Generation of Fast Finite Field Arithmetic for Cortex-M4 with ECDH and SQIsign Applications

Felix Carvalho Rodrigues[1], Décio Gazzoni Filho[1,3], Gora Adj[2], Isaac A. Canales-Martínez[2], Jorge Chávez-Saab[2], Julio López[1], Michael Scott[2] and Francisco Rodríguez-Henríquez[2]

[1] Instituto de Computação, Universidade Estadual de Campinas (UNICAMP), Campinas, Brazil
{felix.rodrigues,decio.gazzoni,jlopez}@ic.unicamp.br
[2] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
{gora.adj,isaac.canales,jorge.saab,michael.scott,francisco.rodriguez}@tii.ae
[3] Department of Electrical Engineering, State University of Londrina, Londrina, Brazil

Abstract. Finite field arithmetic is central to several cryptographic algorithms on embedded devices like the ARM Cortex-M4, particularly for elliptic curve and isogeny-based cryptography. However, rapid algorithm evolution, driven by initiatives such as NIST's post-quantum standardization, might frequently render hand-optimized implementations obsolete. We address this challenge with `m4-modarith`, a library generating C code with inline assembly for the Cortex-M4 that rivals custom-tuned assembly, enabling agile development in this ever-changing landscape. Our generated modular multiplications obtains fast performances, competitive with hand-optimized assembly implementations published in the literature, even outperforming some of them for Curve25519. Two contributions are pivotal to this success. First, we introduce a novel multiplication strategy that matches the memory access complexity of the operand caching method while being applicable to a larger cache size for Cortex-M4 implementations. Second, we generalize an efficient pseudo-Mersenne reduction strategy, and formally prove its correctness and applicability for most primes of cryptographic interest. Our generator allowed agile optimization of SQIsign's NIST PQC Round 2 submission, improving level 1 verification from 123 Mcycles to only 54 Mcycles, a 2.3× speedup. As an additional case study, we use our generator to improve performance of portable implementations of RFC 7748 by up to 2.2×.

Keywords: Finite Field Arithmetic · RFC 7748 · PQC · SQIsign · Cortex-M4.

1 Introduction

The rich expressiveness of elliptic curves has played a key role in cryptography for nearly five decades, showing resilience to evolving security challenges. Today, classical elliptic curve cryptography (ECC) applications are a fundamental backbone of mainstream cryptographic standards, including primitives such as key exchange [39, 40] and digital signatures (ECDSA and EdDSA) [51]. In the 2000s, pairing-based cryptography enabled solutions to long-standing problems such as identity-based encryption, aggregate signatures, and zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARK) protocols, among many others. More recently, over the past two decades, isogeny-based cryptography has emerged as one of the most promising subfields of post-quantum cryptography, thanks to its exceptionally small public key and signature sizes in digital signature schemes, as well as its potential for enabling non-interactive key exchange protocols.

Among the various isogeny-based scheme proposals, two of the most prominent are the Commutative Supersingular Isogeny Diffie-Hellman (CSIDH) scheme and the signature

scheme SQIsign. To offer reasonable security guarantees, it was initially believed that CSIDH required prime field arithmetic with primes of at least 512 bits in length. However, more recent revised estimates now suggest that primes larger than 2048 bits are necessary for sufficient security [19, 18]. SQIsign is the only isogeny-based scheme in the second round of the NIST additional digital signature schemes standardization effort [1]. It achieves NIST security levels 1, 3 and 5 by using primes with a bit size of 251, 383 and 505 bits, respectively, which for efficiency reasons are distinct from the primes used in the round 1 submission. Other recent signing variants, such as AprèsSQI [21] and SQIprime [27], require different sets of primes that are similar in sizes but differ slightly in their shape for their reduction procedure.

The ever-changing landscape of isogeny-friendly primes. Isogeny-based cryptographic schemes rely on carefully selected parameters—most critically, the underlying prime p—to achieve both efficiency and security. This consideration is particularly important in CSIDH and SQIsign (and its variants). These constructions are based on supersingular elliptic curves over finite fields of characteristic p, where the efficiency of isogeny computations is paramount to the overall performance and practicality of the scheme.

Until late 2024 [50, 13], it was common to select p such that $p+1$, or $p-1$, is smooth to facilitate fast isogeny computations. For example, CSIDH adopted primes of the form $p = f \cdot \prod_{i=1}^{n} \ell_i - 1$, where the ℓ_i are distinct odd primes and f is a small power of two. In earlier versions of SQIsign, similar smoothness assumptions were made for both $p+1$ and $p-1$ [56]. Notably, the latest version of SQIsign eliminates this requirement entirely. To achieve a λ-bit security level, it now relies on primes of bitlength approximately 2λ, *without* any particular shape. This way, the NIST round-two version of SQIsign uses primes of the form $p = f \cdot 2^e - 1$, where $e \approx 2\lambda$ and f is a small cofactor.

This illustrates a common trend in isogeny-based cryptography: protocols evolve rapidly, each iteration requiring different specifically tailored primes, in turn rendering hand-tuned field arithmetic implementations obsolete, as parameter changes require costly re-engineering. For resource-constrained ARM Cortex-M4 devices, which powers billions of embedded systems, this creates a critical challenge: maintaining high-performance cryptography across algorithmic iterations without sacrificing development agility.

To address this challenge, we present `m4-modarith`, an open source library that generates constant-time finite field arithmetic code for the Cortex-M4, specific to arbitrarily chosen multiprecision primes, with performance rivaling hand-optimized assembly. By automating low-level optimizations using inline assembly blocks inserted within C code refinement for the Cortex-M4 architecture, our approach decouples performance from prime-specific tuning. Our library supports rapid adaptation to new primes and reduction methods, including pseudo-Mersenne and Montgomery-friendly forms, ensuring relevance across cryptographic standards. Furthermore, all internal procedures were designed with constant-time execution as a top priority, incorporating first-line countermeasures to protect against single-trace side-channel attackers. We demonstrate the efficacy of our library by speeding up RFC 7748 ECDH key exchange (Curve25519, Curve448) and the NIST round-two SQIsign verification procedure [1] by at least a two times acceleration factor.

1.1 Our Contributions

Inspired by `modarith` [57], our `m4-modarith` library generates constant-time finite field arithmetic code for the Cortex-M4, and inherits some countermeasures against power attacks from it. Unlike `modarith`, `m4-modarith` exploits platform-specific ARMv7-EM architecture features, targeting the ARM Cortex-M4 core, which yields highly performant code. The generated code matches the API of `modarith`, facilitating integration and testing. To make this possible, we introduce a novel multi-precision multiplication strategy and an efficient pseudo-Mersenne reduction:

- The core idea behind our multiplication algorithm is to optimize memory access. Our algorithm achieves, for most relevant prime sizes, equivalent memory access complexity to operand caching – a widely used optimization technique for multiplication on embedded devices where intermediate multiplications are strategically divided into rows of size e to improve register usage. While most prior art on Cortex-M4 implementations max out at caching size $e = 3$, our method successfully extends this to $e = 4$, while maintaining compatibility with various prime sizes. This improvement directly translates to reduced memory operations during large-integer arithmetic, which are a bottleneck for non-superscalar cores such as the Cortex-M4.
- We generalize an efficient reduction strategy for pseudo-Mersenne primes using a single generalized carry pass instead of the conventional approach of two carry chains and formally prove its correctness to most cryptographically relevant primes.

These techniques allow our generated modular multiplication and squaring code to have competitive performance compared to publicly-available handwritten assembly code, including outperforming multiple optimized implementations used for Curve25519 [31, 33].

Finally, we showcase the time performance of the generated code in two applications. The first one is the elliptic curve-based shared secret computation specified in the RFC 7748 memorandum [40], where we speedup the portable implementation of `modarith` by up to 2.23×. The second application is the signature verification for the round-two NIST submission of SQIsign. Integrating our generated code results in substantial speedups in finite field arithmetic: multiplication is 2.01× faster and squaring is 2.12× faster than the Cortex-M4 reference implementation [1]. This leads to a notable speedup in the SQIsign verification protocol, with speedups of up to 2.3× compared to the reference.

1.2 Related Work

We present a short summary of related works, both for generating efficient finite field arithmetic and for handmade implementations specific to Cortex-M4 devices.

Generation of finite field arithmetic. Automated code-generation libraries have become useful tools for implementing finite field arithmetic. Fiat-Crypto [42] generates formally verified code using proof assistants like Coq [16], ensuring correctness for its main operations, contingent on compiler trustworthiness, since they do not provide assembly level implementations, opting for backends with high-level languages such as C and Rust, among others. Its focus on formal verification makes it a popular choice for high-assurance applications. A different approach can be seen in RELIC [9], where its primes are defined dynamically, which prevents several possible optimizations.

Recently, the `modarith` library was introduced in [57]. It prioritizes portability by generating portable C and Rust code while maintaining reasonable runtime efficiency through clever implementation of modular arithmetic employing a number of algorithmic tricks like lazy reduction and optimized Montgomery multiplication. This library achieves competitive performance, even when compared to hand-optimized code, for major architectures such as x86. The code generated by `modarith` served as the foundation for improving the performance of 1D SQIsign verification [2], and was later integrated into the round 2 SQIsign reference implementation [1], surpassing the performance of the previous Fiat-Crypto-based backend. While `modarith`'s performance is highly competitive overall [57, §12-13], its portability goal can hinder performance on constrained micro-architectures like the ARM Cortex-M4. On such devices, the use of saturated limbs and access to machine-specific instructions is essential for generating efficient code.

In this work, we build upon the `modarith` framework to create our own API, utilizing its code generation procedures for higher-level functions such as inversion and square-root.

Finite field arithmetic on the Cortex-M4. Prior work has explored the use of `UMAAL` instructions for multi-precision multiplication on the Cortex-M4 (see §2.1). Fujii et

al. [31] combined `UMAAL` with the consecutive operand caching method [58] for 256-bit multiplications, using operand rows of size three for caching (§2.2.1). Their implementation is publicly available[1]. Haase and Labrique [33] employed an irregular operand scanning pattern, specifically optimized for 256-bit operands, with their code also available for comparison[2]. Seo et al. [61] introduced a novel multiplication strategy called *refined operand caching*, which targets primes with more than four hundred bits in length. This method increases the operand row size e to four (instead of the usual three) for multi-precision multiplications on the Cortex-M4, resulting in a slight increase in memory operations compared to the original method. While their paper provides code for primes with larger bitlengths, no code has been released for smaller 256-bit field multiplications. Finally, Emil Lenngren published an optimized implementation of X25519 [44], which uses a non-standard calling convention to reduce function-call overhead for finite-field arithmetic operations. This makes direct comparison with AAPCS32-compliant[3] implementations challenging, although protocol-level comparisons remain fair. The same author recently published a preprint [45] describing an updated version of their X25519 implementation, as well as a new X448 implementation. Both are non-scalable and hand-optimized in assembly, and attain new speed records. We were graciously given access by the author by personal communication to these implementations to use them as examples of optimized assembly RFC 7748 implementations in our benchmarks.

Regarding Curve448, Seo and Azarderakhsh [60] presented an implementation that employs a refined operand caching method for 448-bit multiplications. Anastasova et al. [8] followed a similar overall strategy plus the improvement of other finite field operations, such as squaring and addition. In [6], Anastasova et al. introduced a custom strategy specifically designed for 448-bit multiplications and squaring, setting new speed records for X448 operations. To the best of our knowledge, official implementations for these works have not been made publicly available.

Finally, we highlight the body of work published on isogeny-based cryptography on low-end ARM devices, such as works on CSIDH [64] and its key encapsulation variant, SIKE [61, 7, 5, 48, 59], as well as for SQIsign [2].

