RegRSA: Using Registers as Buffers to Resist Memory Disclosure Attacks

Yuan Zhao[1,2,3], Jingqiang Lin[1,2], Wuqiong Pan[1,2(✉)], Cong Xue[1,2,3], Fangyu Zheng[1,2,3], and Ziqiang Ma[1,2,3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{yzhao,linjq,wqpan,cxue13,fyzheng,zqma13}@is.ac.cn
[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Beijing, China

Abstract. Memory disclosure attacks, such as cold-boot attacks and DMA attacks, allow attackers to access all memory contents, therefore introduce great threats to plaintext sensitive data in memory. Register-based and cache-based schemes have been used to implement RSA securely, at the expense of decreased performance. In this paper, we propose another concept named *register buffer*, which makes use of all available registers as secure data buffer, no matter scalar registers or vector registers. The plaintext sensitive data only appear in register buffer. Based on this concept, we finish a security implementation of 2048-bit RSA called *RegRSA*, to defeat against memory disclosure attacks. The 1024-bit Montgomery multiplication in RegRSA runs entirely in register buffer, by performing computations using scalar instructions and registers, maintaining intermediate variables in vector registers. Due to the size limitation of register buffer, several variables out of Montgomery multiplications are spilled into memory. RegRSA encrypts these variables with AES before saving in memory. Furthermore, RegRSA employs a windowing method and the CRT speed-up to accelerate RSA, and minimizes the data exchange between registers and memory to reduce the workload of AES encryption/decryption. The evaluation on Intel Haswell i7-4770R shows that, the performance of RegRSA achieves a factor of 0.74 compared to the regular RSA implementation in OpenSSL and is much greater than PRIME, the existing register-based scheme for 2048-bit RSA. Moreover, RegRSA allows multiple instances to run on a multi-core CPU simultaneously, which makes it more practical for the real-world applications.

Keywords: Memory disclosure attack · Register · RSA · Montgomery multiplication

Y.Zhao—This work was partially supported by National 973 Program under award No. 2014CB340603 and No. 2013CB338001, and Strategy Pilot Project of Chinese Academy of Sciences under award No. XDA06010702.

J.-H. Hoepman and S. Katzenbeisser (Eds.): SEC 2016, IFIP AICT 471, pp. 293–307, 2016.
DOI: 10.1007/978-3-319-33630-5_20

1 Introduction

RSA [26] is the most prevalent asymmetric cryptographic algorithm. Although this algorithm is considered computationally secure, the RSA implementations face various security threats. Memory disclosure attacks, such as cold-boot attacks [15] and DMA attacks [28], allow attackers to obtain all memory contents, which makes plaintext private keys in memory unsafe. Besides, the sensitive data used or produced during RSA private-key operations, which can be used to derive the private key, should not appear in memory in plaintext. These sensitive data shall be stored in some secure storage when they participate in the private-key computations.

Registers and L1 caches in CPUs dedicated to one core, become the required secure storage because they are exclusive to the thread currently running on this CPU core under certain controls [10,11]. L1 caches are much larger than registers and programmes can be coded in high-level languages. But malicious binaries running on one CPU core could exploit the last-level cache (LLC) to flush a L1 cache line of other cores to memory with the hardware cache coherence mechanism. The existing cache-based scheme [11] forces all other cores that share LLC, into the no-fill cache mode during the cryptographic computations, and sharply reduces the memory access performance of these cores. Moreover, it does not support multiple instances on the cores that shares LLC. The advantage of register-based scheme is that registers are unaffected by other cores, which builds the possibility of executing multiple instances on a multi-core CPU simultaneously. However, the challenge is that registers might be not enough for the asymmetric cryptographic implementation, which requires (sensitive) data to be swapped between registers and memory frequently (results in frequent symmetric encryptions before being written into memory and decryptions after being loaded into registers). So the key point of register-based schemes is to implement the most frequent function entirely in registers as far as possible.

In order to implement an efficient register-based RSA system against memory disclosure attacks, we should choose an implementation method of RSA which has advantages both on speed and storage consuming. Redundant representation [14] is the major method for vector-instruction implementations, which achieves high speed but demands much more storage space. PRIME [10] adopts redundant representation method. In order to finish register-based RSA implementation, PRIME has to abandon CRT method and its performance is greatly degraded. Another vector approach [6] adopts 2-way single instructions to implement Montgomery multiplication, which consumes less storage than redundant representation method. But the authors in [6] point out that its speed is lower than 64-bit scalar implementation. So the register-based scheme adopting 64-bit scalar instructions is our mainly concerned.

