypted by the SME. Intel SGX (Software Guard eXtension) [18] provides process-level TEE. Contrary to ARM TrustZone with which only device vendors manage the programs executed in the TEE, any developer can use the TEE functions provided by Intel SGX.

ARM TrustZone and Intel SGX provide an access control mechanism independent from that provided by the REE. Though these hardware supported access control mechanisms sound like a perfect solution against data leakage by malware infection, they are not in practice. The known issues include vulnerabilities of the TEE itself [21], [22], [23], [24], vulnerabilities in the underlying CPU [26], [27], and side channel attack against the program executed within the TEE [25]. These vulnerabilities tell us that there is no perfect solution and it is hard to fix if a vulnerability is

[1] R&D Group, Hitachi, Ltd., Yokohama, Kanagawa 244–0817, Japan
[2] Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[a)] dai.watanabe.td@hitachi.com

found at hardware level. Therefore, multilevel countermeasures are an important approach including software level approaches in the design of a security system. Another point is that some servicers do not prefer so called vendor lock-in, which prevents them from moving the platform from one to the other, and software level countermeasures may provide them with preferable solutions. This fact encourages researchers to work on the software level technologies to securely execute the program.

In cryptographic applications, security is the primary requirement therefore various adversary models and countermeasures have been discussed so far. Chow et al. proposed the White-box cryptography (WBC) setting in Ref. [12], which allows an adversary to eavesdrop and alter the intermediate values in processing regardless of the place where the data are stored. Many approaches to embed the secret key to the cryptographic implementation including Chow's have been proposed so far, but unfortunately all of them have been broken at a practical level [14]. After Chow's WBC, several moderate adversary models were proposed. Bogdanov et al. assumed that the adversary has communication with malwares in the target device. There is a trivial attack such that the malware sends the full code book of the target cryptographic implementation in this situation and Bogdanov et al. defined the advantage of the adversary, they called it *space hardness*, as the communication consumption for the attack [15]. In addition, they proved that it is possible to construct the secure implementation under this setting [15], [17]. The basic idea is to use huge tables in which the secret information is embedded. If the tables are randomly accessed many times in processing, then some of the accessed points are not known to the adversary. Another example of the moderate adversary was proposed by Oishi et al. in Ref. [28]. They focused on an adversary using a debugger and pointed out that it is not easy even for a skillful hacker to link distinct data (and codes) which are used in separate timing. In addition, they proposed a method to detect alteration of the running program under this setting. In other words, Oishi et al.'s setting puts space and time restrictions on the adversary.

1.3 Contribution of This Paper

This paper deals with the protection of data on memory allocated to processes by a RAM encryption function and considers how to realize it without hardware support. Though the RAM encryption is a limited function compared to the software protection functions provided in the TEE, we believe that it is efficient to RAM scraping malwares which scrape the memory space of the target process in order to get sensitive data such as a credit card number.

We also consider the requirements of the underlying system and the assumptions on abilities of the adversary[*1]. The security of Intel SGX is based on CPU as a black-box model, which assumes the adversary cannot access data on registers. As a result of our consideration, we assume that the adversary is a user-mode process, has root privilege and is semi-honest. We explain that it is possible to construct a software-based RAM encryption scheme to be secure under these assumptions.

[*1] This paper updates and reconstructs the contents of Refs. [10], [11].

Our research intends to illuminate the possibilities and limits of cryptographic countermeasures against malware threats without hardware support, where the concrete barrier to protect a cryptographic key does not exist. We believe that this kind of discussion is also useful to clarify the necessity of hardware support.

1.4 Organization

The rest of this paper is organized as follow: At the beginning the expected system and the ability of the adversary is given in Section 2. Then we propose a software-based RAM encryption scheme in Section 3. The implementation results and the security considerations of the proposed scheme are reported in Section 4 and Section 5 respectively. We conclude this paper in Section 6.

2. Preliminaries

2.1 Expected System

We assume smart phones and other smart mobile devices as the target devices, which are equipped with cameras, various sensors, Wi-Fi in addition to a CPU, memory and storage. We also assume that software (or programs) run on an OS. In other words, the OS manages and allocates the hardware resources of the device such as CPU and memory to the programs. In many systems, the CPU and the OS provide access control mechanisms in various layers, but we only assume one provided by the UNIX-like OS, especially Linux throughout this paper. UNIX-like OS separates a kernel space from a user space. A limited number of pre-defined programs can run in a kernel space and most of the other programs are run on a user space. The latter programs (user-mode programs) are not allowed to direct access to resources in the kernel space and use given APIs such as system calls if necessary. The root is a special user, usually the administrator of the system, who is granted *root privilege*. Other users are granted non-root privileges. We assume that the target program is run in non-root privileged mode and the adversary program is run in root privileged mode.

