PRIME:
Private RSA Infrastructure for Memory-less Encryption

Behrad Garmany
Horst Goertz Institute for IT-Security
Ruhr-University Bochum, Germany
behrad.garmany@rub.de

Tilo Müller
Friedrich-Alexander University
Erlangen-Nuremberg, Germany
tilo.mueller@cs.fau.de

ABSTRACT

Cold boot attacks exploit the fact that data in RAM gradually fades away over time, rather than being lost immediately when power is cycled off. An attacker can gain access to all memory contents by a restart or short power-down of the system, a so called *cold boot*. Consequently, sensitive data in RAM like cryptographic keys are exposed to attackers with physical access. Research in recent years found software-based solutions to the cold boot problem in terms of *CPU-bound* or *memory-less* encryption. To date, however, the focus has been set on symmetric ciphers, particularly concerning disk encryption systems. Contrary to that, the work in hand aims to close the gap to asymmetric ciphers. With *PRIME*, we present a cold boot resistant infrastructure for private RSA operations. All private RSA parameters reside symmetrically encrypted in RAM and are decrypted only within CPU registers. The modular exponentiation algorithm for RSA is implemented entirely on the CPU, such that no sensitive state of RSA ever goes to RAM.

Categories and Subject Descriptors

E.3 [**Data**]: Data Encryption; D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

Security

Keywords

RSA, Cold Boot Attack, CPU-bound Encryption

1. INTRODUCTION

Even after three decades of its existence, the RSA cryptosystem is still considered to be cryptographically secure. However, research done by Halderman et al. [14] disclosed a serious side channel threat against all software-based encryption systems, including RSA. In their renowned paper *Cold Boot Attacks on Encryption Keys*, Halderman et al. showed that an attacker can exploit the *remanence effect* of RAM [13] by cold booting the target system and loading a custom OS kernel that retrieves what is left in memory. Cold booting a device can be done by pressing the reset button or quickly cycling the power off and on; loading a malicious OS kernel can be done by booting a USB flash drive, or alternatively by replugging RAM chips physically into another PC. This concept can be reproduced by existing tools, and it is its easy practical execution that causes cold boot attacks to be a generic and serious threat.

Cold boot attacks expose sensitive key material to an attacker only if he or she gains physical access to a machine that is running. However, mobile devices like laptops and smartphones are frequently lost while they are running (or being in standby mode). Hence, mobile systems used for signing and decrypting emails are likely to disclose RSA private keys via RAM. Above all, stationary SSL servers can be subjected to internal attackers, or to lawful search and seizure, disclosing RSA keys and certificates via RAM that concern even thousands of end-users.

1.1 Contributions

To overcome the threat of cold boot attacks against private RSA keys, we present PRIME – a *Private RSA Infrastructure for Memory-less Encryption*. Our contributions are:

- With PRIME, we give a *CPU-bound* or *memory-less* encryption system for RSA. Memory-less encryption is a widely known method to protect symmetric ciphers against cold boot attacks [19, 26, 31, 27]. However, due to the higher memory footprint of RSA as compared to symmetric ciphers, it has not been clear yet if such an implementation is feasible for RSA.

- Known CPU-bound encryption systems store keys of up to 256 bits inside debug registers of modern 64-bit CPUs. With PRIME, however, we want to support RSA-2048 but cannot store 2048 bits persistently inside the CPU. As a consequence, we store RSA private keys symmetrically encrypted in RAM and decrypt them within Intel's AVX [17] multimedia register set by means of AES-NI.

- Modular exponentiation of big numbers that runs entirely on the microprocessor was never implemented before. Eventually, we solved this task by an optimized variant of *Montgomery's method* [25] in assembly language, and by holding some intermediate values of RSA

ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA
Copyright 2013 ACM 978-1-4503-2015-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2523649.2523656

in RAM. Everything we hold in RAM is either provably non-critical, or gets symmetrically encrypted.

- Memory-less encryption must be run in kernel mode to avoid side effects like context switching that move registers into RAM. Therefore, we provide PRIME as a patch for the Linux kernel (version 3.0 and later). To prove the applicability of our approach, we patched the PolarSSL library and ran a web server on top of PolarSSL performing TLS handshakes by means of PRIME.
- Last, we present a usability and a security analysis of PRIME. We argue that no sensitive data of RSA leaks into RAM by running PRIME inside a virtual machine and observing its memory from outside. As a drawback, PRIME performs about 10 times slower than conventional RSA implementations. Despite this drawback, our prototype can be used for clients and high security servers with low or medium load.

PRIME focuses on private RSA operations. Public RSA operations are not critical and not supported by our implementation.

1.2 Related Work and Motivation

Attacks based on the data remanence of RAM have a long tradition, and the fact that especially keys are unsafe has been known for years. In 1996, Anderson and Kuhn [2] were the first to propose attacks that exploit the remanence effect. Gutmann [13] extended those issues in 2001 and provided a more technical insight into the effect. He suggested not to store cryptographic parameters for long time periods in RAM. This became crucial in 2008, when Halderman et al. [14] showed that with the right tools, the remanence effect can easily be exploited to recover encryption keys on common PCs. Halderman et al. put a focus on breaking disk encryption by recovering symmetric keys, but they also provided tools to recover RSA keys. In 2006, Klein demonstrated [20] that RSA keys can be traced in memory. He wrote a plugin for the IDA debugger which retrieves RSA keys from RAM.

Since often only *partial data* can be retrieved with a cold boot attack—due to the decay of bits in memory—some research has focused on retrieving cryptographic keys in the presence of errors. Halderman et al. introduced an error correcting algorithm that recovers noisy AES and RSA keys. They show that RSA keys can be reconstructed with 6% corruption of arbitrary bits of the key. Heninger and Shachman [15] later introduced a powerful algorithm, which can reconstruct RSA keys with even more corruption of the key. Obviously, these findings make cold boot attacks attractive in practice. Interestingly, a rather different approach than using the remanence effect to attack keys is achieved with TARDIS by Rahmati et al. [30]. In TARDIS, the remanence effect is used to estimate periods of time which can be used for cryptographic protocols on battery-less embedded devices.

In the recent past, many researchers focused on software-based solutions to protect symmetric keys of AES-based disk encryption systems. One contribution to this field is the project TRESOR by Müller, Freiling, and Dewald [26]. TRESOR is a Linux kernel patch that holds cryptographic keys inside debug registers of a CPU. The same authors later introduced TreVisor [27] which goes into the hypervisor layer to add the functionalities of TRESOR, yielding an OS-independent solution. An alternative system called Loop Amnesia [31] by Simmons saves the key inside machine specific registers (MSRs). A rather different approach than storing keys in registers is the Frozen Cache project by Pabel [19] which utilizes CPU caches as key storage. However, this approach can hardly be implemented in practice because CPU caches are designed to act transparently, and thus cannot be controlled well by system programmers.