1.1 Contributions

In this paper, we propose an 2048-bit RSA implementation named *RegRSA*, resistant to memory disclosure attacks. Our basic idea is to keep all plaintext

sensitive data only in registers when being used in RSA private-key computations. The performance of RegRSA is close to the implementation in OpenSSL and it allows multiple instances to run on a multi-core CPU simultaneously. Our major contributions are described as follows:

- We propose a concept named *register buffer* that makes use of all available registers as secure data buffer. We also summarize the available registers in newer CPUs and the instructions used to move data between different kinds of registers.
- We make full use of 704-byte register buffer in Intel Haswell CPU to implement an 1024-bit Montgomery multiplication running entirely in register buffer, by performing computations using scalar instructions and registers, maintaining intermediate variables in vector registers.
- We use a fixed windowing method to speed up Montgomery exponentiation, and finish a CRT-enabled 2048-bit RSA implementation. The precomputed table and intermediate variables are encrypted and stored in OS kernel heap or stack. We present several improvements on using AES-NI [13] to reduce the cost of AES key expansion for each data block.

1.2 Related Work

Cold-boot attacks and DMA attacks have been explored to access sensitive data in memory. The first attack exploiting the remanence effect of RAM was reported in [1]. In 2008, the work in [15] presented an cold-boot attack which recovered cryptographic keys by freezing the RAM chips. The study in [28] provided an overview of cold-boot attacks and the proposed counter-measures. DMA attacks are launched from peripherals through high-speed peripheral ports like PCI [8] and Firewire [4]. TRESOR-HUNT [3] presented an advanced DMA attack to get the AES key in privileged registers by compromising the integrity of OS kernel.

In order to resist memory disclosure attacks, register-based and cache-based schemes are proposed. The register-based schemes employ registers as the secure storage, such as AESSE [23], Amnesia [27] and TRESOR [24] which keep AES keys in registers and computed AES using registers only. PRIME [10] is the first register-based scheme for 2048-bit RSA private-key operations, but its performance is greatly degraded. The study in [29] proposed an elliptic curve cryptography implementation using CPU registers. On the other hand, the cache-based schemes employ caches to store sensitive data. FrozenCache [25] exploited CPU caches to store keys outside RAM. Copker [11] proposed a method to perform RSA private-key operations within CPU caches. Existing RSA implementations against memory disclosure attacks including PRIME and Copker, cannot support simultaneous multiple instances on multi-core CPUs well. Mimosa [12] is the first work to protect sensitive data using hardware transactional memory, which essentially stores data in caches; but it requires special CPU hardware features.

1.3 Outline

The rest of this paper is organized as follows. Section 2 introduces the available registers in CPU. Sections 3 and 4 describe the design and the implementation of RegRSA. In Sect. 5, we evaluate RegRSA in terms of security and performance. We conclude this paper in Sect. 6.

2 Available Registers in Commodity CPUs

Registers are classified into user-accessible registers and special internal registers. User-accessible registers can be read or written by CPU instructions, while internal registers cannot be accessible by instructions. On Intel CPUs, the most commonly-used x86 platform, instructions include scalar instructions and vector instructions. Firstly, scalar integer instructions operate on scalar registers, also called general purpose registers (GPRs); and scalar floating-point instructions operate on floating-point registers. Secondly, Intel vector instruction sets include MMX [9], SSE [18] and AVX [21]. The registers for MMX are called MM registers, which are physically the same registers with floating-point registers. The registers for SSE are called XMM registers, and the registers for AVX are the extensions of XMM registers called YMM registers. XMM registers are the low 128-bit of YMM registers. A 64-bit CPU has more registers and every scalar register is double size. Users on 64-bit OS can manipulate these greater-size registers. For example, on an Intel Haswell CPU, there are sixteen 64-bit GPRs, eight 64-bit MM registers and sixteen 256-bit YMM registers. The total space of these registers is 704-byte. Besides, there are four 64-bit debug registers (DRs) available if the operating system prohibits debugged applications access these registers [24].