Despite the fact that the TEE is of course a part of the research target of secure software implementation, our research focuses on countermeasures without hardware support. Therefore, the existence of the TEE is not expected in this paper.

2.2 Adversary's Object

In this paper, we assume that the adversary intends to acquire some information kept secret in the target device. The attack to take over the device in order to apply Denial of service (DoS) and Dynamic DoS attacks are beyond our concern.

Figure 1 sketches the target system. There are several pieces of secret information in the device such as biometric information of the device holder or the cryptographic key stored in the storage. This information is transferred to processes via the main memory and the adversary (malware) intends to capture them.

Contrary to the black-box model, which assumes that the adversary only has the control of data outside of the device, the malware-infected devices are threatened by eavesdropping inside the device, e.g., the buffer of sensors and the main memory. In other words, the data bus and main memory can be identified as a public communication channel under the assumption.

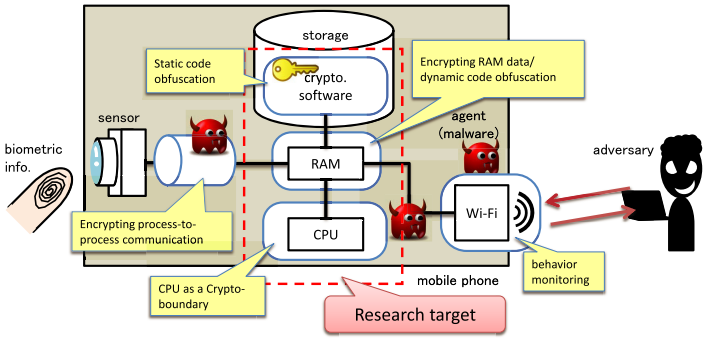

Fig. 1 Malware infected threat model and possible countermeasures.

2.3 Expected Malwares

Most malwares nowadays intrude the target device to collect the personal information of the device holder, or to use the device as a stepping-stone to apply a denial of service attack to other systems [29], [30]. To the best of our knowledge, a malware targeting Android OS for example completes its task within its authority rather than attacks other programs beyond the barrier of the VM. On the other hand, some jail-breaking malwares tend to acquire root privilege for the next step of the attack. Once acquiring the privilege, it is possible to administrate all user-mode running processes.

2.4 Adversary Model

In this paper, we assume an adversary which is weaker than one expected in the TEE and white-box cryptography.

2.4.1 Attack Target

The adversary is assumed to attack only the running process. The execution file of the program in storage is out of our concern. Note that there are some known countermeasures against attacks on execution files such as code encryption and code signing.

2.4.2 Privilege of Adversary

We assume that the adversary is a user-mode process which has a root privilege. The adversary does not run in kernel-mode therefore the attack approaches are limited.

2.4.3 Semi-honest Adversary

We assume a semi-honest adversary in this paper. More specifically, it does not actively run the target program, nor abort, terminate, or alter the running target process. As a natural consequence of these assumptions, the adversary does not run the target program on a debugger. A function to detect a debugger may defuse debugger supported attacks in practice. In addition, we assume that the adversary does not alter the underlying system including the OS and the firmware.

The assumption prevents the adversary from having a direct access measure to the data on registers (in CPU) which are used by the target process. That means it is now possible to consider the analogy of the RAM encryption function provided by Intel SGX. The details on the security considerations are given in Section 5.

3. Proposal for RAM Encryption without Hardware Support

Concealing the memory space of designated processes is the fundamental feature of Intel SGX technology. Let us call this function *RAM encryption* throughout this paper. In this section, we propose a software-based RAM encryption scheme (SBRES) which does not require any special hardware support.

3.1 Rough Sketch of the SBRES

Figure 2 depicts the functions and the process flow of the SBRES. All encryption processes including key management are executed on the registers because the data on registers are not accessible by the adversary. The encryption (and decryption) key is basically generated by using entropy sources available on the CPU such as the CPU timer. Some CPUs provide random number generation functionality and they are also useful entropy sources. The generated encryption key is stored on the registers in plaintext form and the registers are sanitized before the termination of the process.

