xRSA: Construct Larger Bits RSA on Low-Cost Devices

Fan Dang[1], Lingkun Li[2], Jiajie Chen[1]
[1] Tsinghua University, Beijing, China
[2] Michigan State University, East Lansing, USA
dangfan@tsinghua.edu.cn, lilingk1@msu.edu, cjj21@mails.tsinghua.edu.cn

***Abstract*—As the most widely applied public-key cryptographic algorithm, RSA is now integrated into many low-cost devices such as IoT devices. Due to the limited resource, most low-cost devices only ship a 2048-bit multiplier, making the longest supported private key length as 2048 bits. Unfortunately, 2048-bit RSA keys are gradually considered insecure. Utilizing the existing 2048-bit multiplier is challenging because a 4096-bit message cannot be stored in the multiplier. In this paper, we perform a thorough study of RSA and propose a new method that achieves the 4096-bit RSA cryptography with the existing hardware. We use the Montgomery modular multiplication and the Chinese Remainder Theorem to reduce the computational cost and construct the necessary components to compute the RSA private key operation. To further validate the correctness of the method and evaluate its performance, we implement this method on a micro-controller and build a testbed named CanoKey with three commonly used cryptography protocols. The result shows that our method is over 200x faster than the naïve method, *a.k.a.,* software-based big number multiplications.**

***Index Terms*—RSA, Montgomery, low-cost devices**

I. Introduction

Public-key cryptography (PKC) is a part of many security standards and protocols. During the digital transformation, PKC is widely used to provide authentication via electronic signatures or to achieve secure communication across unsecured networks channels [1], [2]. Servers, PCs and high-level consumer electronics usually implement public-key cryptography mechanisms implemented with software at a relatively high speed (*e.g.,* hundreds of signs/s on an Intel i7-8750H CPU), however, such a software-only solution is too slow to satisfy the overall performance of low-performance or low-power devices such as widely applied embedded devices or smart cards. Recent micro-controller units (MCUs), therefore, introduce a public-key accelerator (PKA), an efficient hardware accelerator, to speed up the public-key cryptography operations. Due to the limitation of resources such as RAMs and chip areas, most MCUs and smart cards come equipped with a modular multiplier capable of handling multiplication up to 2048 bits.

With the rapid development of computational technologies, 2048-bit RSA keys are gradually considered insecure. For example, ECRYPT suggests using the private key at least 3072-bit long [3]. More guidelines also encourage or require a 4096-bit private key, *e.g.,* Debian's guide to key creation currently recommends 4096-bit keys [4]; and SSL Labs, one of the most popular SSL testing tools to check all the latest vulnerabilities, requires the 4096-bit private key to get a 100% score when benchmarking the SSL performance [5]. Therefore, there is an urgent requirement to update the key length to 4096 bits. However, replacing all of the existing chips is costly. Instead, it is critical to find an appropriate way to support higher bits of RSA without replacing any existing devices.

For efficiency, modern hardware design usually uses the Montgomery modular multiplier to compute modular multiplication [6]. However, using the existing 2048-bit Montgomery modular multiplier (MMM) to achieve 4096-bit cryptography is not trivial. The key idea of MMM is to transform the number into the Montgomery domain and avoid expensive division operations. Specifically, MMM introduces a constant $R > N$, where N is the modulo and coprime with R. The only division in MMM is a number divided by R, *i.e.,* the multiplication of two numbers a, b in the Montgomery domain is computed as $a \cdot b/R \bmod N$ in the multiplier. In practice, R is the power of two, which makes the division much easier to implement. The Montgomery modular multiplication is detailed in Section II-B. To utilize the 2048-bit MMM, there are several major challenges: 1) The current devices are aiming at achieving 2048-bit RSA cryptography, which only supports 2048-bit operands, *i.e.,* the private key and the message. To make existing devices compatible with 4096-bit operands, we should find an appropriate way to represent and compute 4096-bit operands using the 2048-bit devices. 2) The computation involves two domains, the integer domain and the Montgomery domain. How can we convert a 4096-bit integer domain number to the Montgomery domain with a 2048-bit MMM?

Fortunately, the Chinese Remainder Theorem (CRT) is a powerful tool to reduce computational costs. With the help of CRT, we can use the 2048-bit MMM to compute the medium results of the RSA private-key operation. Finally, we utilize the carefully selected parameters to achieve the 4096-bit operation.

To further validate the correctness of the method and evaluate its performance, we implement this method on a commercial off-the-shelf (COTS) micro-controller, *i.e.,* HED CIU98320B, which equips a 2048-bit MMM. We build a testbed named CanoKey with three commonly used cryptography protocols. The details are elaborated in Section IV-A.

