Leakage Resilient Public Key Authentication for Embedded Devices

Andrew Xia
Massachusetts Institute of Technology
axia@mit.edu

Chiraag Juvekar
Massachusetts Institute of Technology
chiraag@mit.edu

Anantha Chandrakasan
Massachusetts Institute of Technology
anantha@mtl.mit.edu

ABSTRACT

Having an effective and convenient authentication system is necessary for the future development and increased usage of devices in the Internet of Things. This paper presents the implementation of a pairings-based, public key leakage resilient authentication system to improve upon current authentication schemes. We have developed a pairings library in C, and we have implemented software in RISCV assembly that successfully implements such authentication scheme on a FPGA, demonstrating the feasibility and efficiency of such primitive.

1 INTRODUCTION

With the increased number of connected devices (sensors, fitness bands, etc), it has become very important to design a authentication platform to allow devices to confirm the authenticity of the other party. Being able to effectively verify the other party can help prevent fraud. An application of using a secure authentication platform is, for example, scanning an item at a supermarket. The user will scan an item, and the item will send a signed message using a key to the user. This signed message will be verified to confirm whether the item is authentic or a fraud.

To define the problem that we are investigating more formally, we have two parties that are interacting with each other. The first party is the *prover*, which attempts to prove its identity to another party, the *verifier*. By following an authentication protocol, the prover and verifier will interact via a series of messages, and at the end the verifier will use the acquired information to determine whether the prover's identity is actually as it claims. In the example above, the prover can be thought of as an item in a supermarket, and the verifier is the user with a smartphone, attempting to confirm that her purchase will be of a genuine item. The detailed explanation of our authentication protocol will be explained in section 3.3.

If our key on the prover is fixed, then side channel attacks in the form of correlation power analysis (CPA) and differential power analysis (DPA) can effectively uncover our private key and render our device insecure (see Figure 1) [4]. For example, by examining the power differences in computing 0s and 1s, one can gain information on the signed message. Timing attacks can similarly determine the logical operations that are computed by a processor if different inputs take different amounts of time through, for example, branches in the assembly instruction. If we design computations that have the same power consumption across all inputs, we can prevent side channel attacks and help achieve *leakage resiliency*. Previous work on developing a leakage resilient authentication scheme; however such scheme uses a symmetric key scheme that achieves leakage resiliency by updating the key periodically using a pseudo random number generator [10].

However, leakage resiliency is not the only feature we desire in an authentication scheme. With a symmetric key scheme, the key used by the device to sign a message is the same as the key that the user needs to verify the device. However, the verifier cannot have direct knowledge of this key —otherwise the verifier impersonate the prover fraudulent activity by pretending to be the device and signing messages. When the prover sends its signed message to the user, the verifier must contact a secure third party database that knows the key. The database will then verify the device and return the result of the verification to the user. Using a third party database provides extra overhead in the authentication scheme that is not desired.

By using a public key scheme, the database containing all public keys would no longer need to be secure.

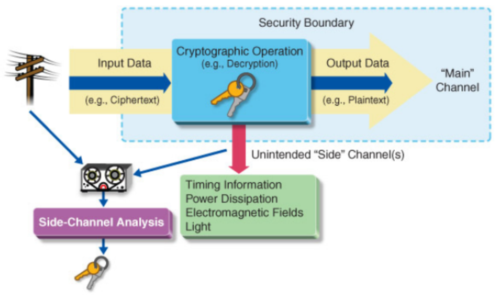

Figure 1: Problems with Side Channel Attacks could potentially compromise our authentication key

Under such a scheme, the prover uses a secret key sk to sign messages, while the verifier uses a public key pk, available to all, to verify the message. However, public key schemes are not necessarily leakage resilient. If the secret key remains fixed, over time there are side channel attacks to uncover the secret key.

In this paper, we will detail the implementation of a public key and leakage resilient authentication scheme. Section 2 will detail previous work on authentication schemes that are only public key or only leakage resilient, and show how this project builds on the previous work. To create a leakage resilient and public key scheme, we will use a pairings-based elliptic curve structure, and section 3 will describe pairings and our three phase authentication protocol. Section 4 will describe our implementation of the pairints library and pairings-based protocols written in Python, C, and for running on a FPGA. Section 5 will show our implementation results. Finally, section 6 will describe areas of future work for this project.