The data stored on memory is encrypted. An instruction to load data is replaced by the SBRES-ldr, which loads the data to registers, and decrypts it with the key on the registers. An instruction to store data is replaced by the SBRES-str in the same manner. After the usual process (on registers), the data is encrypted by the same key, and then the ciphertext is stored on the memory.

3.2 Limitation of the Protection

The SBRES provides only confidentiality of data. And the SBRES function is embedded in the target program therefore the input to the target process is encrypted just after the process starts to run. Therefore, the input (and the output) of the target process is not encrypted. This is also the case of Intel SGX, therefore the data confidentiality of the data outside of the process is not considered in this paper. The data transmitted between processes may be protected by an inter-process encryption mechanism. The data on storage may be protected in other technology, e.g., the data is encrypted by the white-box block cipher, and the protection mechanism is switched to the SBRES when the data is moved to the memory.

3.3 Requirements for Underlying Encryption Algorithm

The SBRES encrypts all data on memory which are not sequentially accessed in general, therefore a chaining mode of operation such as CBC is not suitable for our use in terms of the processing speed. We chose ECB (Electric Code Book) as the mode of operation. If one requires higher security, XTS (XEX-based tweaked-codebook mode with ciphertext stealing) or other

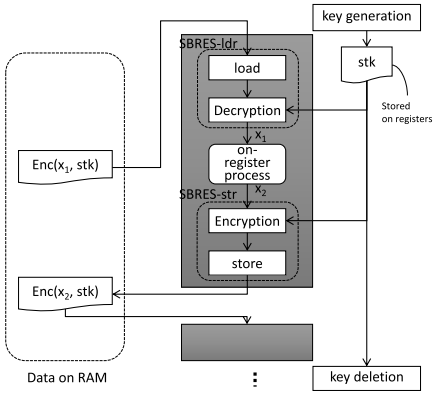

Fig. 2 Data processing in CPU with RAM encryption mechanism.

tweakable mode of operations may be better choices. In addition, it is easy to append a message authentication function to ECB by filling a part of the input to the encryption function by a constant.

The SBRES encrypts and decrypts data on the CPU and the encryption key is stored on registers in plaintext form. Therefore, the SBRES requires registers to store the encryption key and additional work areas. The encryption/decryption is processed there and the target process to be protected can use only the remainder registers. If the underlying CPU has a special instruction set for the AES such as Intel AES-NI supported on and after Westmere architecture, it usually provides dedicated registers for encryption/decryption. Therefore, the target process can use all general purpose registers.

Note that expanding round keys in advance of the encryption process is a common speeding up technique in block cipher implementation such as AES [8], but this is very register consuming. Therefore, it is not applicable to the SBRES.

4. Implementation on Raspberry Pi3

In this section, we examine the embedding of the SBRES to ARMv7 processors, which are widely used in smart devices, and evaluate the penalty of the processing speed.

4.1 Evaluation Environment

We selected Raspberry Pi3 Model B [31] for evaluation. **Table 1** shows the hardware and the software specification of the evaluation environment.

Note that the CPU of Raspberry Pi3 is ARMv8 architecture but the official OS recognizes it as ARMv7 architecture therefore the registers are of 32-bit length in our evaluation environment.

4.2 The Sample of the Target Program

Unfortunately, "real life applications" are too complex for evaluation because the SBRES is manually embedded to the target program. We chose SHA-256 as a target program. Please refer to [9] for the specification.

SHA-256 requires at least 25 32-bit variables for processing while 13 out of 16 general purpose registers are available in ARMv7 processors. In other words, it is not possible to hold all the data on registers during the process and a part of the intermediate value of the process must be evacuated to the memory. In our implementation, 9 variables are allocated to the registers and the remaining 16 variables are allocated to the memory.

Table 1 Evaluation environment.

CPU	Quad Core 1.2 GHz Broadcom BCM2837 64-bit CPU
RAM	1 GB
OS	Raspbian 4.9.2
Compiler	gcc 4.9.2

A hash function itself does not take a piece of secret information as an input and we assume that the whole input is to be protected. That means, the whole input on memory is encrypted and relocated on memory, then the memory space where the input data was located at the beginning is sanitized. The input and output data themselves are out of the protection target of the SBRES as mentioned in Section 3.2.