We also found a related solution for the protection of asymmetric keys. Parker and Xu [28] obfuscate the private key of RSA in RAM and de-obfuscate it in SSE registers of modern x86 CPUs. However, the authors also mention that they cannot prove the security of their system as they obfuscate the key in RAM rather than encrypting it or holding it outside RAM. An attacker who thoroughly studies the obfuscation algorithm would likely be able to reconstruct the key. The lack of research done on the asymmetric field as compared to symmetric ciphers might be due to the fact that utilizing RSA hardware is a general trend. Trusted platform modules (TPMs) are strong and low cost hardware options, and SSL accelerators provide high performance as well as key protection for high-end applications.

Nevertheless, a software-based solution like PRIME is an interesting approach from an academic point of view, which not only has the advantage of reduced costs but also provides flexibility in adapting, extending and improving a system without exchanging hardware.

1.3 Outline

The remainder of this paper is organized as follows: In Section 2, we briefly give necessary background information on the RSA cryptosystem and on AVX. In Section 3, we focus on the design and implementation of PRIME. We describe a variant of Montgomery's method that we implemented for modular exponentiation. In Section 4, we evaluate our implementation regarding its usability and security. Finally, in Section 5, we finish up with a discussion about limitations and future works.

2. BACKGROUND INFORMATION

In the following we give a brief review of the RSA public-key cryptosystem (Section 2.1) and AVX (Section 2.2).

2.1 The RSA Cryptosystem

RSA uses two exponents, e and d, where e is public and d is private. Assume that we have two entities Alice and Bob. Alice wants to send Bob a message over an insecure channel. She calculates the ciphertext $C = M^e \ mod \ n$, where n is the product of two large prime numbers generated during the key generation process listed in Figure 1. To retrieve the plaintext, Bob calculates $P = C^d \ mod \ n$. So encryption and decryption requires modular exponentiation. Modular exponentiation that involves the private exponent is also called the *private RSA operation*.

The idea of RSA is that multiplying two primes is feasible in polynomial time but calculating its inverse operation, i.e., factorization, is a hard problem. If the exponent d is known, the ciphertext can easily be decrypted in polynomial time. Given only the public modulus n and the public exponent e, an adversary must be able to compute the e-th root of $C \ mod \ n$ in order to break the cipher, which is also known as the *RSA problem*. Without knowing the factorization of n, there is no known efficient algorithm that can launch an

attack. So the function that maps x to $x^e \ mod \ n$, which Alice uses to encrypt messages, is a one-way function with a trapdoor (exponent d) known only to Bob.

The security of RSA rests in large part on the idea that the modulus is so big that it is infeasible to factor it. If an adversary can factor the modulus n, he or she can obtain the prime numbers p and q, and hence can calculate $\phi(n) = (p-1)(q-1)$. With $\phi(n)$ and the public exponent e, the adversary can then compute d with the extended Euclidean algorithm [21]. So if factoring large numbers is easy, then breaking RSA is easy. The other direction is an open problem. Note that this specific textbook description of RSA is not secure. RSA implementations require padding schemes to be secure.

1. Select large prime numbers p and q with $p \neq q$.
2. Compute $n = p \cdot q$.
3. Choose e that is relatively prime to $\phi(n)$.
4. The pair (n, e) is published as the public key.
5. Compute $d = e^{-1} \ mod \ \phi(n)$.
6. The private key is d.

Figure 1: RSA key generation.

2.2 Advanced Vector Extensions

Today, video games have become an integral part of many lives of a growing community. For the continuing need of performance with respect to visual processing and gaming, Intel developed the Advanced Vector Extensions (AVX) [24]. AVX is available since Intel's Sandy Bridge processors, shipped in Q1 2011. Besides gaming, these multimedia extensions can be utilized by scientific applications and cryptography.

AVX introduces new instructions and a set of new registers. Contrary to 32-bit and 64-bit general purpose registers, AVX registers are 256 bits wide. There are sixteen 256-bit registers (denoted by YMM0 to YMM15) [17], providing a storage of $16 \times 256 = 4$ kilobits in total. Technically, the SSE registers (denoted by XMM0 to XMM15) were extended such that AVX and SSE registers overlap. Hence, we cannot benefit from both register sets. Nevertheless, 4 kilobits is enough to implement modular exponentiation of big integers solely on the CPU. Basically, we make use of these registers as a surrogate stack or heap, since we do not want to use RAM.

Note that AVX2, which first became available on Intel Haswell CPUs in Q2 2013, does *not* increase the register size or number of registers further. AVX2 only extends the AVX instructions, such that our algorithm is not expected to benefit much from this advancement.

With AVX-512 Intel doubles both the register size and number of registers. Our approach will truly benefit from this architectural advancement. However, AVX-512 will first be supported by Intel with the Knights Landing processors. To date, there is no release date given.

3. DESIGN AND IMPLEMENTATION

The idea to use memory-less encryption for cold boot resistance is not new and has been successfully applied on symmetric ciphers before, particularly for disk encryption. TRESOR [26] is a Linux kernel patch designed to avoid RAM entirely such that all intermediate values of AES are computed within CPU registers only. We were inspired by this solution and learned from it, mainly regarding Linux kernel integration. For the CPU-bound encryption algorithm itself, however, RSA considerably raises the challenge compared to AES, due to its enormous key and block sizes. As stated above, the current RSA standard demands a key of 2048 bits, and this is the size we support in PRIME. Doing arithmetics with 2048-bit RSA numbers, in particular modular exponentiation, requires a lot memory.

We compute private RSA operations without leaking any sensitive information into RAM by means of Montgomery's method, as we explain in Section 3.1. To overcome side effects such as context switching, we run this algorithm inside kernel mode, as we explain in Section 3.2. Finally, in Section 3.3, we provide an overall view of our PRIME infrastructure.

3.1 Memory-less RSA Algorithm

In an early prototype implementation of PRIME, we managed to run RSA solely on the microprocessor by using a modified variant of the modular exponentiation algorithm proposed by Blakely in 1983 [6]. We combined Blakely's method with the *Chinese reminder theorem* (CRT) [22] to speed up our implementation, but after all we reached an encryption speed that was about 1,000 times worse than that of OpenSSL. The enormous loss in performance was mainly owed to the lack in memory. Considering only CPU registers as a safe storage for RSA values is not a trivial task, and we had to introduce costly workarounds by re-computing intermediate values.

