RegKey: A Register-based Implementation of ECC Signature Algorithms Against One-shot Memory Disclosure

YU FU and JINGQIANG LIN, School of Cyber Science and Technology, University of Science and Technology of China, China
DENGGUO FENG, Institute of Software, Chinese Academy of Sciences, China
WEI WANG, MINGYU WANG, and WENJIE WANG, Institute of Information Engineering, Chinese Academy of Sciences, China

To ensure the security of cryptographic algorithm implementations, several cryptographic key protection schemes have been proposed to prevent various memory disclosure attacks. Among them, the register-based solutions do not rely on special hardware features and offer better applicability. However, due to the size limitation of register resources, the performance of register-based solutions is much worse than conventional cryptosystem implementations without security enhancements. This paper presents RegKey, an efficient register-based implementation of **ECC (elliptic curve cryptography)** signature algorithms. Different from other schemes that protect the whole cryptographic operations, RegKey only uses CPU registers to execute simple but critical operations, significantly reducing the usage of register resources and performance overheads. To achieve this goal, RegKey splits the ECC signing into two parts, (1) complex elliptic curve group operations on non-sensitive data in main memory as normal implementations, and (2) simple prime field operations on sensitive data inside CPU registers. RegKey guarantees the plaintext private key and random number used for signing only appear in registers to effectively resist one-shot memory disclosure attacks such as cold-boot attacks and warm-boot attacks, which are usually launched by physically accessing the victim machine to acquire partial or even entire memory data but only once. Compared with existing cryptographic key protection schemes, the performance of RegKey is greatly improved. Regkey is applicable to different platforms because it does not rely on special CPU hardware features. Since RegKey focuses on one-shot memory disclosure instead of persistent software-based attacks, it works as a choice suitable for embedded devices or offline machines where physical attacks are the main threat.

CCS Concepts: • **Security and privacy** → *Digital signatures;*

Additional Key Words and Phrases: One-shot memory disclosure, registers, ECC signature algorithms, cryptographic key protection

This work was supported by National Key RD Plan of China (Grant No. 2020YFB1005803).
Authors' addresses: Y. Fu and J. Lin (corresponding author), School of Cyber Science and Technology, University of Science and Technology of China, 443 Huangshan Road, Hefei, China, 230027; emails: fuyu22@mail.ustc.edu.cn, linjq@ustc.edu.cn; D. Feng, Institute of Software, Chinese Academy of Sciences, 4 South Fourth Street, Beijing, China, 100190; email: fengdg@263.net; W. Wang, M. Wang, and W. Wang, Institute of Information Engineering, Chinese Academy of Sciences, 19 Shucun Road, Beijing, China, 100085; emails: wangwei@iie.ac.cn, wangmingyu@iie.ac.cn, wangwenjie@iie.ac.cn.

1539-9087/2023/11-ART97 $15.00
https://doi.org/10.1145/3604805

ACM Reference format:
Yu Fu, Jingqiang Lin, Dengguo Feng, Wei Wang, Mingyu Wang, and Wenjie Wang. 2023. RegKey: A Register-based Implementation of ECC Signature Algorithms Against One-shot Memory Disclosure. *ACM Trans. Embedd. Comput. Syst.* 22, 6, Article 97 (November 2023), 22 pages.
https://doi.org/10.1145/3604805

1 INTRODUCTION

Cryptosystems are widely used in computer systems and network applications. They rely on the confidentiality of cryptographic keys to ensure data security. However, cryptographic software usually does not provide proper protections for cryptographic keys. Since the key appears in memory in the form of plaintext, it could be obtained by adversaries through various memory disclosure attacks, which include one-shot memory disclosure [3, 7, 24, 37] mainly due to physical attacks and persistent software-based memory disclosure [1, 9, 25, 44] mainly caused by software vulnerabilities. Once adversaries physically access the computer and successfully launch one-shot memory attacks, they could obtain partial or even entire memory data and search the private key from the acquired memory. Software-based attacks usually capture a small area of memory for several times. For example, the OpenSSL Heartbleed attacks [1] allow adversaries to read 64KB memory content at random locations.

Since memory disclosure attacks pose a serious threat to sensitive data, various protection schemes have been proposed. Existing software protection schemes [6, 12, 25, 43] could prevent persistent software-based memory disclosure, but they are unable to prevent one-shot memory disclosure such as cold-boot and warm-boot attacks. In order to protect the key against one-shot memory disclosure, register-based scheme [16, 18, 41, 42, 47, 54, 56] and other schemes [10, 20–22, 29, 36] relying on special CPU hardware features are designed and implemented, but these schemes introduce non-negligible performance degradation. Compared with hardware-assisted schemes, register-based scheme does not require additional CPU hardware features and offers better applicability (see Section 2.2 for details). Register-based schemes implement cryptographic operations within CPU registers to prevent the key from appearing in memory, against memory disclosure attacks. TRESOR [42] stores the AES key and implements AES encryption operations in SSE registers. PRIME [18], RegRSA [56], and VIRSA [16] implement RSA decryption/signing in AVX or AVX-512 registers. Compared with RSA, **elliptic curve cryptography (ECC)** has shorter key size and has been widely used. Yang et al. [54] implements scalar multiplication of the ECC K-163 curve in AVX registers. However, the security strength of K-163 is lower than AES-128 and there is no extra registers to support larger key size. Besides, Yang's scheme focuses on only scalar multiplication and does not finish the complete ECDSA signature algorithm. Current register-based schemes have the following drawbacks.

- The scarce register resource is difficult to afford the space requirement for implementing whole cryptographic operations, resulting in significant performance degradation. Even if the register resource meets the requirements, it still brings two problems. The first problem is poor scalability. The limited register resource makes it difficult to support stronger cryptographic keys. The second is poor compatibility. Current register-based schemes occupy almost all available registers to finish cryptographic operations, which has a negative impact on concurrent programs.
- In order to directly control registers, researchers use platform-related assembly language to implement cryptosystems instead of using high-level programming languages, which significantly increases the effort of developing cryptosystems, especially for complex public key algorithms. So it is difficult to extend for more algorithms.

To solve the above problems, this paper presents RegKey, a register-based implementation for the mainstream ECC signature algorithms, including ECDSA [32], EC-Schnorr [13], and SM2 [2], to resist one-shot memory disclosure. The security goal of RegKey is to protect the private key d and the random number k for generating a signature (r, s), so that they will not appear in memory. In addition to the private key d, the random number k also needs to be kept secret from adversaries to avoid recovering d based on k and (r, s). In order to protect d, RegKey uses registers to store d and execute prime field operations related to d. Although k needs to be protected, considering performance degradation and storage requirement, RegKey does not conduct complex ECC scalar multiplication $k \times G$ in registers where G is the base point of the chosen elliptic curve. Instead, RegKey splits the random number k into two parts $k1$ and $k2$. By reorganizing an ECC signature algorithm into an equivalent algorithm, RegKey calculates $k \times G$ in memory, and only expose $k2$ in memory. The storage of secret $k1$ and the recovery of k from $(k1, k2)$ are conducted entirely inside registers, which prevents adversaries from obtaining k by one-shot memory attacks. More specifically, RegKey has an initialization phase that runs in the trusted environment before any ECC signing. The initialization phase generates secret $k1$ and calculates $k1 \times G$. During each ECC signing, RegKey generates random $k2$, calculates scalar multiplication $k2 \times G$, then uses $k2 \times G$ and precomputed $k1 \times G$ to calculate $k \times G$ in memory, which could be easily finished by calling existing cryptographic libraries. Because of the **elliptic curve discrete logarithm problem (ECDLP)**, adversaries could not recover $k1$ even they obtain $k1 \times G$ in memory through one-shot memory disclosure attacks. RegKey uses $k \times G$ to calculate r in memory, then uses secret $k1$ to recover k and uses (k, d) to calculate s in registers. RegKey does not modify the signature verification algorithms.

RegKey brings the following advantages. First, RegKey offers performance improvements compared with existing ECC-based signature protection schemes. Second, different from the schemes relying on special hardware features, RegKey only uses registers to protect sensitive data without requiring additional features. Last but not the least, compared with existing register-based schemes that protect the whole cryptographic operations, RegKey only uses registers to implement simple but critical operations, which significantly reduces the requirement of register resource and the effort of system development. The essential approach is that RegKey calculates complex ECC scalar multiplication $k \times G$ in memory without disclosing k into memory. RegKey focuses on preventing one-shot memory disclosure instead of persistent memory attacks, so that it could be a useful supplement to existing protection schemes [6, 12, 25, 43] that prevent persistent software-based memory disclosure. RegKey is suitable for scenarios where physical attacks are main threats, such as some embedded devices with specially designed software or offline machines where software attacks through network are avoided. The main contributions of this paper are as follows:

(1) We propose a register-based implementation method for ECC-based signature algorithms, including ECDSA, SM2, and EC-Schnorr, to effectively protect the private key against one-shot memory disclosure attacks. RegKey only protects simple but critical calculations in CPU registers after reorganizing the ECC signing process instead of protecting the whole signing operations, significantly saves required register resources, and reduces the effort of system development.
(2) We construct a RegKey prototype system and provide implementation details. We present the security analysis and evaluate the performance of RegKey. RegKey introduces 18% performance overhead compared with the regular signing implementation in OpenSSL. The performance overhead is acceptable while keeping the private key more secure. Besides, RegKey brings performance improvements compared with other cryptographic key protection schemes.