4.3 Block Cipher Suitable for ARMv7 Processor

The block cipher we use in the evaluation is based on Speck64/96 [2]. The block size of Speck64/96 is 64 bits and the key length is 96 bits. The round function of Speck64 is given as follows:

$$X_{r,L} = ((X_{r-1,L} \ggg r_1) + X_{r-1,R}) \oplus k_r,$$
$$X_{r,R} = (X_{r-1,R} \lll r_2) \oplus X_{r,L},$$
$$r_1 = 8, r_2 = 3,$$

where each operation is 32-bit-wise therefore it is suitable to ARMv7 processors.

Our customization reduces the key length from 96 bits to 64 bits, removes key scheduling function and adopts 1-key Even-Mansour construction [16] instead. Even-Mansour construction just xors the key before and after the permutation, namely there is no key scheduling. The permutation here is the iteration of the round function of Speck64. In this paper we use 8 and 16 iterations of the round functions. Appendix A.1 explains the design rationale and the choice of parameters.

4.4 Implementation Results

Table 2 shows the speeds for an execution of compression function of SHA-256, where "w/ Enc" means the program with the SBRES and "wo/ Enc" means the program without the SBRES. SHA-256 is implemented in assembly language in order to embed the SBRES. We omit the key generation function because it is not essential in the evaluation of processing speed.

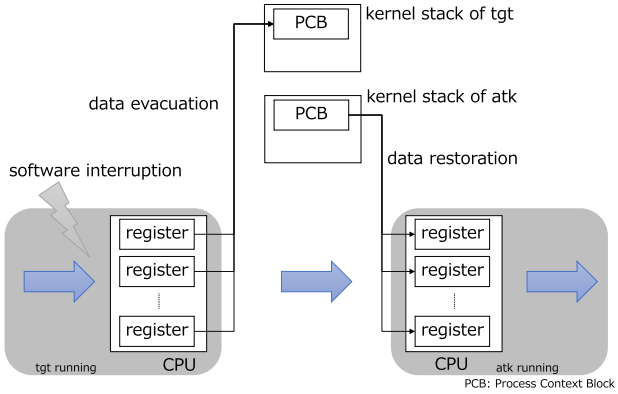

Fig. 3 Rough sketch showing how the data remaining on the registers after software interruption are sanitized.

Table 2 Implementation results on Raspberry Pi3.

Library	cycles/block
w/ Enc (Ours, 16 rounds)	26,530
w/ Enc (Ours, 8 rounds)	15,764
wo/ Enc (Ours)	4,168

As a result, the processing speed of our SHA-256 implementation with the SBRES is 3.8–6.4 times slower than one without the SBRES. The restriction of register usage due to embedding the SBRES affects the flexibility of implementation, but the influence on the speed is not significant in the case of SHA-256.

Note that the penalty in speed heavily depends on the number of memory access instructions. For example, if there is no memory access in the execution of the program, RAM encryption function is not necessary at all. Therefore, if the number of available registers increases, the number of memory access in execution of the program is expected to decrease. In our implementation, we only use integer registers. But if an implementer additionally uses floating point registers, the penalty may be small.

5. Security Consideration of RAM Encryption

In this section we discuss the security issues of the SBRES. There are the following three attack surfaces under consideration to protect the data dealt with by the target process.

- Registers (data in processing)
- Memory (data at rest)
- Input and output of the process (data on transmission)

As mentioned, the third surface is out of the SBRES protection by design, therefore the remaining two cases are discussed. In the following, the attack process and the target process are denoted by `atk` and `tgt` respectively.

5.1 Information Leakage via Registers

The encryption key for the SBRES and a part of the data in processing are on the registers. The access to CPU (or a core in case of multi-core CPU) is exclusive, therefore both `tgt` and `atk` cannot simultaneously run its process on it. If a software interrupt occurs, the process running is interrupted, evacuated then the other process (re-)starts to run. In this paper we assume that the adversary has the root privilege but runs as a user-mode process. If `tgt` is interrupted, the data on registers are evacuated to the kernel stack which is not accessible by user-mode processes[*2].