1.3 Organization

The remainder of this paper is organized as follows. In §2, we review preliminaries on the Cortex-M4 architecture, finite field arithmetic, and elliptic curves. §3 presents our library, highlighting novel multiplication and reduction strategies, along with micro-benchmarks for modular multiplication and squaring across several primes. In §5, we explore the challenges of applying our library to RFC 7748 and SQIsign, and report the results achieved. Finally, §6 contains conclusions and future directions.

2 Preliminaries

2.1 ARM Cortex-M4 Architecture Overview

The ARM Cortex-M4 is a 32-bit RISC microcontroller core targeting embedded systems adhering to the ARMv7-M architecture. As an in-order, non-superscalar processor, it executes instructions sequentially without speculative execution or parallel pipelines. Its load-store architecture restricts data processing operations to registers, with separate instructions for memory access—a design optimized for low-power applications like IoT devices and industrial control systems.

[1] https://github.com/hayatofujii/curve25519-cortex-m4
[2] https://github.com/BjoernMHaase/fe25519
[3] ARM Architecture Procedure Call Standard for 32-bit architecture (AAPCS32) [11].

The ARM Cortex-M4 core features a 32-bit integer register file with 16 registers, 14 of which are general-purpose. Register scarcity on these devices imposes significant overhead when handling multi-precision integers: frequent load/store operations are required to manage large inputs that exceed register capacity. This makes register-aware algorithm design critical for performance.

For cryptographic workloads, the ARMv7E-M DSP extension provides critical arithmetic acceleration. There are several multiple-accumulate instructions and one of them is `UMAAL` which is particularly impactful. For given registers R_l, R_h, R_n, R_m, `UMAAL` computes $(R_h \,||\, R_l) \leftarrow R_h + R_l + R_n \times R_m$, where $||$ denotes 32-bit register concatenation. These instructions can be executed in a single clock cycle, with same cost as a regular arithmetic instruction such as addition and subtraction, hence greatly improving the efficiency of multi-precision arithmetic.

2.2 Prime Fields and Their Arithmetic

Let p be an odd prime and denote by $\mathbb{F}_p$ the finite field with p element. Elliptic-curve-based cryptography operates with primes that are typically hundreds of bits in size. Hence, the elements in $\mathbb{F}_p$, seen as integers in $[0, p-1]$, are represented using n words or *limbs*. With a radix-2^r representation, an m-bit element $a \in \mathbb{F}_p$ can be represented using $n = \lceil m/r \rceil$ limbs, so that $a = \sum_{i=0}^{n-1} a_i \cdot 2^{ir}$, with each a_i being an r-bit integer.

In a *saturated representation*, the radix is equal to the microprocessor's word length, while in an *unsaturated representation*, the radix is slightly smaller than the word length.

The choice of radix is surprisingly impactful, such as in issues of space and complexity. A saturated representation minimizes limb count and is most advantageous when the modulus size is a multiple of the word length, as is often the case. For example, using a radix of 2^{32} on 32-bit CPUs, a 256-bit modulus requires only 8 limbs. In contrast, unsaturated representation require 9 or more limbs. If one uses a multiplication method with quadratic complexity in the size of the operands, this translates to 8^2 and 9^2 word multiplications, respectively. This is a significant increase of 26% in the number of word-level operations.

Let $a, b \in \mathbb{F}_p$. Multiplication in $\mathbb{F}_p$ is defined as $(a \cdot b) \bmod p$. A customary method for performing this modular multiplication consists of first having the integer multiplication $c = a \cdot b$, followed by the modular reduction step $c \bmod p$. In the remaining of this subsection, we discuss in detail efficient methods for performing these two operations.

2.2.1 Integer Multiplication

Multi-precision multiplication involves computing the product $c = a \cdot b$ of two m-bit integers a and b, each decomposed into n-limb operands. The operation proceeds by calculating partial products $a_i \cdot b_j$, for $0 \leq i, j < n$, and accumulating them into the result c. The simplest approach, the *operand-scanning method* [46], directly implements $c = a \cdot b = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (a_i \cdot b_j) \cdot 2^{(i+j)r}$, iterating row-wise over a_i, multiplying each with all b_j, and storing intermediate values as needed. Although intuitive, this requires n^2 partial products and intensive memory accesses. In contrast, the *product-scanning method* [46], also known as the Comba method, processes partial products in a column-wise fashion. In a machine with large number of registers, this technique can be considerably efficient, with only $2n$ loads and $2n$ stores necessary for a multiplication of n-limb integers iterating row-wise over a_i; multiplying each with all b_j; and accumulating intermediate results. This is straightforward and easy to implement, however requires n^2 partial products and has a significant memory access overhead associated to it.

The Karatsuba multiplication algorithm reduces the integer multiplication quadratic cost of n^2 word multiplications to a superlinear cost of just $\lceil n^{\log_2 3} \rceil$ word multiplications, at the cost of an increment in the number of word addition operations. However, on architectures like ARM Cortex-M4, the `UMAAL` instruction enables 32-bit partial product

accumulation with the same cost as an addition, dramatically reducing the practical advantage of Karatsuba for operands of cryptographic interest.

As summarized below, alternative multiplication strategies, such as hybrid scanning and operand caching, have been developed to optimize register usage and memory access patterns on these embedded devices.

Hybrid scanning method [32]. This method combines operand-scanning and product-scanning to optimize multi-precision arithmetic. It employs nested loops: an outer loop iterates over blocks in a column-wise manner (product-scanning), and the inner loop processes rows within each block (operand-scanning). Let d be the number of rows per block. The memory access complexity is $2\lceil n^2/d \rceil + 2n$, but requires at least $3d+2$ registers for operands and a multi-word accumulator. While this approach reduces redundant loads compared to classical methods, it still reloads operands between blocks and requires a large amount of registers, limiting efficiency for architectures with constrained resources.

Operand caching method [35]. Operand caching is a multiplication strategy that optimizes memory-register trade-offs by partitioning the computation into structured blocks and caching operand limbs to minimize redundant memory accesses. To better understand the method, we describe the multiplication using a rhombus format. Figure 1 exemplifies operand scanning and operand caching strategies using this representation.

In this rhombus representation, each point corresponds to a partial product $a_i \cdot b_j$. The indices i and j increase from the rightmost corner to the leftmost corner, reflecting the progression from least to most significant operand limbs. A black arrow superimposed on a point indicates the sequential processing order of partial products, beginning at the rightmost corner of each block. The final results are stored in limbs $c = (c_0, c_1, \ldots, c_{2n-1})$, ordered such that the least significant limb c_0 resides at the rightmost position, and the most significant c_{2n-1} at the leftmost.

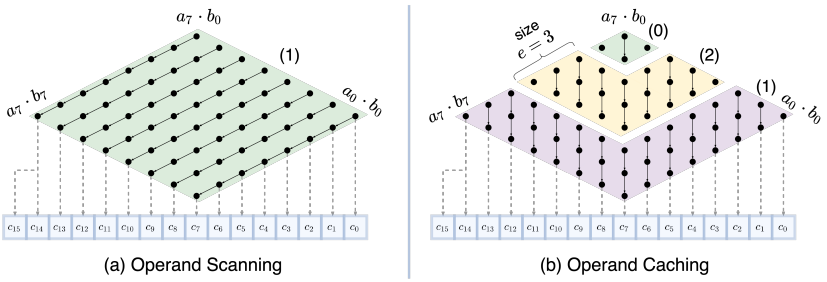

Figure 1: Two multiplication orderings in rhombus format. (a) shows the standard operand scanning method, while (b) illustrates an operand caching strategy with a cache size $e = 3$.

The operand caching method divides the multiplication into $r = \lceil n/e \rceil$ blocks, where e denotes the operand caching size, i.e., the height of each block, defining the number of partial multiplications completed for each column inside a block. The method partitions the computation into an initial block and the subsequent blocks.

The initial block starts the process by computing the upper portion of the multiplication's rhombus structure via classical product-scanning. This block can be omitted if $n \bmod e = 0$, and has dimension $n \bmod e$, otherwise. For each subsequent block, a subset of partial products $a_i \cdot b_j$ is computed column-wise (product-scanning), with operands a_i and b_j cached in registers across multiple iterations within that block. Transitions between blocks require reloading new limbs from the operands.

For the example in Figure 1.b with size $e = 3$, the smaller initial block (0), shown in green, is first computed using traditional product scanning, storing its partial results into c_6, c_7, and c_8. Then, the purple block (1) begins by loading a_0 and b_0, and computing

$c_0 = a_0 \cdot b_0$. The carry-out t from this operation is used in the next column, where a_0 and b_0 are reused, and a_1, b_1 are loaded to compute $c_1 = t + a_1 \cdot b_0 + a_0 \cdot b_1$. The third column then loads a_2, b_2 for the c_3 computation. For the fourth column onward, only b_3 needs to be loaded. This pattern continues for the remaining columns of the block. For columns 6, 7, and 8, the partial results computed in block (0) must be accumulated before final storage, which requires additional load operations. A similar situation occurs in block (2) (shown in yellow), where partial results need to be reloaded: c_4 for its first column computation, c_5 for the second, and so on, until all columns of the block are processed.

With this strategy, the number of memory operations is bounded by $3\lceil n^2/e \rceil + n$ [35]. A further refinement of this method is the *consecutive operand caching method* [62], where blocks are slightly altered for a better interconnection between them, improving register usage. When $n \equiv 0 \pmod{e}$, this method reduces to the operand caching method. The *refined operand caching* [61] is yet another variation which allows, specifically for Cortex-M4 implementations, an increase in size of parameter e from three to four at the cost of additional memory operations. Currently, most speed records on the Cortex-M4 for scalable finite field multiplication use a variation of the operand caching method.

2.2.2 Multi-precision Squaring

While all the multiplication algorithms described earlier remain applicable to squaring, with $a = b$, the symmetry inherent in squaring enables unique optimizations. Specifically, redundant computations arise because partial products involving distinct operand words become duplicates. For example, the term $a_i \cdot b_j + a_j \cdot b_i$, can be simplified to $2(a_i \cdot a_j)$. This redundancy allows certain partial products to be computed once and reused, reducing both computational effort and memory operations compared to multiplication.

Sliding-block-doubling [63]. This method optimizes squaring by deferring the doubling of symmetric partial products until after all intermediate results have been accumulated. By doing so, it replaces explicit additions with more efficient one-bit left shifts, reducing memory operations through an efficient handling of carry propagation and doubling into a single phase. An antecedent of this technique, introducing post-accumulation doubling, was presented in [43]. This was later refined as *sliding-middle-block-doubling* in [65].

2.2.3 Modular Reduction

Given an integer c and a modulus p, the modular reduction of c modulo p computes $(c \bmod p) \in [0, p-1]$. In this work, we focus on reduction techniques that leverage primes of special form, as well as the well-known Montgomery reduction method.

Pseudo-Mersenne primes. Pseudo-Mersenne primes are primes of the form $2^n - k$, where k is typically a small integer. These primes allow for efficient modular reduction, particularly when k fits within a single CPU word. Let $c = ab$, where $0 \leq a, b < 2^n - k$, be a $2n$-bit integer. This product can be decomposed into high and low n-bit components as $c = c_H \cdot 2^n + c_L$. Reduction modulo $p = 2^n - k$ leverages the congruence $2^n \equiv k \pmod{p}$, yielding $c \equiv c_H \cdot k + c_L \pmod{p}$. For small k, this reduces to a linear-time algorithm: multiply c_H by k, add c_L, and iteratively apply the process until the result fits in n bits.

Fujii and Aranha [31] applied this method to the prime of Curve25519, defined as $2^{255} - 19$. Their implementation employs a generalized carry pass to propagate $\lceil \log_2 k \rceil$ bits using the Cortex-M4 `UMAAL` instructions, followed by standard add-with-carry operations and one final propagation from the most significant word to the least significant word.