Our major contributions can be summarized as follows:

- We design an algorithm that uses the most existing 2048-bit Montgomery modular multiplier to achieve a 4096-bit RSA cryptography mechanism without replacing any circuit component.
- We implement the 4096-bit RSA cryptography on an existing device, which is equipped with a 2048-bit Montgomery modular multiplier.
- Experiment results show that our method achieves the correct behavior of 4096-bit RSA cryptography, and makes it over 200x faster than the software-based solution.

The rest of this paper is organized as follows. Section II presents the overview of RSA, Montgomery multiplication and the design of a PKA. Section III describes our proposed algorithm in detail. Section IV presents the design of our testbed CanoKey, and analyses the performance of the algorithm. Section V introduces the related work. Section VI concludes the paper.

II. Preliminaries

A. Public Key Infrastructure and RSA

Public Key Infrastructure (PKI) is widely used among information systems. With PKI, users generate a pair of keys, and the one is called the public key that can be published everywhere to encrypt the message, and another is called the private key, which is used to decipher the message, and it is always secret. Different from Symmetric Key Infrastructure (SKI), which only has a unique key for encryption and decryption, users only need to keep private key secret, which reduces the overhead of keeping the message transmission safe.

RSA, named after Rivest, Shamir, and Adleman, was invented in 1977 [7]. It is the most popular public key cryptosystem. Technically, it randomly grabs two prime numbers p and q at the beginning. Then, it choose (N, e) as the public key, and (N, d) as the private key, where $N = pq$, $GCD(LCM(p-1, q-1), e) = 1$, $ed \equiv 1(\bmod \phi(N))$, and GCD is the greatest common divisor, LCM is the least common multiplier, ϕ is Euler's function. After getting the public key, the sender encrypts the message m by $M \equiv m^e(\bmod N)$ before sending it to the receiver, and the receiver decrypts the cipher M by $m \equiv M^d(\bmod N)$.

RSA is secure because currently, there is no polynomial-time algorithm to split number N into p and q. The only way is to go over every integer within $[2, N-1]$ and see if it can be divided by N. Therefore, when N is extremely large, it takes unacceptable time to find p and q. Currently, most hardware fitted 2048-bits of N is gradually becoming unsafe, so we need to find an appropriate calculation method to support larger bits, saying 4096-bits, of N, with 2048-bits hardware.

B. Montgomery Modular Multiplication

As discussed in section II-A, RSA encryption and decryption should perform multiple modular operations, which means getting remainder after division. Division, however, has a high computationally overhead since it requires quotient digit estimation and then using multiplication and subtraction to correct the quotient.

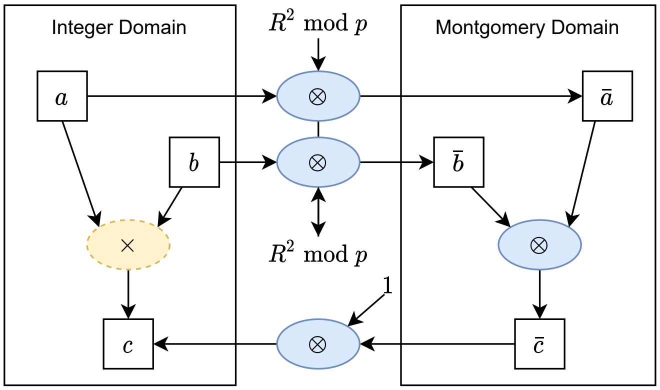

Fig. 1. Compute modular multiplication using the Montgomery product.

Montgomery modular multiplication eliminates division operation by transforming the number to the Montgomery form, further multiplies the number, and performs a certain number of add and subtraction operations, thus reducing the computational overhead. Many IoT devices, therefore, provides hardware implementation of Montgomery modular multiplication to support fast encryption, decryption, or digital signature authentication [8], [9].