Another possibility of the leakage of data on registers is that the registers may not be sanitized at software interruptions. Namely, the processing data of `tgt` on registers is evacuated to the kernel stack and it is not accessible by `atk`, but the data is still on registers and `atk` may be able to read them. We investigated the source code of Raspbian Linux kernel in order to check this possible threat and found that most of the registers are eventually sanitized. **Figure 3** sketches how the registers are sanitized when switching running processes. Assume that `tgt` is running at a software interruption. Then the context of `tgt` including the data on registers are evacuated to the kernel stack. Then the context of the next process, whose priority is the highest in the waiting list, is restored on registers. Namely, secret information of `tgt` is overwritten by the context of the next process. We confirmed that the registers `r4-r11, r13, r14` are directly overwritten by the restored context. The registers `r0-r2, r12` are used in the context switching process itself. And the register `r15`, which is as a program counter, is updated during any process running. There is a register `r3`, for that we could not find the corresponding description in the source code to overwrite it during the context switch. However, our experimental result indicates that `r3` is also sanitized somehow.

5.2 Information Leakage via Memory

The memory space allocated to `tgt` holds secret information and data in processing. `atk`, which has root privilege, and can access the memory space of `tgt`. The SBRES encrypts the data to be transferred to the memory space of `tgt` and there is no encryption key on memory, therefore it is not easy for `atk` to get meaningful information by reading the memory space of `tgt`.

5.3 The Security Not Assured by the SBRES

The adversary considered in Section 2.4 is passive and does not have access to the data on registers during the time that the target process uses it. On the other hand, if one of the conditions is not satisfied, the SBRES cannot be secure. Here we mention some cases which are out of the protection offered by the SBRES. The adversaries mentioned in Sections 5.3.1 and 5.3.2 violate the

[*2] There is a compile option generating `/dev/kmem`, which enables user-mode process to access kernel stack, but it is usually invalidated.

first requirement and one mentioned in Section 5.3.3 violates the second requirement.

5.3.1 Active Adversary

Different from the RAM encryption with hardware support, the SBRES implementation is just snippets of code embedded in the target program. Therefore, an active adversary can remove the snippets and insert a new code which reveals the secret on registers to memory before encryption. Therefore, the SBRES cannot be a countermeasure against this kind of adversary. In order to prevent these kinds of attacks, countermeasures to detect unexpected code modifications are necessary. The examples are code signing and control flow integrity [1]. Code obfuscation is an optional countermeasure. To obfuscate the SBRES code in the execution file of `tgt` may increase the cost of attacks.

5.3.2 Side Channel Attacks

The SBRES prevents the adversary from reading the secret information on the memory by encrypting the data. However, some side channel attacks do not directly read the memory but investigate the memory access pattern of the target process. This kind of adversary does not directly modify the program but is still active; It flushes the cache memory for example in order to control the behavior of the program. The SBRES does not provide protection against this kind of attacks. Another countermeasure such as Oblivious RAM [6], which randomizes the memory access pattern in processing, should be considered.

5.3.3 Kernel Mode Adversary

If the CPU or the kernel of the OS has a critical vulnerability, the adversary may be able to run in kernel-mode. In this case, the ability of the adversary is far beyond our assumption. `atk` is able to access to the kernel stack of `tgt` and to get the secret information on registers including the encryption key of the SBRES.

6. Concluding Remarks

In this paper, we discussed whether the RAM encryption function is feasible without hardware support. We first clarified the adversary model, then proposed a software-based RAM encryption scheme. The assumed adversary is semi-honest and cannot use debugging functions. As a result, it cannot access the data on registers. In other words, the data processing within the CPU is secure against this kind of adversary, therefore the proposed SBRES is secure under this setting. In addition, we proposed an encryption algorithm suitable for SBRES on ARMv7 processor and embedded it to SHA-256. The execution time for hashing becomes 3.8–6.4 times slower than one without SBRES. The optimization of the implementation and the automation of embedding SBRES are future works of this research.

References

[1] Abadi, M., Budiu, M., Erlingsson, Ú. and Ligatti, L.: Control-Flow Integrity, *CCS '05* (2005).

[2] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B. and Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers, Cryptology ePrint Archive, Report 2013/404 (2013), available from ⟨http://eprint.iacr.org/2013/404⟩.

[3] Biryukov, A., Velichkov, V. and Le Corre, Y.: Automatic Search for the Best Trails in ARX: Application to Block Cipher Speck, Cryptology ePrint Archive, Report 2016/409 (2016), available from ⟨http://eprint.iacr.org/2016/409⟩.

[4] Biryukov, A., Roy, A. and Velichkov, V.: Differential Analysis of Block Ciphers SIMON and SPECK, *Fast Software Encryption, FSE 2014*, Lecture Notes in Computer Science, Vol.8540, pp.546–570, Springer (2014).

[5] FIDO Alliance, available from ⟨https://fidoalliance.org/⟩ (accessed 2016-09-15).

