# Protecting Elliptic Curve Cryptography Against Memory Disclosure Attacks

Yang Yang[1,2,3], Zhi Guan*[1,2,3], Zhe Liu[4], and Zhong Chen[1,2,3]

[1] Institute of Software, School of EECS, Peking University, China
[2] MoE Key Lab of High Confidence Software Technologies (PKU)
[3] MoE Key Lab of Network and Software Security Assurance (PKU)
[4] University of Luxembourg
{yangyang,guanzhi,chen}@infosec.pku.edu.cn zhe.liu@uni.lu

**Abstract.** In recent years, memory disclosure attacks, such as cold boot attack and DMA attack, have posed huge threats to cryptographic applications in real world. In this paper, we present a CPU-bounded memory disclosure attacks resistant yet efficient software implementation of elliptic curves cryptography on general purpose processors. Our implementation performs scalar multiplication using CPU registers only in kernel level atomatically to prevent the secret key and intermediate data from leaking into memory. Debug registers are used to hold the private key, and kernel is patched to restrict access to debug registers. We take full advantage of the AVX and CLMUL instruction sets to speed up the implementation. When evaluating the proposed implementation on an Intel i7-2600 processor (at a frequency of 3.4GHz), a full scalar multiplication over binary fields for key length of 163 bits only requires 129 $\mu s$, which outperforms the unprotected implementation in the well known OpenSSL library by a factor of 78.0%. Furthermore, our work is also flexible for typical Linux applications. To the best of our knowledge, this is the first practical ECC implementation which is resistant against memory disclosure attacks so far.

**Keywords:** Elliptic Curve Cryptography, Efficient Implementation, Memory Disclosure Attack, Cold Boot Attack, AVX, CLMUL

## 1 Introduction

Main memory has long been commonly used to store private keys at runtime for various cryptosystems because of the assumption that memory space isolation mechanism of operating system and volatility of dynamic RAM (DRAM) prevent access from adversaries both logically and physically. However, the presence of *cold boot attack*[5] shows acquiring memory contents is much easier than most people thought. Cold boot attack leverages the data remanence property which is a fundamental physical property of DRAM chips, because of which DRAM contents take a significant time to fade away gradually. This property exists in

* Corresponding Author

all DRAM chips and it determines how DRAM chips works: holding memory contents by refreshing the state of each memory unit periodly. The problem is that the time before memory contents fading away after the moment at which memory chips lost power can be significant extended at a low temperature, and memory contents stop fading away and become readable again once memory chips power on, so memory contents may survive across boots. As a result, for a running target machine physically accessible to adversaries, adversaries are able to transplant the memory chips from the machine to a well prepared attack machine after cooling down the memory chips, and cold boot the attack machine with a customized boot loader to bypass all defence mechanisms of the target machine and dump the memory contents to some persistent storage to get a snapshot of the memory contents.

Cold boot attack is not the only attack can be used to acquire the memory contents. DMA attack is another powerful attack which uses direct memory access through ports like Firewire, IEEE1394 to access physical memory space of the target machine. And the recently exposed vulnerable of OpenSSL called "HeartBleed"[5] which leaks memory contents onto network can also be used for memory acquisition. These attacks are called **memory disclosure attacks** in general, since they disclose contents in memory partially or entirely to adversaries while adversaries are unable to actively change the contents in the memory. Not only PCs and laptops, smart phones have also been demonstrated to be vulnerable to these attacks[13]. These attacks pose huge and prevalent threats to the implementation of both public-key and symmetric-key cryptosystems which hold secret keys in memory at runtime, and adversaries are able to reconstruct the key efficiently when only a part of the key or key schedule is acquired[5,7]. Compared to symmetric-key cryptosystems, these attacks may cause more damage on public-key cryptosystems, since revoking a compromised private key is very expensive, sometimes even impossible: private keys often have a long life cycle such as several years, leakage of high sensitive private keys leads to serious consequences.

Memory disclosure attacks have become a research hotspot since the presence of cold boot attack[5]. Akavia et al. proposed a new security model to formally describe memory disclosure attacks[1], in which the total amount of key leakage is bounded while "only computation leaks information" in traditional model of side channel attacks. Based on this model, several schemes have been proposed to resist memory disclosure attacks theoretically[1] [2] [15]. These solutions are not practical and cannot be used to protect existing cryptographic primitives, such as AES, RSA, etc.