Table 1. Instructions used to move data between different kinds of registers

	GPR	MM	XMM	YMM	DR
GPR	MOV	MOV	VMOV/PINR/VPINR	-	MOV
MM	MOV	MOV	MOVQ2DQ	-	-
XMM	VMOV/PEXTR/VPEXTR	MOVDQ2Q	MOVDQA/VMOVDQA	VINSERTI128	-
YMM	-	-	VEXTRACTI128	VMOVDQA	-
DR	MOV	-	-	-	-

Scalar registers, MM registers and YMM registers can be used as secure storage for sensitive computations. The data in different kinds of registers may be exchanged frequently. Table 1 summarizes the instructions used to move data between different kinds of registers. Note that, scalar registers and XMM registers can exchange data with most other registers, so they could be the hub of data exchange or keep the most commonly used data. Specifically, instructions PINR and VPINR insert a 64-bit data item from a scalar register to the particular location in a XMM register. The difference between PINR and VPINR is

that, PINR is a legacy instruction that keeps the high 128-bit of the destination YMM register unchanged, but with performance penalty. VPINR is an new AVX instruction which will clear the high 128-bit of destination YMM register with no performance penalty.

3 System Design

In this section, we present the design goals of RegRSA. Then, we propose the concept of register buffer and describe the architecture of RegRSA on top of register buffer.

3.1 Design Goals

The target of this work is to design a *secure* and *efficient* 2048-bit RSA implementation as follows.

Security Goal. All the sensitive data including cryptographic keys (AES keys and RSA private keys) and intermediate variables does not appear in the memory in the form of plaintext, against cold-boot attacks [15] and other memory disclosure attacks.

Performance Goal. The speed of RegRSA should be close to regular implementations, e.g., OpenSSL. Multiple optimization techniques are expected to be exploited in RegRSA, such as Montgomery multiplication [22], the windowing method [19] for modular exponentiation, and the CRT speed-up [19].

Assumptions. First, the OS kernel is trustworthy, which means an attacker can not tamper the OS kernel to launch attacks such as TRESOR-HUNT [3]. Second, the system initialization before any user-space process is safe for users to input a password to derive an AES key in privileged debug registers [24]. Finally, the register features are available in hardware and software platform, that is, CPUs and the OS are 64-bit, and necessary instruction extensions including AVX, AES-NI and MULX are ready.

3.2 Register Buffer

Existing register-based schemes [10,24,29] have investigated the usage of registers for storing sensitive data, but only focused on some registers (not all available registers). The high-speed implementations of cryptographic algorithms have explored the cooperation of different kinds of instructions and registers to accelerate cryptograph computing, but such implementations do not systematically consider the security of keys.

In this paper, we propose a concept named register buffer, which makes use of all available registers as secure data buffer, no matter scalar registers or vector registers. As the registers are used to provide operands and accept results for certain instructions, register buffer requires the comprehensive cooperation of different kinds of instructions and registers for efficient computing and secure storage. As described in Fig. 1, register buffer is divided into two sets of registers:

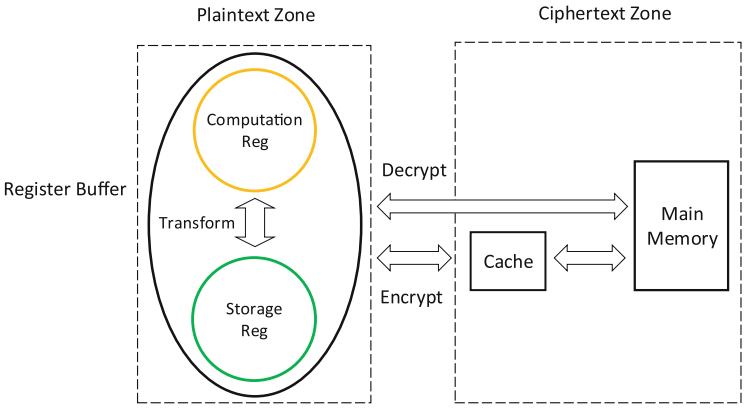

Fig. 1. Register buffer

computation-reg and *storage-reg*. Computation-reg is a set of registers which provides inputs and receives results for on-going instructions. Storage-reg is a set of registers for maintaining data unused in the current period. The registers are ready to be converted between computation-reg and storage-reg depending on which kind of instructions are being executed. Data are plaintext in register buffer. When data have to be stored in memory (RAM or caches), they are encrypted with AES. In brief, register buffer deems that all available registers are secure storage resources which should be fully utilized for efficiency and security.

3.3 RegRSA Architecture

From an implementation point of view, RegRSA is divided into three levels: (1) the modular multiplication level, (2) the modular exponentiation level and (3) the RSA level. The high level calls the lower level by sending parameters and receiving results. The architecture and the data transfer between registers and main memory are depicted in Fig. 2.