[6] Goldreich, O. and Ostrovsky, R.: Software protection andsimulation on oblivious rams, *Journal of ACM* (1996).

[7] HCE Work Group: The Host Card Emulation in Payments: Options for Financial Institutions, Mobey Forum (2014), available from ⟨http://www.mobeyforum.org/w/wp-content/uploads/20141118-Mobey-Forum-HCE-Report-FINAL.pdf⟩ (accessed 2016-09-15).

[8] National Institute of Standards and Technology (NIST), Advanced Encryption Standard (AES), Federal Information Processing Standards, FIPS 197 (online), DOI: 10.6028/NIST.FIPS.197 (2001).

[9] National Institute of Standards and Technology (NIST), Secure Hash Standard (SHS), Federal Information Processing Standards, FIPS 180-4 (online), DOI: 10.6028/NIST.FIPS.180-4 (2015).

[10] Watanabe, D.: A note on the treat of malware infection for cryptographic functions in software implementation, IEICE Technical Report, ICSS2017-44, Vol.117, No.316, pp.35–40 (2017) (in Japanese).

[11] Watanabe, D., Yoshida, N., Sakamoto, J., Miyachi, R. and Matsumoto, T.: A note on system respecting adversary and the security of cryptographic process under well-behaved operation system, *Computer Security Symposium 2018*, 2A4-1 (2018) (in Japanese).

[12] Chow, S, Eisen, P.A., Johnson, H. and van Oorschot, P.C.: White-Box Cryptography and an AES Implementation, Nyberg, K. and Heys, H.M. (Eds.), *Selected Areas in Cryptography*, Lecture Notes in Computer Science, Vol.2595, pp.250–270, Springer (2002).

[13] Billet, O., Gilbert, H. and Ech-Chatbi, C.: Cryptanalysis of a White Box AES Implementation, *Selected Areas in Cryptography, SAC 2004*, Lecture Notes in Computer Science, Vol.3357, pp.227–240 (2005).

[14] CHES 2017 Capture the Flag Challenge, available from ⟨https://whibox.cr.yp.to/⟩ (accessed 2017-10-30).

[15] Bogdanov, A. and Isobe, T.: White-Box Cryptography Revisited: Space-Hard Ciphers, *ACM Conference on Computer and Communications Security 2015*, pp.1058–1069 (2015).

[16] Dunkelman, O., Keller, N. and Shamir, A.: Minimalism in Cryptography: The Even-Mansour Scheme Revisited, *Advances in Cryptology, EUROCRYPT 2012*, Lecture Notes in Computer Science, Vol.8238, pp.336–354 (2012).

[17] Fouque, P.-A., Karpman, P., Kirchner, P. and Minaud, B.: Efficient and Provable White-Box Primitives, Cryptology ePrint Archive: Report 2016/642 (2016).

[18] Matthew, H.: Intel SGX for Dummies (Intel SGX Design Objectives) (2015), available from ⟨https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx⟩ (accessed 2017-09-11).

[19] ARM Ltd.: TrustZone Technology Overview, available from ⟨http://www.arm.com/products/esd/trustzone_home.html⟩.

[20] Kaplan, D., Powell, J. and Woller, T.: AMD Memory Encryption, White paper (2016), available from ⟨https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf⟩.

[21] Beniamini, G.: Android linux kernel privilege escalation vulnerability and exploit (CVE-2014-4322), blog-post (2015), available from ⟨http://bits-please.blogspot.com/2015/08/android-linux-kernel-privilege.html⟩ (accessed 2019-06-17).

[22] Beniamini, G.: Exploring Qualcomm's Secure Execution Environment, blog-post (2016), available from ⟨http://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html⟩ (accessed 2019-06-17).

[23] Beniamini, G.: QSEE privilege escalation vulnerability and exploit (CVE-2015-6639), blog-post (2016), available from ⟨http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html⟩ (accessed 2019-06-17).

[24] Beniamini, G.: War of the Worlds - Hijacking the Linux Kernel from QSEE, blog-post (2016), available from ⟨http://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html⟩ (accessed 2019-06-17).

[25] Moghimi, A., Irazoqui, G. and Eisenbarth, T.: CacheZoom: How SGX Amplifies The Power of Cache Attacks, *Cryptographic Hardware and Embedded Systems, CHES 2017*, Lecture Notes in Computer Science, Vol.10529 (2017).