2 RELATED WORK

Currently, the Energy Efficient Circuits & Systems Group at MIT has designed a leakage resilient symmetric key protocol[10]. The design of this authentication protocol uses a Keccak-f[400] permutation to create a pseudo-random number generator (PRNG) that uses a seed agreed upon by both the verifier and the prover. However, because this authentication protocol uses a symmetric key for signing and verifying, we must use a third party database containing all of the keys. This database would require additional security constraints, which adds further vulnerabilities and inefficiencies for practical use cases.

Brakerski, Kalai, Katz, and Vaikuntanathan present a leakage resilient public key encryption scheme in this paper [4]. The idea is to periodically "update" the secret key sk to a new randomized secret key sk' which can still be decrypted by a non-mutating public key pk. The private key update process, along with other operations, have been proven to be leakage resilient. By performing pairings operations in the exponent, obfuscation of the operation is achieved. More details of the correctness of this scheme will be shown in section 3.2.

Previous work on hardware accelerators have focused on improving the multiply-and-add pairings operation by optimizing hardware and software simultaneously [14].

In order to implement the pairings function, a pairings library which would include finite field and elliptic curve operations must be implemented. A Python-based pairings library has been previously implemented Energy Efficient Circuits & Systems Group, which uses optimization techniques described in [12]. The work of this project includes implementing authentication and encryption schemes based on this library. In addition to verifying the correctness of the library in a desktop environment, we have also ported the pairings library to run on an embedded processor with memory constraints.

3 PRELIMINARIES

In this section, we present the mathematical background related to our pairings-based authentication scheme in section 3.2, and we describe our three phase authentication protocol in section 3.3.

3.1 Elliptic Curves

Pairings operations are based on elliptic curves, which is a mathematical structure that has the following property:

Definition 3.1 (Elliptic Curve). An elliptic curve is a plane curve that consists of points satisfying the Weierstrass equation[1]

$$y^2 = x^3 + ax + b$$

One key property of elliptic curves is that all lines that intersect two points on an elliptic curve will cross a third point on the curve. We can define an addition on an elliptic curve as follows: given two points x, and y, find the third point z' that connects the line between x and y. The point $z = x + y$ is defined as the point z' reflected

[1]There are other representations of Elliptic Curves such as projective coordinates

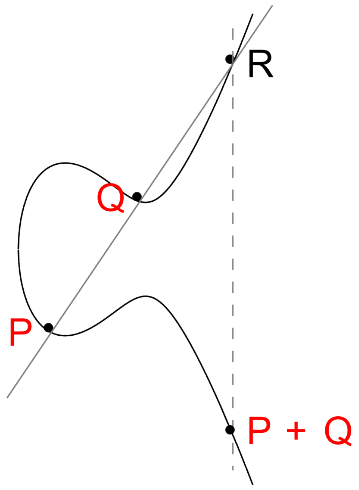

Figure 2: An example calculation on an Elliptic Curve. The operation $P \oplus Q = R$ as the line through P and Q intersects at $-R$.

across the x axis. See figure 2 for an example of a point addition.

In Elliptic Curve Cryptography, the curve E is defined on a finite field of order p. When operating on a finite field, any point on the elliptic curve in the finite field $E/\mathbb{F}_p$ can, through additions and doublings, generate other points in the field. We can define the additive identity as ∞.

The discrete log problem in elliptic curves is hard. More specifically, given a an elliptic curve defined over a prime field, $E/\mathbb{F}_p$, with n elements, the fastest attack on the discrete log problem is the Pollard-Rho algorithm, that takes $O(\sqrt{n})$ time. Thus, a $2n$-bit elliptic curve will have n-bit security. A 256-bit elliptic curve will have 128 bits of security. [2]

In addition to an elliptic curve being defined on a prime field F_p, we can define the *twist curve* of an elliptic curve to be defined on the extension field F_{p^2}.

3.2 Pairings

We define the pairings function below as follows:

Definition 3.2 (Pairings). A pairing is a bilinear map

$$e : G_1 \times G_2 \longrightarrow G_T$$

[2]On the other hand, subexponential algorithms, such as the index calculus attack, exist on finite fields. To also achieve 128 bits of security on a finite field, the order of the finite field would have to be 3072 bits [1].

that satisfies bilinearity, non-degeneracy, and computability.