Consider a k-bit prime modulus p, and let $R = 2^k$. A number a's Montgomery form is

$$\bar{a} = a \cdot R \bmod p.$$

Assume we have a Montgomery multiplier $\otimes$. It implements the Montgomery multiplication of two numbers, a and b, in Montgomery form. The definition of $a \otimes b$ is

$$a \otimes b = a \cdot b \cdot R^{-1} \bmod p, \tag{1}$$

where R^{-1} is the inverse of R of the modulo p. Given a Montgomery multiplier, $a \cdot b \bmod p$ can be calculated as Fig. 1. Rather than directly computing $a \cdot b \bmod p$, we use four Montgomery modular multiplications instead. The two core operations are $\otimes R^2$ and $\otimes 1$, where the operation $\otimes R^2$ converts a number from the integer domain to the Montgomery domain, and *vice versa*. The correctness is obvious. Note that

$$\begin{aligned}
\bar{a} &= a \otimes R^2 = a \cdot R \bmod p, \\
\bar{b} &= b \otimes R^2 = b \cdot R \bmod p, \\
\bar{c} &= \bar{a} \otimes \bar{b} = a \cdot b \cdot R \bmod p, \\
c &= a \cdot b \bmod p = \bar{c} \otimes 1.
\end{aligned}$$

We implement a Rust-based Montgomery modular multiplication and compare it to an open-source big number library `num-bigint` [10]. The result shown in Table I is based on a single multiplication, which contains four Montgomery modular multiplications. In the real-world practice, saying computing $a^b \bmod p$, we only need to convert a into the

TABLE I
RUNNING TIME OF MODULAR MULTIPLICATIONS

	Intel Xeon E5-2699 v4	Apple M1
`num-bigint`	405 ns	460 ns
Montgomery modular multiplication	380 ns	264 ns

Montgomery domain once, then perform the Montgomery modular multiplication $b-1$ times, and finally convert the result to the integer domain once, which makes the running time significantly less than the naïve multiplications.

C. RSA with CRT

For efficiency, the Chinese Remainder Theorem (CRT) is usually used for reducing the computation. In RSA-CRT, the private key is denoted as $(p, q, d_p, d_q, q_{inv})$, where $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, and $q_{inv} = q^{-1} \bmod p$. Then the private key operation is computed as Algorithm 1, which is four times faster than direct exponentiation.

Algorithm 1 Private-key operation of RSA-CRT.

```
Require: message $m$, private key $(p, q, d_p, d_q, q_{inv})$
Ensure: $m^d \bmod N$
$S_p = m^{d_p} \bmod p$
$S_q = m^{d_q} \bmod q$
$h = q_{inv} \cdot (S_p - S_q) \bmod p$
$S = S_q + h \cdot q \bmod N$
return $S$
```

D. Public-key Accelerator

Fig. 2 demonstrates the block diagram of a typical public-key accelerator (PKA). A PKA contains three major components: the control and status registers, the adder and the multiplier, and the independent RAM. An independent memory called PKA RAM is used for providing initial data to the PKA, and for holding the results after computation is completed. Access is done through the PKA AHB interface. Typically, the PKA RAM is several KiBs (3.5 KiB for STM32L5X2, and 3 KiB for HED CIU98320B). Once the data is loaded to the PKA RAM, the operations are triggered by writing the operation code to the control register. After the calculation is done, an interrupt is triggered, and then the result can be read from the PKA RAM.

III. THE XRSA ALGORITHM

In this section, we propose the method to construct a 4096-bit RSA private key operation using use a 2048-bit Montgomery modular multiplier. This method is based on the RSA-CRT algorithm. To be concise, we introduce $\oplus$ and $\ominus$, and let $a \oplus b = a + b \bmod p$, $a \ominus b = a - b \bmod p$.

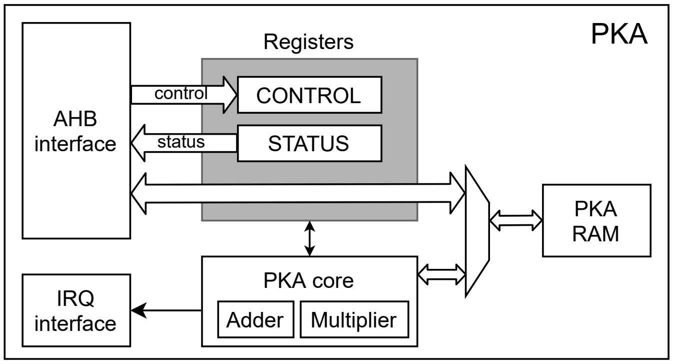

Fig. 2. Block diagram of PKA

A. Compute R^2

As shown in Fig. 1, turning a number from the integer domain to the Montgomery domain is essential before performing the Montgomery modular multiplication. Therefore, we need to compute $R^2 \bmod p$ first. Assume that p is a n-bit prime number, the procedure to compute $R^2 \bmod p$ is shown in Algorithm 2.