[26] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M.: Meltdown, Technical Report, arXiv:1801.01207 [cs.CR] (2018), available from ⟨https://arxiv.org/abs/1801.01207⟩.

[27] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution, Technical Report, arXiv:1801.01203 [cs.CR] (2018), available from ⟨https://arxiv.org/

abs/1801.01203⟩.
[28] Oishi, K. and Matsumoto, T.: Self destructive tamper response for software protection, *Proc. 6th ACM Symposium on Information, Computer and Communications Security AsiaCCS 2011*, pp.490–496 (2011).
[29] Malwarebytes 2017 State of Malware Report, available from ⟨https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf⟩ (accessed 2018-03-01).
[30] Exploit Database, available from ⟨https://www.exploit-db.com/⟩ (accessed 2018-03-05).
[31] Raspberry Pi, available from ⟨https://www.raspberrypi.org/⟩.

Appendix

A.1 Parameter choices of the encryption algorithm

In this section we explain the reason for the parameter choices of the block cipher specification given in Section 4.3.

A.1.1 Block Size and Key Length

The proposed SBRES assumes that the data on registers are not encrypted and the encryption/decryption is done when the data is transferred from the registers to the memory and vice versa. Therefore, to reduce the data transfer between them obviously improves the processing speed. The example target program (SHA-256) has eight working variables a, b, c, d, e, f, g, h (The labels refer to [9]) which are more frequently accessed than other variables. Therefore, our implementation allocates eight registers to them and an additional register to a variable which holds intermediate values in calculations. On the other hand, the SBRES at least requires registers to hold an encryption key and the intermediate value of the processing.

Let us consider the AES-128 as the encryption algorithm for the SBRES for example. Both the block size and the key length of AES-128 are 128-bit so that at least 8 registers are allocated to hold the state of the AES-128. Remind that 13 general purpose registers are available in ARMv7 processor therefore we have only five free registers. That means, we cannot hold all the eight working variables on registers. As a result, the processing speed of SHA-256 with SBRES-AES-128 is expected to be much slower than that shown in Section 4.4[*3].

From the viewpoint of the security, larger block size and longer key length are desirable so that the use of 64-bit block size and 64-bit key length should be restricted. In addition, the security of Even-Mansour construction is upper-bounded by a meet-in-the-middle attack and the attack requires about $2^{n/2}$ known plaintexts to recover the secret key, where n is the block size of the underlying block cipher. Therefore, the security of the proposed block cipher in Section 4.3 is upper-bounded to 2^{32} block encryptions. In this paper, we assume that the adversary has limited computation power and this limitation is not critical. If the program is expected to run for a long time beyond the above limitation, the encryption key for SBRES should be updated.

[*3] The parameter selection here is valid only for ARMv7. For example, ARMv8 processors have more general purpose registers. In addition, they have the AES instruction therefore the use of AES-128 may be a better choice

A.1.2 Related Plaintext Differential Attack

In this paper, the adversary is assumed to be able to access only the encrypted data on memory. The data on memory is decrypted on registers with an encryption key, processed by some instructions, encrypted again with the same key, and re-written to memory. Let us assume that the target process F is decomposed to sub-processes $F = F_m \circ F_{m-1} \circ \cdots \circ F_1$. Each sub-process F_i is further decomposed to $Write_i \circ \tilde{F}_i \circ Read_i$, where $Read_i$ and $Write_i$ are memory accesses and $\tilde{F}_i$ does not have memory access. Then the ciphertexts accessible by the adversary is depicted as follows:

$$P_i = \mathrm{Dec}(C_i, K),$$
$$P_{i+1} = \tilde{F}_i(P_i)$$
$$C_{i+1} = \mathrm{Enc}(P_{i+1}, K).$$

A naive attack assumes $\mathrm{Enc} \circ \tilde{F}_i \circ \mathrm{Dec}$ as the target function. Another possible attack is a related plaintext attack. Let us consider the simplest example of the sub-process $\tilde{F}_i$ xoring, i.e., F_i just xors a constant Δ to P_i. In this case, the adversary does not knows the plaintext P_i but knows the input differential Δ, and gets the ciphertexts C_i and C_{i+1} corresponding to the plaintexts P_i and $P_i \oplus \Delta$ respectively. Therefore, this setting is considered as a differential attack under the assumption that the adversary does not know plaintexts. The advantage of this attack compared to the naive attack is to reduce the complexity of the target function; the target function is not the composition of the functions but only an encryption function.