Several solutions based on CPU-bounded encryption have been proposed to protect existing cryptographic primitives. The underlying idea of which is to avoid DRAM usage completely by storing the secret key and sensitive inner state in CPU registers, since there has been no known attacks can be used to acquire contents of CPU registers. AESSE [11] from Müller et al. implemented an AES cryptosystem using X86 SSE registers as the key storage. The perfor-

[5] `http://heartbleed.com`

mance of AESSE is about 6 times slower than a standard implementation. As a successor of AESSE, TRESOR [12] utilizes the new X86 AES-NI instructions for AES encryption to provide a better performance and a better compatibility with Linux distributions. Loop-Amnesia [16] stores a master key inside Machine Specific Registers (MSRs), and supports multiple AES keys securely inside RAM by scrambling them with the master key. TreVisor [14] builds TRESOR into the hypervisor layer to yield an OS-independent disk encryption solution.

On the other hand, resisting memory disclosure attacks for public key cryptosystem is more challenging. Garmany et al.[4] have proposed a memory disclosure attacks resistant RSA implementation based on CPU-bounded encryption. They achieved 21 ms per operation for modular exponentiation, which is about ten times slower than off-the-shelf cryptographic libraries. And the TLS server based on their implementation achieved a significant higher latency under high load in the benchmark.

An important reason for the unsatisfied performance in PRIME is the extreme large key size of RSA for CPU registers. On the other hand, elliptic curve cryptography (ECC)[10][8] provides the same level of security as RSA with a much smaller key size, and thus, requires less memory storage. In this case, efficient implementation of ECC which resists against memory disclosure attacks is still a challenging work. Our work presented in this paper is going to make up for the gap.

### 1.1 Contributions

In this paper, we propose an efficient elliptic curve cryptography (ECC) implementation that also resists memory disclosure attacks by keeping the private key and sensitive intermediate state out of the memory. In detail, our major contributions are listed as follows:

- We proposed an efficient and memory disclosure attack resistant CPU-bounded elliptic curve cryptography cryptosystem using new features on modern X86-64 CPUs. The cryptosystem keeps the private key and sensitive intermediate data within CPU registers after the private key is loaded so that attackers have no chance to retrieve the key or key schedule from the memory.
- The cryptosystem makes full use of AVX and CLMUL instruction sets of X86-64 CPU architecture to speed up the computation. The performance evaluation of our solution on an Intel i7-2600 processor found our implementation achieves a performance of 129 $\mu s$ for a scalar multiplication operation over binary fields for key length of 163 bits. This result outperforms the unprotected fashion of OpenSSL by a factor of 78.0%.
- To the best of our knowledge, this work provides the first memory disclosure attacks resistant ECC implementation, and the first public-key cryptographic implementation that resists these attacks, and provides a better performance than off-the-shelf cryptographic libraries at the same time.

The rest of this paper is organized as follows. A brief introduction of elliptic curve cryptography is given in section 2 together with architecture we focus on.

Next we describe how we design and implement the solution in section 3. Then the evaluation on security and performance is given in section 4. Finally, we conclude this paper in section 5.

## 2 Preliminaries

In this section, we first recap the basic knowledge of elliptic curve cryptography and then give a brief description of the CPU architecture and key instruction sets we used for our implementation.

### 2.1 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) can be considered as an approach of public-key cryptography based on the algebraic structure of elliptic curves over finite fields[10][8]. An elliptic curve over a field $K$ is defined by Equation 1.

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1}$$

The equation can be simplified for the characteristic of $K$ is 2, 3, or greater than 3, respectively. The points satisfying the equation and a distinguished point at infinity which is the identity element forms a set, together with group operations of the elliptic group theory form an Abelian group. The group is used in the construction of elliptic curve cryptographic system.

There are two basic group operations in ECC: a point addition adds two different points together and a point doubling operation doubles a single point. Point addition and point doubling are computed according to a *chord-and-tangent* rule. *Scalar multiplication* computes $k \cdot P$ where $k$ is a scalar and $P$ is a point on an elliptic curve, that is adding $P$ together $k$ times. A scalar multiplication is computed as a serious of point additions and point doublings. Like the integer exponentiation in RSA cryptosystem, scalar multiplication is the basic cryptographic operation of various ECC schemes.