This performance drawback forced us to think about another approach. In 1985, Montgomery introduced an efficient algorithm for modular exponentiation [25]. This algorithm turned out to be more suitable for our purpose, i.e., more suitable as a basis for a CPU-bound encryption. With the help of Montgomery's method, we were able to decrease the performance drawback of PRIME down to factor 10.

Another improvement we introduced together with Montgomery's method was to relax our *security policy* slightly, meaning that we now store well-chosen intermediate values of RSA in RAM. Note that adversaries cannot observe RAM contents over a period of time, but that they can—due to the nature of cold boot attacks—retrieve only static RAM images at a certain point in time. We take advantage of this fact carefully and store some values in RAM briefly. These values by themselves do not affect the security of RSA and we wipe each affected RAM line shortly afterwards.

3.1.1 Montgomery's Method

The basic idea of Montgomery's method is to transform numbers into a system where division is performed by a power of 2. For example, a division by 2^l (where $l > 0$) only requires us to drop the least significant l bits, because a division by 2 is a right shift. In microprocessors the size of registers is usually a power of 2, as it is the case with AVX (256 bits). Therefore, Montgomery's method is particularly suited for an implementation on CPU registers. It is basically an algorithm for modular multiplication, but it is also a common technique for RSA private operations. A profound introduction to Montgomery's method can be found in the literature [22]; in the following, we give necessary basics and explain our variant of this algorithm.

Let the modulus n be an l-bit integer and let r be 2^l.

The Montgomery method requires that r and n be relatively prime, i.e., $gcd(r, n) = 1$. This requirement is satisfied in RSA where n is odd. To use the Montgomery algorithm, each number is transformed to what is called the *n-residue form* by multiplying the numbers with r modulo n. For example, the n-residue form of an integer a is defined as

$$\bar{a} = a \cdot r \pmod{n}, \; with \; a < n \; .$$

Since $a \mapsto a \cdot r$ is a bijection, the set

$$\{a \cdot r \pmod{n} \mid a \in [0, n-1]\}$$

is a complete residue system.

Given two integers $\bar{a}$ and $\bar{b}$ in n-residue form, the Montgomery product is defined as

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \; ,$$

where r^{-1} is the inverse of r modulo n. Note that the result of the Montgomery product is also in n-residue form:

$$\begin{aligned} \bar{a} \cdot \bar{b} \cdot r^{-1} &\equiv a \cdot r \cdot b \cdot r \cdot r^{-1} \\ &\equiv (a \cdot b) \cdot r \pmod{n} \; . \end{aligned}$$

An additional parameter n' is needed to compute the Montgomery product efficiently. The parameter n' is an integer with the property

$$r \cdot r^{-1} - n \cdot n' = 1 \; .$$

Such an n' and r^{-1} exist, because of the requirement that $gcd(r, n) = 1$. These parameters can be computed using the extended Euclidean algorithm [21]. They are computed once per modulus. The computation of the Montgomery product is described in Algorithm 1.

Algorithm 1: Montgomery Product (MonPro)

Input: $\bar{a}$, $\bar{b}$
Output: $\bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$
1 $t = \bar{a} \cdot \bar{b}$
2 $m = t \cdot n' \pmod{r}$
3 $u = \frac{t + m \cdot n}{r}$
4 **return** $u \bmod n$

To realize how the modulo n operation in step 4 of Algorithm 1 is achieved, it is necessary to have a look at the bounds. From step 1 it follows that

$$t \leq n^2 - 2n + 1 < n^2 - n.$$

Step 2 implies that $m \leq r - 1 < r$. With $r \geq n + 1 > n$, the following upper bound can be given:

$$\begin{aligned} \frac{t + m \cdot n}{r} &< \frac{n^2 - n + rn}{r} \\ &< \frac{n^2 - n + n^2}{n} \\ &= 2n - 1 \end{aligned}$$

The bound implies that at most one subtraction by n is required for the modulo n operation in step 4. Thus, step 4 can be replaced by an if-construct, which determines whether u is greater than or equal to n and subtracts u by n accordingly. The last step is known as the *final subtraction*. This conditional subtraction might not be an issue at first glance, but in fact it offers a side channel leakage (timing attack) [10]. Therefore, there exist techniques to avoid the final subtraction, but we approach the easiest solution by always performing the subtraction and check the sign bit to determine which result to use.

Note that division and reduction modulo r is easy since r is a power 2 number. This is the remarkable feature of the Montgomery product. However, the conversion to n-residue form and the computation of n' is rather time consuming. Thus, the method is not recommendable when a single modular multiplication is to be performed, but if modular exponentiation is performed (as in RSA) it is well suited. Obviously, for modular exponentiation several modular multiplications are performed with respect to the same modulus. The exponentiation algorithm given in Algorithm 2 uses the right-to-left binary method and makes use of the Montgomery product, which is computed by the MonPro function.

Algorithm 2: Montgomery Exponentiation

Input: $a = (a_{l-1}, \ldots, a_0)_2$, $d = (d_{l-1}, \ldots, d_0)_2$,
$n = (n_{l-1}, \ldots, n_0)_2$, $r = 2^l$
Output: $a^d \; mod \; n$
1 $result = 1$
2 $base = \bar{a} = a \cdot r \pmod{n}$
3 **for** $i = 0$ **to** $l - 1$ **do**
4 | **if** $d_i = 1$ **then** $result = MonPro(result, base)$;
5 | $base = MonPro(base, base)$
6 **end**
7 **return** $MonPro(result, 1)$

Step 7 of Algorithm 2 transforms the result in n-residue form, which is equal to $\overline{a^d} \bmod n$, back to an ordinary residue number:

$$\begin{aligned} MonPro(result, 1) &= \overline{a^d} \cdot 1 \cdot r^{-1} \pmod{n} \\ &= a^d \cdot r \cdot r^{-1} \pmod{n} \\ &= a^d \bmod n \; . \end{aligned}$$

Note that step 5 of Algorithm 2 can be precomputed (also in parallel).

A variety of techniques exist to implement the Montgomery product in software. As there are many different platforms with different register sizes, some techniques perform better on specific hardware than others. We refer the reader to the work by Koç et al. [23] for an overview of these techniques. One of these techniques is called CIOS (*Coarsely Integrated Operand Scanning*). Basically, in PRIME, where we want to implement the Montgomery product on 256-bit registers (AVX), we use a modified variant of the CIOS technique.