The rest of this paper is organized as follows. Section 2 presents the background and related work. Section 3 introduces preliminaries. The design and implementation of RegKey are described

in Sections 4 and 5, respectively. Section 6 evaluates RegKey from the aspects of performance and security, and compares RegKey with other cryptographic key protection schemes. Section 7 concludes this paper.

2 BACKGROUND AND RELATED WORK

2.1 One-shot Memory Disclosure Attacks

One-shot memory disclosure attacks have different patterns from persistent software-based memory disclosure attacks. Persistent software-based memory disclosure is mainly caused by software vulnerabilities, and could be remotely launched for multiple times without interrupting the running victim system. One-shot memory disclosure is another type of attack where adversaries obtain part of or even all the data remaining in RAM but only once. Then, adversaries search plaintext private keys or key-related data in the dumped memory content. After a one-shot memory disclosure attack, the state of the victim machine is changed. Therefore, the adversaries could not constantly monitor memory. One-shot memory disclosure is mainly caused by physical attacks such as cold-boot attacks [3, 24, 37] and warm-boot attacks [7], and also by crashing the system [9]. Cold-boot attacks and warm-boot attacks are launched based on the physical character of DRAM chips, so that they are difficult to completely defeat. If adversaries are physically accessible to the target machine, they could launch these attacks to acquire memory data.

Adversaries exploit the remanence effect of DRAM chips to launch cold-boot attacks [3, 24, 37]. Since the memory content lasts longer at low temperatures after out of power and is readable after the power is turned on again, adversaries could freeze the memory chip and plug it into another malicious machine to access the memory content.

BootJacker [7] proposed the concept of warm-boot attacks. BootJacker acquires full memory data on the victim machines across a warm boot. Since there is no power interruption during rebooting, the memory content is usually intact and the CMOS flag instructs BIOS to bypass memory tests in Linux kernel. Therefore, adversaries could reboot the target machine to a prepared malicious kernel and recover the memory content based on the data retention property of DRAM. These physical attacks pose a serious threat to sensitive data in memory when a cryptographic system is running.

2.2 Cryptographic Key Protection

In order to prevent the cryptographic key from various memory disclosure attacks, researchers have proposed register-based protection schemes and other protection schemes based on special CPU hardware features.

2.2.1 Register-based Protection Schemes. Since registers are inside the processor and RAM is outside the processor, register-based schemes could make full use of registers to store sensitive data and execute cryptographic operations without disclosing secrets into memory. As register-based implementations of symmetric cryptography, AESSE [41], Amnesia [47], and TRESOR [42] store the AES key and complete AES encryption/decryption entirely inside CPU registers.

Compared with symmetric cryptography, the implementation of public key cryptography needs much more storage space while the space provided by registers is very limited, which significantly reduces the performance. As register-based RSA implementations, PRIME [18], RegRSA [56], and VIRSA [16] conduct RSA decryption/signing inside registers, and optimize the performance through vector instructions and various acceleration algorithms in modular exponentiation. Due to the limitation of registers, the key size of current RSA protection schemes is only 2048-bit and is difficult to support stronger keys. Although ECC has shorter keys, the difficulty of implementing ECC is increased because of the complex ECC calculations. At present, the required security

strength is ECC P-256, which equals to AES-128 and RSA-2048. Although Yang et al. [54] completes scalar multiplication of K-163 based on YMM registers, the security strength of K-163 is lower than AES-128 and there is no extra register space to support larger ECC keys. Besides, Yang's scheme achieves only scalar multiplication instead of complete ECDSA signature algorithms. To the best of our knowledge, there is no register-based implementation of complete ECC signing.

2.2.2 Other Protection Schemes. Other protection schemes rely on special CPU hardware features to resist memory disclosure. ERIM [51] and CryptoMPK [31] use Intel **MPK (Memory Protection Key)** to protect the private key in RAM by changing the access permission of memory pages where sensitive data resides. However, MPK only resists memory disclosure attacks caused by software vulnerabilities, and could not resist physical attacks, such as cold-boot attacks. Intel **SGX (Software Guard Extensions)** [11] provides the trusted execution environment for sensitive applications, which is called *enclave*. SGX-based scheme [29] executes cryptographic operations inside an enclave to protect sensitive data from software vulnerabilities and physical attacks. Mimosa [21, 36] utilizes Intel **TSX (Transactional Synchronization Extensions)** to protect the private key during RSA decryption. Mimosa completes RSA decryption in CPU caches and keeps sensitive data in the TSX write set. Once other threads access sensitive data in the write set, TSX triggers a conflict, resulting in the termination of the transaction. However, Intel does not support SGX and TSX on recent desktop CPUs. Copker [20, 22] adopts the **CAR (Cache-as-RAM)** mechanism to protect RSA and ECDSA private-key operations, which is built on top of CPU cache-filling modes in x86 platforms. Copker stores sensitive data in CPU caches instead of DRAM. However, the performance of Copker is seriously degraded under the multi-thread mode. Sentry [10] employs cache locking in ARM platforms to resist physical attacks.

Among various cryptographic key protection schemes, register-based schemes have special advantages. They do not require additional CPU hardware features, and only rely on register resources of the processor to prevent memory disclosure. Therefore, register-based solution is more applicable to different platforms.

3 PRELIMINARIES

3.1 Elliptic Curve Cryptography

ECC has been widely used in the real world because it has the advantages of small key length, fast speed, and fewer computing resources. ECC was proposed by Miller [40] and Koblitz [35], and the time complexity of solving k from scalar multiplication $k \times P$ is exponential [17] where k is a scalar and P is a point on the chosen elliptic curve. There are two types of operations on a finite cyclic group: point addition and point doubling on elliptic curves, while scalar multiplication will call point addition and point doubling for multiple times. The underlying operations are prime field arithmetic operations, including modular addition, modular multiplication, modular square, modular inversion, and reduction. The upper-level operations are group operations based on the underlying prime field operations. Scalar multiplication is the most complex and time-consuming operation in ECC.

Several digital signature algorithms are designed on ECC. The signature algorithms rely on the private key d to sign messages and use the public key $Q = d \times G$ to verify a signature. Each signing uses a different random number k to generate a signature. The random number is regarded as an ephemeral private key and should not be exposed. Once adversaries acquire the random number, they could recover the private key based on the obtained random number and signature. The mainstream ECC-based signature algorithms include ECDSA, EC-Schnorr, and SM2. As the most commonly adopted signature algorithm, ECDSA [32] offers a variant of DSA on elliptic curves. The ECDSA signing is shown in Algorithm 1. The EC-Schnorr signature algorithm [13] is considered

as a replacement of ECDSA in bitcoin, which shares the same elliptic curve P-256. SM2 is specified by Chinese State Cryptography Administration and the SM2 digital signature algorithm [2] is officially accepted in ISO/IEC14888-3/AMD1.

ALGORITHM 1: ECDSA Signing

Input: Message m, private key d
Output: Signature (r, s)

1: Calculate $e = H(m)$ where H is a hash function
2: Generate a random number $k \in [1, n-1]$ where n is the order of the base point G
3: Calculate $(x, y) = k \times G$
4: Calculate $r = x \bmod n$; if $r = 0$, go back to Step 2
5: Calculate $s = k^{-1} \times (e + r \times d) \bmod n$; if $s = 0$, go back to Step 2
6: Return (r, s)

3.2 User-accessible Registers in x86-64

Registers are a set of high-speed storage components with limited storage capacity inside a processor. Registers are categorized into user-accessible registers and special internal registers depending on whether they could be read or written by instructions. User-accessible registers are classified into two categories, which are scalar registers and vector registers. The x86-64 platform provides sixteen 64-bit **general-purpose registers (GPRs)** as scalar registers. Vector registers [28, 39] include MM, XMM, YMM, and ZMM registers. The least significant 256-bit of ZMM registers are YMM registers and the least significant 128-bit of YMM registers are XMM registers. Both scalar and vector registers could be used to store data and perform calculations. Besides, four 64-bit privileged debug registers dr0-dr3 could be used in Linux kernel, for occupying these privileged registers does not cause obvious impacts on kernel functions [42].

Some vector instructions can be applied to efficiently implement cryptographic operations on vector registers. For instance, Intel has proposed the AES-NI [23] instruction set for x86 platforms in 2008, which implements AES encryption/decryption with high performance. At present, Intel, AMD, and ARM issue a wide variety of products with AES support. However, there is no instruction set directly related to public key cryptography for users.

4 DESIGN

In this section, we first describe the threat model, then introduce important design principles of RegKey. Next, we present how to use RegKey to protect ECDSA signing in the modes of single-thread and multi-thread. Finally, we describe other RegKey-protected ECC-based signing.