In the modular multiplication level, we design and implement an all-register 1024-bit Montgomery multiplication which computes Montgomery multiplication by using scalar instructions and registers, maintaining parameters in vector registers. In the modular exponentiation level, we apply the windowing method for 1024-bit Montgomery exponentiation. We compute, encrypt and save precomputed table in the kernel heap, and then load the precomputed values depending on the exponents and decrypt the values. Based on the CRT method, we implement 2048-bit RSA by performing two 1024-bit Montgomery exponentiations. The encrypted Montgomery exponents are loaded from memory and the results of Montgomery exponentiations are encrypted and swapped between registers and the kernel stack. In all levels, all the sensitive data in RAM are encrypted with AES and the AES key is in debug registers. Note that, the major computations,

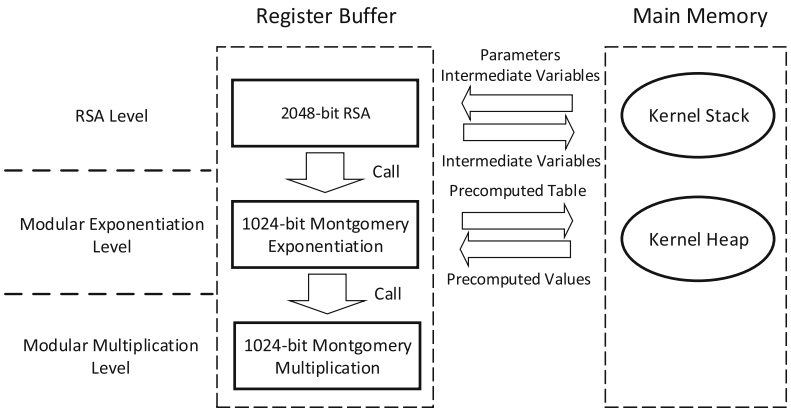

Fig. 2. RegRSA architecture

Montgomery multiplication, do not need to exchange data between register and memory, so the performance degradation from data encryption/decryption and exchange is not significant.

4 Implementation

In this section, we describe the detailed implementation of RegRSA: in particular, the assembly codes of RegRSA to gain the complete control of registers, and the integration of RegRSA into Linux kernel because such implementation must run in kernel mode otherwise task switching may dump registers into RAM.

4.1 Montgomery Multiplication Implementation

1024-bit Montgomery multiplication [22] performs the computation $S = A \times B \times R^{-1} \pmod{M}$, $R = 2^{1024}$, $0 \leqslant A, B < M < R$. Coarsely Integrated Operand Scanning (CIOS) [20] is an interleaved Montgomery multiplication method with three 1024-bit inputs A, B, M and one 64-bit input μ $(-M^{-1} \bmod 2^{64})$. As depicted in Fig. 3, we employ scalar registers and scalar instructions to perform Montgomery multiplication, while keeping three 1024-bit inputs in YMM registers, 64-bit input in a scalar register and saving the 64-bit intermediate variables q_j $(q_j = S_j \times \mu \bmod 2^{64})$ in MM registers. We make full use of 704-byte register buffer to finish the first all-register 1024-bit Montgomery multiplication implementation.

According to 64-bit Linux call convention, registers RBX, RBP, RSP, R12, R13, R14, R15 must be protected, we push these registers except RSP into stack. Since the left fifteen 64-bit scalar registers are not enough for computing the whole 1024-bit Montgomery multiplication in one time, we split Montgomery multiplication into four parts. The first part performs $A[0 \sim 7] \times B[0 \sim 7] + M[0 \sim 7] \times (q_0 \sim q_7)$, the second part performs $A[8 \sim 15] \times B[0 \sim 7] + M[8 \sim 15] \times$

$(q_0 \sim q_7)$, the third part performs $A[0 \sim 7] \times B[8 \sim 15] + M[0 \sim 7] \times (q_8 \sim q_{15})$ and the fourth part is $A[8 \sim 15] \times B[8 \sim 15] + M[8 \sim 15] \times (q_8 \sim q_{15})$. The j-th round computation of the first part is depicted in Fig. 3. Instruction MULX [18] is used to perform 64-bit scalar multiplication. The summation variable S occupies nine scalar registers and is updated in every round, and q_j in MM registers is moved back when needed. Eight MM registers are enough for storing $q_0 \sim q_7$ or $q_8 \sim q_{15}$ for eight rounds. Besides, the final subtraction in Montgomery multiplication is always performed, whether or not S is no less than M, to eliminate the timing side-channel [7].