Algorithm 2 Compute R^2.

```
Require: a $n$-bit prime number $p$
Ensure: $R^2 \bmod p$, where $R = 2^n$
1: $s = (R-1) \oplus 1$
2: $s_1 = s \oplus s$
3: for $i = 2$; $i \leq n$; $i \leftarrow i+1$ do
4:     $s_i = s_{i-1} \otimes s_{i-1}$
5: end for
6: return $s_n$
```

To see the correctness of Algorithm 2, note that $s_1 = 2 \cdot 2^n \bmod p$, and $s_i = 2^i \cdot 2^n \bmod p$. Besides, Step 1 in Algorithm 2 is necessary because $R = 2^n$ can not be represented in n bits ($R-1$ is the largest number that can be represented in n bits). Obviously, Algorithm 2 requires two modular additions and $n-1$ (or $\lceil \log_2 p \rceil - 1$) Montgomery modular multiplications.

While in the RSA-CRT algorithm, computing R^2 needs to be performed twice, *i.e.,* for p, and q, respectively.

B. Compute $m^{d_p} \bmod p$

Once we have $R^2 \bmod p$ and $R^2 \bmod q$, we are able to compute the components in Algorithm 1. For a 4096-bit RSA key, m is 4096-bit long, while p is 2048-bit long. Therefore, the computation can be performed within the 2048-bit MMM. We only need to deal with the 4096-bit m. Let $R = 2^{2048}$, and $m = m_1 \cdot R + m_2$, where $m_1 = \lfloor m/R \rfloor$ and $m_2 = m \bmod R$. Then we have

$$\begin{aligned} m \bmod p &= (m_1 \cdot R + m_2) \bmod p \\ &= (m_1 \otimes R^2) \oplus m_2 \end{aligned} \tag{2}$$

After computing $m \bmod p$, we can now compute $m^{d_p} \bmod p$ using the fast exponentiation algorithm [11]. In xRSA, we propose a variant of the fast exponentiation algorithm, which is shown as Algorithm 3. The major differences are the additional Steps 3 and 7, which ensure that it runs in a constant time regardless of the input values.

Algorithm 3 A variant of the fast exponentiation algorithm.

```
Require: $\overline{m} = m \bmod p$, and $d_p$
Ensure: $m^{d_p} \bmod p$
 1: $y = 1 \otimes R^2$
 2: $t = \overline{m} \otimes R^2$
 3: for $i = 1$; $i \leq 2048$; $i \leftarrow i + 1$ do
 4:     if the rightmost bit of $d_p$ is 1 then
 5:         $y \leftarrow y \otimes t$
 6:     else
 7:         $dummy \leftarrow y \otimes t$
 8:     end if
 9:     $t \leftarrow t \otimes t$
10:     $d_p \leftarrow d_p >> 1$
11: end for
12: return $y \otimes 1$
```

Algorithm 3 requires 4,099 Montgomery modular multiplications.

C. Private-key Operation

With the above results, we are finally able to finish the private-key operation. Step 3 in Algorithm 1 can be implemented using a modular subtraction and four Montgomery modular multiplications (note that a modular multiplication requires four $\otimes$ operations since we need the result h in the integer domain).

However, the rest of the private-key operation ($S = S_q + h \cdot q \bmod N$) relies on modulo N, a 4096-bit number, which cannot be done directly. Therefore, let us consider $x \cdot y$, where x, y are 2048-bit numbers. Let $x = x_1 \cdot 2^{1024} + x_2$ and $y = y_1 \cdot 2^{1024} + y_2$, where $x_1, x_2, y_1,$ and y_2 are 1024-bit numbers. Then we have $x \cdot y = x_1 y_1 \cdot 2^{2048} + (x_1 y_2 + x_2 y_1) \cdot 2^{1024} + x_2 y_2$. Let HI$(x) = \lfloor x/2^{1024} \rfloor$ and LO$(x) = x \bmod 2^{1024}$, and the result of $x \cdot y$ can be rewritten as Table II, where the header indicates the bits of the result.

TABLE II
THE COMPOSITION OF $x \cdot y$

4096∼3073	3072∼2049	2048∼1023	1024∼1

HI($x_1 y_1$)	LO($x_1 y_1$)
	HI($x_1 y_2$)
	HI($x_2 y_1$)
		LO($x_1 y_2$)
		LO($x_2 y_1$)
		HI($x_2 y_2$)
			LO($x_2 y_2$)

Note the fact that $R^{-1} \equiv 1 \bmod (R-1)$. If we let $R = 2^{2048}$, we have $a \otimes b = a \cdot b \bmod (R-1) = a \cdot b$, for $a, b < 2^{1024}$, since $a \cdot b < R - 1$. Therefore, we are able to use the Montgomery modular multiplier to compute the product of two 2048-bit numbers. The computation requires four multiplications and five additions (additions to compute 1023∼2048 bit may overflow and the carry must be taken into account).