4.1 Threat Model

The primary goal of RegKey is to protect the private key d against one-shot memory disclosure during ECC signing. Adversaries could access the target machine to launch one-shot memory disclosure attacks and search sensitive data in dumped memory data. When one-shot memory disclosure is successful launched, the attack will be detected because the state of the target machine is inevitably changed. For example, the machine running signing services is powered off after a cold-boot attack [3, 24, 37], and is rebooted after a warm-boot attack [7]. Since the signing service is interrupted after a one-shot memory disclosure attack, adversaries obtain all signing-related memory contents only once before the victim user restarts the signing service.

Before any ECC signing, the offline key generation phase and system initialization phase need to be executed (see Section 4.2 Principle 3 for details). The key generation phase is only executed once,

and the initialization phase is executed after the machine is restarted or a one-shot memory disclosure attack. A reasonable assumption is that the key generation phase and the initialization phase are finished without any attacks. These procedures are transient and performed by the system user usually in an offline environment before any ECC signing. After the initialization phase, ECC signing phase is started. Adversaries could launch one-shot memory disclosure during any ECC signing to obtain signing-related memory contents at any certain time, but only once because a successful attack will interrupt the signing service. After the victim user restarts the signing service, adversaries could launch attacks again to capture more signing-related contents in memory. RegKey ensures that the adversaries never find sensitive data related to d and k from the dumped memory.

RegKey assumes a trusted **operating system (OS)** kernel. Adversaries could not write malicious codes to the target machine's OS kernel and access the private key through injected malicious codes, such as TRESOR-HUNT [5]. Existing OS integrity solutions [26, 30, 33] could work complementarily with RegKey. Also, RegKey does not consider side-channel attacks and assumes that cryptographic algorithms are semantically secure. Last, RegKey is designed to resist one-shot attacks instead of persistent memory attacks.

4.2 Design Principles

In order to defeat against one-shot memory disclosure attacks, RegKey ensures that sensitive data will never appear into memory, including the private key d and the random number k. Once adversaries obtain d from memory, they could forge signature for any message. The random number k has the same security requirements as d because adversaries could recover d based on k and a signature (r, s). The design of RegKey needs to satisfy the following principles.

Principle 1. RegKey uses registers to protect d and k during each ECC signing. RegKey regards on-chip CPU registers as secure areas to store sensitive data and execute sensitive operations. RegKey protects d and k in different ways. During each ECC signing, the calculation related to d is a simple prime field operation, which can be entirely implemented inside limited registers. However, directly implementing calculations related to k inside registers is rather challengable. The most complex and time-consuming group operation in ECC is scalar multiplication. The computation associated with k happens to be scalar multiplication $k \times G$. Executing $k \times G$ within limited registers is very difficult due to the requirements of storage space and probably brings remarkable performance degradation. Although Yang's scheme [54] implements scalar multiplication of K-163 inside registers, the required register space of P-256, P-384, and P-512 greatly exceeds K-163.

In order to protect k in registers, RegKey divides k into two independent integers $k1$ and $k2$, and ensures that only $k2$ is exposed in memory for calculating $k \times G$. The secret $k1$ in RegKey has the same security requirements as k in original ECC signature algorithms. That is, $k = (k1+k2) \bmod n$, where k, $k1$ and $k2$ have the same range. When RegKey chooses P-256, both k, $k1$ and $k2$ could be 256-bit integers. Generating constant secret $k1$ and calculating scalar multiplication $k1 \times G$ are finished in the initialization phase, which runs in the secure environment before any ECC signing. During each signing, RegKey generates random $k2$, calculates $k2 \times G$, and then uses $k2 \times G$ and precomputed $k1 \times G$ to calculate $k \times G$ in memory. Due to the ECDLP, adversaries could not recover $k1$ even if they obtain $k1 \times G$ in memory.

With the effective protection of d and k, RegKey could be used to calculate a signature (r, s). The process of calculating r includes complex scalar multiplication $k \times G$, and does not involve sensitive $k1$ and d. Therefore, RegKey calculates r in memory without special protections, which could be easily finished by calling existing common cryptographic libraries. The process of computing s from r involves sensitive $k1$ and d, but requires only simple prime field operations, such as modular addition, modular multiplication and perhaps an inversion. Therefore, RegKey calculates s inside CPU registers. Since calculations of s consist of only prime field operations, the usage of register

resources and the performance overhead are significantly reduced. In addition, if any part of the computation for obtaining s from r is non-critical (does not involve sensitive data), it can also be performed in memory. Section 4.3 describes RegKey-protected ECDSA signing with details.

In summary, because d and $k1$ are protected in registers, adversaries could not directly obtain or infer them even if $(k2, r, s)$ is learned from memory.

Principle 2. RegKey ensures that each tuple ($k2$, r, s) acquired by adversaries corresponds to different $k1$. If different $(k2, r, s)$ tuples acquired by adversaries correspond to one $k1$, the private key d could be inferred, as described in Sections 4.3 and 4.4. RegKey considers the situations of multiple attacks and multiple threads.

RegKey is supposed to resist multiple one-shot memory disclosure attacks (but not persistent memory attacks) as follows. (1) Adversaries could launch the first attack, and acquire one tuple $(k2, r, s)$ from the thread running signing service. (2) Because of the "one-shot" feature, the signing service is terminated after an attack. (3) RegKey regenerates $k1$, then restarts the signing service with newly-generated $k1$. (4) Adversaries could launch the second attack, and acquire another $(k2, r, s)$ from the thread running signing service. Since RegKey changes $k1$ after an attack, each tuple $(k2, r, s)$ acquired from multiple attacks corresponds to different $k1$. In fact, RegKey always regenerates $k1$, when the machine reboots.

RegKey supports multiple threads of ECC signing. For a machine with j logical cores, j threads running signing service will fully occupy the CPU at a certain time, which implies that adversaries could obtain up to j different tuples $(k2, r, s)$ from j threads through a one-shot memory attack. RegKey assigns different $k1$ to each concurrent thread. Therefore, each tuple $(k2, r, s)$ acquired from multiple threads still corresponds to different $k1$. When adversaries launch one-shot memory disclosure attacks for multiple times, RegKey updates $k1$ for each thread after an attack, then restarts the signing service.

Principle 3. RegKey generates ciphertext parameters and plaintext parameters used for signing in the initialization phase. An offline key generation phase and a system initialization phase need to be performed before any ECC signing. The key generation phase produces the AES key and ECC private key d. The initialization phase provides ciphertext parameters ($Enc(k1)$, $Enc(d)$) and plaintext parameters $k1 \times G$ for each ECC signing thread. More specifically, the initialization phase (1) generates random $k1 \in [1, n-1]$, (2) calculates scalar multiplication $k1 \times G$, and (3) uses the AES key to encrypt $k1$ and d, which outputs ciphertext ($Enc(k1)$, $Enc(d)$). Ciphertext parameters ($Enc(k1)$, $Enc(d)$) and plaintext parameter $k1 \times G$ are stored in hard disk or any other non-volatile storage. These parameters are loaded into memory from hard disk in the subsequent ECC signing phase. Register-based TRESOR [42] is applied to protect the AES key. During the initialization phase, RegKey integrates TRESOR, which stores the AES key in debug registers with Ring-0 privilege to prevent access from user applications. During each ECC signing, ciphertext ($Enc(k1)$, $Enc(d)$) are passed from memory into registers. Then, RegKey uses the AES key stored in debug registers to recover $k1$ and d inside registers. After a one-shot memory disclosure attack (or reboot), the signing service is interrupted, and then Regkey performs the initialization phase again to generate new ($k1$, $k1 \times G$, $Enc(k1)$).

4.3 Using RegKey to Protect ECDSA Signing

This section describes how to use RegKey to protect ECDSA signing in single thread.

Prior to all protected ECDSA signing, RegKey needs to perform the initialization operations as described above, which calculates plaintext parameters $k1 \times G$ and ciphertext parameters ($Enc(k1)$, $Enc(d)$) used for generating signatures.

Next, RegKey performs the protected ECDSA signing. The specific operation of the protected ECDSA signing is given in Algorithm 2. The random number k used for signing in Algorithm 2

is the result of the modular addition of $k1$ and $k2$, that is, $k = (k1 + k2) \bmod n$. Thus, ECDSA signing is divided into two parts of sequential executions. Steps 1–5 in Algorithm 2 do not involve sensitive data but involve complex scalar multiplication, running in memory. The results of Steps 1–5 (e, $k2$, r) and the encrypted secrets ($Enc(k1)$, $Enc(d)$) are passed to Steps 7–10 protected in CPU registers. Inside CPU registers, Algorithm 2 decrypts ($Enc(k1)$, $Enc(d)$) to obtain plaintext ($k1$, d) by the AES key in privileged registers, then uses ($k1$, $k2$) to recover k, finally calculates $s = k^{-1} \times (e + r \times d) \bmod n$, which are simple operations compared with scalar multiplication. Although Algorithm 2 has introduced additional AES decryption, the execution time of AES operation is negligible compared with ECDSA signing.