We will now define the properties of bilinearity, non-degeneracy, and computability.

Let P be the generator of G_1 and Q be the generator of G_2. Let the order of G_1 be p_1 and the order of G_2 to be p_2. A mapping function is *bilinear* if, given any scalar $a \in Z_{p_1}$ and $b \in Z_{p_2}$, the map has the following property:

$$e(aP, bQ) = e(P, Q)^{ab} = e(bP, aQ)$$

In other words, scalars applied to either of the inputs from the two pre-image groups can be alternated.

A mapping function is *non-degenerate* if for all $A \in G_1, A \neq P$ and $B \in G_2, B \neq Q$, the pairing function $e(A, B) \neq e(P, Q)$, where $e(P, Q)$ is the identity element of the target group G_T.

A mapping function is *computable* if there exists an efficient algorithm to compute e. This property is mainly for the practicality of using pairings to perform cryptographic protocols. The Miller Loop was the first efficient implementation of computing a pairings function [11].

For the purposes of our pairings library, we let G_1 be the Elliptic Curve defined over the prime field E/F_p, and G_1 be the twist curve defined on the extension field E/F_{p^2}. We use Barreto-Naehrig curves, such that the target group is an extension field $F_{p^{12}}$, such that the *embedding degree* is defined as $k = 12$ [1]. Specifically, a Barreto-Naehrig curve is an elliptic curve E over a finite field F_p with the elliptic curve having order $r = \#E/F_p$. Furthermore, p and r are both prime, define as

$$p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$$

$$r = 36u^4 + 36u^3 + 18u^2 + 6u + 1$$

for some $u \in \mathbb{Z}$. Our implementation uses $u = -(2^{62} + 2^{55} + 1)$. Our finite field order p is approximately 256 bits, giving the elliptic curve a security level of approximately 128 bits. Similarly, the target group is a multiplicative group of 3072-bits, which according to NIST recommendations is also at the 128-bit security level [1]. The pairings function that we use in our library is the Optimal Ate pairing, which maps elements from the elliptic curve group and the twist curve group to the multiplicative group, $e(E/F_p, E/F_{p^2}) \longrightarrow F_{p^{12}}$.

In the the leakage resilient public key scheme[4], the pairing is used in the decryption process. The scheme is described as follows: our public key is a series of elements in group g^A, where A is a length l vector. Our secret key is g^Y, where Y is a length l vector and each column of Y is in the kernel of A. We can encrypt a message m by mapping each bit m_i of m to g^{v^T}. If m_i is 0, then we set v^T to a linear combination of rows in A, and if m_i is 1 then we set v^T to be a random vector. As our encryption of a single bit is in the form g^{v^T}, and our secret key is g^Y, we use pairings to perform decryption, in which $e(g^{v^T}, g^Y) = e(g,g)^{v^T Y}$. If $e(g,g)^{v^T Y} = e(g,g)^0$ then return 0, otherwise return 1.

While this decryption may seem unnecessarily complicated, the advantage of using pairings lies in the ability to update our secret key without leakage. We update g^Y by selecting a random scalar r, and updating the secret key to $sk' = g^{Yr}$ in a leakge resilient process [4].

Work has been done on implementing the scheme described above, but such scheme requires the number of pairings computations to be asymptotic to the security level [4]. We will look into implementing a leakage resilient public key encryption scheme that requires a constant number of pairings operations.

Computing a pairing is about 4-6 times more expensive than doing scalar multiplication operations in the original elliptic curve [9]. For this project, we first implemented the pairings library and authentication scheme on a RISCV FPGA microcontroller to benchmark our performance. Future work for this project could include developing a hardware accelerator to reduce the computation overhead of a pairings operation. Implementation of the pairings-based library can be done in Python on a computer, on C for the RISC-V processor, and can be built in hardware through BlueSpec Verilog.

3.3 Leakage Resilient, Public Key Primitive

In our authentication scheme, devised by Juvekar and Kalai (paper to appear), we have a 3 step authentication system, as shown in figure 3. The authentication scheme composes of 3 messages sent between the prover and the verifier.