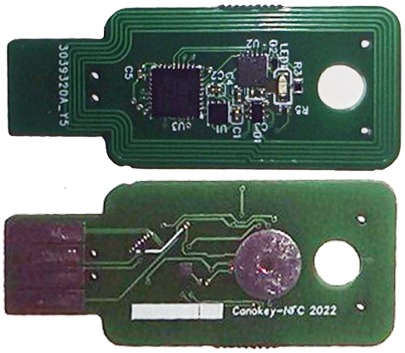

Fig. 3. The testbed CanoKey

To compute the result modulo N, let $M = S_q + h \cdot q$, then we have

$$M \bmod N = \begin{cases} M - N, & \text{if } M \geq N; \\ M, & \text{otherwise.} \end{cases} \quad (3)$$

Proof. The primes p and q in a legal RSA key should satisfy the constraint $p, q \geq \sqrt{2}(2^{\text{nlen}/2-1})$, where nlen is the bit length of N [12]. Therefore, $N = p \cdot q \geq 2^{\text{nlen}-1}$. The maximum possible value of M is $2^{\text{nlen}}-1$, so $M - N < N$. □

D. Complexity Analysis

This section discusses the complexity of the xRSA method. Since the Montgomery modular multiplication takes the most considerable time, we only measure how many multiplications are in the implementation of this method. Note that Steps 1 and 3 in Algorithm 1 share the same modulo p, it would be more reasonable to perform Step 2 first.

The complexity in each step is as follows:

- Step 1 & 2: 2047 $\otimes$ ops for computing $R^2 \bmod q$, 1 $\otimes$ op for computing $m \bmod q$, and 4099 $\otimes$ ops for computing the exponentiation, and 1 additional $\otimes$ op for converting the result to the integer domain, *i.e.,* 6,148 $\otimes$ ops in total for each step.
- Step 3: q_{inv} and S_q needs the conversion to the Montgomery domain, and the result needs the conversion to the integer domain. Therefore, 4 $\otimes$ ops are needed in total.
- Step 4: as shown in Table II, this step requires 4 $\otimes$ ops.

In summary, the xRSA method requires 12,304 Montgomery modular multiplications.

IV. Implementation and Evaluation

To further prove the effectiveness of the proposed xRSA method, we develop a testbed named CanoKey and implement several protocols that use RSA-4096. This section first introduces the details of the platform we use and how we implement CanoKey.

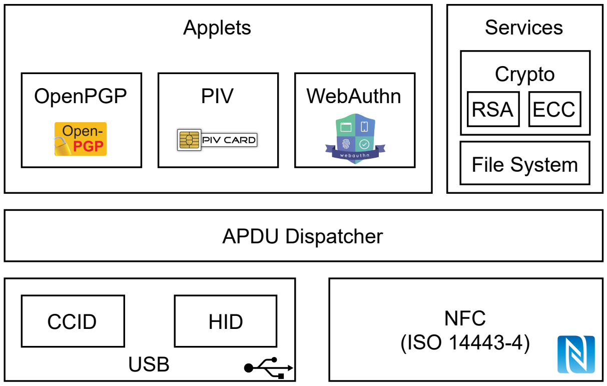

Fig. 4. The architecture of the testbed

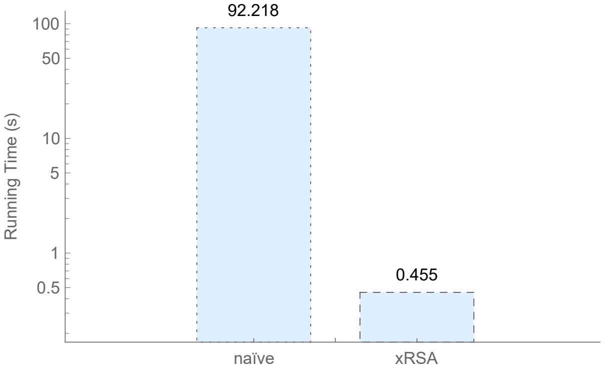

Fig. 5. The running time compared to the naïve multiplication

A. Implementation

We use the HED CIU98320B micro-controller to implement the testbed. Fig. 3 shows our testbed CanoKey. The core of CanoKey is the HED CIU98320B micro-controller. We also use a Fudan FM11NC08 NFC channel chip to support contactless communication. HED CIU98320B uses an ARM SecurCore SC000 32bit-RISC core with a $16\,\mathrm{KiB}$ RAM, a $3\,\mathrm{KiB}$ PKA RAM, and a $320\,\mathrm{KiB}$ flash. The PKA of HED CIU98320B supports up to 2048-bit modular multiplications, and the library provided by the vendor supports the RSA-2048 private key operations at the speed of 37 ops (operations per second).