In Algorithm 2, $k \times G = (k1 + k2) \times G = k1 \times G + k2 \times G$. The value of $k1 \times G$ has been prepared in advance during the initialization phase and is passed to Algorithm 2 as a parameter. The operations required to obtain $k \times G$ contain scalar multiplication $k2 \times G$ and a point addition that adds $k2 \times G$ to $k1 \times G$. Compared with the original steps of ECDSA signing, the introduced performance overhead after splitting k is only a point addition, about $\frac{1}{1.5L}$ of scalar multiplication $k \times G$ where L is the bit length of k. There is no modification for the ECDSA verification algorithm.

ALGORITHM 2: The Protected ECDSA Signing

Input: Plaintext parameters including message m and $k1 \times G$, ciphertext parameters including $Enc(k1)$ and $Enc(d)$
Output: Signature (r, s)

1: In memory, calculate $e = H(m)$ where H is a hash function
2: In memory, generate a random number $k2 \in [1, n-1]$
3: In memory, calculate the scalar multiplication $k2 \times G$
4: In memory, calculate $(x, y) = (k1 + k2) \times G = k1 \times G + k2 \times G$
5: In memory, calculate $r = x \bmod n$; if $r = 0$, go back to Step 2
6: Pass the parameters e, $k2$, r, $Enc(k1)$, $Enc(d)$ from memory to CPU registers
7: Inside CPU registers, decrypt $Enc(k1)$ to recover $k1$
8: Inside CPU registers, decrypt $Enc(d)$ to recover d
9: Inside CPU registers, recover $k = (k1 + k2) \bmod n$
10: Inside CPU registers, use the plaintext $k1$ and d to calculate $s = k^{-1} \times (e + r \times d) \bmod n$
11: Return (r, s)

According to Algorithm 2, adversaries could obtain one tuple $(e, k2, r, s)$ from memory through a one-shot memory disclosure attack under the single-thread mode. Since $k1$ is secret, adversaries could not recover private key d with the equation $d = r^{-1} \times [s \times (k1 + k2) - e] \bmod n$. However, adversaries might try to launch attacks for multiple times. We assume adversaries acquire $(k2, e, r, s)$ in the first attack and acquire $(k2', e', r', s')$ in the second attack. If two tuples correspond to the same $k1$, $(k1, d)$ could be solved by associating Equation (1) with Equation (2), that is $k1 = (s \times r' - s' \times r)^{-1} \times (e \times r' - e' \times r + k2' \times s' \times r - k2 \times s \times r') \bmod n$. Therefore, RegKey needs to change $k1$ after detecting a one-shot memory disclosure attack.

$$s = (k1 + k2)^{-1} \times (e + r \times d) \bmod n \tag{1}$$

$$s' = (k1 + k2')^{-1} \times (e' + r' \times d) \bmod n \tag{2}$$

However, when a one-shot memory disclosure attack is successful launched, the running state of the target machine is changed. Cold-boot attack [3, 24, 37] makes the machine powered off and warm-boot attack [7] reboots the machine, both of which would result in the termination of the running ECDSA signing service. With the unauthorized interruption of the signing service,

RegKey enters the initialization phase again, which regenerates $k1$, computes updated $k1 \times G$ and encrypts $k1$ using the AES key. Then, RegKey restarts the signing service with different $k1$, so this attack is impossible.

4.4 Multi-thread Support

Taking the protected ECDSA signing in Algorithm 2 as an example, when the multi-thread mode is adopted, multiple signing instances are running in parallel in memory at a certain time. On a multi-core device, the number of concurrently-running threads is consistent with the number of cores. For a device with j logical cores, since one thread used for signing occupies one logical core, the maximum number of concurrent signing instances is j. Therefore, RegKey starts j threads running signing services at most. More threads do not improve performance because j threads could fully occupy CPU at a certain time. The tuple $(k2_i, e_i, r_i, s_i)$ represents parameters in the i-th thread ($i \in [1, j]$). We assume adversaries could launch a one-shot memory disclosure attack and successfully learn every tuple from j parallel signing instances, including $(k2_1, e_1, r_1, s_1)$, $(k2_2, e_2, r_2, s_2)$, ..., $(k2_j, e_j, r_j, s_j)$. When these tuples acquired from different threads correspond to same $k1$, adversaries could exploit the following equations to infer $(k1, d)$. With Equation (3) and Equation (4), the adversaries solve $k1 = (s_1 \times r_2 - s_2 \times r_1)^{-1} \times (e_1 \times r_2 - e_2 \times r_1 + k2_2 \times s_2 \times r_1 - k2_1 \times s_1 \times r_2) \bmod n$. Then, adversaries could recover $d = r_1^{-1} \times [s_1 \times (k1 + k2_1) - e_1] \bmod n$ from the determined $k1$.

$$s_1 = (k1 + k2_1)^{-1} \times (e_1 + r_1 \times d) \bmod n \tag{3}$$

$$s_2 = (k1 + k2_2)^{-1} \times (e_2 + r_2 \times d) \bmod n \tag{4}$$

...

$$s_j = (k1 + k2_j)^{-1} \times (e_j + r_j \times d) \bmod n \tag{5}$$

Therefore, RegKey needs to assign different $k1$ to each thread. For a device with j logical cores, j threads running protected signing service are started with different $k1$. The tuple $(k1_i, k2_i, e_i, r_i, s_i)$ represents parameters in the i-th thread ($i \in [1, j]$). Plaintext $(k1_1, k1_2, ..., k1_j, d)$ only appears inside registers during protected ECDSA signing. If adversaries could launch one-shot memory disclosure attacks for multiple times, RegKey changes $k1$ for each thread, similar to the single-thread mode. After detecting a one-shot memory disclosure attack, RegKey regenerates $(k1_1, k1_2, ..., k1_j)$, then restarts the threads running signing services with updated $(k1_1, k1_2, ..., k1_j)$.

In our 4-core 8-thread device with Hyper-Threading support, RegKey starts eight threads and assigns eight different $k1$ to corresponding threads. The signing instances in the same thread share the same $k1$ and the signing instances in different threads have various $k1$, which ensures that RegKey could resist one-shot memory disclosure attacks under the multi-thread mode.

4.5 Using RegKey to Protect Other ECC-Based Signing

Since ECC-based digital signature algorithms have the similarity that complex scalar multiplication related to k happens during calculating r, RegKey is also well applied to other ECC-based signing. This section describes RegKey-protected EC-Schnorr signing and RegKey-protected SM2 signing in the single thread. For the multi-thread mode, RegKey assigns different $k1$ to different threads running signing services.

4.5.1 RegKey-Protected EC-Schnorr Signing. Prior to all EC-Schnorr signing, the offline key generation phase and initialization phase need to be performed as described in Principle 3 in Section 4.2. The protected EC-Schnorr signing is presented in Algorithm 3.

ALGORITHM 3: The Protected EC-Schnorr Signing

Input: Plaintext parameters including message m and $k1 \times G$, ciphertext parameters including $Enc(k1)$ and $Enc(d)$
Output: Signature (r, s)

1: In memory, generate a random number $k2 \in [1, n-1]$
2: In memory, calculate the scalar multiplication $k2 \times G$
3: In memory, calculate $r = (k1 + k2) \times G = k1 \times G + k2 \times G$
4: In memory, calculate $H(Q||r||H(m))$ where Q is the public key and H is a hash function
5: Pass the parameters $k2$, $H(Q||r||H(m))$, $Enc(k1)$ and $Enc(d)$ from memory to CPU registers
6: Inside CPU registers, decrypt $Enc(k1)$ to recover $k1$
7: Inside CPU registers, decrypt $Enc(d)$ to recover d
8: Inside CPU registers, recover $k = (k1 + k2) \bmod n$
9: Inside CPU registers, calculate $s = k + H(Q||r||H(m)) \times d$
10: Return (r, s)

ALGORITHM 4: The Protected SM2 Signing

Input: Plaintext parameters including message m and $k1 \times G$, ciphertext parameters including $Enc(k1)$ and $Enc(d)$
Output: Signature (r, s)

1: In memory, calculate $e = H(\overline{M})$ where H is a hash function
2: In memory, generate a random number $k2 \in [1, n-1]$
3: In memory, calculate the scalar multiplication $k2 \times G$
4: In memory, calculate $(x, y) = (k1 + k2) \times G = k1 \times G + k2 \times G$
5: In memory, calculate $r = (e + x) \bmod n$; if $r = 0$, go back to Step 2
6: Pass the parameters $k2$, r, $Enc(k1)$, $Enc(d)$ from memory to CPU registers
7: Inside CPU registers, decrypt $Enc(k1)$ to recover $k1$
8: Inside CPU registers, decrypt $Enc(d)$ to recover d
9: Inside CPU registers, recover $k = (k1 + k2) \bmod n$
10: Inside CPU registers, calculate $s = (1 + d)^{-1} \times (k - r \times d) \bmod n$
11: Return (r, s)

Algorithm 3 executes operations without sensitive data in memory, including calculating r and $H(Q||r||H(m))$. Since both Q and m are public parameters, $H(Q||r||H(m))$ is calculated in memory after obtaining r. Operations involving sensitive data are implemented inside CPU registers, including recovering the random number k, using k and d to calculate $s = k + H(Q||r||H(m)) \times d$. Again, there is no modification for the EC-Schnorr verification algorithm.