We can define g_1 as the generator for the elliptic curve defined over F_p, and g_2 to be the generator for the twist curve defined over F_{p^2}. For the purposes of notation, let capital letter variables S, A, U represent length-three vectors, and lower case letters and greek letters be a single element.

Before sending the messages, the prover first needs to set up a key generation. The prover selects length-three vectors S, A with each each element in the vector as a random element in Z_p such that $S \cdot A = 0$. We define the vector S as the secret key and the vector A as the private key. The prover will also send the verifier the public key A before beginning the authentication protocol.

The authentication protocol is described below:

(1) **Commitment**. The prover generates a random $\alpha \in Z_p$ and length-three vector U with each element in U as a random element in Z_p. The prover first sends a commitment message consisting of g_1^α on the elliptic curve and g_2^U as three elements on the twist curve to the verifier.
(2) **Challenge**. The verifier generates a random $\alpha' \in Z_p$ and sends it to the prover.
(3) **Response**. The prover now computes $h_1^Y = (g_1^u \cdot g_1^S)^{\alpha+\alpha'}$, which is a length-three vector on the elliptic curve. The prover also computes $h_1^A = (g_1^A)^{\alpha+\alpha'}$. The prover sends h_1^Y and h_1^A to the verifier.

Once the verifier has received the information from the response stage, it will now need to run a verify function to accept or reject the prover's identity. The verifier first computes $h_1 = g_1^{\alpha+\alpha'}$. The verifier will then confirm that the following three properties hold, for the optimal Ate pairing $e(E/F_p, E/F_{p^2}) \to F_{p^{12}}$:

$$\prod_{i=1}^{3} e((h_1^A)_i, (g_2^U)_i) = \prod_{i=1}^{3} e((h_1^Y)_i, (g_2^A)_i)$$

For i = 1,2,3: $e(h_1, (g_2^U)_i) \neq e((h_1^Y)_i, g_2)$

$$\prod_{i=1}^{3} e((h_1^A)_i, g_2) = \prod_{i=1}^{3} e(h_1, (g_2^A)_i)$$

If all three proprties hold, then the verifier knows that the prover is authentic. If any of the three properties do not hold, then the verifier can reject the prover's claim of authenticity. In this scheme, there are eighteen total pairings operations computed on the verifier's side, and no pairings computed on the prover's side.

We can see that this scheme is public key because the secret key here is the vector S and the public key is vector A.

S is updated periodically using a pseudo-random random key generation, similar to the one in [10]. However, for every update of S, the same public key A can still verify the private key. The updating of the private-public key is done in a leakage-resilient manner, and so this three-phase authentication protocol is also leakage resilient.