Algorithm 3 lists the pseudocode of our CIOS variant. The first inner loop (lines 4-8) performs a row-wise multiplication, updating the result in variable t. The rest of the algorithm performs the reduction. Note the alternation between multiplication and reduction. The update in the first inner loop is made on t_j and the value of m is obtained by multiplying t_0 by n'_0 (line 14). This is due to the fact that the result is shifted one word to the right (division by 2^w) in the second j-loop. The variable n'_0 stands for the least significant 256 bits of n'. This specific use of n'_0 in the computation process of m is an observation due to Dusse and Kaliski [11]. The variable t can therefore be split into $k + 2$ words of 256 bits.

Our CIOS variant listed in Algorithm 3 can be transformed to an assembly language program. All variables of this algorithm are 256-bit values ($w = 256$), such that each $regX$ variable can be implemented as a 256-bit wide AVX

Algorithm 3: MonPro CIOS

Input: $\bar{a} = (a_{k-1}, \ldots, a_0)_{2^w}$ and $\bar{b} = (b_{k-1}, \ldots, b_0)_{2^w}$ in n-residue form, $r = 2^{kw}$, $n = (n_{k-1}, \ldots, n_0)_{2^w}$, n'_0
Output: $\bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n}$

```
1  Set t = (t_{k-1}, ... t_0) = 0
2  for i = 0 to k-1 do
3      regC = 0
4      for j = 0 to k-1 do
5          regS = t_j + a_j · b_i + regC (mod 2^w)
6          regC = (t_j + a_j · b_i + regC)/2^w
7          t_j = regS
8      end
9      regS = t_k + regC (mod 2^w)
10     regC = (t_k + regC)/2^w
11     t_k = regS
12     r_{k+1} = regC
13     regC = 0
14     m = t_0 · n'_0 (mod 2^w)
15     regS = t_0 + m · n_0 + regC (mod 2^w)
16     regC = (t_0 + m · n_0 + regC)/2^w
17     for j = 1 to k-1 do
18         regS = t_j + m · n_j + regC (mod 2^w)
19         regC = (t_j + m · n_j + regC)/2^w
20         t_{j-1} = regS
21     end
22     regS = t_k + regC (mod 2^w)
23     regC = (t_k + regC)/2^w
24     t_{k-1} = regS
25     t_k = t_{k+1} + regC
26 end
27 final subtraction
```

register (YMM0 to YMM15). However, 256-bit arithmetics must be achieved by defining macros, which are in fact costly workarounds because processors do not support 256-bit arithmetics even though AVX is supported.

It is worth mentioning that two AVX registers are sufficient to hold the number resulting from

$$t_j + a_j \cdot b_i + regC \pmod{2^w}$$

because the whole term is bounded as follows:

$$\begin{aligned} t_j + a_j \cdot b_i + regC &\leq (2^w - 1) + (2^w - 1)^2 + 2^w - 1 \\ &= 2^{2w} - 1 \end{aligned}$$

Hence, with $w = 256$, two AVX registers are always enough to hold this intermediate result.

Another interesting issue with our CIOS variant is that we can hold non-critical intermediate results in RAM. Variable t of Algorithm 3 and the *result* variable of Algorithm 2 can be held in RAM to save some of the scarce YMM registers. Due to the design of our infrastructure as described in Section 3.2, only one copy of t and *result* resides in RAM. Of course, the question arises whether this approach is secure. Let l be the number of iterations in Algorithm 2 depending on the time when the cold boot attack is performed. An adversary can gain the following value:

$$\overline{a^{d'}} \bmod n, \ where\ d' = d \bmod 2^l$$

As we know, with a big modulus, this function is a one-way function. To obtain d' from it, an adversary would have to solve the discrete logarithm for a composite modulus. Due to Eric Bach's work [4], this problem is proved to be as hard as factorizing the modulus. Specific algorithms like *Pollard's rho* can be used, but they are all not feasible, when the modulus is big. Another approach to obtain d' is by utilizing a *Meet-in-the-Middle* attack that has a complexity of $O(2^{\frac{l}{2}})$ in time and memory. So if l is low, which is the case at the beginning of our algorithm, a fraction of the key can eventually be retrieved. But first of all, this requires very precise timing of the cold boot attack, and second, even if the specific moment in time can be hit, the obtained fraction of the key is too small to pose a risk. The reason is that for any l, only the least significant l bits of the key can be recovered. For example, if we assume that a Meet-in-the-Middle attack is feasible for up to $l = 128$ in a reasonable time, then the least 128 significant bits of the key are exposed to an adversary. However, for RSA-2048 this amount of bits is far away from being critical for the entire key. Due to Boneh, Durfee, and Frankel [8], we know that the critical limit for an RSA-2048 private key is at least 512 bits (or in general, at least one fourth of the key size). Notice that errors in t or *result*, which can occur in the course of a cold boot (decay of bits) result into a complete loss of d' as there are no other key related information in RAM.

When we implemented Algorithm 3, the use of assembly language was necessary to gain full control over CPU registers and to use them in a non-standard manner, because our implementation is not supported by any high-level language compiler.

3.1.2 Performance Methods

The previous section focused on the Montgomery algorithm. Originally, this algorithm was invented to boost the performance of modular exponentiation, but we could additionally use it as a basis for CPU-bound encryption. Besides the Montgomery algorithm, there are other methods to boost the performance of private RSA operations, which we want to explain briefly regarding their applicability for PRIME.

Special *windowing techniques* as shown by Koç [22] perform better than the square-and-multiply method used in Algorithm 2 and can save up to 25% in the number of multiplications. The term *window* refers to the number of bits scanned at a time. This technique, however, involves big precomputed values which must be held in RAM. Another technique involves the *Chinese remainder theorem* (CRT), which is used to boost the performance of modular exponentiation up to a factor of 4. This technique, however, involves 5 private parameters to be protected, which is difficult to handle without main memory.

Using various of these boosting techniques together is possible and has a clear benefit when it comes to performance. However, combining several techniques securely is not a trivial task. Boneh and Brumley [9] exploited information leaked by the combination of boosting techniques used by OpenSSL (which involves Montgomery's method, window techniques, CRT, and the *Karatsuba* multiplication method). Hence, to keep things simple and secure, we decided to only use the plain Montgomery method, as explained in Section 3.1.1. Further techniques like CRT might be an option for future versions of PRIME, but must be carefully chosen with respect to cold boot attacks.