4.5.2 RegKey-Protected SM2 Signing. The protected SM2 signing is presented in Algorithm 4.

Algorithm 4 executes operations without sensitive data in memory, including calculating $hash(\overline{M})$ and calculating r where $\overline{M}$ is a known parameter related to message. Operations involving sensitive data are implemented inside CPU registers, including recovering the random number k, using k and d to calculate $s = (1 + d)^{-1} \times (k - r \times d) \bmod n$. There is no modification for the SM2 verification algorithm.

5 IMPLEMENTATION

In this section, we build the RegKey prototype system. We present system architecture and implementation details. In particular, we integrate register-protected ECC signing operations into Linux kernel and ensure its atomicity.

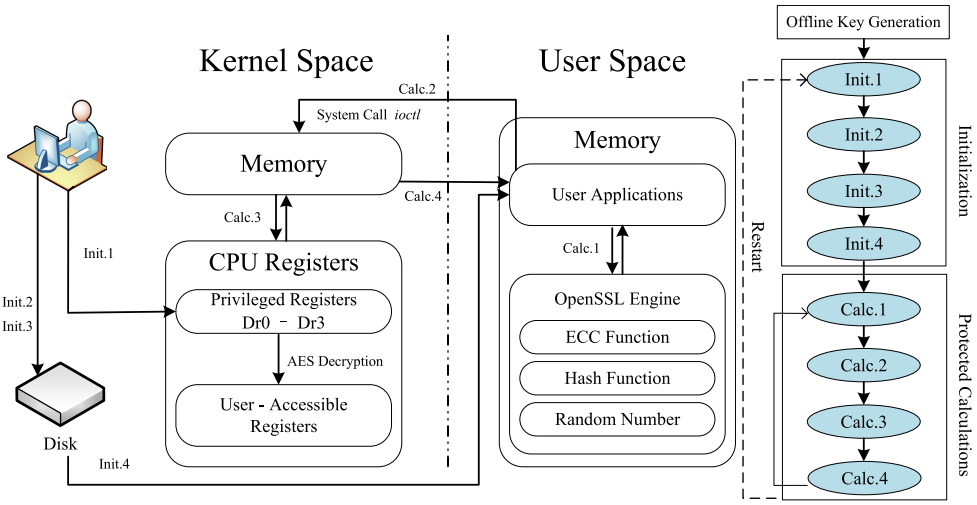

Fig. 1. The system architecture of RegKey.

5.1 System Architecture

The system allocates different running environments for protected and unprotected data as shown in Figure 1, which are CPU registers of kernel space and memory in user space, respectively. Memory of user space is an unprotected area and could directly call interfaces of existing cryptographic libraries. The widely-used OpenSSL library is chosen to provide various cryptographic functions for user applications. Calculations executed inside CPU registers are integrated into Linux kernel and provide an interface for user-space applications. User space accesses the kernel module through `ioctl` system call, which passes parameters and requests to perform register-protected ECC signing. Kernel RegKey module receives parameters, completes operations involving sensitive data inside registers, then returns the signature.

In order to access the privileged registers and ensure atomicity, the system implements register-protected operations as a kernel module. The system utilizes TRESOR [42], which selects debug registers at Ring-0 privilege to store an AES key, so that user-space applications at Ring 3 privilege could not directly access this AES key. Debug registers dr0-dr3 are enough to hold an 128/256 bits AES key that is used to decrypt ciphertext parameters passed from user applications. The kernel module also employs the interrupt controls of OS to ensure atomicity of the register-protected operations.

The flow of the system is shown in Figure 1. The system user first performs offline key generation operations only once, before the initialization phase and the signing phase. When the machine is started, initialization operations (Init.1, Init.2, Init.3 and Init.4) are performed. Besides, after a one-shot memory disclosure attack, the signing service is terminated. The system needs to restart the initialization phase, which regenerates $k1$ for each thread, calculates new $(k1 \times G, Enc(k1))$. After the initialization operations, the system performs repeated protected signing calculations (Calc.1, Calc.2, Calc.3 and Calc.4). The key generation phase and initialization phase are assumed secure because these procedures are transient and performed by the system user in the offline environment before any signing.

- **Offline Key Generation.** Generate the AES key and ECC private key d.
- **Init.1.** Store the AES key in debug registers dr0-dr3 with Ring-0 privilege.

Table 1. The Interfaces Provided by OpenSSL

Function	Interface
Hash function	EVP_DigestInit_ex EVP_DigestUpdate EVP_DigestFinal_ex
ECC scalar multiplication	EC_POINT_mul
ECC point addition	EC_POINT_add
Get r from (x, y)	EC_POINT_get_affine_coordinates
Reduction	BN_nnmod
Judge the value is zero or not	BN_is_zero
Verify the ECDSA signature	ECDSA_do_verify

- **Init.2.** Calculate $Enc(d)$ with the AES key, and store $Enc(d)$ in hard disks or other non-volatile storage.
- **Init.3.** Generate $k1$, calculate $k1 \times G$ and $Enc(k1)$, then store $(k1 \times G, Enc(k1))$ in hard disks or other non-volatile storage.
- **Init.4.** A file containing plaintext $k1 \times G$ and ciphertext $(Enc(k1), Enc(d))$, is loaded into memory.
- **Calc.1.** Use non-sensitive data to calculate r in user-space memory based on OpenSSL.
- **Calc.2.** Pass parameters from user applications to Linux kernel through `ioctl` system call.
- **Calc.3.** Calculate s inside CPU registers in kernel space.
- **Calc.4.** Return signature (r, s) from kernel space to user space.

5.2 Implementation Details

We build the prototype system on Intel Core i7-1065G7 CPU, with Ubuntu 18.04 as the OS. The version of OpenSSL cryptographic library is OpenSSL-1.1.1k.

Operations in Memory. The system employs OpenSSL engine [45] to realize various unprotected operations in memory including hash functions, necessary prime field calculations, and group operations on elliptic curves, such as scalar multiplication and point addition. The required interfaces of OpenSSL are shown in Table 1. After the whole signing process is finished including operations in memory and operations inside registers, the generated signature is returned back to user space. The system could call OpenSSL interface `ECDSA_do_verify` to verify the signature.

Data Conversion. After operations in memory are finished, the computed intermediate results are transferred to CPU registers for calculating s. However, OpenSSL defines a type of BIGNUM for large integer calculations while the operands in registers are represented as 64-bit unsigned longs. For example, `MULX` multiplies two 64-bit integers stored in GPRs but the size of a BIGNUM variable is larger than 64 bits. A conversion between BIGNUM and unsigned long array is required. When the P-256 elliptic curve is selected, the operand a in BIGNUM is 256-bit and could be turned into an unsigned long array $b[]$ through the newly-defined `BN_2_Ulong` function. `Ulong_2_BN` is used to transform the unsigned long array $b[]$ to a BIGNUM variable a. The code of `BN_2_Ulong` and `Ulong_2_BN` are listed in the Appendix.

Operations inside Registers. The plaintext parameters $(e, k2, r)$ and ciphertext parameters $(Enc(k1), Enc(d))$ are passed from memory to registers as the form of 64-bit unsigned long array. The system utilizes the AES key stored in debug registers dr0-dr3 to decrypt $Enc(k1)$ and $Enc(d)$ to obtain plaintext $(k1, d)$. The AES key is moved from debug registers to general-purpose registers, then is moved to XMM registers. The system makes full use of Intel AES-NI instructions [23] to efficiently realize AES decryption based on XMM registers. The ASM code of AES-NI is presented

Table 2. Performance Overhead of Additional Functions

Supplementary Function	Interface	Execution Time (μs)
ECC point addition	EC_POINT_add	11
System call	IOCTL	16
AES Decryption	Intel AES-NI	2
Conversion from BIGNUM to Unsigned Long Array	BN_2_Ulong	4
Conversion from Unsigned Long Array to BIGNUM	Ulong_2_BN	3

in the Appendix. User-accessible registers, including scalar registers and vector registers, are regarded as secure buffers to execute signing operations involving sensitive data. In order to directly control registers, the system uses assembly language to perform calculations, including using ($k1$, $k2$) to recover k and using (k, d, e, r) to calculate s. Since registers are scarce resources inside processors, RegKey only implements simple prime field operations inside registers rather than complex group operations on elliptic curves. When the RegKey-protected ECDSA signing is finished, the system clears all the data remaining inside registers and returns signature (r, s) to user space.

5.3 Building Execution Environments for RegKey

The system regards operations inside registers as a `char` module and compile it to Linux kernel. The module provides services through the `ioctl` system call, which is the interaction channel between user space and kernel space. User-space applications use `ioctl` to request the register-protected calculations in Linux kernel and receive signatures. The system also uses interrupt controls of OS to ensure atomicity of operations inside registers.