Domain parameters define the field and the curve of an ECC system, including the field order $f$, the curve constants $a$ and $b$, the base point(subgroup generator) $G$, the order of the base point $n$ and the cofactor $h$. The generation of domain parameters is time consuming, therefore several standard bodies published domain parameters for several common field sizes, which are commonly known as “standard curves” or “named curves”. Domain parameters must be agreed on by all parties before use.

Each party must also generate a public-private key pair before use. The private key $d$ is a randomly selected scalar. The public key $Q$ is computed as $Q = d \cdot G$, where $G$ is the base point. The security of an ECC cryptosystem relies on the assumption that finding the discrete logarithm of a random elliptic curve point with respect to a public known base point is infeasible.

Elliptic Curve Diffie-Hellman(ECDH) is an anonymous key agreement protocol that allows two parties to establish a shared secret over an insecure channel.

It is a variant of the Diffie-Hellman protocol using elliptic curve cryptography. Denote two parties **A** and **B**, the key pairs of them are $(d_A, Q_A)$ and $(d_B, Q_B)$, then they can compute $d_A Q_B$ and $d_B Q_A$ respectively to get a shared secret, as $(x_k, y_k) = d_A Q_B = d_A d_B G = d_B Q_A$ where $x_k$ is the shared secret. A symmetric key can be derived from the shared secret to encrypt subsequent communications. The key exchange protocol is one of the most important applications for public key cryptography.

### 2.2 The x86-64 CPU Architecture

x86-64 is the 64-bit version of x86 instruction set. It supports larger memory address space and register files. X86-64 is supported by mainstream operating systems and widely used in modern desktop and laptop computers. Besides the base instruction set, we mainly use its two extensions: CLMUL and AVX.

**Carry-Less Multiplication** The Carry-Less Multiplication(CLMUL)[6] instruction set is an X86 extension that can be used to compute a polynomial multiplication, which is the product of two operands without the generation or propagation of carry values. Carry-less multiplication is an essential processing component of several cryptographic systems and standards, including of the Galois Counter Mode(GCM) and elliptic curve cryptography over binary fields. Software implementations of carry-less multiplication are often inefficient and suffer from potential side channel attacks, while CLMUL instruction set provides an convenient and efficient way to calculate carry-less multiplications.

**Advanced Vector Extensions** The Advanced Vector Extensions (AVX/AVX2 /AVX–512) [7] are X86 SIMD extensions proposed by Intel. AVX and AVX2 are supported by recent processors, while AVX–512 will be supported in 2015. AVX/ AVX2 doubles the amount of SIMD registers from 8 to 16, which are named as YMM0–YMM15 respectively, and extends the width of each register from 128 bits to 256 bits. AVX–512 doubles the amount and width of SIM registers again to 32 and 512 bits resprectively. AVX introduces a non-destructive three-operand SIMD instruction format. Mixed usage of AVX and legacy SSE instructions should be avoided to prevent AVX-SSE transition penalties. For any legacy SSE instruction, there is an equivalent VEX encoded instruction which should be used instead to avoid the penalty.

## 3 System Design and Implementation

### 3.1 System Overview

We hold the private key and intermediate data in CPU registers to avoid the leakage of secret data. We implement the ECC scalar multiplication using assembly language to control the usage of registers precisely to ensure no secret

[6] `http://en.wikipedia.org/wiki/CLMUL_instruction_set`
[7] `http://en.wikipedia.org/wiki/Advanced_Vector_Extensions`

data leaks into memory explicitly. To avoid the implicit leakage of secret data by context switch mechanism, we deploy the ECC implementation in kernel level in a loadable kernel module(LKM) and make the computation atomically by disabling the interruption and kernel preemption during the computation. Netlink based interfaces are provided for user level applications. We also implement an OpenSSL engine to provide an interface based on our implementation for ECDH operations to demonstrate the possibility of integrating our implementation with existed applications. A private key is imported into the cryptosystem before use. The imported key is stored in debug registers and loaded into YMM registers before each scalar multiplication. The kernel of the operating system is patched to restrict access to debug registers. The architecture of the system is shown in Figure 1.