In order to understand our security concerns, we briefly show the procedure of using windowing techniques in combination with CRT, or more precisely Garner's algorithm [12], which is listed in Algorithm 4. Let us assume that due to a cold boot attack, an adversary can retrieve the precomputed values resulting from a windowing technique and addition-

Algorithm 4: Garner's algorithm

Input: Ciphertext message C; p and q are prime numbers with $p > q$ known by the private key owner; d_p,d_q,q^{-1} in steps 1 to 3 are precomputed private parameters
Output: Plaintext message M

1 Compute d_p, such that $d_p e = 1 \, mod \, p - 1$
2 Compute d_q, such that $d_q e = 1 \, mod \, q - 1$
3 Compute $q^{-1} \, mod \, p$, where $p > q$
4 $M_p = C^{d_p} \equiv M^{ed_p} \, mod \, p = M \, mod \, p$
5 $M_q = C^{d_q} \equiv M^{ed_q} \, mod \, q = M \, mod \, q$
6 $h = q^{-1}(M_p - M_q) \, mod \, p$
7 $M = M_q + hq$

ally knows C from Algorithm 4. Providing that the window length is 5, an adversary knows:

1. C
2. $C^x \bmod p$, where $x = (x_4, x_3, x_2, x_1, x_0)_2$

The value of x corresponds to an arbitrary 5-bit windowing value. Since an adversary knows C, he or she can compute all 32 possible values of C^x without modulo reduction. By definition it holds:

$$\begin{aligned} C^x \bmod p &= C^x - kp, \; k > 0 \\ \Longrightarrow C^x - (C^x \bmod p) &= kp \end{aligned}$$

For each of the C^x values an adversary can now compute $gcd(C^x - (C^x \bmod p), n)$, where n is the public modulus. For one of these values, $gcd(C^x - (C^x \bmod p), n)$ reveals the prime number and consequently discloses the private exponent. One of the precomputed values is sufficient to break the system. Hence, using windowing techniques in combination with CRT is insecure in PRIME, where we treat RAM as insecure.

3.2 Linux Kernel Patch

In the previous section, we have shown that it is possible to implement private RSA operations solely on the microprocessor with a variant of Montgomery's method. However, if we implement this variant straightforward in userland, PRIME is *not* secure against cold boot attacks. The reason is that CPU registers are regularly swapped out to RAM due to context switching. As a consequence, we have to implement PRIME inside kernel mode and run it atomically.

3.2.1 Context Switching and Atomicity

It is necessary to run PRIME in kernel mode because problems arise when *context switching* or *swapping* occur. Interrupts, including timing interrupts from scheduling, cause the system to save the CPU context of the running process to RAM and to load the context of a new process into the CPU. This context (in Linux presented as a *process control block (PCB)*) includes all registers and consequently, it includes sensitive values of PRIME. Therefore, PRIME must be run *atomically* in kernel mode. Atomic sections are not allowed in user mode to prevent starvation through unprivileged processes.

To define atomic sections in kernel mode, Linux provides `preempt_enable()` and `preempt_disable()`, for enabling and disabling preemption, respectively. Additionally, hardware interrupts must be disabled right after an atomic section is entered, and be re-enabled only before an atomic section is left. This can be achieved by calling `local_irq_save()` and `local_irq_restore()`. These functions, however, only guarantee atomicity *per CPU* which does not suffice for PRIME. As we use static RAM locations to cache intermediate values of RSA, concurrently running instances of PRIME would mutually falsify their results. Of course, this behavior must be enhanced in future releases of PRIME with engineering efforts, but so far we solved the problem by introducing *global* atomic sections. Global atomic sections do not imply that other CPUs stop working, but that only one CPU can execute PRIME at a time. To disallow several PRIME instances in parallel, we use the mechanism of spin locks. We span the atomicity of PRIME over all CPUs by "spinning" until a lock is free.

3.2.2 Key Management

We are only able to store 256-bit values securely inside debug registers over the entire uptime of a system, similar to symmetric CPU-bound encryption. The reason is that *debug registers* of modern 64-bit CPUs are the only meaningful storage for cryptographic keys. We cannot utilize AVX or general purpose registers as permanent key storage, because those are accessible and required by userland programs. Contrary to that, the user mode in Linux has to ask the kernel via the `ptrace` system call to read or set debug registers. We thus can patch `arch/x86/kernel/ptrace` to prohibit access to them.

Since there are four *breakpoint registers*, 64-bit each, we can store 256 bits inside the CPU. As we want to support RSA-2048, we manage an RSA key in RAM that is encrypted with AES-256. With AES-NI, Intel introduced an efficient instruction set to implement AES-256 inside the CPU. The benefit of AES-NI for us is the possibility to decrypt RSA keys completely within CPU registers (in detail, those are the SSE registers). An RSA private key never has to enter RAM, neither as a whole nor partially, in its unencrypted form. (For more information on how the secret AES key is copied securely into debug registers and an in-depth security analysis of this concept, we refer to the original literature of symmetric CPU-bound encryption [26].)

Interestingly, the performance drawback of PRIME is *not* owed to the extra operations arising from decrypting the RSA key. Implementing AES with AES-NI is so fast compared to RSA that the operation becomes negligible. According to our tests, the performance drawback from the key decryption step is below 1% of the overall performance. Hence, from a performance point of view, using AES-128 would have no advantage so we chose AES-256.

3.3 Architecture and Interface

Figure 2 lists the basic operating principle of PRIME. After a *set-up phase* which is not cold boot resistant, we have an RSA private key symmetrically encrypted on a USB flash drive. In the *initialization phase* during boot, we load this RSA key together with a secret AES key into the kernel. Sysfs is used as an API between user and kernel mode. In the *exponentiation phase*, user data is signed or decrypted.

After loading our kernel module specific sysfs attributes are available in `/sys/kernel/prime/`. We use these attributes to get RSA parameters into the kernel space and to display results to the user mode. The corresponding store and show methods of each attribute organize the data, e.g., writing to the attribute `rsa_message` invokes our exponen-

tiation algorithm. The initializing attributes `rsa_priv_key` and `rsa_aesk` can only be set by a root user. Only the `rsa_message` attribute can be read and written by unprivileged users. Before a message is sent to kernel space, it needs to be transformed to n-residue form. This can be done in user space and is non-critical. We have exemplarily patched the PolarSSL library to do so (see Section 4).

PRIME is safe with respect to concurrent userland threads trying to access the `rsa_message` attribute in parallel. The result of each private RSA operation is put into a data structure that is bound to the PID of a calling process. This approach ensures that a process can only get its own results, and that it cannot falsify or steal other results. The PID of the calling process is queried in this data structure and corresponding results are only made available for the reading process. Upon read, all issued data is removed and cleared from the system space by zeroing and freeing its memory.