A.1.3 Number of Rounds

Biryukov et al. evaluated the differential probability of Speck in Ref. [3]. They reported that the maximum characteristic differential probability of Speck64 is smaller than 2^{-31} at 9 rounds. Another result of theirs via truncated evaluation expected that the differential probability is around 2^{-60} at 14 rounds. The security evaluation of the proposed block cipher is not our main concern in this paper, we choose 16 rounds which is expected to be sufficient for preventing differential attack and 8 rounds, its half rounds.

Note that Even-Mansour construction limits the number of encryption block by 2^{32}. The research of Biryukov indicates that around 10 iterations of a round function is moderate even if the related plaintext attack is applied. On the other hand, the appropriate number of rounds for our use is still unclear. The complexity of the related plaintext differential attack heavily depends on the complexity of the processing after decryption. If all of $\tilde{F}_i$ are not so simple, then we may be able to reduce the number of rounds.

A.1.4 Optimization for Register Consumption

To remove the key scheduling of Speck and to adopt 1-key Even-Mansour construction instead in the block cipher in Section 4.3 reduces the register consumption. It requires only two registers to hold the encryption key and two registers to hold the intermediate value of the processing. Our implementation encrypts only a 32-bit variable at a block encryption and the remainder 32-bit of the input is a fixed constant. The register which holds the data to be encrypted is used during the encryption to

hold the intermediate value. As a result, three additional registers in total are necessary in order to embed the SBRES with the proposed block cipher.

Tsutomu Matsumoto is a professor of the Faculty of Environment and Information Sciences, Yokohama National University, and directs the Research Unit for Information and Physical Security at the Institute of Advanced Sciences. He also serves as the Director of the Cyber Physical Security Research Center (CPSEC) at the National Institute of Advanced Industrial Science and Technology (AIST). Starting from Cryptography in the early '80s, he has opened up the field of security measuring for logical and physical security mechanisms. He received a Doctor of Engineering degree from the University of Tokyo in 1986. Currently, he is interested in research and education of Embedded Security Systems such as IoT Devices, Cryptographic Hardware, In-vehicle Networks, Instrumentation and Control Security, Tamper Resistance, Biometrics, Artifact-metrics, and Countermeasure against Cyber-Physical Attacks. He serves as the chair of the Japanese National Body for ISO/TC68 (Financial Services) and the Cryptography Research and Evaluation Committees (CRYPTREC) and as an associate member of the Science Council of Japan (SCJ). He was a director of the International Association for Cryptologic Research (IACR) and the chair of the IEICE Technical Committees on Information Security, Biometrics, and Hardware Security. He received the IEICE Achievement Award, the DoCoMo Mobile Science Award, the Culture of Information Security Award, the MEXT Prize for Science and Technology, and the Fuji Sankei Business Eye Award.

Ryo Miyachi received his Bachelor of Engineering and Master of Information Science from Yokohama National University, Kanagawa, Japan, in 2017 and 2019, respectively. His research interests include post-quantum cryptography and tamper-resistant implementation for cryptographic modules.

Junichi Sakamoto received his M.I.S and Doctor of Informatics degrees from Yokohama National University, Japan, in 2017, and 2020, respectively. He is currently working as a postdoctoral researcher at Yokohama National University. He has engaged in various researches regarding information security, including the methodology of secure implementation of the cryptographic algorithms, side-channel attacks to pairing computation, and laser-based fault attacks.

Manami Suzuki received her B.E. degree in information engineering and her M.S. degree in information sciences from Tohoku University, Sendai, Japan, in 2016 and 2018, respectively. She is currently with the Yokohama Research Laboratory, Hitachi Ltd. Her research interests include hardware security and physically unclonable functions.

Dai Watanabe received his B.S. and M.S. degrees from Tohoku Univeristy, Sendai, Japan, in 1994 and 1996 respectively, and received his Doctor degree from Tokyo University of Science in 2007. He has been engaged in research on information security, cryptography and cryptographic protocol at R&D Group, Hitachi, Ltd. since 1999. He is a member of the Information Processing Society of Japan (IPSJ) and The Institute of Electronics, Information and Communication Engineers (IEICE).

Naoki Yoshida is a Specially Appointed Assistant Professor in the Institute of Advanced Sciences at Yokohama National University. He received his M.S. and Ph.D. of Informatics degree from Yokohama National University in 2014, 2017. His research interests include embedded system security, artifact metrics and instrumentation security.