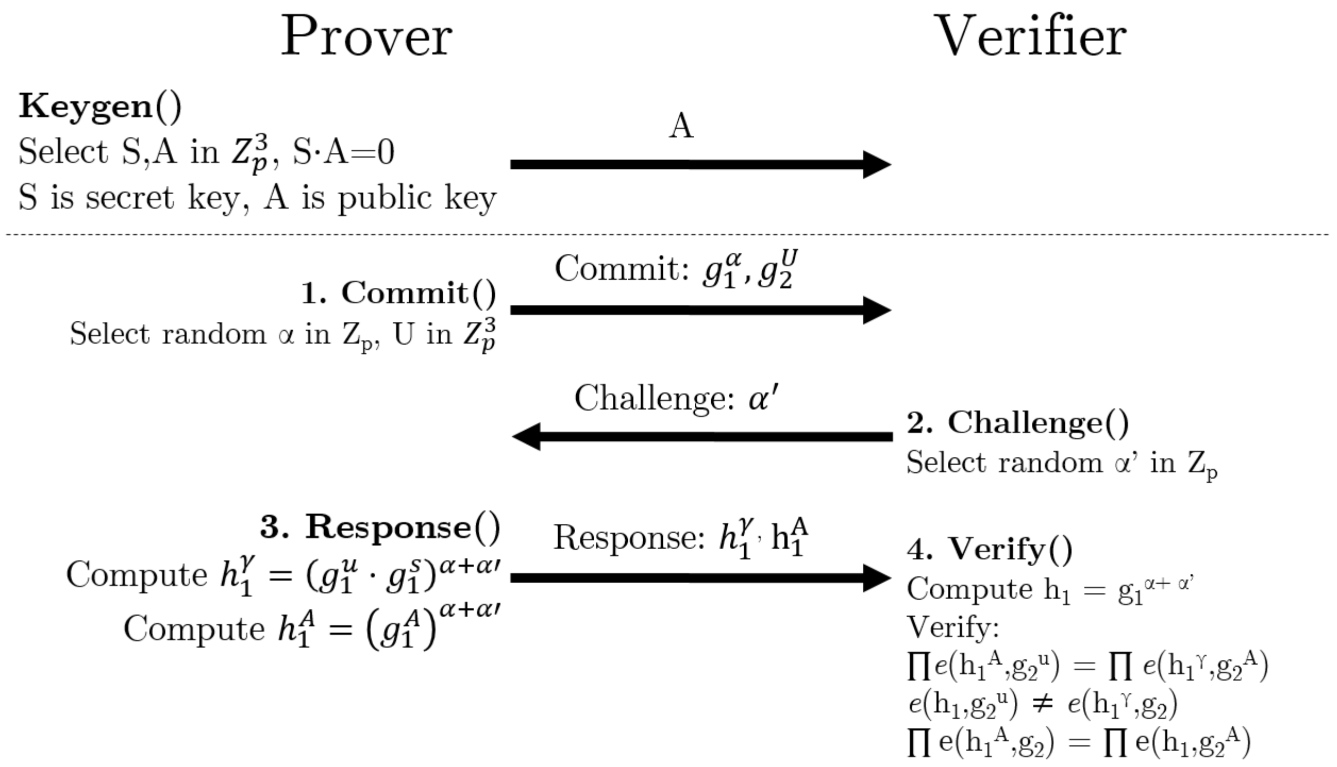
Prover
Verifier
Keygen()
Select S,A in Z_p^3, S·A=0
S is secret key, A is public key
A
1. Commit()
Select random α in Z_p, U in Z_p^3
Commit: g_1^α, g_2^U
Challenge: α'
2. Challenge()
Select random α' in Z_p
3. Response()
Compute $h_1^\gamma = (g_1^u \cdot g_1^s)^{\alpha+\alpha'}$
Compute $h_1^A = (g_1^A)^{\alpha+\alpha'}$
Response: h_1^γ, h_1^A
4. Verify()
Compute $h_1 = g_1^{\alpha+\alpha'}$
Verify:
$\prod e(h_1^A, g_2^u) = \prod e(h_1^\gamma, g_2^A)$
$e(h_1, g_2^u) \neq e(h_1^\gamma, g_2)$
$\prod e(h_1^A, g_2) = \prod e(h_1, g_2^A)$

Figure 3: The three phase authentication protocol. The secret key S is periodically updated. Keygen is run before the 3 phase (and verification) protocol. Keygen provides public key A for the verifier. 18 pairings are computed by the verifier to verify the prover's identity.

3.4 Other Applications

In addition to computing our three-phase authentication protocol by using pairings, other cryptographic primitives can be implemented based on pairings. One application for pairings is the construction of short signatures. Most discrete logarithm signature schemes such as the ElGamal signature scheme are composed of a pair of integers within a group Z_p [7]. However, Boneh, Lynn, and Shacham (BLS) proposed the first signature scheme such that signatures only use a single integer within a group [2].

Another application for pairings lies in Identity-Based Encryption (IBE). In standard public key cryptography, the public key and the private key associated with a user may be a random string of bits that may not be related in any way to the user. However, in IBE we can use unique characteristics of the user, such as her email address or any other unique feature, to create a public-private key pair. Boneh and Franklin were the first to implement an Identity-Based Encryption protocol through the use of pairings [3].

4 IMPLEMENTATION

In this section, we will discuss our implementation of the pairings library. In order to implement the pairings function, we would need to have a library capable of computing the Miller loop to calculate a pairing. For the Miller Loop to function, we would also need to write libraries for Elliptic Curves defined on $\mathbb{F}_p$ and also $\mathbb{F}_{p^2}$, while the output of the pairing function would be an element in $\mathbb{F}_{p^{12}}$. We can build operations in $\mathbb{F}_{p^{12}}$ through $\mathbb{F}_p$, $\mathbb{F}_{p^2}$, and $\mathbb{F}_{p^{12}}$, with $\mathbb{F}_{p^6}$ functions.