One issue appears when n is not a multiple of the limb size. For the $2^{255} - 19$ example, word sizes of 32, 64 bits misalign with 255, necessitating multiprecision shifts to isolate c_H. These shifts can be avoided through partial reduction modulo $2p = 2^{256} - 38$, realigning with the word size. To obtain a canonical representative, only the final reduction performs

a conditional subtraction by p – a reasonable trade-off for better performance. Larger word size mismatches, on the other hand, call for per-limb shifts.

Generalized Mersenne or Solinas primes. Generalized Mersenne or Solinas primes [66] are of the form $p = 2^n - 2^a \pm 2^b \pm 2^c \pm \ldots \pm 1$, ideally with few terms (at least three terms if not a Mersenne primes). For best performance, the exponents should be a multiple of the CPU word size, making them more difficult to find.

Efficient reduction procedures use a similar observation as for pseudo-Mersenne primes: $2^n \equiv 2^a \mp 2^b \mp 2^c \mp \ldots \mp 1 \pmod{p}$. However, as one term ($2^n$) is replaced by multiple terms, the reduction expression is more complicated, and may require a series of conditional subtractions to obtain a canonical representative in constant time; this is exemplified in [3] for the NIST P-256 prime. A notable exception are the class of *golden-ratio primes* [34] such as $2^{448} - 2^{224} - 1$, for which there exist formulas of comparable complexity to pseudo-Mersenne primes (see §5.1.1).

Montgomery arithmetic. Peter Montgomery proposed in [49] a method for modular arithmetic that replaces costly (and often variable-time) division operations by multiplications and bitwise shifts. It may be applied to arbitrary moduli without special form, although it admits further optimizations for moduli p such that $p \pm 1$ has a large power-of-two factor.

Fix $R > p$, where usually $R = 2^k$ for some k. We define the *Montgomery representation* of an integer c modulo p as $\overline{c} = cR \bmod p$. Addition (resp. subtraction) of two integers in Montgomery representation can be realized as regular multiprecision operations, ending with a conditional subtraction (resp. addition) by p to ensure a canonical representation; this is easily realized in constant time to deflect timing side-channel attacks.

Multiplication of two integers $\overline{c_1}, \overline{c_2}$ results in $\overline{c_1} \cdot \overline{c_2} \equiv c_1 c_2 R^2 \pmod{p}$, which is not the desired representation. Montgomery's REDC algorithm computes $\mathrm{REDC}(c, p) = cR^{-1} \bmod p$; applied to the previous calculation, we have $\mathrm{REDC}(\overline{c_1} \cdot \overline{c_2}, p) = c_1 c_2 R \bmod p = \overline{c_1 c_2}$, allowing modular operations to be chained. Essentially, REDC adds a carefully chosen multiple of p to its input which renders it divisible by R, so that multiplication by $R^{-1} \pmod{p}$ can be replaced by a division by R (realized using shifts or, if R is chosen as a power of the CPU radix, by repositioning full limbs).

Conversion to and from Montgomery representation can be performed fairly cheaply: the former by multiplying by a precomputed constant $R^2 \bmod p$ followed by application of REDC, and the latter by direct application of REDC.

Although REDC may be stated as an algorithm on multiprecision integers, Montgomery also described a word-by-word variant [49, Section 2] which processes one limb of the input at a time. It is displayed as Algorithm 1.

Algorithm 1 REDC-WBW(c, p)

Require: R, w, n s.t. $R = w^n$ ($\log_2 w$: CPU word length), $\mu = -p^{-1} \bmod w$ (precomputed), $c = (c_{n-1}c_{n-2}\cdots c_1 c_0)_w$, $p = (p_{n-1}p_{n-2}\cdots c_1 c_0)_w$ (base-w representations of c and p)
Ensure: $cR^{-1} \pmod{p}$
1: **for** $i = 0, \ldots, n-1$ **do**
2: $\quad q \leftarrow c_0 \mu \pmod{w}$
3: $\quad c \leftarrow (c + qp)/w$ ▷ Word (q) by bigint (p) multiplication
4: **if** $c \geq p$ **then**
5: $\quad c \leftarrow c - p$
6: **return** c

Further optimizations are admissible if $p \pm 1 \equiv 0 \pmod{2^n}$ for $n > \log_2 w$; such primes are usually called *Montgomery-friendly*. We consider w.l.o.g. the case where $2^n \mid (p+1)$. Faz-Hernández et al. [30] show that multiplication by p in line 3 of Algorithm 1 can be replaced by a multiplication by $p + 1$; although w no longer divides $c + q(p+1)$, one can replace line 3 by $c \leftarrow \lfloor (c + q(p+1)) \rfloor / w$ without changing the result, since $q < w$;

this truncated division is realized by a bitwise shift by $\log_2 w$. Since the least significant $\lfloor n/\log_2 w \rfloor$ limbs of $p+1$ are zero, the corresponding single-word multiplications and additions in the calculation of $c + q(p+1)$ need not be performed.

When choosing R as a power of w, for some primes of interest it holds that $R > 4p$. In that case, it can be shown that one may eschew the final conditional subtraction in REDC (line 5 in Algorithm 1). Although the output of REDC is now in the range of 0 to $2p-1$, the condition $R > 4p$ ensures that the algorithm remains correct even for inputs in that range. However, modifications to the modular addition (resp. subtraction) algorithms are required: rather than conditionally subtracting (resp. adding) p, which might result in continuous growth in the size of the output, one must conditionally subtract (resp. add) $2p$ instead. Also, many protocols require that its output is a canonical representative modulo p; to achieve this, after the conversion from Montgomery to conventional representation is performed (by applying REDC) at the end of the protocol, a single conditional subtraction can be performed at negligible cost.

2.3 Elliptic Curves

An elliptic curve is a smooth, non-singular projective algebraic curve of genus 1, whose points form a group under an addition formula given by rational functions. Classical elliptic curve cryptography is based on the *discrete logarithm problem*: given a point G, a generator of a large subgroup in a curve, and a point $P := [n]G$ (i.e., G added to itself $n-1$ times), find the secret integer n. This problem is the base of a large fraction of public-key cryptography deployed today, but is known to be susceptible to quantum attacks.

Elliptic curves in the Montgomery model can be written as $E: \; y^2 = x^3 + Ax^2 + x$ for $A^2 \neq 4$. These curves support efficient formulas for point addition using x-only arithmetic [22, Sec 2.2] that makes them particularly amenable for computing the left-to-right Montgomery ladder [40, 52, 22], which remains state-of-the-art for computing elliptic-curve Diffie-Hellman key-exchange. Moreover, Montgomery curves are used by virtually all the isogeny-based cryptosystems proposed to date [19, 24, 18, 25].

2.4 Notation for our Selection of Primes

Although our library supports arbitrary prime sizes, for the purpose of showcasing its performance, we focus on a representative set of primes commonly used in elliptic-curve-based and isogeny-based cryptographic schemes:

- p_{25519} and p_{448} are the 255-bit pseudo-Mersenne prime $2^{255} - 19$ and the 448-bit Goldilocks generalized Mersenne prime $2^{448} - 2^{224} - 1$, respectively, both endorsed in the RFC 7748 memorandum [41].
- p_{266}, p_{336}, p_{383} and p_{512} are pseudo-Mersenne primes defined as, $2^{266} - 3$, $2^{336} - 3$, $2^{383} - 187$ and $2^{512} - 569$, respectively [37, 10, 17].
- p_{521} is the 521-bit NIST-recommended Mersenne prime, also known as P-521 and secp521r1, given by $2^{521} - 1$ [51].
- p_{1973}, p_{47441} and p_{318233} are the 254-, 378- and 502-bit primes used in round-one SQIsign to meet NIST security levels I, III and V, respectively [20, Chapter 5].
- p_{5248}, p_{65376} and p_{27500} are the primes defined as $5 \cdot 2^{248} - 1$, $65 \cdot 2^{376} - 1$ and $27 \cdot 2^{500} - 1$, respectively. These primes are used to achieve NIST security levels I, III and V in the round-two SQIsign signature scheme [1, Chapter 5].

3 Efficient Finite Field Arithmetic for the Cortex-M4

In this section we present our library `m4-modarith`[4], a specialized arithmetic library optimized for ARM Cortex-M4 based on Michael Scott's `modarith` framework [57]. The remainder of this section is organized as follows. In §3.1, we describe some of the main design choices behind our library. In §3.2, we present our novel multiplication and squaring strategies. In §3.3, we detail our efforts to achieve efficient modular reduction, both for pseudo-Mersenne primes and for general primes using Montgomery reduction.

3.1 Library Design

The Cortex-M4 architecture presents characteristics and constraints that strongly motivate the development of specialized implementations for finite field arithmetic. One key factor is the larger performance gap between optimized specialized implementations and portable ones on the Cortex-M4 compared to other architectures. This discrepancy arises due to the specialized instruction set of the architecture, including efficient multiply-accumulate operations and other features difficult to fully exploit through high-level code alone.

`m4-modarith` focuses on generating efficient finite field arithmetic implementations for Cortex-M4 devices with applicability to other devices with access to `UMAAL` instructions. It prioritizes critical finite field operations (particularly addition, subtraction, multiplication, and squaring) for ECC and isogeny-based post-quantum cryptographic schemes such as SQIsign. Unlike general-purpose libraries (e.g., RELIC [9]) that accommodate dynamically defined primes, `m4-modarith` operates exclusively with fixed prime moduli specified during generation. This constraint enables deep architectural optimizations tailored to the modulus structure, a strategy impractical for runtime-configurable primes. The library was tested for prime numbers of size 225 up to 2048 bits.

The core of the library resides in our code generating script, where a given prime and reduction method are used to generate constant-time C code with inline assembly blocks tailored for Cortex-M4 devices. Its generated code adheres to the `modarith` API, simplifying validation and integration. The library is capable of generating code for any given prime through two main reduction methods: pseudo-Mersenne prime reduction for primes with such shape, such as $p_{25519} = 2^{255} - 19$, and using Montgomery reduction for general primes. We also employ optimizations for Montgomery-friendly primes when applicable.

Radix representation. On Cortex-M4 CPUs, the typical drawbacks of full-radix systems—such as serialization—are less pronounced due to the in-order, non-superscalar pipeline. Additionally, efficient instructions like `UMAAL` simplify partial product accumulation, reducing the overhead of using a saturated representation.

Considering the above, in our library, integers are represented as n-limb integers in full-radix $2^r = 2^{32}$, aligning with the CPU's 32-bit word size. To address the challenges posed by the saturated representation, we employ inline assembly to handle overflow efficiently. This approach allows us to fully exploit the hardware capabilities of the Cortex-M4, overcoming the limitations of high-level languages in dealing with these issues.

Coding language. The decision to build our arithmetic using mainly C code with inline assembly in selected areas has both down and upsides. One of the major disadvantages relates to relying on the compiler register allocation strategies, which limits some of the possible strategies for our generator.

A key advantage of this approach is its ease of integration into existing software. By relying on a higher-level language, modifications and adaptations become simpler. Additionally, avoiding full assembly code allows the compiler to optimize inlining strategies

[4]The library `m4-modarith` is available at https://github.com/Crypto-TII/m4-modarith.

more effectively. For instance, the compiler can share variables across functions as they are inlined, bypassing explicit reloads required in assembly code. On the Cortex-M4's in-order, non-superscalar pipeline, every assembly instruction incurs at least one cycle of overhead, making such optimizations significant. Finally, while performance can vary with compiler versions and flag selections, this flexibility is generally beneficial. Newer compilers often introduce optimizations that enhance efficiency, offsetting potential inconsistencies.