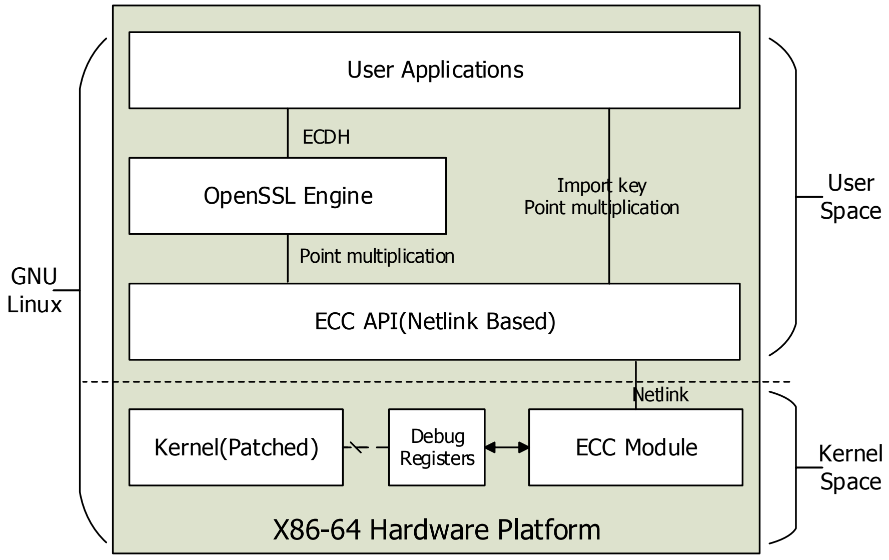

**Fig. 1.** The architecture of the proposed ECC cryptosystem.

### 3.2 Implementation of Secure Scalar Multiplication in ECC

**Field Operations** For domain parameters "sect163k1", there are two sizes of polynomial involved in field operations: 163-bit and 325-bit. The former is the size of polynomials over the field $\mathbb{F}_{2^{163}}$, while the latter is the size of the product of two polynomial and will be further reduced to 163 bits. As we implement field operations mainly with AVX instructions and YMM registers, we use one YMM register for a 163-bit polynomial and two YMM registers for a 325-bit polynomial.

Operations over binary field are carry-less operations, which means the operation is performed without the generation or propagation of any carry values. In this case, for any polynomial $a$ and $b$ over the binary field, the following equation holds:

$$a + b = a - b = a \oplus b$$

Namely that an addition and a subtraction over binary field can be simply calculated by XORing two operands.

The presence of CLMUL instruction set makes it much easier to implement carry-less multiplication efficiently and securely. Denote the 163-bit input operands $A$ and $B$ by $[A_2 : A_1 : A_0]$ and $[B_2 : B_1 : B_0]$, where $A_i, B_j, 0 \leq i, j \leq 2$ is a 64-bit quad-word. The carry-less multiplication between $A_i$ and $B_j$ can be simply calculated with a `VPCLMULQDQ` instruction:

$$C_{ij} = A_i \cdot B_j = \texttt{VPCLMULQDQ}(A_i, B_j), 0 \leq i, j \leq 2$$

Then the product $C$ of $A$ and $B$ can be calculated as:

$$C = A \cdot B = [C_{22} : C_{21} \oplus C_{12} : C_{20} \oplus C_{11} \oplus C_{02} : C_{10} \oplus C_{01} : C_{00}]$$

Therefore we can implement a carry-less multiplication of two 163-bit operands with only 9 PCLMULQDQ instructions for partial products, 4 bitwise XOR instructions to combine partial products together, and several register manipulation instructions to put the partial products into right place in the result. A squaring operation can be implemented similarly as multiplicate the operand with itself.

The result of a multiplication or a squaring is a 325-bit polynomial, therefore it has to be reduced to 163 bits before further use. We use the NIST fast reduction algorithm and modulo $f(z) = z^{163} + z^7 + z^6 + z^3 + z^1$ defined in FIPS 186-2[3] for reduction.