4.2 Montgomery Exponentiation Implementation

We use the fixed windowing method [19] to speed up Montgomery exponentiation. The size of window is 6-bit (64 entries). For CRT-enabled RSA, two different moduli are needed; i.e., for the private-key parameters p and q, two tables of ${C_p}^k$ and ${C_q}^k$ $(k = 0, \cdots, 63)$ which share a memory space. At the beginning of Montgomery exponentiation, we prepare precomputed table: compute and encrypt ${C_p}^2 \sim {C_p}^{63}$ (or ${C_q}^2 \sim {C_q}^{63}$), save them into kernel memory. ${C_p}^0$, ${C_p}^1$ (or ${C_q}^0$, ${C_q}^1$) are also encrypted and saved in the precomputed table for the const time of table lookup. Then we get the precomputed value from the precomputed table to YMM registers depending on the exponent, and decrypt them using 128-bit AES. Since the maximum size of kernel stack is 8KB which still has to save struct `thread_info` at the stack bottom, 6-bit precomputed table which needs 8 KB memory cannot be saved in kernel stack. So we use system function `kmalloc` to allocate 8KB memory on the kernel heap for precomputed table before beginning RegRSA, and use function `kfree` to free memory after

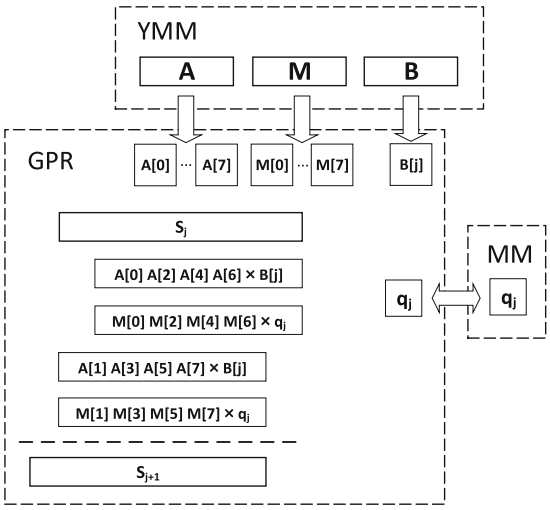

Fig. 3. First part of our 1024-bit Montgomery multiplication implementation

finishing RegRSA. We load all entries of the precomputed table in sequence when we need certain precomputed values.

AES-NI instruction extension [13] is used to implement AES encryption/decryption and key expansion by hardware. In this study, we present several improvements on using AES-NI. First, AES-128 and AES-256 only need one 128-bit temporary register in the key expansion. Second, we derive the round keys from the last round to the first round which is very useful for AES decryption. Third, we perform on-the-fly bulk AES encryption/decryption which uses one round key to process multiple 128-bit data blocks and then the next round key, which sharply reduces the cost of key expansions for each data block.

4.3 RSA Implementation

We utilize the CRT method and our 1024-bit Montgomery exponentiation implementation to finish 2048-bit RSA private-key operations. The input parameters are copied from the user space memory to the kernel space memory, including the CRT parameters p, C_p, d_p, q, C_q, d_q, $q^{-1} \bmod p$, and Montgomery parameters $R^2 \bmod p$, $R^2 \bmod q$, $-p^{-1} \bmod 2^r$, $-q^{-1} \bmod 2^r$. The parameters d_p, d_q, $q^{-1} \bmod p$, $R^2 \bmod p$, $R^2 \bmod q$, $-p^{-1} \bmod 2^r$, $-q^{-1} \bmod 2^r$ are constant for each private key, while $C_p = C \bmod p$ and $C_q = C \bmod q$ which need to be computed on the fly for each ciphertext C. All the above parameters are encrypted with AES when they are stored in memory. As described in Algorithm 1, the CRT speed-up requires two 1024-bit Montgomery exponentiations for 2048-bit RSA. As register buffer is not enough to hold the result of one Montgomery exponentiation while performing another, we keep the intermediate variables encrypted in memory.

Algorithm 1. Implementation of 2048-bit RSA Private-key Operation

Input: The CRT parameters p, C_p, d_p, q, C_q, d_q, $q^{-1} \bmod p$, and Montgomery parameters $R^2 \bmod p$, $R^2 \bmod q$, $-p^{-1} \bmod 2^r$, $-q^{-1} \bmod 2^r$

Output: Plaintext M.