- Set-up phase (only once):
  1. Generate RSA parameters p, q, d, e.
  2. Encrypt the binary data of d with AES-256.
  3. Keep the encrypted key on an external drive.
- Initialization phase (on each boot):
  4. Write the encrypted RSA key to attribute `/sys/kernel/prime/rsa_priv_key`.
  5. Write the symmetric AES key to attribute `/sys/kernel/prime/rsa_aesk`.
- Exponentiation phase (for each encryption):
  6. Write message in n-residue form to attribute `/sys/kernel/prime/rsa_message`.
  7. Read decrypted/signed message from attribute `/sys/kernel/prime/rsa_message`.

Figure 2: Basic operating principle of PRIME.

4. EVALUATION

To test and evaluate our implementation, and to prove the applicability of PRIME, we patched the open source library PolarSSL. PolarSSL is comparable to OpenSSL but more lightweight. Based on our PolarSSL patch, we evaluated the usability and security of PRIME in realistic scenarios. To this end, we performed benchmark and security tests while running a Hiawatha web server inside a PRIME-based virtual machine. Hiawatha is a web server based on PolarSSL.

4.1 Usability

In what follows in this section is an analysis of the usability of PRIME regarding its compatibility (Section 4.1.1) and its performance (Section 4.1.2 and Section 4.1.3).

4.1.1 Compatibility

PRIME makes extensive use of AVX in the computing process of a RSA private operation, and it requires AES-NI to decrypt private keys within CPU registers. These requirements are fulfilled by modern CPUs such as Intel Core i5 and i7 processors based on the *Sandy Bridge* microarchitecture. From AMD's side of view, these requirements are fulfilled by processors based on the *Bulldozer* microarchitecture. On older CPUs, PRIME cannot be executed.

On the software-side, we currently support Linux kernels since version 3.0, but our code might be ported to other OSs in future. In the userland, cryptographic libraries must be patched to work together with PRIME. We have shown that for the PolarSSL library as an example.

4.1.2 Private RSA Performance

We now examine the performance of RSA private operations in isolation. We examine the performance of our CPU-bound exponentiation algorithm in comparison to conventional exponentiation algorithms, in particular to those of OpenSSL and PolarSSL. The standard exponentiation algorithm in PolarSSL uses the Montgomery method with CRT and additional windowing techniques (see Section 3.1.2). Since the CRT method might be prone to some attacks, especially side channel attacks [3], PolarSSL is configurable to run without CRT, involving Montgomery's method along with windowing techniques only. Contrary to that, OpenSSL uses Montgomery's method with CRT and windowing techniques but cannot be configured to run without them.

Table 1 shows the average performance of RSA private operations for PRIME, PolarSSL, and OpenSSL. Our implementation is slower by factor 9 in comparison to the best PolarSSL algorithm, and slower by factor 12 in comparison to the OpenSSL implementation. Recall that our implementation does neither use CRT nor windowing techniques for security reasons, and that it must therefore compute a higher number of multiplications. If we take this into account and compare PRIME to the PolarSSL variant without CRT, the performance drawback is about factor 3.

Despite these drawbacks, we think that our implementation is suitable for high security servers with little or medium load, as we show in the next section.

4.1.3 SSL/TLS Performance

We now examine the performance of PRIME in practical scenarios, particularly we measure TLS handshakes within the Hiawatha web server. Our tests were performed on localhost to avoid network noise. Additionally, we disabled optimizing features like caching and keep-alive to get more reliable results. Table 2 shows benchmark values averaged over 1,000 TLS handshakes. Comparing these values with Table 1, we can see that roughly 41 ms of additional overhead are spent for TLS handshakes. As a comparison, normal HTTP connections take about 1 ms. Hence, the performance drawback of PRIME is alleviated when taking the TLS protocol as a whole into account.

Note that the potential target audience of PRIME are high security servers with medium or low load. Nevertheless, we also stressed our PRIME server with maximum load to examine its behavior in extreme situations. Using JMeter, we simulated a scenario where 9,000 users access the server within 60 seconds (i.e., 150 requests per second on average). For this test, we see a significant difference between Hiawatha/CRT and Hiawatha/PRIME. Figure 3 illustrates the response latencies in relation to the elapsed time. As the figure illustrates, the Hiawatha/CRT server needs 73 seconds to respond to all 9,000 requests, i.e., the average response time amounts to 47 ms. The Hiawatha/PRIME server needs almost 3 times longer to process all requests, and the average response time rises up to 225 ms.

The increased performance drawback under high load is no surprise, since the private RSA algorithm of PRIME

Crypto Library / Algorithm	Performance
OpenSSL / CRT	1.8 ms
PolarSSL / CRT	2.4 ms
PolarSSL / No CRT	8.0 ms
PolarSSL / PRIME	21.0 ms

Table 1: Average performance of a private RSA-2048 operation on an Intel i5-2320 CPU.

Web Server / Algorithm	Performance
Hiawatha / CRT	43 ms
Hiawatha / No CRT	50 ms
Hiawatha / PRIME	62 ms

Table 2: Average performance of a TLS handshake with the PolarSSL-based Hiawatha web server.

is accessed in a strictly serial manner. Recall that we do not support multiple PRIME instances in parallel, but that we enforce globally defined atomic sections. Hence, each process running into an RSA private operation pends until it catches the free spin lock. Hiawatha threads do not sleep but "spin" till the lock is available, which results in higher CPU utilization (see Figure 4). Also note the irregular maxima and minima values in Figure 3. Since spin locks do not provide fairness, it might happen that a process constantly fails to catch the lock.

4.2 Security

In this section, we show that PRIME provides the desired security by protecting RSA private keys. For convenience, our security tests are run inside a virtual machine. Running a PRIME guest inside a VM, we can conveniently monitor its RAM contents from outside. In detail, we used QEMU-KVM which is a virtualization infrastructure that lets the Linux kernel act as hypervisor. QEMU-KVM provides a monitor for both looking into RAM contents and into CPU registers (e.g., to check the state of debug and AVX registers). A convenient way to acquire memory dumps is to run the `pmemsave` command in the QEMU-KVM monitor. A memory dump can then be analyzed and searched for private RSA parameters.