The PKA of the MCU provides four basic operations: loading data to the PKA RAM, fetching data from the PKA RAM, adding two numbers, and multiplying two numbers. Based on these operations, we implement two RSA operations following the xRSA method:

- `int rsa_get_public_key(rsa_key_t *key, uint8_t *n)`, which computes the public key given a RSA private key, *i.e.,* $p \cdot q$;
- `int rsa_private(rsa_key_t *key, const uint8_t *in, uint8_t *out)`, which performs the private key operation, used in signing or decrypting, *i.e.,* the Algorithm 1.

Here, `rsa_key_t` is the RSA key with the CRT components, *i.e.,* $(e, p, q, d_p, d_q, q_{inv})$.

Fig. 4 shows the architecture of the testbed CanoKey. Based on the segregation of these RSA operations and other necessary cryptographic algorithms (*e.g.,* ECDSA, Ed25519, and SHA-2), we implement three popular cryptographic protocols (applets in Fig. 4), *i.e.,* OpenPGP [13], Personal Identity Verification (PIV, as specified in NIST SP 800-73-4) [14], and WebAuthn [15]. OpenPGP is mostly used for signing and encrypting emails as well as SSH authentication. While PIV and WebAuthn are typically used to identify users with digital signatures. We implement all the mandatory features of the OpenPGP 3.4 and the PIV (NIST SP 800-73-4) specification. The private key and other data are organized in a fail-safe filesystem designed for microcontrollers – `littlefs` [16].

We also implement two kinds of interfaces for testing: USB and NFC. The USB interface supports chip card interface device (CCID) [17] and human interface device (HID) [18] protocols, where OpenPGP and PIV use the CCID protocol while WebAuthn uses the HID protocol. The NFC interface is much simpler. We implement the ISO 14443-4 protocol [19] for transceiving data via NFC. On top of these two interfaces, we encapsulate the command and the response into the application protocol data units (APDUs). APDUs are dispatched to each applet globally. The above implementations have been published open-source at https://github.com/canokeys/canokey-core.

B. Evaluation

This section presents the evaluation result of the xRSA method based on the testbed CanoKey.

1) Raw RSA-4096 performance: We use the SysTick Timer [20] to measure the running time of the xRSA method. The SysTick Timer is a standard 24-bit countdown timer with auto reload. The timer source of the whole system is from the USB, *i.e.,* $48\,\mathrm{MHz}$, with the accuracy of $\pm 0.25\%$. Therefore, in the evaluation, we measure the running time with the resolution of 1 millisecond. Fig. 5 shows the average running time of the xRSA method compared to the naïve method. The naïve method is implemented using Mbed TLS [21], which is a pure software solution based on the big number multiplication. The result shows that our method is 203x faster than the naïve method.

While comparing to the vendor-provided RSA-2048, our method achieves 2.2 ops, 17x slower than RSA-2048.

2) Correctness: To validate the correctness of the implementation, we utilize GnuPG [22] to run automatic tests. Each round of tests contains the following procedures:

- Reset data in the OpenPGP applet;
- Generate three independent keys, *i.e.,* signing, encrypting, and authenticating keys and import them to the OpenPGP applet;
- Sign the current timestamp with the applet and verify the signature on PC;

TABLE III
RUNNING TIME OF SIGNING USING GNUPG

	CanoKey	YubiKey 5 NFC
Average running time	869 ms	670 ms

- Encrypt the current timestamp with the applet and decrypt it on PC;
- Launch an SSH server and use the private key in the applet to authenticate the SSH service.

The test script has been published as part of the testbed too. During the 24-hour test on a laptop running Ubuntu 20.04, the implementation passes every round.

3) The real-world performance: In this section, we discuss the real-world performance in two folds. We first compare our implementation with YubiKey 5 NFC [23]. YubiKey 5 NFC employs an Infineon SLE 78CLUFX5000PH micro-controller, which supports RSA 4096 natively. The conducted evaluation is signing with a 4096-bit RSA private key using GnuPG. Table III shows the average running time of signing an empty string using the `gpg` command. Our method is 29.7% slower than the native RSA-4096 support, indicating that this method can provide a comparable result.

The second evaluation is performed on an Android phone, *i.e.,* a Redmi Note 8 Pro mobile phone with a Helio G90T CPU running Android 10. The average power that NFC can provide is several mWs [24]. NFC also limits the maximum interval between the end of a frame sent by the reader and the start of the response frame from the device. Therefore, the implementation should balance the power consumption and the computation speed. With a carefully selected frequency, *i.e.,* 20 MHz, our testbed passes the signing and decrypting operations on OpenKeychain, an OpenPGP implementation for Android, showing that this method can be applied to a low-power scenario.