The inversion operation is to find a polynomial $g = a^{-1} \text{mod} f$ over $F_{2^m}$ for the polynomial $a$ satisfying $ag \equiv 1 (\text{mod} f)$. The inverse can be efficiently calculated by the extended Euclidean algorithm for polynomials. In our solution, we use the "modified almost inverse algorithm for inversion in $\mathbb{F}_{2^m}$" algorithm[6] to calculate the inversion of a polynomial in $\mathbb{F}_{2^{163}}$. An inversion operation takes much longer time than other basic field operations, therefore the group operation algorithm must be carefully selected to reduce the number of inversion operations.

**Group Operations** We employ Montgomery ladder for elliptic curves over binary fields to compute a scalar multiplication [9]. One advantage of this algorithm is that it does not have any extra storage requirements, which is suitable for our CPU-bounded ECC cryptosystem since the storage space we can use is limited. It is also efficient enough and has been used in mainstream cryptographic libraries, such as OpenSSL. We use the projective version of the algorithm in order to reduce field inversions, as described by López and Dahab[9].

In this algorithm, each iteration $j$ between line 4–13 performs an addition (line 7 and 10) and a doubling (line 8 and 11) to compute the x-coordinates only of $T_j = [lp, (l + 1)p]$, where l is the integer represented by the $j$th left most bit of $k$. After the final iteration, $x$-coordinates of $kP$ and $(k + 1)P$ have been computed, and line 14 and 15 are used to compute the affine coordinates of $kP$. Using temporary variables for intermediate result, one addition requires 4 field multiplications and one squaring, and one doubling requires 2 field multiplications and 4 squarings. The number of temporary variables used by an addition and a doubling is 2 and 1 respectively. The operation used to convert $kP$ and $(k+1)P$ back to affine coordinates requires 3 temporary variables, 10 field multiplications, 1 field squaring and 1 field inversion.

We allocate the YMM registers as follows. One YMM register for storing a dimension of a point on elliptic curve over $\mathbb{F}_{2^{163}}$, and thus, two YMM registers are sufficient for a point represented in affine coordinates. Following the equations listed above, we implement addition and doubling with field operation in MACROs. Consequently, a doubling operation requires five YMM registers and an addition requires eight YMM registers, as well as the scalar multiplication algorithm (i.e. doubling-and-addition) requires 12 YMM registers. At the end, the ECC private key is a polynomial over the field and can be stored into one YMM register.

## 3.3 Deployment of ECC Cryptosystem in the Operating System

The implementation of ECC scalar multiplication should be carefully deployed in the operating system to make it be secure and accessible to user space applications. As described above, we implement the ECC cryptosystem as a loadable kernel module(LKM) to make it run atomically and patched the kernel to protect the private key. We demonstrate the modification and deployment successfully on Ubuntu 14.04, the kernel version is 3.13.0-34.

**Atomicity** The private key and sensitive intermediate data is kept in YMM registers during cryptographic computations, so we have to make the computation atomic to prevent sensitive intermediate data being swapped into the memory. This can be achieved by disabling the interruption and the kernel preemption by `preempt_disable()`, `local_irq_save()` and `local_irq_disable()` before the computation, and enable the interruption and the kernel preemption again after the computation by `local_irq_enable()`, `local_irq_restore()` and `preempt_enable()` to make the system run normally.

**User Space Interface** We provide two interfaces based on netlink mechanism, which is widely used for communications between kernel and user space processes of the same host:

**private_key_import** Import the private key into the debug registers. The input is the plain text of the private key, which is a 163-bit big number.

**private_key_operation** Calculate the product of the point multiplication between the private key and a given point. The input is a point which is represented by two big numbers.

Each interface is implemented as a request-response round trip, namely the user space process send the request consisting of the function name and input values into the kernel then block until a response is received.

**Private Key Protection** Access to debug registers should be prevented from either user space or kernel space other than the code of our system to prevent key damage and loss of secret data. Debug registers can only be accessed with ring 0 privilege directly. Access to debug registers in user space and kernel space are finally delegated to kernel functions `ptrace_set_debugreg`, `ptrace_get_debugreg`, `native_set_debugreg` and `native_get_debugreg`. We modified these functions to discard any change to debug registers and inform the caller there is no debug registers available. We searched the source of the kernel we are using thoroughly and found no other accesses to debug registers besides in these functions, which means only our module has access to debug registers in our patched system in both user space and kernel space.