4.2.1 Cold-Boot Resistance

To make different RSA implementations compatible with each other, standard representations were introduced. The standard representation of RSA private keys is described in RFC-5958, and the representation of SSL/TLS certificates is described in RFC-5280. Commonly, such representations are standardized in ASN1.1 DER or BER format [18]. What is additionally dealt with are "PEM" files. A PEM file is a Base64-encoded version of an RSA key or certificate in DER format. It is primarily used to provide a safe and portable inclusion in ASCII text. Such encodings leave identifiable patterns of RSA parameters in RAM. This fact is, for instance, used by the tool *rsakeyfind* from Halderman et al. The tool works perfectly on systems running an SSL/TLS Apache server and successfully retrieves BER-encoded RSA keys from RAM.

However, such tools are useless for our evaluation, because PRIME does not use RSA parameters in BER or PEM format. What we did instead was to search for known bytes of the RSA key directly. Unlike real adversaries, we know our key before, and so we can look for exact matches of the key inside memory. We looked for the key and any fraction of it in different representations (e.g., in little and in big endian), and at different system states (e.g., in standby and during swapping), but we never found a significant match of it in RAM. We also revised Linux' process control blocks by hand, to look into the context of SSE and AVX registers, but could not find suspicious bytes.

In 2009, it was shown by Heninger and Shacham [15] that a 2048-bit key can be recovered with 43% of unknown bits, given that e is small. But since we never found any part of the RSA key in RAM, we do not face any threat by recovery methods.

But what is about the AES key that we use to encrypt the RSA key? In PRIME, the AES secret key is basically as critical as the RSA private key. Therefore, we took the same approaches into account to verify that no residues of the AES key remain in RAM as we did for the RSA key. This might be the case, for example, after we initialized PRIME via sysfs. If we do not clean affected memory lines and buffers thoroughly afterwards, parts of the AES secret key would remain in RAM. However, according to our tests, PRIME is secure a few seconds after its initialization phase. During initialization, RAM must be used to load the AES key into debug registers, but afterwards this key cannot be traced anymore.

Last but not least, note that according to the authors of TRESOR [26], CPU registers themselves are not vulnerable to cold boot attacks. Rebooting a machine in order to read out registers (or even by cooling down and unplugging a CPU) is not feasible.

4.2.2 Other Attacks

The designated design goal of PRIME was its resistance against cold boot attacks. We now consider attacks other than cold boot attacks briefly. *DMA-based attacks* [5, 7] and *local privilege escalations*, for example, both enable an attacker to execute arbitrary code in system mode. Obviously, PRIME does not defeat such attacks because system level privileges can be used to read out debug registers, which in turn allows an adversary to decrypt the RSA private key in RAM.

For local attackers without system privileges, side channel attacks are a well-known risk to cryptographic keys. *Cache timing attacks*, for example, rely on the multi-threading capabilities of modern CPUs. For the case of RSA, Percival [29] showed that CPUs with hyper-threading capabilities can leak key related information. His proposed attack uses the fact that processes on hyper-threading CPUs share the same cache. Although no data is shared between threads, they can spy out each other by forcing data out of the cache and using timing differences for memory accesses to decode information. A spy process running in parallel with a crypto process can thus gain valuable information about the key. A similar attack is due to Aciiçmez, Seifert and Koç [1] and uses the branch prediction capabilities of modern CPUs. Again, a shared resource is used as a covert channel. All these approaches require a spy process to simultaneously run on the same CPU as the crypto process. Contrary to that, Yarom and Falkner [32] introduce an attack that utilizes the

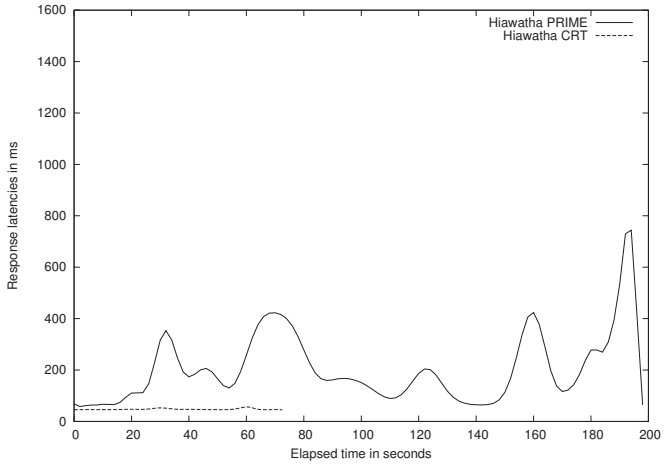

Figure 3: Latencies for PRIME and PolarSSL.

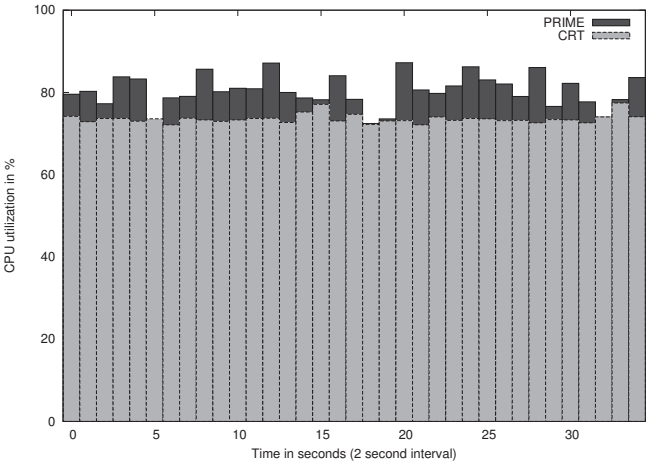

Figure 4: CPU utilization of PRIME vs. PolarSSL.

L3 cache and overcomes the limitation of processes to run on the same CPU.

We can thwart this type of attacks by simply setting the CPU core that runs a PRIME process into *no-fill* mode [19]. In no-fill mode read and write misses access RAM directly and cache lines are never replaced. Recall that a PRIME process additionally runs in an atomic section. Cache timing attacks against AES are thwarted due to the use of AES-NI, which is by design secure against such attacks according to Intel [16] as it runs entirely without memory.

Also note that Algorithm 2 has a branch that is dependent on a bit of the private key, which might lead to a timing attack. A common technique to break the dependency of input and key operations is achieved by *blinding*. The idea is to multiply the payload by a random number before a private RSA operation is applied. Let C be the ciphertext and P be the corresponding plaintext:

Select a random number $b \in [1, n-1]$, and calculate:

$$\begin{aligned} C_1 &= C \cdot b^e \bmod n \\ &= (P \cdot b)^e \bmod n \\ P_1 &= C_1^d \bmod n \\ &= (P \cdot b)^{ed} \bmod n \\ &= P \cdot b \bmod n \\ P &= P_1 \cdot b^{-1} \bmod n \end{aligned}$$

This solution does not have to take place in kernel mode, but can be adopted by any user mode process that calls PRIME, e.g. by PolarSSL, to strengthen the implementation against timing attacks.