Operations in each finite field include addition, subtraction, inversion, negation, multiplication, and scalar multiplication. Operations on the elliptic curve include point addition and point doubling, as well as scalar multiplication. The pairings function is an optimal Ate pairing that composes of computing the Miller Loop and the Final Exponentiation on two inputs, an element in the elliptic curve and one on the twist curve [6].

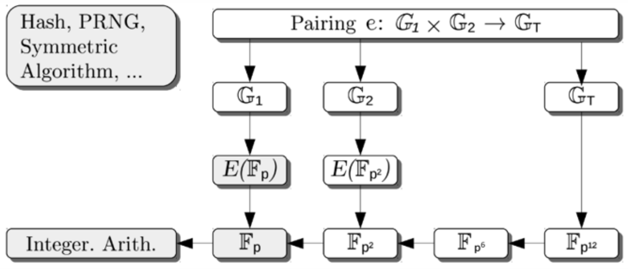

Figure 4: A diagram of our pairings library, showing the dependencies of the mathematical structures [14]

Both the Python and the C library implementations are written using the architecture described in figure 4.

4.1 Python Implementation

Building upon previous work, a pairings library in Python was previously written in the lab. On top of this library, the Boneh-Franklin Identity Based Encryption scheme was implemented and verified for correctness, showing the correctness of encryption on the library [3]. In addition, we also implemented the BLS Signature Scheme, as a demonstration that an authentication scheme could be done [2].

Work was also done to implement the leakage resilient, public key authentication scheme as devised by Juvekar and Kalai, as a proof of concept of correctness. This authentication scheme was shown to work on a desktop setting. However, because we eventually want to deploy the authentication scheme in an embedded environment, the next step would be to port the software to be executable on a RISCV processor implemented on a FPGA. We can achieve this by porting the Python library to C and compiling the library to a RISCV instruction set.

4.2 C Implementation

Because our application target is an embedded device, hardware constraints dictate that our memory consumption and energy consumption must be low. The *C* programming language does not natively support integers larger than 32 bits, while our curve uses numbers on the order of 256 bits. We initially decided to use the GMP library for our implementation of big integers [8]. When the Python Library was initially ported over to C, the memory consumed was too large; memory leakage was causing the whole program to run using over 65MB on the heap. As our prime p is on the order of 256 bits, we wanted a target of memory consumption on the order of Kilobytes. We used Valgrind, which is a programming tool for memory debugging, leak detection, and profiling, to determing where to optimize our pairings library.

First, the codebase was rewritten such that the big integers stored in GMP would not leak memory. After this stage, the total memory allocated on the heap was reduced to 36MB, while the maximum live heap usage was at 20KB. Another aspect of optimization in the library was the elimination of using Montgomery numbers for the pairings library. We were also able to eliminate the usage of certain constant parameters previously used in the Python library. In addition, because the hardware accelerator that we were to use for the pairings computation did not have a divider, functions in the Python library that had used division had to be rewritten. Specifically, the inversion function for finite field elements was previously written using the extended Euclidean algorithm. This fuction was rewritten to use the extended binary Euclidean algorithm instead, which uses right and left shifts to replace the more costly division function. Memory was eventually reduced down to 10MB of total memory allocation, with a maximum live heap usage of approximately 11.7KB.

However, 10MB of total memory allocation would still be way too large for running the compiled RISCV code on an embedded processor. Our target memory constraints for the 32-bit embedded processor involved 64KB of random access memory, and also 64KB of instruction set memory for the binary executable. The GMP library implementation of bigInt stores the numbers on the heap instead of the stack. In a heap, the dynamic memory allocator allocates memory at an address when an object is stored in memory. Following this, the allocator's next memory allocation index is incremented. When objects are cleared in the code, the dynamic memory allocator will not decrement the location for the next heap allocation.