Testing methodology. Towards promoting correctness, our library employs a comprehensive testing methodology. We implement an extensive test suite covering all supplied functions, which evaluates random inputs and edge cases. Additionally, since we use the library to implement RFC7748, as is detailed in §5.1.1, we further use RFC7748 test vectors to ensure correctness for this implementation. The library also has the capability of testing across a diverse set of over 100 prime moduli ranging from 256 to 2048 bits. Finally, an automated bound checking via interval arithmetic is integrated during the code generation that uses our new pseudo-Mersenne reduction method shown in §3.3.1. While not constituting formal verification, this multi-layered approach provides high engineering confidence in our generator's correctness.

3.2 Multiplication Strategy

Efficient multiplication and squaring are fundamental to many cryptographic algorithms and high-precision arithmetic routines. On the Cortex-M4, optimizing these operations is key to achieving high performance. While all approaches based on the traditional schoolbook algorithm require n^2 word multiplications, their efficiency largely depends on the number of memory operations needed to carry out the computation.

In a standard implementation of the operand caching strategy on the Cortex-M4, the cache size e is typically limited to three due to register constraints. To extend this to $e = 4$, a variant called *refined operand caching* was introduced in [61]. However, achieving this requires loading additional operands from the stack at specific points, which increases the total number of memory operations. The main contribution of our multiplication strategy is that, by combining elements of the hybrid method with operand caching, we enable the use of blocks with size $e = 4$ on these devices without incurring extra memory operations, when the number of limbs is divisible by e.

A novel hybrid operand caching multiplication. Our approach developed into a different path than the traditional operand caching in §2.2.1, where blocks are defined for and computed by a product-scanning method. Instead, for n-limb inputs, our algorithm divides the process into $\lceil n/e \rceil$ blocks of width e and length n, each implemented as a single large inline assembly command, using an operand scanning method within the block. With this approach, we were able to fix $e = 4$ for Cortex-M4 devices, maximizing register usage.

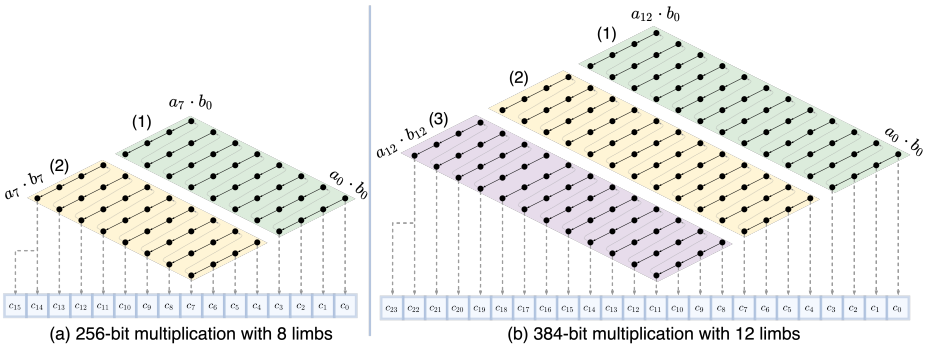

Figure 2: Block Multiplication Strategy for 256-bit and 384-bit operands on Cortex-M4.

Each block performs a portion of the full multiplication, and the partial results are combined to produce the final product. Figure 2 illustrates this strategy for 256-bit and 384-bit operands, using the same rhombus format as described in §2.2, using 2 and 3 blocks, respectively. Note how our strategy also differs from a traditional hybrid scanning method as multiple blocks are joined into a single larger block, greatly improving register usage. Other approaches such as refined and consecutive operand caching were considered; however, to achieve peak efficiency for this approach, registers must be preserved between blocks, which the compiler does not guarantee between blocks of inline assembly.

To illustrate our register allocation strategy, we first examine a typical line within a block, as shown in Algorithm 2. In this line, registers are used to load operands, perform partial multiplications via `UMAAL`, and store intermediate results into c and temporary registers $t0$ through $t4$.

In Algorithm 2, registers are renamed to variables $t0$–$t4$ and t, $b0$–$b3$, b, and c for clarity; i, j, and k denote precomputed address offsets, which do not require register usage.

Algorithm 2 LINE-MUL(`a, b0-b3, t0-t4, i, j, k, c`)

Require: Operand a, registers $b0, b1, b2, b3$ containing 4 limbs of operand b, registers $t0, t1, t2, t3$ containing previous partial results, with i, j, k as address offsets.

Ensure: For $t = a_i$, compute partial products $t \cdot b0, t \cdot b1, t \cdot b2$ and $t \cdot b3$, with c_k and $t1, t2, t3, t4$ storing its results for the next line.

```
1: ldr t, [a, #i]                          ▷ Load line limb from a into t
2: umaal t0, t4, t, b0                     ▷ Perform all line multiply-accumulate operations
3: umaal t1, t4, t, b1
4: umaal t2, t4, t, b2
5: umaal t3, t4, t, b3
6: ldr t, [c, #j]                          ▷ Reuse t for loading partial result, not done on first block
7: str t0, [c, #k]                         ▷ Store partial result from t0 into c
8: mov t0, #0                              ▷ Clear t0 for usage in the next line
9: adcs t4, t4, t                          ▷ Merge partial results, not done on first block
```

Each execution of Algorithm 2 computes four partial products, with the carry-out limb stored in $t4$. The first partial product ($t0$) is not reused by other calls to LINE-MUL within the same block computation and so is immediately stored in c, while the remaining partial products are kept in the temporary registers $t0$–$t3$. This approach avoids the need to reload partial products between successive LINE-MUL calls. To achieve this, the temporary registers $t0, \ldots, t4$ are rotated in the next invocation of LINE-MUL to $t1, \ldots, t4, t0$. This eliminates the need for additional registers for the next lines, allowing a block computation to be carried out using only ten registers for variable storage (five for operands and five for partial results $t0$–$t4$), with three additional registers managing the memory addresses of a, b, and c. Moreover, except in the lines corresponding to the first block, each line must load and accumulate partial results with a carry addition (lines 6 and 9), continuing until all partial results have been processed.

Algorithm 3 displays the method for a full block computation. $t0, \ldots, t3$ are zeroed if they are the initial line for the first block. Otherwise, they are initially loaded with partial results from the previous block, with subsequent lines using the (conceptual) rotation of line 7 to avoid extra loads. Algorithm 2 is run n times for each block. At the end of the block computation, $t0, \ldots, t4$ are stored into c so that it either has the final results (if it's the final block), or we can later reuse the partial results in the next block. Note that as stated previously BLOCK-MUL is implemented as a single inline assembly block, with all calls for LINE-MUL inlined together and no actual loop or register reassigment taking place.

Memory operations. To count memory operations required by our method, we must consider all loads and stores for operands as well as partial results, for each block.

In a block computation, shown in Algorithm 3, first e limbs from operand b are loaded

Algorithm 3 BLOCK-MUL(`u, n, a, b, c`)

Require: Operands a and b, size n and u as the block index, c starts zeroed for $u = 0$.

Ensure: Compute $c' = \sum_{i=0}^{n} \sum_{j=4u}^{4u+4} a_i b_j \cdot 2^{32(i+j)}$.

1: `b0, b1, b2, b3` $\leftarrow b_{4u}, b_{4u+1}, b_{4u+2}, b_{4u+3}$ ▷ Load b limbs
2: `t0, t1, t2, t3, t4` $\leftarrow c_{4u}, c_{4u+1}, c_{4u+2}, c_{4u+3}, 0$ ▷ Load partial results
3: **for** $0 \leq i < n$ **do**
4: $j \leftarrow 4u + 4 + i$ ▷ Calculate address offsets (precomputed)
5: $k \leftarrow 4u + i$
6: LINE-MUL(a,`b0-b3,t0-t4`$, i, j, k, c$)
7: `t0, t1, t2, t3, t4` $\leftarrow$ `t1, t2, t3, t4, t0` ▷ Rotate `t0-t4` registers
8: $c_{k+1}, c_{k+2}, c_{k+3}, c_{k+4} \leftarrow$ `t0, t1, t2, t3` ▷ Store final partial results into c

a single time at the beginning of the block (line 1), and then for each use of LINE-MUL, a new limb from operand a must be loaded. For all blocks except the first one, n additional loads of partial results must also be counted, giving a total of $e + 2n$ for these blocks, and $e + n$ for the first block. In terms of stores, it is easy to see that a block will perform $e + n$ stores, either as partial values or for its final result. If $e \mid n$, there are n/e blocks, resulting in a total of $(n/e)(2(e + n)) + ((n/e) - 1)n = 3n^2/e + n$, the same as operand caching.

Table 1 summarizes memory access counts for various integer multiplication strategies, with concrete examples for comparison. The parameters used in these comparisons are set to $e = 4$ for our method and for refined operand caching, and $e = 3$ for standard operand caching. Note that in the case of refined operand caching, this corresponds to $e = 3$ as defined in the original paper, since the authors describe their block size as $e + 1$.

Memory access counts for our method assume that the bitlengths of the operands are multiples of 128. For other operand sizes, slight modifications to the construction are necessary, as discussed in Remark 1. We show that when $n \bmod e = 0$, our method outperforms other strategies in practice with respect to the number of memory operations. When $n \bmod e \neq 0$, we describe below how our approach can be adapted accordingly.

Table 1: Comparison of memory operations for different multiplication strategies, assuming no reuse of variables between columns or lines for both product and operand scanning.

Method	Block size (e)	Count formula	Total for m-bit a, b		
			256	384	512
Product Scanning	–	$2n(n + 1)$	144	312	544
Operand Scanning	–	$3n^2 + 2n$	208	456	800
Operand Caching [35]	3	$3\lceil n^2/e \rceil + n$	74	156	274
Refined Operand Caching [61]	4	$3\lceil n^2/e \rceil + 3\lfloor n/e \rfloor + n$	62	129	220
Hybrid Operand Caching (this work)	4	$3(n^2/e) + n$	56	120	208

Remark 1 (Case $n \mod e \neq 0$). Although several primes of interest are represented with a number of limbs that is multiple of four, e.g., $p_{25519} = 2^{255} - 19$ and the SQIsign primes, others such as $p_{448} = 2^{448} - 2^{224} - 1$ are not. For these cases, we construct a first block of width $n \mod e$ and length n. For this smaller block, we maintain the same overall strategy as for a full sized one. In terms of memory operation costs, it adds a full block worth of operations $(2(e+n)+n)$ minus $2(e-(n \bmod e))$ loads and stores that represent the unused limbs, resulting in a total amount of operations of $\lceil n/e \rceil (2e + 3n) - n - 2(e - (n \bmod e))$. This means that for such sizes, performance may be slightly worse than traditional operand caching with $e = 4$, but crucially still better than the more commonly used operand caching with $e = 3$. For example, consider multiplication modulo p_{448}: using standard operand caching with $e = 3$, the total number of memory operations is 210. In comparison, our

method requires only 182 operations, while operand caching with $e = 4$ achieves 161.

Remark 2 (Cortex-M4 memory optimizations). An often overlooked aspect of large integer multiplication implementations, particularly on in-order, single-issue cores such as the Cortex-M4, is the varying cost of memory operations. As briefly noted in ARM's technical reference manual,[5] load instructions may be pipelined with other memory operations. Particularly, our experiments showed that sequences of load instructions followed by a store operation rather than an arithmetic one, are less likely to incur pipeline stalls. This suggests that a strategy involving more memory operations may be preferable if those operations can be scheduled more efficiently. In our approach, we can interleave loads from different line multiplications with stores to minimize isolated load instructions.

3.2.1 Squaring Strategy

Squaring, as a special case of multiplication, can be further optimized. A first option is adapting the multiplication routine to reduce register usage, as both input operands are the same. However, this does not exploit the inherent symmetry of squaring; concretely, for an operand $a = (a_n, \ldots, a_0)$, the calculations $a_i \cdot a_j$ and $a_j \cdot a_i$ for $i \neq j$ can be performed only once (say for $i < j$), doubled and finally added to the a_i^2 computation. This results in $n(n-1)/2$ word multiplications, significantly better than a full multiplication.