V. RELATED WORK

Before FPGA gets popular, researchers tried digital signal processors (DSPs) to build a custom chip for RSA encryption. Kochanski [25] was the first to try this. He implemented 512-bit RSA on Fujitsu's CMOS array, with the running time of 100 ms.

Kaedi *et al.* implements the RSA algorithm on a Spartan 6 LX75 FPGA [26], enabling lower power when using RSA in IoT devices. Dai *et al.* proposes to use the Fast Fourier Transform to help reduce the area and time for implementing the Montgomery multiplication [27]. Verma *et al.* uses a carry-save design for the Montgomery modular multiplier to achieve a better area performance [28]. Bansal *et al.* presents a hybrid Cryptosystem using RSA and Blowfish algorithm that suits the cloud computing environment [29]. Most works focus on the operand size less than or equal to 2048 bits, except for Dai's work [27]. The implementation results of [27] show that supporting a 4096-bit operand costs twice the resources as a 2048-bit operand. Moreover, the look-up tables (LUTs) used in this work are over 14k, which is relatively large for a chip. For comparison, note that a RISC-V CPU core may cost less than 2k LUTs, *e.g.,* PicoRV32 [30].

On the contrary, many researchers focus on implementing a small-sized Montgomery modular multiplier, *e.g.,* less than 1000 bits, for the Elliptic Curve Cryptography (ECC). For example, Abd-Elkade *et al.* improves the radix-2 MMM structure and reduces the critical path delay [31]. Khan *et al.* incorporates the Karatsuba algorithm to improve the efficiency of the Montgomery modular multiplication [32].

VI. CONCLUSIONS AND FUTURE WORK

A. Conclusions

This paper performs a thorough study of RSA and proposes a new method that achieves the 4096-bit RSA cryptography with the existing hardware. We use the Montgomery modular multiplication and the Chinese Remainder Theorem to reduce the computational cost and construct the necessary components to compute the RSA private key operation. To further validate the correctness and evaluate the performance of the method, we implement this method on a micro-controller and build a testbed named CanoKey with three commonly used cryptography protocols. The result shows that our method is over 200x faster than the naïve method, *a.k.a.,* software-based big number multiplications.

B. Future Work

As this work has shown, the xRSA method provides the possibility to achieve larger bits RSA operations on the COTS devices. However, the following questions are left for future research:

- Is it possible to use a 2048-bit MMM to construct even higher, *e.g.,* 7680-bit, RSA operations?
- How do we generate 4096-bit private keys using the 2048-bit MMM? Since generating private keys is also important to real-world applications.

ACKNOWLEDGMENT

We sincerely thank anonymous reviewers for their insightful comments to improve our work. We thank Yuxiang Zhang, Yichuan Gao, Bowen Xia, and Hongren Zheng for their contributions to the testbed CanoKey. This work is partially supported by the National Science Foundation under grant NSF 1919154, and the China Postdoctoral Science Foundation under Grant 2019M650685.

REFERENCES

[1] S.-Y. Chang, Y.-H. Lin, H.-M. Sun, and M.-E. Wu, "Practical RSA Signature Scheme Based on Periodical Rekeying for Wireless Sensor Networks," *ACM Transactions on Sensor Networks*, vol. 8, no. 2, Mar. 2012.

[2] M. Suárez-Albela, P. Fraga-Lamas, and T. M. Fernández-Caramés, "A Practical Evaluation on RSA and ECC-Based Cipher Suites for IoT High-Security Energy-Efficient Fog and Mist Computing Devices," *Sensors*, vol. 18, no. 11, 2018.

[3] N. S. et al., "Ecrypt ii yearly report on algorithms and keysizes (2011–2012)," Network of Excellence, Tech. Rep., 2012.

[4] "keyring.debian.org," https://keyring.debian.org/creating-key.html, (Accessed on 05/2021).

[5] "SSL Server Rating Guide · ssllabs/research Wiki," https://github.com/ssllabs/research/wiki/SSL-Server-Rating-Guide, (Accessed on 05/2021).

[6] P. L. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[7] R. L. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978.

[8] A. A. Abd-Elkader, M. Rashdan, E.-S. A. Hasaneen, and H. F. Hamed, "Advanced implementation of Montgomery Modular Multiplier," *Microelectronics Journal*, vol. 106, p. 104927, 2020.