1: Load parameters p, C_p, d_p, $R^2 \bmod p$, $-p^{-1} \bmod 2^r$ from RAM to registers and decrypt them with AES
2: $M_p \leftarrow {C_p}^{d_p} \mod p$
3: Encrypt M_p with AES and save it in RAM
4: Load parameters q, C_q, d_q, $R^2 \bmod q$, $-q^{-1} \bmod 2^r$ from RAM to registers and decrypt them with AES
5: $M_q \leftarrow {C_q}^{d_q} \mod q$
6: Encrypt M_q with AES and save it in RAM
7: Load M_p, M_q, q, $q^{-1} \bmod p$, $R^2 \bmod p$, $-p^{-1} \bmod 2^r$ from RAM to registers and decrypt them with AES
8: $M \leftarrow M_q + [(M_p - M_q) \times (q^{-1} \bmod p) \mod p] \times q$
9: **return** M

4.4 Integration in Linux Kernel

We integrate RegRSA into Linux kernel to ensure no data in registers would leak into main memory.

Char module. We integrate RegRSA into a char module and compile this module into Linux kernel. The module provides an interface for user space with the system call ioctl. The user processes can use the interface to send the inputs to RegRSA and receive the results. In kernel space, RegRSA can access privileged debug registers for AES keys (and all other registers).

Atomicity. Before the execution of RegRSA, kernel preemption is suspended by calling `preempt_disable` and interrupts are disabled by calling `local_irq_save`. So data in registers will not be written into main memory by context switch during the RegRSA computations. Finally, kernel preemption is restored by calling `preempt_enable` and interrupts are enabled by calling `local_irq_restore`. As non-maskable interrupts (NMIs) cannot be disabled by software settings, we adopt the solution in Copker [11]. We modify NMI handlers to clear registers with sensitive data, including scalar registers, MM registers and YMM registers.

4.5 AES Key

AES key is securely produced and maintains safety after OS initialization.

AES key derivation. By utilizing an existing technique in TRESOR [24], we have patched the linux kernel to let user input a password before any userland process startup. We assume the password is strong enough to defeat against brute-force attacks. Moreover, the user can update AES keys by changing the password after a while.

AES key protection. Also like in TRESOR [24], AES key is stored in debug registers which cannot be accessed from user space. System functions `ptrace_set_debugreg` and `trace_get_debugreg` are patched to ensure the user process cannot set four debug registers dr0 to dr3 or get their values.

5 Evaluation

In this section, we conduct security analysis on RegRSA, evaluate its performance and the impact of RegRSA. In the end, we discuss some further considerations. The system is evaluated on this platform: Intel Haswell i7-4770R CPU, 8GB memory and OS is Ubuntu 14.04 64-bit. We turn off Turbo Boost of Intel Haswell i7-4770R for stable frequency 3.2GHz, in the performance experiments.

5.1 Security Analysis

Memory Disclosure Attacks. First, we explore the security for the situation of only one RegRSA running instance on CPU. The input parameters

are all sensitive data, including CRT parameters and Montgomery parameters, which are encrypted with AES before passing from user space to kernel space. AES key is in privileged debug registers, no user process could read or write these registers. Due to the atomic execution of RegRSA, no data in registers will leak into main memory by task switch. In the Montgomery multiplication level, all the intermediate variables are produced and kept in registers. In the Montgomery exponentiation level, the precomputed table is encrypted before stored in RAM, and the precomputed values are decrypted in XMM registers. In the RSA level, the results of Montgomery exponentiation are encrypted before stored in memory and decrypted after reading into registers. All the data in registers are eliminated before leaving the atomic region. In a word, all the sensitive data in main memory are encrypted, and the plaintext sensitive data appear in registers only.

Also, we have done experimental evaluation on RegRSA. We used Kdump to dump kernel memory while RegRSA was performing RSA private-key operations. We searched AES keys and RSA private keys in the captured image and found no matching strings.

When there are multiple instances of RegRSA on several CPU cores, no matter how many requests received from user space, only one RegRSA instance is running on a single CPU core in a moment due to the atomic execution. As each RegRSA instance owns its own variables in kernel stack and heap, RegRSA can execute multiple instances on different CPU cores which will not interfere with each other.

Cache-based Timing Side-Channels. RegRSA is resistant to cache-based timing side channel attacks [2,5,7]. We employ the fixed windowing method in Montgomery exponentiation and the final subtraction in Montgomery multiplication is always performed, so there is no branch in the execution flow. Thus, there is no timing side channels in RegRSA based on instruction paths. When we perform the table-lookup in Montgomery exponentiation, we load the precomputed table as a whole. This makes attackers could not learn which entry RegRSA accesses and deduce the exponents. Therefore, there is no timing side channels attacks based on data access.