Since the plaintext private key, the random number used for signing and sensitive intermediate results are kept inside CPU registers, atomicity of register-protected operations needs to be guaranteed to prevent context switch, which passively transfers sensitive data from registers into memory. Existing works [16, 18, 22, 42] ensure atomicity by disabling kernel preemption and interrupts. RegKey adopts the same method. Before executing register-protected calculations, RegKey calls `preempt_disable` to disable kernel preemption and calls `local_irq_save` for interrupts. When the register-protected calculations are finished, RegKey uses `preempt_enable` and `local_irq_restore` to restore kernel preemption and enable interrupts. Therefore, the duration of atomic operations at each ECDSA signing equals to the time for calculating s inside registers based on (e, $k2$, r, $Enc(k1)$, $Enc(d)$), which is Steps 7–10 in Algorithm 2.

6 EVALUATION

6.1 Performance

We evaluate the performance of RegKey and its impact on concurrent tasks. The elliptic curve we choose is NIST P-256. The machine is Core i7-1065G7 CPU, with Ubuntu 18.04 as the OS.

6.1.1 Performance of RegKey-protected ECDSA Signing. The performance evaluation is carried out from two aspects. Compared with the regular ECDSA implementation in OpenSSL, RegKey introduces some additional operations for protecting the private key. We first present the performance overhead caused by supplementary operations in Table 2 for better comparisons. The execution time of these supplementary functions is negligible compared with OpenSSL ECDSA implementation that costs 0.242 *ms*.

Furthermore, the performance of our protected ECDSA implementation is evaluated. RegKey performs the protected ECDSA signing through `ioctl` system call in user space while OpenSSL directly calls the interfaces of ECDSA signing in user applications. Since Core i7-1065G7 is a

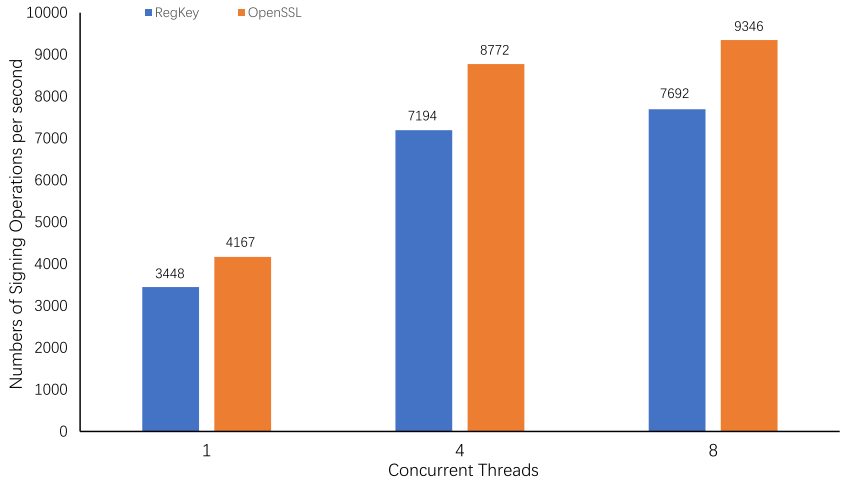

Fig. 2. Performance comparison between RegKey-protected ECDSA and OpenSSL.

4-core 8-thread CPU with Hyper-Threading support, eight threads fill up the CPU. We configure the single thread, four threads, and eight threads respectively to evaluate the performance. Each thread performs abundant signing operations. The number of signing operations per second is given in Figure 2.

Figure 2 indicates that the performance of RegKey is 83% (3448/4167 = 83%), 82% (7194/8772 = 82%), and 82% (7692/9346 = 82%) of regular OpenSSL ECDSA implementation in single thread, four threads and eight threads, respectively. The performance overhead is acceptable, while the private key is effectively protected and is resistant to one-shot memory disclosure attacks. More threads are not necessary because Core i7-1065G7 CPU supports up to eight logical cores. On the contrary, since the scheduling of threads also takes time, more threads will degrade performance.

6.1.2 Impact on Concurrent Tasks. RegKey occupies a part of registers and disables kernel preemption and interrupts, which may affect concurrent tasks. We use SysBench benchmark to evaluate this performance impact. Four scenarios are analyzed by running SysBench CPU test and memory test. The baseline is measured in a clean environment without computing-intensive tasks except SysBench. In the second scenario, we start eight threads to execute RegKey-protected ECDSA signing when SysBench is working. The third scenario is User-mode RegKey that does not disable kernel preemption and interrupts and uses the AES key stored in XMM registers running in user space. The last scenario is OpenSSL ECDSA signing in eight threads. In CPU test, SysBench launches eight threads to perform 10K requests and prime numbers are up to 20K. The score of CPU test is the average time for each request. In memory test, SysBench launches eight threads to perform 1KB each read or write operation for 3GB data.

The result of SysBench test is shown in Table 3. Since there is no significant difference between RegKey, User-mode RegKey, and OpenSSL for both CPU test and memory test, occupying a part of registers and disabling kernel preemption and interrupts in RegKey will not cause obvious negative impact.

6.2 Security Analysis

6.2.1 Resistance to One-shot Memory Disclosure Attacks. As shown in [32], disclosing either the private key d or the random number k undermines the security of ECDSA.

Table 3. CPU and Memory Impact on Concurrent Tasks

	CPU Test (ms)	Memory Test (MiB)
Baseline	1.67	16760.28
RegKey	3.53	7691.25
User-mode RegKey	3.47	7957.38
OpenSSL	3.34	7256.13

To protect the private key d, RegKey stores d and performs prime field operations related to d entirely inside CPU registers without using memory. Therefore, even if adversaries obtain all memory contents through one-shot memory disclosure attacks, d could not be directly found from the dumped memory.

Attacks on k will also expose the private key d in ECDSA [32]. To protect k, RegKey needs to satisfy the following two conditions.

(I) The secret k is not obtained by adversaries. Otherwise, adversaries could recover $d = r^{-1} \times (s \times k - e) \bmod n$ when k is exposed.
(II) Unique secret k is generated for each message signed. Otherwise, adversaries could recover $k = (s_1 - s_2)^{-1} \times (e_1 - e_2) \bmod n$, then recover d with determined k.

In RegKey, the random number is $k = k1 + k2 \bmod n$, and $k2$ could be exposed to adversaries through one-shot memory disclosure. The secret is $k1$ for each ECDSA signing as $k1$ is protected by CPU registers. From the perspective of adversaries, $k1$ is regarded as random and uniformly distributed over $[1, n - k2 - 1] \cup [n - k2 + 1, n - 1]$ ($k1 \neq n - k2$ because $k1 + k2 \neq 0 \bmod n$). As for original ECDSA, k is required as random and uniformly distributed over $[1, n - 1]$. Therefore, $k1$ in RegKey has the same security strength as k in original ECDSA when n is sufficiently large, which satisfies Condition-I.

For different messages signed, adversaries could obtain different message-related tuples $(k2, r, s)$ through one-shot memory disclosure attacks. However, RegKey ensures that each tuple $(k2, r, s)$ acquired by adversaries corresponds to different $k1$, as described in Principle 2 in Section 4.2, which satisfies Condition-II. From the above analysis, RegKey is secure even if adversaries have the ability to launch one-shot memory disclosure attacks.

During RegKey-protected signing, the sensitive data are stored inside CPU registers and we do not explicitly write them into memory and disable task scheduling to prevent them from being passively swapped into memory. We only write the final result (r, s) to memory and carefully clear all the data resident in registers at the end of each signing.

We verify our analysis using a Linux memory extractor, `lime`, which works for volatile memory acquisition from Linux and Linux-based devices. During the continuous operations of RegKey-protected ECDSA signing, the entire memory is dumped and saved as a `mem` file. We search d, $k1$ and k in the `mem` file and find no instance.

6.2.2 Other Attacks. We consider different side channel attacks, including timing side channel attacks and cache side channel attacks. RegKey is carefully implemented to resist timing side channel attacks. RegKey involves two kinds of cryptographic operations, which are AES encryption and ECDSA signing. The AES-NI RegKey adopted is free of timing side channel attacks [23]. As for ECDSA signing, the main weakness of side channel analysis for ECDSA is scalar multiplication. RegKey uses the latest OpenSSL interface `EC_POINT_mul` [45] to calculate scalar multiplication $k2 \times G$, which is a constant time implementation by OpenSSL developers. Therefore, adversaries could not obtain $k2$ through timing side channel attacks.

Table 4. Comparison between RegKey and Other Key Protection Schemes

		Performance	Security		Special CPU Features
			One-shot	Persistent	
ECDSA Signing	RegKey	82% OpenSSL	√		No
	Copker	30% PolarSSL	√	√	CAR
RSA Signing	PRIME	8.6% OpenSSL	√		No
	RegRSA	74% OpenSSL	√	√	No
	VIRSA	102% OpenSSL	√	√	AVX-512F
	Mimosa	85% PolarSSL	√	√	TSX
	CryptoMPK	90% OpenSSL		√	MPK

Cache side channel attacks also pose threats to regular ECDSA implementations [4, 14, 52, 55]. Again, scalar multiplication of RegKey employs the latest OpenSSL, which improves the Montgomery ladder algorithm to protect $k2$ against cache side channel attacks during ECDSA signing. Mitigations based on isolation [19, 34, 38] and randomization [46, 50] have been proposed. These mitigations could prevent adversaries from probing for sensitive activities in cache lines while a service is running.