[9] W. Wang, J. Han, Z. Xie, S. Huang, and X. Zeng, "Cryptographie coprocessor design for IoT sensor nodes," in *2016 International SoC Design Conference (ISOCC)*, 2016, pp. 37–38.

[10] "rust-num/num-bigint: Big integer types for Rust," https://github.com/rust-num/num-bigint, (Accessed on 08/2021).

[11] D. M. Gordon, "A Survey of Fast Exponentiation Methods," *Journal of Algorithms*, vol. 27, no. 1, pp. 129–146, 1998.

[12] P. Gallagher, D. D. Foreword, and C. F. Director, "FIPS PUB 186-3 Federal Information Processing Standards Publication Digital Signature Standard (DSS)," 2009.

[13] "Functional Specification of the OpenPGP applicationon ISO Smart Card Operating Systems," https://gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.4.pdf, (Accessed on 08/2021).

[14] "SP 800-73-4, Interfaces for Personal Identity Verification — CSRC," https://csrc.nist.gov/publications/detail/sp/800-73/4/final, (Accessed on 08/2021).

[15] "Web Authentication: An API for accessing Public Key Credentials - Level 2," https://www.w3.org/TR/webauthn-2/, (Accessed on 08/2021).

[16] "littlefs-project/littlefs: A little fail-safe filesystem designed for microcontrollers," https://github.com/littlefs-project/littlefs, (Accessed on 08/2021).

[17] "USB Integrated Circuit(s) Cards Interface Devices," https://www.usb.org/sites/default/files/DWG_Smart-Card_CCID_Rev110.pdf, (Accessed on 08/2021).

[18] "Human Interface Devices (HID) Information — USB-IF," https://www.usb.org/hid, (Accessed on 08/2021).

[19] "Cards and security devices for personal identification — Contactless proximity objects — Part 4: Transmission protocol," International Organization for Standardization, Geneva, CH, Standard, Jun. 2018.

[31] A. A. H. Abd-Elkader, M. Rashdan, E.-S. A. M. Hasaneen, and H. F. A. Hamed, "FPGA-Based Optimized Design of Montgomery Modular

[20] T. Martin, "Chapter 3 - Cortex-M Architecture," in *The Designer's Guide to the Cortex-M Processor Family (Second Edition)*, second edition ed., T. Martin, Ed. Newnes, 2016, pp. 71–130.

[21] "Mbed TLS - Trusted Firmware," https://www.trustedfirmware.org/projects/mbed-tls/, (Accessed on 08/2021).

[22] "The GNU Privacy Guard," https://gnupg.org/, (Accessed on 08/2021).

[23] "USB-A YubiKey 5 NFC Two Factor Security Key — Yubico," https://www.yubico.com/product/yubikey-5-nfc/, (Accessed on 08/2021).

[24] A. Lazaro, R. Villarino, and D. Girbau, "A Survey of NFC Sensors Based on Energy Harvesting for IoT Applications," *Sensors*, vol. 18, no. 11, 2018.

[25] M. Kochanski, "Developing an RSA Chip," in *Advances in Cryptology — CRYPTO '85 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 350–357.

[26] S. Kaedi, M. A. Doostari, and M. B. Ghaznavi-Ghoushchi, "Low-complexity and differential power analysis (DPA)-resistant two-folded power-aware Rivest–Shamir–Adleman (RSA) security schema implementation for IoT-connected devices," *IET Computers & Digital Techniques*, vol. 12, no. 6, pp. 279–288, 2018.

[27] W. Dai, D. Chen, R. C. Cheung, and C. Koc, "Area-Time Efficient Architecture of FFT-Based Montgomery Multiplication," *IEEE Transactions on Computers*, vol. 66, no. 03, pp. 375–388, March 2017.

[28] R. Verma, M. Dutta, and R. Vig, "FPGA Implementation of RSA based on Carry Save Montgomery Modular Multiplication," in *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, 2016, pp. 107–112.

[29] V. P. Bansal and S. Singh, "A hybrid data encryption technique using RSA and Blowfish for cloud computing on FPGAs," in *2015 2nd International Conference on Recent Advances in Engineering Computational Sciences (RAECS)*, 2015, pp. 1–5.

[30] "cliffordwolf/picorv32: PicoRV32 - A Size-Optimized RISC-V CPU," https://github.com/cliffordwolf/picorv32, (Accessed on 08/2021).
Multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 6, pp. 2137–2141, 2021.

[32] "High-speed FPGA implementation of full-word Montgomery multiplier for ECC applications," *Microprocessors and Microsystems*, vol. 62, pp. 91–101, 2018.