5.2 Performance

Comparison with OpenSSL. We launch different numbers of threads in user space to send RSA private-key operation requests to RegRSA. Because of simultaneous multithreading (SMT), eight threads make RegRSA occupy the CPU fully. RegRSA is compared with OpenSSL version 1.0.1f, in different concurrent levels. The numbers of RSA private-key operations per second and the ratios of the performances between RegRSA and OpenSSL are given in Table 2.

As the number of threads increases, the performance becomes better, either for RegRSA or OpenSSL. RegRSA achieves at least a factor of 0.74 compared to the speed of OpenSSL. The efficiency degradation of 26 % is acceptable, to defeat against memory disclosure attacks.

Table 2. Comparison with OpenSSL

# of Concurrent Threads	1	4	8
RegRSA	637	2537	2638
OpenSSL	858	3308	3571
RegRSA / OpenSSL	0.74	0.77	0.74

Comparison with PRIME. We expect to compare RegRSA with PRIME [10] on the same platform. Because we do not have the source code of PRIME, the comparison with PRIME is conducted through OpenSSL – we assume the OpenSSL in [10] is identical with that in this paper (version 1.0.1f). Table 3 presents the speed ratio of PRIME and RegRSA to OpenSSL, respectively. So we can see that the efficiency of RegRSA is far beyond PRIME; moreover, RegRSA can execute 8 instances on quad-core CPUs simultaneously.

5.3 Impact on Concurrent Tasks

As RegRSA disables kernel preemption and interrupts during its running, the stable of the operating system and the performance of other applications may be effected. We initiate eight user threads to continuously send RSA private-key operation requests. Through a period of observation, we see that the operating system works properly without disruptions, and the response of OS is normal as well. Then we use the SysBench benchmark to evaluate the performance impact on concurrent tasks. We run the CPU test of SysBench and execute the test in four situations. The first is no computing-intensive tasks performing during the test. The second is same as OS stable test with running eight user threads to send requests to RegRSA. The third is to start eight threads to perform a "mock" RegRSA in user space. This mock RegRSA does not disable kernel preemption and interrupts, and use a fixed value as the AES key – other configurations are the same as those of RegRSA. The fourth is to perform OpenSSL speed test for 2048-bit RSA private-key operations in eight threads. Test parameters of SysBench are 8 threads, 10,000 requests and prime numbers up to 20000. The test score is the average time for each request.

The results are presented in Table 4. There is no significant difference between the situations of RegRSA, mock RegRSA and OpenSSL. So disabling kernel preemption and interrupts in RegRSA does not cause obvious negative effects.

Table 3. Comparison with PRIME

	Latency (ms)	Speed Ratio
PRIME	21.0	-
OpenSSL [10]	1.8	-
PRIME / OpenSSL [10]	-	0.086
RegRSA / OpenSSL 1.0.1f	-	0.74

Table 4. Impact on other applications

	Idle	RegRSA kernel space	Mock RegRSA user space	OpenSSL
SysBench (ms)	2.83	5.94	5.87	5.98

5.4 Discussions

SMT. SMT on Intel CPUs also known as Hyper-Threading (HT), which is used to improve the efficiency of processing units in CPUs. HT provides two hardware threads on each core. Each thread has a separate set of registers that can be used for RegRSA. So RegRSA can run at most eight instances on a quad-core HT CPU simultaneously. Two threads on one CPU core will facilitate instruction pipelining which improves performance of RegRSA slightly; see Table 2 for details.

Full Memory Encryption. The aim of full memory encryption is to provide confidentiality of the entire software stack outside the CPU [17]; therefore, memory disclosure attacks are defeated. However, existing memory encryption suffer significant performance degradation [16].

Other CPUs. If a CPU possesses multiple registers including scalar registers, MM registers and YMM registers, and supports AES-NI instruction set and MULX instruction, it may be a suitable platform for RegRSA. So besides Intel CPUs, other CPUs like AMD can also be considered. For example, AMD Carrizo CPUs also support AVX2, MULX and AES-NI, it can be a potential candidate.

6 Conclusion

We propose a concept named register buffer that makes use of all available registers as secure data buffer, and design and implement 2048-bit RSA named RegRSA. In RegRSA, all the sensitive data appeared in main memory are encrypted, and the plaintext data are protected in registers to defeat against memory disclosure attacks. The evaluation on Intel Haswell i7-4770R showed that, the performance of RegRSA achieves at least a factor of 0.74 compared to the RSA implementation of OpenSSL. Moreover, RegRSA supports multiple instances on multi-core CPUs simultaneously, which makes RegRSA more practical for the real-world applications against memory disclosure attacks. We will explore to use two sets of registers of Hyper-Threading for one RSA computation instance in the future.