Our main goal is to improve from a naive squaring implementation while reusing most building blocks from multiplication for better maintainability. For this, all $a_i \cdot a_j$ operations are divided into blocks of height $e = 4$ and decreasing width starting with n. We can see these blocks as incomplete versions of the multiplication case, with our generator handling their construction in the same general manner. When all these operations are computed, a final integration block performs the final doubling of partial results and calculation of a_i^2.

Figure 3 shows this method for 9- and 12-limb squaring. The red block represents the final integration block. Each triangle depicts $n(n-1)/2$ word multiplications. Note also that integration block contains n dots corresponding to a_i^2 for $i = 0, \ldots, n-1$.

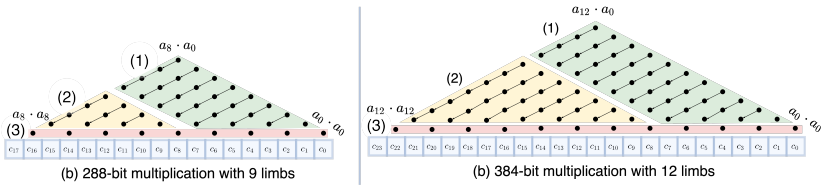

Figure 3: Our squaring strategy for 288 and 384-bit operands.

Note that the triangular block does not follow the standard height of $e = 4$. For example, a naive version of our strategy might divide a 14 limb squaring such as the one for p_{448} into three blocks with height 4 and a small block of size one before the final integration block. Instead, we divide into two full blocks and a base triangle block of size five, minimizing the number of needed blocks. This is possible due to the shape of this triangular block, which allows us to use minimal register spilling for these special cases. We generate a block for each base triangle with sizes ranging from 4 to 7. Figure 4 shows examples of how blocks are arranged for different operand sizes.

Based on the code written for the base triangle of size 7, we notice that without the pressure of loading previous partial results, we are able to generate a more efficient squaring that handles the full 256-bit case with minimal register spilling, incorporating the integration block and avoiding most partial results reloading.

[5] https://tinyurl.com/ARM-Doc-Instructions.

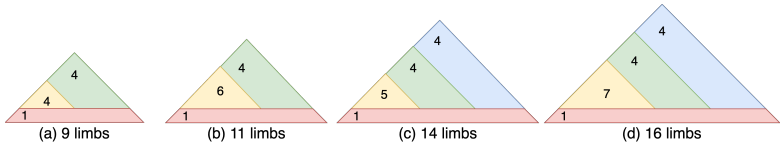

Figure 4: Difference in block allocation for various n-limb squarings.

3.3 Modular Reduction

Our library `m4-modarith` focuses on two modular reduction methods: pseudo-Mersenne-prime-specific reduction techniques, that takes of advantage of the shape of these primes for efficient computation, and the Montgomery reduction for arbitrary primes, exploiting the Montgomery-friendly optimizations described in §2.2.3 when possible. `m4-modarith` uses the latter for Solinas primes, which have some degree of Montgomery-friendliness. However, for one particular case (the prime p_{448}), we have hand-written code that demonstrates the performance benefits of Solinas reduction, at least for such "golden ratio" primes, although it is unclear if this technique would outperform Montgomery reduction for Solinas primes with less favorable properties.

3.3.1 An Efficient Reduction Strategy for Pseudo-Mersenne Primes

In the method described in §2.2.3 for pseudo-Mersenne primes $p = 2^n - k$, we see that there are two complete carry chains, a general one using `UMAAL` instructions, and a second with add-with-carry operations for 1-bit carries. Since these operations have the same cost for the Cortex-M4, if it were possible to preempt the final carry chain and include it into the first carry pass, it would be possible to save roughly half the computational overhead.

We demonstrate that this is possible for a large number of pseudo-Mersenne primes. Essentially there are two methods, one for $d = n \bmod 32 > 16$ and another for $d = n \bmod 32 \leq 16$. The maximum allowed size for k for these methods to work is also dependent on the gap $32 - d$. We note that a very similar approach was done specifically for the Curve25519 prime p_{25519} by Thomas Pornin [54], and by Emil Lengrenn [44] in his implementation of X25519; our contribution is to generalize this to other primes, lay out the conditions for this technique to work, and prove it is correct if conditions are met.

When $d = n \bmod 32 > 16$, i.e., the gap is smaller than 16 bits, the method is shown in Algorithm 4. For this case, we first compute and properly shift the last "folded" limb of c, so that it can be separated into a d-bit low part, which is then added to the first limb of the result and propagated using the generalized carry pass, with the high part being added to the last limb of the result. As Theorem 1 shows, this can be accomplished only when it remains some space in the last limb, specifically when $k \cdot 2^{32-d} < 2^{d-1}$.

Algorithm 4 MODRED(n, k, c)

Require: $n > 32$, $k > 0$ and $0 < c < 2^{2(n+1)}$, with $d = n \bmod 32$ and $k \cdot 2^{32-d} < 2^{d-1}$
Ensure: $(c \bmod p)$ or $(c \bmod p) + p$, with $p = 2^n - k$
1: $l \leftarrow \lceil \frac{n}{32} \rceil$
2: $(c_0, \ldots, c_{2l-1}) \leftarrow c$, with $0 < c_i < 2^{32}$ for $0 \leq i < 2l$
3: $t \leftarrow c_{l-1} + k2^{32-d} \cdot c_{2l-1}$
4: $v_L \leftarrow t \pmod{2^d}$, $v_H \leftarrow \lfloor t/2^d \rfloor$
5: $(r_0^H | r_0^L) \leftarrow c_0 + k2^{32-d} c_l + kv_H$
6: $r_0 \leftarrow r_0^L$
7: **for** $1 \leq i < l - 1$ **do**
8: $\quad (r_i^H | r_i^L) \leftarrow r_{i-1}^H + c_i + k2^{32-d} c_{i+l}$
9: $\quad r_i \leftarrow r_i^L$
10: $r_{l-1} \leftarrow r_{l-2}^H + v_L$
11: **return** $(r_0, \ldots, r_{l-1})$

When $0 < d \leq 16$, the algorithm is similar, with the only difference being that we must start the preemption process earlier, at the penultimate limb. The corresponding

algorithm (with $k \cdot 2^{32-d} < 2^{31}$) and its proof are shown in Appendix B. For cases where neither algorithm applies, we revert to the traditional method, involving two carry passes.

Theorem 1. *Given an integer $p = 2^n - k$ with $d = n \bmod 32$ and $k \cdot 2^{32-d} < 2^{d-1}$, let $c = a \cdot b < 2^{2(n+1)}$ be the product of two integers smaller than $2p$. Then Algorithm 4 reduces c modulo p while maintaining the upper bound of $2p$.*

Proof. Note first that $n = 32(l-1) + d$. Then, rewriting c as

$$c = \sum_{0 \le i < l} (c_i + 2^{32l} c_{i+l}) 2^{32i},$$

and using $2^n = 2^{32l-(32-d)} \equiv k \pmod{p}$, we obtain a first reduction of c modulo p:

$$r' = \sum_{0 \le i < l} (c_i + k 2^{32-d} c_{i+l}) 2^{32i}.$$

Now, define $s = \sum_{0 \le i < l-1} (c_i + k 2^{32-d} c_{i+l}) 2^{32i}$ and $t = c_{l-1} + k 2^{32-d} \cdot c_{2l-1}$ so that

$$r' = s + t 2^{32(l-1)}.$$

Setting $v_L = t \pmod{2^d}$ and $v_H = \lfloor t/2^d \rfloor$ then implies

$$t 2^{32(l-1)} = v_L 2^{32(l-1)} + 2^d v_H 2^{32(l-1)} = v_L 2^{32(l-1)} + 2^n v_H.$$

Thus we can reduce further c using again $2^n \equiv k \pmod{p}$:

$$r = s + v_L 2^{32(l-1)} + k v_H.$$

To show that $0 \le r < 2p$, we first note that $v_L 2^{32(l-1)} < 2^d 2^{32(l-1)} = 2^n$. Then, for $0 \le i < l$, we have $c_i + k 2^{32-d} c_{i+l} \le (2^{32} - 1) + (2^{d-1} - 1)(2^{32} - 1) = 2^{d-1}(2^{32} - 1)$. Hence

$$s \le 2^{32(l-1)+d-1} - 2^{d-1} < 2^{n-1}.$$

Next, from $t \le 2^{d-1}(2^{32} - 1) < 2^{31+d}$, we have $v_H < 2^{31}$. Therefore, observing from $k \cdot 2^{32-d} < 2^{d-1}$ and $d \le 31$ that $k < 2^{d-2}$, we obtain $k v_H < 2^{29+d}$. It follows that

$$\begin{aligned} 0 \le r &< 2^{n-1} + 2^n + 2^{29+d} = 2^{n+1} - 2^{n-1} + 2^{29+d} \\ &< 2^{n+1} - 2^{31+d} + 2^{29+d} < 2^{n+1} - 2^{30+d} < 2p. \end{aligned}$$

To conclude, one can easily see that $r = (r_0, \ldots, r_{l-1})$, the output of Algorithm 4, since each of the values $(c_0 + k 2^{32-d} c_l + k v_H)$ and $(r^H_{i-1} + c_i + k 2^{32-d} c_{i+l})$ for $1 \le i < l$ fits into two 32-bit limbs, and $(r^H_{l-2} + v_L)$ into one limb. Indeed, we have that $c_0 + k 2^{32-d} c_l + k v_H < 2^{31+d} + 2^{29+d} < 2^{64}$, $r^H_{i-1} + c_i + k 2^{32-d} c_{i+l} < 2^{32} + 2^{31+d} < 2^{32+d} < 2^{64}$, and $r^H_{l-2} + v_L < 2^d + 2^d = 2^{d+1} \le 2^{32}$. □

4 Performance of Modular Multiplication and Squaring

4.1 Benchmark Methodology

All the compared source codes were compiled using GCC version 13.2.1 with optimization flags `-O3` and `-funroll-loops`. The benchmarks measured performance using the built-in cycle counting register of the Cortex-M4's DWT debugging unit, with interrupts turned off to avoid measurement fluctuations. The target hardware is an ST Microelectronics NUCLEO-L4R5ZI evaluation board for the STM32L4R5ZI microcontroller. Following

standard benchmarking practices such as those used by the `pqm4` project [36], the core clock speed is reduced to 20 MHz to ensure deterministic execution times.

While ARM specifies instruction timings for the Cortex-M4 from the point of view of the core, the memory system is implemented by the vendor and may necessitate additional wait states. An issue in the `pqm4` GitHub repository[6] discusses different memory timings in the STM32F407VG MCU at the heart of the STM32F4DISCOVERY board (previously the default board for `pqm4`); it arises from its banked memory organization, split as 112 KB in SRAM1 and 16 KB in SRAM2, with the latter being slower. Thus, `pqm4` supplies two linker scripts for that board, one using only SRAM1 (so that schemes can only access 112 KB of RAM) and a second one using both banks, for schemes that require the full 128 KB of RAM, with a tradeoff of worse timings for accesses to the additional RAM.

In the course of developing our library, we have observed a similar phenomenon in our target NUCLEO-L4R5ZI board, which also sports three memory banks: SRAM1 (192 KB), SRAM2 (64 KB) and SRAM3 (384 KB), with the first being faster than the other two. However, unlike the STM32F4DISCOVERY, `pqm4` does not supply different linker scripts for schemes that use less memory. It is usual to place the stack pointer at the end of RAM (in this case, SRAM3 if using all the available memory); as most memory accesses target the stack (e.g. local variables), this causes a non-trivial performance impact.