Debug registers are per-core registers, therefore we have to copy the key into debug registers on all CPUs to prevent logical errors. We implement this procedure with the help of the kernel function `smp_call_function` which runs a certain function on all other CPUs.

## 4 Evaluation and Discussion

### 4.1 Security Verification

We assume the procedure of loading the key from a secure storage to the registers is safe and the memory trace of the key is erased immediately. This assumption is reasonable, because this procedure is transient and the user of the system has to physically access the computer to load the key, which also prevent the physical access from attackers. Therefore our system focus on the life cycle since the private key has been loaded into the registers.

We implement ECC in a way that no private key or sensitive intermediate data leaks into memory once the private key has been loaded into debug registers. Considering about the threats posed by memory disclosure attacks, our approach resists these attacks since adversaries cannot get the private key or any private key related data which can be used for key recovery with these attacks from our cryptosystem. In this section, we analyze and evaluate the approach to verify its security under the threats of these attacks.

Since we have already patched kernel functions which are used to access debug registers, and Müller et al. have verified debug registers are reset to zero after both cold and warm reboot, there is no way for the key to be moved into memory from debug registers or be accessed with cold boot attack.

For intermediate data and the private key in YMM registers, they will not appear in memory unless we write them to memory explicitly or be swapped into memory due to interruptions during the execution. We reviewed our code

**Table 1.** Performance comparison of scalar multiplication between OpenSSL and this work. ⋆: unprotected version. †: protected version

| Implementation | `Operations Per Second` |
|---|---|
| OpenSSL | 4346⋆ |
| This work | 7734† |
| Improvements | 78.0% |

thoroughly and made sure only the final result is written into memory. As preemption and interruption are disabled during the computation, our system is theoretically not vulnerable to memory disclosure attacks.

We also verified the correctness of the system on the test machine. Since we have already known the private key, we do this by acquire a series of snapshots of the memory contents after the key is loaded and search for the private key in them. If our system works correctly, there will be no match in snapshots. Performing a memory disclosure attack such as cold boot attack actually is time consuming, and these attacks get no more than memory contents, so we used a tool called **fmem** [8] to acquire the whole range of memory contents instead. We also performed this test on a system running a process using OpenSSL ECC library with the same private key as a comparison. The result shows there is a match of the private key on the system using OpenSSL while there is no significant match on the system using our approach.

As should be noted that the security of cryptographic application in real world is not trivial, single countermeasure can not mitigate different attacks. But our method is compatible with other countermeasures such as software based power analysis defeat method and can be used together.

### 4.2 Performance

Our benchmark is running on a desktop with an Intel Core i7-2600 processor set to be constantly running at 3.4 GHz. The operating system is Ubuntu Linux 13.10 64-bit with our modified kernel at version 3.11.0. We implemented an OpenSSL ECDH engine using scalar multiplication in our approach, and compared the performance of ECDH operation on curve SECT163K1 in our implementation with the same operation provided by OpenSSL[9] at version 1.0.1e. The evaluation shows the performance improvements precisely and practically, since ECDH is widely used and each operation comprises mainly a single scalar multiplication. The performance is measured by operations per second with the OpenSSL built-in speed tool. the result is shown in Table 1.

As shown in the table, an ECDH operation, namely a scalar multiplication in our solution is faster than that in OpenSSL by a factor of 78.0%. The result of performance evaluation is encouraging, since our implementation is memory disclosure attacks resistant and that in OpenSSL is not, ours is also much more

[8] `http://hysteria.sk/~niekt0/fmem/`

[9] `https://www.openssl.org/`

**Table 2.** Performance comparison of field arithmetic operations between OpenSSL and this work (in $\mu$s per operation)

| Implementation | `Modular Add.` | `Modular Mul.` | `Modular Sqr.` | `Inversion` |
|---|---|---|---|---|
| OpenSSL | 0.013 | 0.160 | 0.117 | 3.670 |
| This work | 0.004 | 0.084 | 0.083 | 3.133 |
| Improvements | 225% | 90.1% | 41.0% | 17.1% |

efficient. We also compared the performance of running scalar multiplications directly and running with the APIs exposed into user space, and found communications between user space and kernel brings 2%–5% overheads, which are acceptable.