Finally, DMA attacks could be launched through high-speed peripherals and bypass OS security mechanisms. This persistent attack could obtain cryptographic keys in memory. TRESOR-HUNT [5] is an advanced DMA attack, which accesses AES key in debug registers by injecting malicious codes into OS kernel. Anyway, DMA attacks could be prevented by monitoring bus activities [48] or configuring IOMMU [49].

6.3 Comparison with Other Key Protection Schemes

We compare and analyze current key protection schemes from the perspectives of performance, prevented attacks, and required CPU features. The result is shown in Table 4. One-shot attacks in Table 4 indicate the protection scheme defends against one-shot memory disclosure mainly caused by physically accessing the target machine, and persistent attacks represent the protection scheme defends against persistent software-based memory disclosure.

Among the protections for ECDSA signing, Copker builds on the **CAR (Cache-as-RAM)** character and cache-filling modes, which brings 70% performance loss. Compared with Copker, Regkey brings obvious performance improvements. As for the RSA protection schemes, although Mimosa and CryptoMPK show better performance than register-based RSA implementations (i.e., PRIME and RegRSA), they rely on special CPU hardware features. Besides, CryptoMPK cannot resist physical attacks such as cold-boot attacks (i.e., typical one-shot memory disclosure). VIRSA makes full use of special AVX-512F vector instructions to implement vectorized RSA calculations inside registers, thus it offers better performance than RegRSA. However, most desktop CPUs do not support AVX-512 instructions at present. Some schemes [8, 15, 27, 53] use vector instructions to improve the speed of ECC computations. RegKey is well compatible with these methods, which can further improve the performance of the protected ECC signing. In summary, RegKey demonstrates advantages, because it defends against one-shot memory disclosure and introduces moderate performance overheads without requiring special CPU hardware features.

Furthermore, we compare the number of lines of ASM code between different register-based protection schemes. Since there are no instruction extensions directly related to public key cryptography, developers need to implement complex large integer calculations inside registers manually instead of using existing cryptographic libraries. Compared with RSA, although ECC has shorter key size, the complexity of mathematical calculations of ECC is greatly increased, which

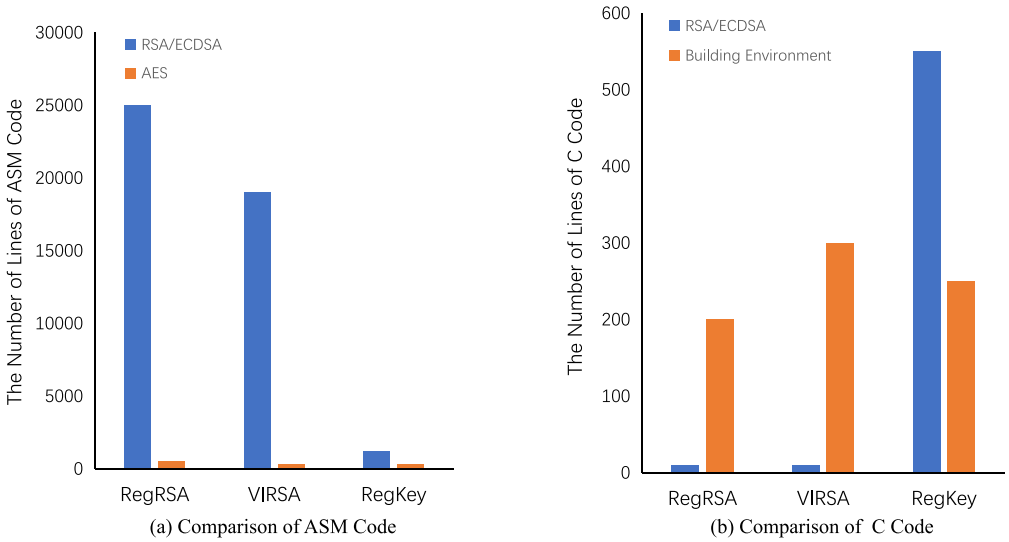

Fig. 3. Comparison of ASM and C Code between RegKey and other register-based protection schemes.

makes it more difficult to implement ECC inside registers using assembly language. In addition, platform-specific assembly language makes the system deployment and code porting very difficult. For this reason, one of the goals of RegKey is to reduce required ASM code.

Figure 3 presents the comparison of ASM and C code of RegKey, RegRSA, and VIRSA. The result illustrates the number of lines of ASM code used for signing in RegKey is significantly decreased, while the number of lines of C code used for signing is increased. The reason is that RegKey does not achieve the whole signing operations inside registers. RegKey transfers complex ECC group operations into memory and only executes simple prime field operations inside registers. In-memory operations of RegKey are finished by calling the interfaces of existing OpenSSL engines.

7 CONCLUSION

In this paper, we propose RegKey, a register-based implementation for ECC signature algorithms, which effectively prevents adversaries from obtaining the private key in memory through one-shot memory disclosure attacks. Furthermore, we build the prototype system of RegKey and evaluate the security and performance. RegKey incurs 18% performance overhead compared with the regular ECDSA implementation of OpenSSL while protecting the private key. Compared with other cryptographic key protection schemes, RegKey does not require special CPU hardware features and shows better performance.

APPENDIX

A CODE OF SUPPLEMENTARY FUNCTIONS

A.1 Conversion from BIGNUM to Unsigned Long Array

```
1 BN_2_Ulong(BIGNUM *a, unsigned long b[]){
2   BIGNUM *c=BN_new();
3   int i;
4   int n=3;//A 256-bit BIGNUM equals four 64-bit elements
5   for(i=0; i<n; ++i){
```

```
6           BN_copy(c,a);
7           BN_mask_bits(c,64);
8           b[i]=BN_get_word(c);
9           BN_rshift(a,a,64);
10      }
11      b[n]=BN_get_word(a);
12      free(c);
13  }
```

Listing 1. Conversion from BIGNUM to Unsigned Long Array

A.2 Conversion from Unsigned Long Array to BIGNUM

```
1  Ulong_2_BN(unsigned long b[], BIGNUM *a){
2      int i;
3      int n=3;//A 256-bit array contains four 64-bit elements
4      BN_set_word(a,b[n]);
5      for(i=n-1; i>=0; --i){
6           BN_lshift(a,a,64);
7           BN_add_word(a,b[i]);
8      }
9  }
```

Listing 2. Conversion from Unsigned Long Array to BIGNUM

A.3 AES Encryption

```
1  .macro      key_shedule     r0      r1      rcon
2      pxor    rhelp,      rhelp
3      movdqu      \r0,    \r1
4      shufps      $0x1f,      \r1,    rhelp
5      pxor    rhelp,      \r1
6      shufps      $0x8c,      \r1,    rhelp
7      pxor    rhelp,      \r1
8      aeskeygenassist     $\rcon,     \r0,    rhelp
9      shufps      $0xff,      rhelp,  rhelp
10     pxor    rhelp,      \r1
11 .endm
12
13 .macro      aes_enc
14     key_shedule     rstate      rk1     0x1
15     key_shedule     rk1     rk2     0x2
16     key_shedule     rk2     rk3     0x4
17     key_shedule     rk3     rk4     0x8
18     key_shedule     rk4     rk5     0x10
19     key_shedule     rk5     rk6     0x20
20     key_shedule     rk6     rk7     0x40
21     key_shedule     rk7     rk8     0x80
22     key_shedule     rk8     rk9     0x1b
23     key_shedule     rk9     rk10    0x36
24     pxor    rstate,     mes
```

```
25    aesenc      rk1 ,      mes
26    aesenc      rk2 ,      mes
27    aesenc      rk3 ,      mes
28    aesenc      rk4 ,      mes
29    aesenc      rk5 ,      mes
30    aesenc      rk6 ,      mes
31    aesenc      rk7 ,      mes
32    aesenc      rk8 ,      mes
33    aesenc      rk9 ,      mes
34    aesenclast      rk10 ,      mes
35  .endm
```

Listing 3. AES Encryption by AES-NI Intrustions

REFERENCES

[1] 2014. *OpenSSL Heartbleed.* (2014). Retrieved October 28, 2022 from https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

[2] 2016. *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves Part 2: Digital Signature Algorithm.* (2016). Retrieved October 28, 2022 from http://www.gmbz.org.cn/main/bzlb.html.