As the applications of §5 easily fit into the fast SRAM1 bank, we opted to run benchmarks with a linker script that restricts the use of RAM to this bank only, for better performance. Indeed, we recommend that end-users of our library (or other high-performance code) prioritize the use of SRAM1 if their application fits in it, to get a free performance boost. To allow direct comparisons to previously published results [2, 1, 36] using full-RAM linker scripts, we also provide performance figures for this case.

4.2 Performance Comparison of Pseudo-Mersenne Reduction Strategies

Our method applies to moduli where, for a given prime $p = 2^n - k$, $d = n \bmod 32 > 0$, as long as k is small (see §3.3.1). For instance, when dealing with the primes (defined in §2.4) p_{266} (with size 266 bits and $d = 10$), p_{336} (336 bits, $d = 16$), p_{383} (383 bits, $d = 31$) and p_{521} (521 bits, $d = 9$), one can exploit the respective gaps to enable more efficient carry handling. These compose the majority of pseudo-Mersenne primes of interest. One case where our algorithm cannot be applied is for primes such as p_{512} (512 bits, $d = 0$) that lack such a gap, necessitating the traditional two-pass approach.

Table 2 compares the computational costs of modular multiplication and squaring under the traditional pseudo-Mersenne reduction strategy (§2.2.3) and our proposed method (§3.3.1) for these five pseudo-Mersenne primes of cryptographic interest.

Table 2: Comparison of the effect of different pseudo-Mersenne reductions in modular multiplication (mm) and modular squaring (ms).

Reduction	p_{25519}		p_{266}		p_{336}		p_{383}		p_{521}	
	mm	ms	mm	ms	mm	ms	mm	ms	mm	ms
Standard (§2.2.3)	257	180	365	286	454	369	516	415	1024	733
New (§3.3.1)	210	153	308	238	389	300	433	342	905	626
Speedup	1.22	1.18	1.19	1.20	1.17	1.23	1.19	1.21	1.13	1.17

We indirectly compare our reduction strategies, as a direct comparison would not fully capture practical performance, since the reductions are inlined within multiplication and squaring operations. For Curve25519, our method reduces modular multiplication and

[6] https://github.com/mupq/pqm4/issues/180#issuecomment-775981267

squaring costs by 18% and 15%, respectively, achieving speedups of 1.22× and 1.18×. Since these speedups consider the full operation with reduction, this means that performance improvement when comparing only reductions must necessarily be even greater. Similar improvements are observed across all primes, with speedups ranging from 1.13× to 1.23×.

4.3 Performance Comparison of Multiplication with State-of-the-Art

Our implementation achieves substantial performance improvements over existing portable libraries while matching or approaching the efficiency of hand-optimized assembly, all through maintainable generated C code. Table 3 compares cycle counts for modular multiplication (mm) and squaring (ms) against state-of-the-art libraries on ARM Cortex-M4. We generated code for comparison for the Curve25519 prime p_{25519}, Curve448 prime p_{448} and for the pseudo-Mersenne prime p_{512}, as well as the three Montgomery-friendly primes used for the SQIsign signature scheme named p_{5248}, p_{65376} and p_{27500}.

On the portable side, we compare with the library we based our API, `modarith` [57] (`MA`). We also compare against Fiat-Crypto [29] (`FC`), a formally verified cryptographic arithmetic implementation generated from high-level code that is used for ECC in important libraries such as the BoringSSL, used inside Android. For p_{512}, we also compare against the Microsoft Research Elliptic Curve Cryptography (`MS`) library by Costello et al. [17].

Table 3: Comparison of modular arithmetic implementations in cycles for modular multiplication (mm) and modular squaring (ms). Speedups are given between our best timings versus either handmade assembly best results or best portable timings.

Categ	Library	p_{25519}		p_{448}		p_{5248}		p_{65376}		p_{27500}		p_{512}	
		mm	ms	mm	ms	mm	ms	mm	ms	mm	ms	mm	ms
Ptb	`MA`[57]	465	322	1337	932	439	315	931	658	1403	1011	1886	1397
	`FC`[29]	512	396	2436	1423	1757	1754	4048	4057	7212	7421	2965	2274
	`MS`[17]	–	–	–	–	–	–	–	–	–	–	3795	4287
Ours	Gen	**210**	**153**	937	771	221	163	**554**	**423**	**796**	**556**	**798**	**593**
	Custom	–	–	745	**565**	**209**	**152**	–	–	–	–	–	–
ASM	`FA`[31]	259	–	–	–	–	–	–	–	–	–	–	–
	`HL`[33]	226	157	–	–	–	–	–	–	–	–	–	–
	`SA`[60]	–	–	**659**	659	–	–	–	–	–	–	–	–
Spdup	Ours/ASM	0.93	0.97	1.13	0.87	–	–	–	–	–	–	–	–
	Ptb/Ours	2.21	2.12	1.80	1.63	2.10	2.07	1.68	1.55	1.76	1.82	2.36	2.36

The code generated by `m4-modarith` consistently outperforms portable alternatives by significant margins. In particular, for the SQIsign primes, our approach shows significant advantages over Fiat-Crypto. For instance, with p_{5248}, our implementation achieves a multiplication time of just 221 cycles, an 8.0× speedup over `FC`, and a 2.0× improvement over `modarith`. Similar gains are observed for larger SQIsign primes: for p_{65376}, multiplication is reduced to 554 cycles, representing a 7.3× improvement over `FC` (4048 cycles); and for p_{27500}, our implementation is 9.1× faster than `FC` (7212 cycles). These results demonstrate how our systematic code generation and domain-specific optimizations for structured primes can enable efficient arithmetic required by schemes like SQIsign.

Regarding handmade assembly implementations, several were included into our benchmarks. For p_{25519} operations, we include comparisons with implementations by Fujii and Aranha [31] (`FA`), who focused on optimizing p_{25519} specifically for the Cortex-M4, and Haase and Labrique [33] (`HL`), who developed highly optimized field arithmetic as part of their AuCPace protocol. Note that the implementation reported by Seo et al. [61] is not included in our benchmarks, as no source code has been made available for p_{25519}. For p_{448}, we include results from the implementation by Seo and Azarderakhsh [60] (`SA`), which

is an inline assembly implementation using a Solinas prime reduction method. Though the paper does not mention whether the code is publicly available, we have located a GitHub repository of one of the authors containing p_{448} code;[7] we cannot confirm that it corresponds to the implementation as described in that paper.

Our generated implementation rivals performance of handwritten assembly. For p_{25519}, our modular multiplication and squaring outperforms Fujii and Aranha's [31] and Haase and Labrique's [33] assembly. For p_{448}, our custom implementation is outperformed by Seo and Azarderakhsh's implementation [60] by only 13%, while our generated version is still closer to assembly performance than to portable performances.

Recently, Lengrenn [45] reported a non-scalable but highly optimized implementation for both X25519 and X448. Though we were graciously given access to the implementation for RFC 7748 benchmarks (see §5.1.1), differences in benchmarking setups and its usage of non-standard call conventions complicate direct comparisons for these micro-benchmarks.

For primes with further special structures, such as the golden-ratio trinomial p_{448}, our methodology simplifies further optimizations. Starting from our strategy for pseudo-Mersenne reduction, we implemented a Solinas-specific reduction for p_{448} (see §5.1.1), reducing mm from 955 to 742 cycles. For p_{5248}, we built a custom Montgomery reduction routine atop our generated base (see §5.2.1), achieving 2.1× speedups over `modarith`. These optimizations highlight the flexibility of our framework: developers can iteratively refine performance-critical sections without rebuilding the entire arithmetic stack.

5 Case Studies

In this section we explore the two main case studies of our library: RFC 7748 as a representation of elliptic curve cryptography applications, and SQIsign for isogeny-based cryptography. For each application, first we briefly summarize their workings, then present the challenges associated with its implementation on Cortex-M4 devices using our generated code. Finally, we benchmark our implementations with the state-of-the-art.

5.1 RFC 7748

In 2014, the Transport Layer Security (TLS) working group of the Internet Engineering Task Force (IETF) sought recommendations from the Crypto Forum Research Group (CFRG) for new elliptic curves to be included in the next version of the TLS protocol [55]. The criteria for selecting these curves, drawing from [15, 53], prioritized key design practices and elliptic curve properties, such as the rigidity of the curve generation process and the simplicity of cryptographic algorithm implementations. After thorough deliberation, two prime elliptic curves, Curve25519 and Curve448, were chosen to offer 128-bit and 224-bit security levels, respectively. RFC 7748 [40] provides the details of these selections, including the curve parameters and the Montgomery ladder-based scalar multiplication algorithms (X25519 and X448).

The [40] memorandum specifies two Montgomery elliptic curves. The recommendation for the 128 bits of security level uses the prime $p_{25519} = 2^{255} - 19$, and the curve parameter is given by $A = 486662$. This curve is commonly known as Curve25519 and was proposed in 2005 by Bernstein [14]. The curve group order is $h \cdot r \approx 2^{255}$, with $h = 8$ and r a large prime. On the other hand, the recommendation for the 224-bit security level is to use $p_{448} = 2^{448} - 2^{224} - 1$ and $A = 156326$. This curve was originally proposed by Hamburg in Edwards form as Ed448-Goldilocks [34], but it is referred to in [40] as Curve448. The group order is $h' \cdot r' \approx 2^{448}$, with $h' = 4$ and r' a large prime.

[7] https://github.com/solowal/DEVELOP/blob/master/Source%20Code/ICISC'20/Ed448.c

5.1.1 RFC 7748 Implementation in the Cortex-M4

Our library includes an implementation of RFC 7748 ECDH for both Curve25519 and Curve448, based on that of `modarith`, as we share their API. For the former curve, we use the output of our code generator configured for pseudo-Mersenne prime reduction using the technique described in §3.3.1, without further tweaks. For the latter curve, we generate code for Montgomery reduction with Montgomery-friendly optimizations; however, we have also written a hand-optimized version taking advantage of the Solinas reduction technique.

At a high level, our method for Solinas reduction resembles that of pseudo-Mersenne prime reduction, exploiting the fact that $2^{448} \equiv 2^{224}+1 \pmod{p_{448}}$; however, the additional high-order term 2^{224} in the RHS requires multiple applications of this identity to arrive at a fully simplified expression. We find it conceptually easier to visualize this procedure as polynomial reduction: starting from the output of a multiplication or squaring of 448-bit integers, which is composed of 28 32-bit limbs $c_0, c_1, \ldots, c_{27}$, we write this in polynomial form $c(w)$, from which we can recover the corresponding integer by evaluation at $w = 2^{32}$:

$$c(w) = c_0 + c_1 w + \ldots + c_{27} w^{27}.$$

Similarly, write $p(w) = w^{14} - w^7 - 1$, so that $p_{448} = p(2^{32})$. The first reduction pass is obtained as $c'(w) = c(w) \bmod p(w)$:

$$\begin{aligned} c'(w) =& (c_0 + c_{14} + c_{21}) \quad + (c_1 + c_{15} + c_{22})w \quad + \ldots + (c_6 + c_{20} + c_{27})w^6 \quad + \\ &+ (c_7 + c_{14} + 2c_{21})w^7 + (c_8 + c_{15} + 2c_{22})w^8 + \ldots + (c_{13} + c_{20} + 2c_{27})w^{13}. \end{aligned} \tag{1}$$

Evaluation of $c'(2^{32})$ produces a partially reduced version of the desired output $c \pmod{p_{448}}$, needing further carry propagation passes to arrive at a 448-bit representative. Our first reduction pass sets the i-th limb of the output to the coefficient of w^i in (1).

Note that each coefficient of (1) fits exactly into the pattern $R_h + R_l + R_n \times R_m$ (for $R_n \in \{1, 2\}$) used as input to the `UMAAL` instruction, as described in §2.1, resulting in a neat and efficient implementation. Also note the reuse of limbs of c between coefficients of w^i and w^{i+7} for $i \in \{0, 1, \ldots, 6\}$ in (1); this observation led us to schedule together the instructions that compute the i-th and $(i+7)$-th limbs, to minimize loads.