5. CONCLUSIONS AND FUTURE WORK

To conclude, with PRIME we presented an infrastructure that protects RSA private keys against cold boot attacks. The main contribution of our work is that we have shown that CPU-bound encryption is possible for asymmetric cryptosystems. This was not clear before and considered as difficult, if possible at all, because of the enormous memory footprint of RSA compared to AES.

With PRIME, we put a focus on making cold boot resistant RSA possible on the latest x86 *standard hardware* without requiring high-end products like SSL accelerators. Our proof of concept went into patching PolarSSL and testing our infrastructure with the Hiawatha web server. Experiments have shown that PRIME is well suitable for servers with low and medium throughput, while being secure and not leaking sensitive data into RAM. This makes PRIME particularly interesting for use cases like SSH servers where throughput is not high, and where asymmetric operations are rarely required. Hence, OpenSSH is a target of interest for us that we want to support in future. Obviously, PRIME will benefit from future advancements like AVX-512, since it provides us with four times more register memory than we have with the current version.

Another plan for future versions of PRIME is a more efficient support for multiprocessing architectures (SMP). Since all CPUs provide their own set of registers, PRIME can theoretically be run on all CPUs in parallel, which would increase its throughput many times over.

Last but not least, we want to support multiple RSA keys in future. At the present, PRIME only supports the usage of a single key. However, this is an unnecessary limitation because the AES master key can be used to hold an encrypted key ring of RSA private keys in RAM.

6. REFERENCES

[1] O. Aciiçmez, c. K. Koç, and J.-P. Seifert. On the Power of simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, Singapore, 2007. ACM.

[2] R. Anderson and M. Kuhn. Tamper Resistance: A Cautionary Note. In *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 1–1, Oakland, CA, 1996. USENIX Association.

[3] C. Arnaud and P.-A. Fouque. Timing attack against protected RSA-CRT implementation used in PolarSSL. In *Proceedings of the 13th international conference on Topics in Cryptology*, CT-RSA'13, pages 18–33, Berlin, Heidelberg, 2013. Springer-Verlag.

[4] E. Bach. Discrete Logarithms and Factoring. Technical

Report UCB/CSD-84-186, EECS Department, University of California, Berkeley, Jun 1984.

[5] M. Becher, M. Dornseif, and C. N. Klein. FireWire - All your memory are belong to us. In *Proceedings of the Annual CanSecWest Applied Security Conference*, Vancouver, British Columbia, Canada, 2005. Laboratory for Dependable Distributed Systems, RWTH Aachen University.

[6] G. R. Blakely. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Trans. Comput.*, 32(5):497–500, May 1983.

[7] E.-O. Blass and W. Robertson. TRESOR-HUNT: Attacking CPU-Bound Encryption. In *2012 Annual Computer Applications Conference*, Orlando, Florida, Dec. 2012. Northeastern University, College of Computer and Information Science, ACSAC 28.

[8] D. Boneh, G. Durfee, and Y. Frankel. An Attack on RSA Given a Small Fraction of the Private Key Bits. In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '98, pages 25–34, London, UK, 1998. Springer-Verlag.

[9] D. Brumley and D. Boneh. Remote Timing Attacks are practical. *Computer Networks*, 48(5):701–716, Aug. 2005.

[10] W. D. Colin. Leakage from Montgomery Multiplication. In c. Koç, editor, *Cryptographic Engineering*, pages 431–449. Springer, 2009.

[11] S. R. Dussé and B. S. Kaliski, Jr. A Cryptographic Library for the motorola DSP56000. In *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques on Advances in Cryptology*, EUROCRYPT '90, pages 230–244, Aarhus, Denmark, 1991. Springer-Verlag.

[12] H. L. Garner. The Residue Number System. In *Western Joint Computer Conference*, IRE-AIEE-ACM '59, pages 146–153, San Francisco, CA, Mar. 1959. ACM.

[13] P. Gutmann. Data Remanence in Semiconductor Devices. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 4–4, Washington, D.C., 2001. USENIX Association.

[14] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryptions Keys. In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60, San Jose, CA, Aug. 2008. Princeton University, USENIX Association.

[15] N. Heninger and H. Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '09, pages 1–17, Santa Barbara, CA, 2009. Springer-Verlag.

[16] Intel. *Intel Advanced Encryption Standard (AES) Instructions Set*, Jan. 2010.

[17] Intel. *Intel Advanced Vector Extensions Programming Reference*. Number 319433-011. Intel Corporation, June 2011.

[18] International Telecommunication Union. Information Technology — ASN.1 Encoding Rules — Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). ITU-T Recommendation X.690, July 2002.

[19] Jürgen Pabel. Frozen Cache. http://frozenchache.blogspot.com/, Jan. 2009.

[20] T. Klein. All Your Private Keys are Belong to Us, 2006. URL: `http://www.trapkit.de/research/sslkeyfinder/keyfinder_v1.0_20060205.pdf`.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1997.

[22] Ç. K. Koç. High-Speed RSA Implementation. Technical report TR201, RSA Data Security, Inc., Nov. 1994.

[23] c. K. Koç, T. Acar, and B. S. Kaliski, Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[24] C. Lomont. Introduction to Intel Advanced Vector Extensions. Intel Corporation, May 2011.

[25] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.

[26] T. Müller, F. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security Symposium*, pages 17–17, San Francisco, California, Aug. 2011. University of Erlangen-Nuremberg, USENIX Association.

[27] T. Müller, B. Taubmann, and F. Freiling. TreVisor: OS-Independent Software-Based Full Disk Encryption Secure Against Main Memory Attacks. In *10th International Conference on Applied Cryptography and Network Security (ACNS '12)*, Singapore, June 2012. University of Erlangen-Nuremberg, Springer-Verlag.

[28] T. P. Parker and S. Xu. A Method for Safekeeping Cryptographic Keys from Memory Disclosure Attacks. In *Proceedings of the First international conference on Trusted Systems*, INTRUST'09, pages 39–59, Beijing, China, Dec. 2009. Springer-Verlag.

[29] C. Percival. Cache Missing for Fun and Profit. In *Proceedings of BSDCan 2005*, May 2005.

[30] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu. TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 36–36, Bellevue, WA, 2012. USENIX Association.

[31] P. Simmons. 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*, ACSAC '11, pages 73–82, Orlando, Florida, 2011. ACM.

[32] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.