[3] Johannes Bauer, Michael Gruhn, and Felix C. Freiling. 2016. Lest we forget: Cold-boot attacks on scrambled DDR3 memory. *Digital Investigation* 16 (2016), S65–S74. https://doi.org/10.1016/j.diin.2016.01.009

[4] Naomi Benger, Joop Van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems–CHES 2014: 16th International Workshop*. Springer, 75–92. https://doi.org/10.1007/978-3-662-44709-3_5

[5] Erik-Oliver Blass and William Robertson. 2012. TRESOR-HUNT: Attacking CPU-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 71–78. https://doi.org/10.1145/2420950.2420961

[6] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 193–204. https://doi.org/10.1145/3052973.3052983

[7] Ellick M. Chan, Jeffrey C. Carlyle, Francis M. David, Reza Farivar, and Roy H. Campbell. 2008. BootJacker: Compromising computers using forced restarts. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. 555–564. https://doi.org/10.1145/1455770.1455840

[8] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. 2020. High-throughput elliptic curve cryptography using AVX2 vector instructions. In *International Conference on Selected Areas in Cryptography*. Springer, 698–719. https://doi.org/10.1007/978-3-030-81652-0_27

[9] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*. 321–336.

[10] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal De Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. 177–189. https://doi.org/10.1145/2694344.2694380

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016), 1–118.

[12] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. 2012. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Operating Systems Design and Implementation*. 61–75.

[13] J. Elbahrawy, J. Lovejoy, A. Ouyang, and J. Perez. 2020. Analysis of Bitcoin improvement proposal 340-Schnorr signatures. (2020).

[14] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. 2016. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1505–1515. https://doi.org/10.1145/2976749.2978400

[15] Armando Faz-Hernández, Julio López, and Ricardo Dahab. 2019. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Transactions on Mathematical Software (TOMS)* 45, 3 (2019), 1–35. https://doi.org/10.1145/3309759

[16] Yu Fu, Wei Wang, Lingjia Meng, Qiongxiao Wang, Yuan Zhao, and Jingqiang Lin. 2021. VIRSA: Vectorized in-register RSA computation with memory disclosure resistance. In *International Conference on Information and Communications Security*, Vol. 12918. Springer, 293–309. https://doi.org/10.1007/978-3-030-86890-1_17

[17] Steven D. Galbraith and Pierrick Gaudry. 2016. Recent progress on the elliptic curve discrete logarithm problem. *Designs, Codes and Cryptography* 78, 1 (2016), 51–72. https://doi.org/10.1007/s10623-015-0146-7
[18] Behrad Garmany and Tilo Müller. 2013. PRIME: Private RSA infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*. 149–158. https://doi.org/10.1145/2523649.2523656
[19] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*. 217–233.
[20] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. 2014. Copker: Computing with private keys without RAM. In *NDSS*. 23–26. https://doi.org/10.14722/ndss.2014.23125
[21] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 3–19. https://doi.org/10.1109/SP.2015.8
[22] Le Guan, Jingqiang Lin, Ziqiang Ma, Bo Luo, Luning Xia, and Jiwu Jing. 2016. Copker: A cryptographic engine against cold-boot attacks. *IEEE Transactions on Dependable and Secure Computing* 15, 5 (2016), 742–754. https://doi.org/10.1109/TDSC.2016.2631548
[23] Shay Gueron. 2010. Intel advanced encryption standard (AES) new instructions set. (2010).
[24] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98. https://doi.org/10.1145/1506409.1506429
[25] Keith Harrison and Shouhuai Xu. 2007. Protecting cryptographic keys from memory disclosure attacks. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 137–143. https://doi.org/10.1109/DSN.2007.77
[26] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring operating system kernel integrity with OSck. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 279–290. https://doi.org/10.1145/1961295.1950398
[27] Junhao Huang, Zhe Liu, Zhi Hu, and Johann Großschädl. 2020. Parallel implementation of SM2 elliptic curve cryptography on Intel processors with AVX2. In *Australasian Conference on Information Security and Privacy*. Springer, 204–224. https://doi.org/10.1007/978-3-030-55304-3_11
[28] Intel. 2022. Intel 64 and ia-32 architectures software developer's manual volume 2 (2a, 2b, 2c & 2d): Instruction set reference, a-z. (2022).
[29] Intel. 2017. SGX SSL. (2017). Retrieved October 28, 2022 from https://github.com/intel/intel-sgx-ssl.
[30] Fangjie Jiang, Quanwei Cai, Jingqiang Lin, Bo Luo, Le Guan, and Ziqiang Ma. 2019. TF-BIV: Transparent and fine-grained binary integrity verification in the cloud. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 57–69. https://doi.org/10.1145/3359789.3359795
[31] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. 2022. Annotating, tracking, and protecting cryptographic secrets with CryptoMPK. In *IEEE Symposium on Security and Privacy*.
[32] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security* 1, 1 (2001), 36–63. https://doi.org/10.1007/s102070100002
[33] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight kernel protection against return-to-user attacks. In *USENIX Security Symposium*, Vol. 16.
[34] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security Symposium*. 189–204.
[35] Neal Koblitz. 1987. Elliptic curve cryptosystems. *Math. Comp.* 48, 177 (1987), 203–209.
[36] Congwu Li, Le Guan, Jingqiang Lin, Bo Luo, Quanwei Cai, Jiwu Jing, and Jing Wang. 2019. Mimosa: Protecting private keys against memory disclosure attacks using hardware transactional memory. *IEEE Transactions on Dependable and Secure Computing* 18, 3 (2019), 1196–1213. https://doi.org/10.1109/TDSC.2019.2897666
[37] Simon Lindenlauf, Hans Höfken, and Marko Schuba. 2015. Cold boot attacks on DDR2 and DDR3 SDRAM. In *2015 10th International Conference on Availability, Reliability and Security*. IEEE, 287–292. https://doi.org/10.1109/ARES.2015.28
[38] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 406–418. https://doi.org/10.1109/HPCA.2016.7446082
[39] Chris Lomont. 2011. Introduction to Intel advanced vector extensions. *Intel White Paper* 23 (2011), 1–21.
[40] Victor S. Miller. 1985. Use of elliptic curves in cryptography. In *Conference on the Theory and Application of Cryptographic Techniques*, Vol. 218. Springer, 417–426. https://doi.org/10.1007/3-540-39799-X_31
[41] Tilo Müller, Andreas Dewald, and Felix C. Freiling. 2010. AESSE: A cold-boot resistant implementation of AES. In *Proceedings of the Third European Workshop on System Security*. 42–47. https://doi.org/10.1145/1752046.1752053

[42] Tilo Müller, Felix C. Freiling, and Andreas Dewald. 2011. Tresor runs encryption securely outside RAM. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, 1–16. https://www.usenix.org/conference/usenix-security-11/tresor-runs-encryption-securely-outside-ram.

[43] T. Paul Parker and Shouhuai Xu. 2009. A method for safekeeping cryptographic keys from memory disclosure attacks. In *International Conference on Trusted Systems*. Springer, 39–59. https://doi.org/10.1007/978-3-642-14597-1_3

[44] Torbjörn Pettersson. 2007. Cryptographic key recovery from Linux memory dumps. *Chaos Communication Camp* (2007), 1–14.

[45] OpenSSL Project. 2021. OpenSSL-1.1.1k. (2021). Retrieved October 28, 2022 from https://www.openssl.org.

[46] Moinuddin K. Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787. https://doi.org/10.1109/MICRO.2018.00068

[47] Patrick Simmons. 2011. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 73–82. https://doi.org/10.1145/2076732.2076743

[48] Patrick Stewin. 2013. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In *Research in Attacks, Intrusions, and Defenses: 16th International Symposium (RAID 2013)*. Springer, 1–20. https://doi.org/10.1007/978-3-642-41284-4_1

[49] Patrick Stewin and Iurii Bystrov. 2013. Understanding DMA malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference*. Springer, 21–41. https://doi.org/10.1007/978-3-642-37300-8_2

[50] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating cache conflicts with localized randomization. In *NDSS*. https://dx.doi.org/10.14722/ndss.2020.24086

[51] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[52] Wenbo Wang and Shuqin Fan. 2018. Attacking OpenSSL ECDSA with a small amount of side-channel information. *Science China Information Sciences* 61 (2018), 1–14. https://doi.org/10.1007/s11432-016-9030-0

[53] Wenjie Wang, Wei Wang, Jingqiang Lin, Yu Fu, Lingjia Meng, and Qiongxiao Wang. 2021. SMCOS: Fast and parallel modular multiplication on ARM NEON architecture for ECC. In *International Conference on Information Security and Cryptology*. Springer, 531–550. https://doi.org/10.1007/978-3-030-88323-2_28

[54] Yang Yang, Zhi Guan, Zhe Liu, and Zhong Chen. 2014. Protecting elliptic curve cryptography against memory disclosure attacks. In *International Conference on Information and Communications Security*, Vol. 8958. Springer, 49–60. https://doi.org/10.1007/978-3-319-21966-0_4

[55] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Paper 2014/140. (2014). https://eprint.iacr.org/2014/140.

[56] Yuan Zhao, Jingqiang Lin, Wuqiong Pan, Cong Xue, Fangyu Zheng, and Ziqiang Ma. 2016. RegRSA: Using registers as buffers to resist memory disclosure attacks. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, Vol. 471. Springer, 293–307. https://doi.org/10.1007/978-3-319-33630-5_20

Received 23 December 2022; revised 9 April 2023; accepted 28 May 2023