Carry propagation follows a similiar strategy as in §3.3.1; our first block computes the coefficients of w^0, w^6, w^7 and w^{13} of c' in (1). In parallel with the calculation of the remaining coefficients, a first carry propagation pass starts by propagating the high word of the coefficient of w^{13} back to limb 0 (this coefficient must also be added later to limb 7). Next, a second pass is performed, which also adds the carry from limb 13 to limbs 0 and 7.

To see why this second propagation pass is sufficient, let $q_i < w$ and s_i be all the computed limbs and carry values of c' from (1) for $0 \leq i \leq 13$. It is easy to see that for any $i \leq 6$, s_i cannot be greater than two, while for $7 \leq i \leq 13$, $s_i \leq 3$. Since s_{13} can be larger than zero, a carry pass must take place, starting at the first limb and adding s_{13}, i.e. $q_0 + s_{13}$. Call the resulting carry $t_0 \leq 1$. Limbs $i < 7$ can be computed by $q_i + s_{i-1} + t_{i-1}$, with a resulting carry t_i for each of them. The 7th limb is computed as $q_7 + s_6 + t_6 + s_{13}$ since we need to add the carry s_{13}, and for limbs $7 < i \leq 13$, $q_i + s_{i-1} + t_{i-1}$ again computes each limb i. Now consider the case for limb $i = 13$, computed as $q_{13} + s_{12} + t_{12} \leq w + 3$. Assuming a carry is produced, the new value of limb 13 is $(q_{13} + s_{12} + t_{12}) \mod w \leq 3$, with a carry of at most 1. Hence if a carry is propagated until limb 13 on the next carry pass, no other carry can be produced.

The Montgomery ladder as described in RFC 7748, defines an exact sequence of modular operations which are mapped directly to our API. The standard calls for a field inversion at the end of the algorithm, which we compute following `modarith` by first computing the progenitor $y = x^{(p-5)/8}$ for Curve25519, or $y = x^{(p-3)/4}$ for Curve448, to then obtain the inverse of x as $1/x = y^8 \cdot x^3$, or $1/x = y^4 \cdot x$, respectively.

5.1.2 Performance Comparison of RFC 7748

In Table 4 we compare our RFC 7748 generated code to `modarith` (`MA`) and also to the optimized implementation due to Emil Lenngren [44] (`Lpub`), which uses a non-standard function calling convention to improve performance, with also a version (`Lm4f`) which utilizes floating-point registers to further improve performance. Additionally, we were graciously given access to the RFC 7748 implementation discussed in [45], where novel non-scalable strategies are presented for both X448 and X25519 implementations. They correspond to `Lsiz` for its size sensitive versions and `Lspd` for its speed optimized variants.

Table 4: Benchmarks from our `m4-modarith` library (Custom and Generated) and other available code for RFC 7748, in thousands of cycles. Speedup refers to our best times divided by other implementation.

Categ	Library	Curve25519				Curve448			
		Red	KCycles	Spdup	1/Spdup	Red	KCycles	Spdup	1/Spdup
Ptb	`MA`[57]	PM	1394	0.45	2.23	Mont	6215	0.60	1.66
Ours	Custom	PM	624	1.00	1.00	Solinas	3739	1.00	1.00
	Generated	PM	634	0.98	1.02	Mont	4695	0.80	1.26
ASM	`Lpub`[44]	PM	563	1.11	0.90	–	–	–	–
	`Lm4f`[44]	PM	476	1.31	0.76	–	–	–	–
	`Lsiz`[45]	PM	521	1.20	0.83	Solinas	2836	1.32	0.76
	`Lspd`[45]	PM	428	1.46	0.69	Solinas	2273	1.64	0.61

Our `m4-modarith` library demonstrates good performance in RFC 7748 elliptic curve operations. For Curve25519, our custom implementation has very similar performance to the generated version. This is due to its core operations being mostly the same between the versions, with changes mostly due to inlining strategies. Our generated version outperforms `modarith` by a factor of 2.20×. For assembly-optimized implementations, while Lenngren's speed-focused `Lspd` retains a 1.46× advantage over our custom code, we note that for the only implementation which does not rely on the floating point unit present in Cortex-M4F cores, `Lpub`, our gap is much closer, with their code being only 13% faster than our generated code. This illustrates that our generator produces code closer to handwritten assembly than to efficient portable implementations.

For Curve448, the lack of Solinas reduction support for our generated code impacts performance, but the results overall reinforce this trend. Our custom Solinas reduction outperforms `modarith`'s implementation by 1.66×, while speed hand-optimized code of [45] beats our code by a slightly smaller margin.

Our RFC 7748 implementation prioritized speed over code size, yielding Curve25519 and Curve448 footprints of 23.8KB and 11.6–15.6KB versus 1.1–3.1KB for Lengrenn's hand-tuned assembly. This gap partially reflects our design focus – rather than using custom calling conventions and a tight integration of arithmetic for a particular application, we prioritized scalability and maintainability. Compiling with size optimization (`-Os`) shrinks our code to 3KB and 9KB (8× smaller for Curve25519), closer to assembly sizes but incurring 25–30% performance loss for Curve25519; Curve448 sees only a 3–6% degradation. As will be seen in §5.2.1, our SQIsign implementation balances code size and performance considerations.

5.2 SQIsign

Isogeny-based cryptography is a quantum-secure extension of elliptic curve cryptography. An *isogeny* is a rational map between elliptic curves that is also a non-constant group homomorphism. An isogeny that maps a curve to itself is called an *endomorphism*. The set

of all endomorphisms of a curve E, together with the zero map, form a ring which we denote $\mathrm{End}(E)$ (where composition plays the role of multiplication). Cryptographic applications work over a finite field of characteristic p and with *supersingular* elliptic curves, which are those having endomorphism ring isomorphic to a maximal order in the quaternion algebra $B_{p,\infty}$ ramified at p and infinity. The security guarantees of isogeny-based schemes rely on the supersingular isogeny problem, which is to find an isogeny between two given supersingular elliptic curves. This problem is closely related to determining the structure of the endomorphism ring of a given curve: given two supersingular curves E_0 and E where only the endomorphism ring of E_0 is known, the problem of finding an isogeny $\varphi : E_0 \to E$ is equivalent to the problem of computing the endomorphism ring of E [28, 68].

The SQIsign signature scheme [24] is currently the most relevant isogeny-based scheme, and the only one included in the second round of the NIST additional digital signature scheme standardization effort [1]. The signature scheme originates from the Fiat-Shamir transform of a sigma protocol, in which the prover holds a supersingular curve E_{pk} as public key and its endomorphism ring as secret key. Knowledge of the secret is demonstrated through the prover's ability to find an isogeny from E_{pk} to a challenge curve E_{chal}.

The different phases of the protocol can be summarized as follows:

1. **Setup:** starting from a curve E_0 of known endomorphism ring, the prover generates a random isogeny $\varphi_{sk} : E_0 \to E_{pk}$ to some other curve E_{pk} and uses it to compute $\mathrm{End}(E_{pk})$. Both φ_{sk} and $\mathrm{End}(E_{pk})$ are kept secret, while E_{pk} is made public.

2. **Commitment/Challenge:** the prover generates another secret isogeny $\varphi_{com} : E_0 \to E_{com}$, and sends E_{com} to the verifier, and the verifier in turn generates an isogeny $\varphi_{chal} : E_{com} \to E_{chal}$, and sends both φ_{chal} and E_{chal} to the prover.

3. **Response:** the prover uses φ_{chal} to compute $\mathrm{End}(E_{chal})$, then responds with an isogeny $\varphi_{resp} : E_{pk} \to E_{chal}$ which it finds using its knowledge of both endomorphism rings.

4. **Verification:** the verifier evaluates φ_{resp} and checks that the codomain is E_{chal}.

The response isogeny must be randomized in order to avoid leaking any private information. By using the Deuring correspondence [26], this is reduced to finding a left ideal of $\mathrm{End}(E) \subset B_{p,\infty}$ of given norm. This is a computationally expensive task involving the manipulation of different mathematical objects (ideals, quaternion elements, and rational numbers of arbitrary size), making signing a complex procedure. On the other hand, verification is a much simpler procedure requiring mostly just the evaluation of an isogeny, making it not only more lightweight, but also comprised almost entirely of finite field operations. Thus, we focus only on verification, since it is more apt for embedded devices and also benefits more directly from improvements to the field arithmetic.

Two-dimensional SQIsign. Later works [13, 50] improved the scheme by embedding the response isogeny $\varphi_{resp} : E_{com} \to E_{chal}$ into a two-dimensional isogeny $\Phi : E_{com} \times E_{aux} \to E_{chal} \times E'_{aux}$. Using techniques based on [23], this framework allows for an efficient evaluation of φ_{resp} even when its degree is not smooth, thus lifting a restriction which slowed down signing and significantly increased the total degree of the isogeny. This improvement made its way to the SQIsign NIST submission, which adopted the two-dimensional variant for round 2 of the process [1].

Although evaluating isogenies in two dimensions is more complex, the total verification time remains roughly unchanged due to the smaller degrees involved, as first pointed out in [2]. Moreover, the algorithms behind two-dimensional isogenies still break down almost entirely to finite field operations, so the optimizations presented in this paper remain important. Especially important for this work is also the fact that this technique enables the use of Montgomery-friendly primes of the form $p = c \cdot 2^f - 1$, where c is a small odd integer, which allows for drastic optimizations in the finite field reduction algorithm.

5.2.1 SQIsign Verification in the Cortex-M4

We further showcase the performance of `m4-modarith` by integrating its field arithmetic code with SQIsign, both the optimized one-dimensional variants presented in [2] and the round-2 submission to the NIST Additional Digital Signatures standardization process, based on two-dimensional techniques. A total of 6 different primes are used among these variants, and writing Cortex-M4-optimized code for all of them would represent a major undertaking, if it were not for the code generation approach of our library[8].

Due to the specific form of the primes, we generate code using Montgomery reduction with Montgomery-friendly optimizations. Although the generated code already achieves excellent performance, we also wrote a hand-tweaked version of the p_{5248} reduction procedure to extract some additional performance, made possible by its fairly small size (8 limbs) and high degree of Montgomery-friendliness; this hand-written code does not follow a regular pattern that could be backported to the code generator for other cases.

An additional source of performance improvement became apparent by inspection of the compiled binaries. In essence, we noticed that while the compiler was inlining our library code into the $\mathbb{F}_p$ arithmetic API of SQIsign's `gf` module[9], these functions were not in turn inlined into the $\mathbb{F}_{p^2}$ arithmetic API. Such inlining avoids overhead in terms of e.g. function call and return instructions and setting up the stack frame, which is significant for small functions such as modular addition and subtraction.

Since the remainder of SQIsign's modules use almost exclusively $\mathbb{F}_{p^2}$ and not $\mathbb{F}_p$ arithmetic, we merged all files in the `gf` module, to give the compiler opportunity to inline our library code into $\mathbb{F}_{p^2}$ functions as well. As expected, performance improved, but code size also increased significantly. By experimenting with the `inline` keyword in our library code, and moving large but rarely-used $\mathbb{F}_{p^2}$ functions to a separate file to prevent inlining, we kept most of the speedup with only a small impact in code size, as discussed next.

5.2.2 Performance Comparisons for SQIsign

Table 5 presents performance figures for variants of one-dimensional SQIsign (based on source code accompanying [2]), as well as the second round submission of the scheme to NIST's Additional Digital Signatures contest [1], based on two-dimensional techniques. For the work of [2], the versions are an optimized Round 1 implementation (R1+o), a variant using a different sampling technique (Smart) and an uncompressed variant (Unc).