Using a heap prevents the dynamic memory allocator of our embedded processor to function correctly once we exhausted all onboard memory. For example, we can create bigInt a, for which the dynamic memory allocator will assign a hypothetically to memory address $0x8$. Now, if we create bigInt b and c, the dynamic memory allocator will assign b to memory address $0xB$ and c to $0x10$. If we are done with using a, the dynamic memory allocator will clear a off the heap, but when we create d, it will not be inserted at memory address $0x8$, but instead at the dynamic memory allocator's incremented location of $0x14$. We see that the amount of memory remaining is directly related to the number of computations we run, rather than the maximum live memory consumption of the computations, which will eventually cause our dynamic memory allocator to run out of memory.

Our options to reduce this total memory allocation would be to either a) rewrite the memory allocator to more efficiently use memory on the heap, or b) rewrite our bigInt library to store variables on the stack instead of the heap. The latter option was chosen on the grounds of modularity, such that we wanted to make sure our code could run on other embedded environments with other dynamic allocators. See figure 5 to see the decreased memory usage over time.

Thus, we built a custom bigInt library capable of supporting addition, subtraction, multiplication, shifts, comparisons, and modulos. Since our prime is on the order of 252 bits, by using eight 32-bit wide limbs for a total of 256 bits, our bigInt library would be able to fit all arithmetic operations. As negative numbers are used in the binary extended Euclidean algorithm, each bigInt structure is in a two's complement, big endian form. There are certain cases when the extended Euclidean algorithm requires 256 positive bits when computing the inverse on 252 bit finite field, and in this case having a leading 1 in the largest bit can represent a large positive instead of a large negative number.

4.3 Embedded Processor Implementation

In this section, we detail how our authentication scheme was implemented on hardware.

By using a custom-written big integer library, we were able to reduce the maximum allocate stack usage to 10.7KB for a single pairing, which is well under our physical limitation of 64KB.

We compiled our code into a RISCV instruction set using the RISCV gcc compiler. We could simulate the RISCV environment on a intel x86 processor using the spike RISCV architectural simulator [13].

While the memory usage after using the stack-based bigInt library decreased the memory consumption such that the pairings library could be run on an embedded processor, the cycle count of running the pairings operation could be further optimized. To achieve this, inline assembly code was added to the *C* code in the pairings library. For example, in the addition operation of two bigInt integers, if there is a overflow in a limbwise addition the carry bit would be stored in the carry flag. As we are iterating through limbs to compute a addition operation, the *C* code without inline assembly would not know to look for the carry flag. By writing inline assembly code we would be able to save cycles in recomputing the carry flag. Other optimizations by writing inline assembly have also contributed to the decreased runtime of the pairings function.

Further work in the embedded processor environment would include testing the code on the lab's previously developed finite field hardware accelerator. We would also hope to develop a pairings hardware accelerator such that the computation time and energy consumption could be further decreased.

5 RESULTS

The main contribution of this work is in the memory optimization of the pairings library, and the successful demonstration of the implementation of the pairings library in an embedded environment. We used a Barreto-Naehrig curve such that the optimal Ate pairing could be used.

We were also able to demonstrate the successful implementation of BLS Signatures and Identity-Based Encryption (BF-IBE) by using our pairings library on the Python implementation of the pairings library.

We were able to reduce the memory footprint of the program from approximately 65MB to 10.7KB. We were able to test a pairings function in an embedded environment, by compiling our pairings library in C to RISCV assembly code, that could be run on a FPGA simulating a RISCV processor.

6 CONCLUSION

Designing a leakage resilient public key authentication scheme achieves more security and convenience over previously implemented methods. While a symmetric key scheme requires storing the verifying key in a secure database, a public key scheme would not. By using a pairings based encryption scheme, we can update the secret key while being leakage resilient.

In terms of the future work, further optimization in both the authentication protocol design and the implementation can be improved to achieve better performance. Currently, the verifier needs to complete 18 pairings in the authentication protocol. As pairings are 4-6 times more expensive in computation than elliptic curve