References

1. Anderson, R., Kuhn, M.: Tamper resistance - a cautionary note. In: 2nd Usenix Workshop on Electronic Commerce, vol. 2, pp. 1–11 (1996)
2. Bernstein, D.: Cache-timing attacks on AES (2005)

3. Blass, E.-O., Robertson, W.: TRESOR-HUNT: Attacking CPU-bound encryption. In: 28th Annual Computer Security Applications Conference, pp. 71–78. ACM (2012)
4. Böck, B., Austria, S.B.: Firewire-based physical security attacks on windows 7, efs and bitlocker. Secure Business Austria Research Lab (2009)
5. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: 8th Workshop on Cryptographic Hardware and Embedded Systems, pp. 201–215 (2006)
6. Bos, J.W., Montgomery, P.L., Shumow, D., Zaverucha, G.M.: Montgomery multiplication using vector instructions. In: Lange, T., Lauter, K., Lisoněk, P. (eds.) SAC 2013. LNCS, vol. 8282, pp. 471–490. Springer, Heidelberg (2014)
7. Brumley, D., Boneh, D.: Remote timing attacks are practical. Comput. Netw. **48**(5), 701–716 (2005)
8. Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digit. Invest. **1**(1), 50–60 (2004)
9. DuLong, C., Gutman, M., Julier, M., et al.: Complete Guide to MMX Technology. McGraw-Hill Professional, New York (1997)
10. Garmany, B., Müller, T.: PRIME: Private RSA infrastructure for memory-less encryption. In: Proceedings of the 29th Annual Computer Security Applications Conference, pp. 149–158. ACM (2013)
11. Guan, L., Lin, J., Luo, B., Jing, J.: Copker: Computing with private keys without ram. In: 21st ISOC Network and Distributed System Security Symposium (NDSS) (2014)
12. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: 36th IEEE Symposium on Security and Privacy (Oakland) (2015)
13. Gueron, S.: Intel Advanced Encryption Standard (AES) New Instructions Set (2010)
14. Gueron, S., Krasnov, V.: Software implementation of modular exponentiation, using advanced vector instructions architectures. In: Özbudak, F., Rodríguez-Henríquez, F. (eds.) WAIFI 2012. LNCS, vol. 7369, pp. 119–135. Springer, Heidelberg (2012)
15. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: Cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009)
16. Henson, M., Taylor, S.: Beyond full disk encryption: Protection on security-enhanced commodity processors. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 307–321. Springer, Heidelberg (2013)
17. Henson, M., Taylor, S.: Memory encryption: A survey of existing techniques. ACM Comput. Surv. (CSUR) **46**(4), 53 (2014)
18. Intel. Intel 64 and ia-32 architectures software developer's manual volume 2 (2a, 2b & 2c): Instruction set reference, a-z (2015)
19. Koc, C.K.: High-speed RSA implementation. Technical report, RSA Laboratories (1994)
20. Koç, Ç.K., Acar, T., Kaliski, B.S.: Analyzing and comparing montgomery multiplication algorithms. Micro, IEEE **16**(3), 26–33 (1996)
21. Lomont, C.: Introduction to intel advanced vector extensions. Intel White Paper (2011)
22. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)

23. Müller, T., Dewald, A., Freiling, F.C.: AESSE: A cold-boot resistant implementation of AES. In: 3rd European Workshop on System Security, pp. 42–47. ACM (2010)
24. Müller, T., Freiling, F.C., Dewald, A.: TRESOR runs encryption securely outside RAM. In: USENIX Security Symposium, p. 17 (2011)
25. Pabel, J.: Frozen cache. Blog (2009). http://frozenchache.blogspot.com
26. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
27. Simmons, P.: Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In: 27th Annual Computer Security Applications Conference, pp. 73–82. ACM (2011)
28. Wetzels, J.: Hidden in snow, revealed in thaw: Cold boot attacks revisited. arXiv preprint (2014). arXiv:1408.0725
29. Yang, Y., Guan, Z., Liu, Z., Chen, Z.: Protecting elliptic curve cryptography against memory disclosure attacks. In: Hui, L.C.K., Qing, S.H., Shi, E., Yiu, S.M. (eds.) ICICS 2015. LNCS, vol. 8958, pp. 49–60. Springer, Heidelberg (2015)