We also compared the performance improvements of each field operations, the result is shown in Table 2. The performance of field addition, modular multiplication and modular squaring in our solution is faster than that in OpenSSL by a factor of 225%, 90.1% and 41.0% respectively. The performance of inversion in our solution is faster than that in OpenSSL by a factor of 17%. The performance improvements due to the reduction of branches, unrolled loops and the power of CLMUL and AVX instruction sets, etc.

## 5 Conclusion and Future Work

Memory disclosure attacks such as cold boot attack have become a threat that must be considered by designers and developers of cryptographic libraries and applications. To mitigate such threats, we present a systematic solution on protecting public key cryptography based on the idea that restricts the private key and the intermediate states during the cryptographic operations inside CPU. Our solution is implemented as a Linux kernel patch with interfaces in the user space to provide secure elliptic curve scalar multiplications, and various secure ECC schemes can be constructed based on it. Evaluation shows our approach leaks none of the information that can be exploited by attackers to the memory, therefore it resists the cold boot attack and other memory disclosure attacks effectively. An ECC scalar multiplication over binary fields for key length of 163 bits in our solution is about 78.0% faster than that in OpenSSL. To the best of our knowledge, our solution is the first efficient ECC implementation that is memory disclosure attacks resistant.

One of our future work is to support multiple private keys in our cryptosystem so that it supports multiple applications at the same time. After the release of processors supporting AVX-512, we will provide support of larger key size since AVX-512 has 4 times more register space than AVX/AVX2.

## Acknowledgements

This work is supported by National High Technology Research and Development Program of China under Grant No. 2014AA123001.

## References

1. Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *Theory of Cryptography*, number 5444 in Lecture Notes in Computer Science, pages 474–495. Springer Berlin Heidelberg.
2. Yevgeniy Dodis, Kristiyan Haralambiev, Adriana Lpez-Alt, and Daniel Wichs. Efficient public-key cryptography in the presence of key leakage. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, number 6477 in Lecture Notes in Computer Science, pages 613–631. Springer Berlin Heidelberg, January 2010.
3. PUB FIPS. 186-2. digital signature standard (DSS). *National Institute of Standards and Technology (NIST)*, 2000.
4. Behrad Garmany and Tilo Mller. PRIME: private RSA infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, page 149158. ACM.
5. J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX Security Symposium*, pages 45–60, 2008.
6. Darrel Hankerson, Julio Lpez Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. In etin K. Ko and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems  CHES 2000*, number 1965 in Lecture Notes in Computer Science, pages 1–24. Springer Berlin Heidelberg.
7. N. Heninger and H. Shacham. Reconstructing RSA private keys from random key bits. *Advances in Cryptology-CRYPTO 2009*, page 117, 2009.
8. Neal Koblitz. Elliptic curve cryptosystems. 48(177):203209.
9. Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over gf (2m) without precomputation. In *Cryptographic Hardware and Embedded Systems*, pages 316–327. Springer, 1999.
10. Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology  CRYPTO 85 Proceedings*, number 218 in Lecture Notes in Computer Science, pages 417–426. Springer Berlin Heidelberg.
11. Tilo Müller, Andreas Dewald, and Felix C. Freiling. AESSE: a cold-boot resistant implementation of AES. In *Proceedings of the Third European Workshop on System Security*, EUROSEC '10, pages 42–47, New York, NY, USA, 2010. ACM.
12. Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR runs encryption securely outside RAM. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.
13. Tilo Mller and Michael Spreitzenbarth. FROST. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, number 7954 in Lecture Notes in Computer Science, pages 373–388. Springer Berlin Heidelberg, January 2013.
14. Tilo Mller, Benjamin Taubmann, and Felix C. Freiling. TreVisor. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, number 7341 in Lecture Notes in Computer Science, pages 66–83. Springer Berlin Heidelberg, January 2012.
15. Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. 41(4):772–814.
16. Patrick Simmons. Security through Amnesia: a software-based solution to the cold boot attack on disk encryption. In *ACSAC*, pages 73–82, 2011.