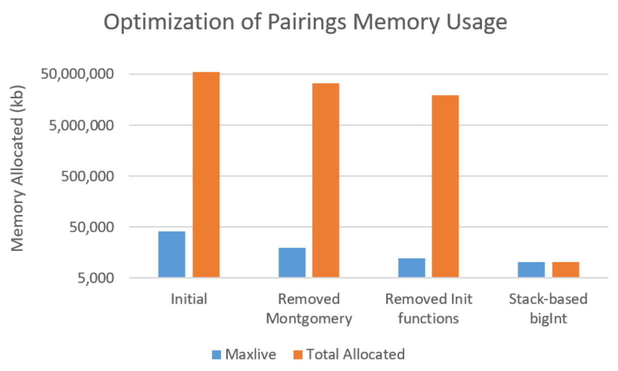

Figure 5: Reducing memory usage through rewriting the pairings based library. In the end, writing a bigInt library built on the stack instead of the heap was necessary to reduce maximum memory consumption. We were able to reduce memory consumption (maximum live / total allocated) from 16MB/65MB using a heap-based bigInt approach using GMP to just 10.7KB/10.7KB of memory with our bigInt library built on the stack

scalar multiplications, improving the protocol by reducing the number of pairings would increase performance. To further improve the computation speed of the authentication protocol, we are also hoping to develop a hardware accelerator for computing pairings in the future.

This scheme will allow for future Internet of Things devices to achieve better security by having authentication scheme with both a public key and leakage-resiliency.

REFERENCES

[1] Barreto, P. S., & Naehrig, M. (2006). Pairing-Friendly Elliptic Curves of Prime Order. *Selected Areas in Cryptography Lecture Notes in Computer Science*, 319-331. doi:10.1007/11693383_22

[2] Boneh, D., Lynn, B., & Shacham, H. (2001). Short Signatures from the Weil Pairing. *Advances in Cryptology - ASIACRYPT 2001 Lecture Notes in Computer Science*, 514-532. doi:10.1007/3-540-45682-1_30

[3] Boneh, D. & Franklin, M., “Identity-based encryption from the Weil pairing", *Advances in Cryptology -CRYPTO 2001*, Lecture Notes in Computer Science, 2139 (2001), 213âĂŞ229. Full version: *SIAM Journal on Computing*, 32 (2003), 586âĂŞ615.

[4] Brakerski, Zvika, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. “Overcoming the Hole in the Bucket: Public-Key Cryptography Resilient to Continual Memory Leakage." *2010 IEEE 51st Annual Symposium on Foundations of Computer Science (2010)*: n. pag. Web.

[5] Costello, C. (n.d.). Pairings for Beginners. Retrieved June 6, 2016, from http://www.craigcostello.com.au/pairings/PairingsForBeginners.pdf

[6] Duquesne, S., N. E. Mrabet, S. Haloui, & F. Rondepierre, “Choosing and generating parameters for low level pairing implementation on BN curves", *Cryptology ePrint Archive, Report 2015/1212*, (2015).

[7] ElGamal, T., “A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory*, 31 (1985), 469âĂŞ472.

[8] Granlund, Torbjörn, et al., "GNU Multiple Precision Arithmetic Library 4.1.2", (December 2002), Web, from https://gmplib.org/

[9] Guillevic, A., & Vergnaud, D. (2015). Algorithms for Outsourcing Pairing Computation. *Smart Card Research and Advanced Applications Lecture Notes in Computer Science*, 193-211. doi:10.1007/978-3-319-16763-3_12

[10] Juvekar, Chiraag S., Hyung-Min Lee, Joyce Kwong, and Anantha P. Chandrakasan. "A Keccak-based Wireless Authentication Tag with Per-query Key Update and Power-glitch Attack Countermeasures." *2016 IEEE International Solid-State Circuits Conference (ISSCC)* (2016): 290-92. Web.

[11] Miller, V. S., “The Weil pairing, and its efficient calculation", *Journal of Cryptology*, 17 (2004), 235âĂŞ261.

[12] Naehrig, M., Niederhagen, R., & Schwabe, P. (2010). New Software Speed Records for Cryptographic Pairings. *Lecture Notes in Computer Science Progress in Cryptology - LATINCRYPT 2010*, 109-123. doi:10.1007/978-3-642-14712-8_7

[13] Nguyen, Q., et. al, RISCV Tools (GNU Toolchain, ISA Simulator, Tests) (2014), GitHub repository, https://github.com/riscv/riscv-tools.

[14] Unterluggauer, T., & Wenger, E. (2014). Efficient Pairings and ECC for Embedded Systems. *Lecture Notes in Computer Science Cryptographic Hardware and Embedded Systems*. CHES 2014, 298-315. doi:10.1007/978-3-662-44709-3_17