For one-dimensional variants, although [2] already includes Cortex-M4-specific code, we further improve upon their results by as much as 11% by using only code generated by our library (Ours, gen); the gap widens to 21% when considering our hand-optimized implementation for p_{5248} in the uncompressed variant (Ours, cust). The use of the fast SRAM1 memory bank increases performance by up to 17% compared to a full-RAM memory map. Finally, inlining optimizations (Opt) for $\mathbb{F}_{p^2}$ arithmetic improve performance by up to 16% compared to our original code. Combining all improvements (field arithmetic, hand-optimized versions, use of the fast SRAM1 and $\mathbb{F}_{p^2}$ inlining), gains of up to 44% are realized in comparison to previous Cortex-M4-specific code, with the gains being significantly more pronounced compared to the reference code based on Fiat-Crypto.

Moving to the second round submission of SQIsign, code generated by our library is faster than the `modarith`-based reference code by up to 73%, considering field arithmetic improvements alone. When considering hand-optimized code for p_{5248} as well, the speedup compared to the reference 83%. Use of the fast SRAM1 bank results in an additional gain of up to 16%. $\mathbb{F}_{p^2}$ inlining optimizations also contribute up to 16% extra performance. Combining all of these improvements results in up to 126% faster code than the reference.

[8] Our SQIsign implementation is available at https://github.com/Crypto-TII/m4-modarith-pqm4.

[9] This inlining is made possible because the library code is directly included into the file which serves as an API adapter for SQIsign's $\mathbb{F}_p$ arithmetic layer, through the C preprocessor's `#include` directive.

Table 5: SQIsign verification performance using our `m4-modarith` library and other implementations. Timings in millions of cycles for the faster SRAM1 (FM) and slower full-RAM (SM) configurations. Speedups computed between the reference code for each variant and our fastest implementation in the FM configuration.

Imp	NIST Level I								NIST Level III				NIST Level V			
	R1+o[2]		Smart[2]		Unc.[2]		R2[1]		R1+o[2]		R2[1]		R1+o[2]		R2[1]	
	p_{1973}		p_{5248}		p_{5248}		p_{5248}		p_{47441}		p_{65376}		p_{318233}		p_{27500}	
	FM	SM	FM	SM	FM	SM	FM	SM	FM	SM	FM	SM	FM	SM	FM	SM
`FC` (R1 Ref)	1051	1176	354	391	236	260	–	–	4113	4600	–	–	9917	11230	–	–
`MA` (R2 Ref)	482	557	124	143	82.4	95.3	107	123	1784	2083	319	375	3644	4314	632	751
`M4`	340	382	82.9	94.6	54.8	62.6	–	–	1168	1324	–	–	2605	2946	–	–
Ours (gen)	315	349	78.4	90.7	51.8	60.0	67.0	77.5	1067	1193	207	238	2385	2666	376	435
Ours (cust)	–	–	70.3	78.6	46.2	51.7	60.0	67.1	–	–	–	–	–	–	–	–
Opt (gen)	293	325	67.7	79.5	44.5	52.3	57.8	67.7	1028	1149	197	226	2334	2607	363	421
Opt (cust)	–	–	66.5	74.5	43.6	48.8	54.5	61.2	–	–	–	–	–	–	–	–
Ref./Ours	3.59		5.32		5.41		1.96		4.00		1.73		4.25		1.74	

Code size and RAM usage results are presented in Table 6. In general, our code is smaller than Fiat-Crypto, and somewhat bigger than that of `modarith`, although similar in some cases and even a bit smaller in one case (security level I of the second round submission). The impact of $\mathbb{F}_{p^2}$ inlining optimizations is variable, but as small as 1.5%; we consider it a good tradeoff for the performance gained, although such a decision is of course best left to the end-user of the library. RAM usage is similar across implementations based on a saturated representation (Fiat-Crypto, the Cortex-M4 code of [2] and the code generated by our library), whereas `modarith` requires up to 16% extra RAM, compared to our library code, due to the extra limbs required for unsaturated representations.

Table 6: Code size (C) and RAM usage (R) in KB for our `m4-modarith` library and other implementations for verification across different SQIsign variants.

Imp	NIST Level I								NIST Level III				NIST Level V			
	R1+o[2]		Smart[2]		Unc.[2]		R2[1]		R1+o[2]		R2[1]		R1+o[2]		R2[1]	
	p_{1973}		p_{5248}		p_{5248}		p_{5248}		p_{47441}		p_{65376}		p_{318233}		p_{27500}	
	C	R	C	R	C	R	C	R	C	R	C	R	C	R	C	R
`FC` (R1 Ref)	111	8.66	86.0	4.46	86.4	5.00	–	–	150	12.8	–	–	202	16.8	–	–
`MA` (R2 Ref)	85.5	9.09	60.2	4.87	60.6	5.37	40.3	30.9	106	14.0	43.8	50.0	128	17.9	46.3	64.2
`M4`	88.4	8.52	61.5	4.40	61.9	4.84	–	–	104	12.8	–	–	128	16.7	–	–
Ours (gen)	98.2	8.53	63.0	4.37	63.4	4.84	38.4	27.6	111	12.8	43.6	43.1	138	16.7	51.1	57.2
Ours (cust)	–	–	61.9	4.37	62.3	4.84	38.3	27.6	–	–	–	–	–	–	–	–
Opt (gen)	99.7	8.58	64.3	4.39	64.7	4.86	42.1	27.6	113	12.9	52.0	43.2	141	16.8	61.3	57.4
Opt (cust)	–	–	63.3	4.39	63.7	4.86	42.5	27.6	–	–	–	–	–	–	–	–

6 Conclusions

This work presented `m4-modarith`, a library for generating efficient finite field arithmetic code targeting the ARM Cortex-M4 microcontroller. Our library performs automated code generation of C code with targeted inline assembly to achieve performance competitive with hand-optimized implementations while retaining flexibility. Integration with the SQIsign verification protocol yields a significant improvement of up to 2.3× over the reference. For high-performance Cortex-M4 parts at 216 MHz, the results of Table 5 translate to eminently practical 250 ms level I security verification times, challenging the perception that SQIsign is inherently slow and too complex for use in constrained

environments. Similarly, we demonstrate the value of platform-specific optimizations by speeding up RFC 7748 by up to 2.2× compared to the portable `modarith` code generator [57].

Our library occupies a critical middle ground between portable code and handwritten assembly. Portable implementations often fail to exploit architectural features like the Cortex-M4's `UMAAL` instructions, while manual assembly optimizations lack adaptability to parameter changes. By automating low-level optimizations, `m4-modarith` reduces the performance gap without sacrificing development agility. Multiplication and reduction strategies introduced here also hold broader relevance: the former offers a simpler alternative matching operand caching's memory operation count in multi-precision multiplication for embedded systems, while the latter's single generalized carry handling improves reduction performance and can be used in most cryptographically relevant pseudo-Mersenne primes.

One of the key takeaways of this work is that, despite the ever-evolving landscape of isogeny-friendly primes, our `m4-modarith` library can readily generate efficient field arithmetic for any such prime, thus making it a highly practical tool for implementing high-performance isogeny-based cryptographic protocols.

As a future work, a few promising directions emerge. First, integrating formal verification tools like Jasmin [4] and EasyCrypt [12] could extend our methodology with machine-checked correctness guarantees for critical operations such as modular reduction. This would complement—without replacing—the C/inline-assembly paradigm, enabling verified low-level snippets to coexist with high-level code while preserving performance. Second, including support for other embedded architectures such as Helium MVE enabled devices are a natural follow-up path for our library. Finally, the exploration of alternative scalable strategy methods for multiplication, such as Karatsuba for very large primes, or research into the viability of a full Solinas reduction strategy might be valid directions.

Acknowledgments

We thank João Victor Oliveira Paliares and Luis Felipe Lapa Barcelos Coutinho for their contributions for our testing and benchmarking, as well as the anonymous reviewers for the helpful suggestions. The first, second and sixth author thank the Technology Innovation Institute for partial financial support during the development of this work.

A Performance of Additional Primes

For completeness, we present comparisons for modular multiplication and squaring for a few other primes of cryptographic interest. The primes are as follows: the NIST prime p_{256r1}, usually called P-256 or secp256r1, the prime secp256k1 (which we label here as p_{256k1}), both using Montgomery reduction, and the pseudo-Mersenne primes $p_{266}, p_{336}, p_{383}, p_{521}$, defined in §2.4. We compare when possible to other libraries such as `modarith` [57] (`MA`), Fiat-Crypto [29] (`FC`), μ-ECC [38] and p256-m [47], used in Mbed-TLS [67].

Table 7: Comparison in cycles for modular multiplication (mm) and squaring (ms). Speedups are given between our best timings versus best results for other libraries.

Categ	Library	p_{256r1}		p_{256k1}		p_{266}		p_{336}		p_{383}		p_{521}	
		mm	ms	mm	ms	mm	ms	mm	ms	mm	ms	mm	ms
Ours	Generated	**344**	**289**	**387**	**346**	**308**	**238**	**413**	**323**	**457**	**356**	**931**	**639**
Others	`MA`[57]	628	492	654	524	595	358	641	411	1025	698	1299	769
	`FC`[29]	1455	1469	1920	1921	503	298	836	492	1427	1175	1613	923
	μ-`ecc`[38]	1333	1380	708	764	–	–	–	–	–	–	–	–
	`p256-m`[47]	1814	–	–	–	–	–	–	–	–	–	–	–
Spdup	Others/Ours	1.83	1.70	1.69	1.51	1.63	1.25	1.55	1.27	2.24	1.96	1.40	1.20

B Modular Reduction for the condition $k \cdot 2^{32-d} < 2^{31}$

Given an integer $p = 2^n - k$ and a product $c = a \cdot b < 2^{2(n+1)}$ of two integers smaller than $2p$, Algorithm 4 reduces c modulo p while maintaining the upper bound of $2p$ when $k \cdot 2^{32-d} < 2^{d-1}$ for $d = n \bmod 32$. Algorithm 5 below generalizes to the case $k \cdot 2^{32-d} < 2^{31}$.

Algorithm 5 MODRED'(n, k, c)

Require: $n > 64$, $k > 0$ and $0 < c < 2^{2(n+1)}$, with $d = n \bmod 32$ and $k \cdot 2^{32-d} < 2^{31}$
Ensure: $(c \bmod p)$ or $(c \bmod p) + p$, with $p = 2^n - k$

$l \leftarrow \left\lceil \frac{n}{32} \right\rceil$
$(c_0, \ldots, c_{2l-1}) \leftarrow c$, with $0 < c_i < 2^{32}$ for $0 \le i < 2l$
$(t_H | t_L) \leftarrow c_{l-2} + k2^{32-d} c_{2l-2}$
$v \leftarrow t_H + c_{l-1} + k2^{\delta} c_{2l-1}$
$v_L \leftarrow v \pmod{2^d}$, $v_H \leftarrow \lfloor v/2^d \rfloor$
$(r_0^H | r_0^L) \leftarrow c_0 + k2^{32-d} c_l + k v_H$
$r_0 \leftarrow r_0^L$
for $1 \le i < l - 2$ **do**
    $(r_i^H | r_i^L) \leftarrow r_{i-1}^H + c_i + k2^{32-d} c_{i+l}$
    $r_i \leftarrow r_i^L$
$(r_{l-2}^H | r_{l-2}^L) \leftarrow r_{l-3}^H + t_L$
$r_{l-2} \leftarrow r_{l-2}^L$
$r_{l-1} \leftarrow r_{l-2}^H + v_L$
return $(r_0, \ldots, r_{l-1})$

Theorem 2. *Algorithm 5 is correct.*

Proof. Note first that $n = 32(l-1) + d$. Rewriting c as $c = \sum_{0 \le i < l} (c_i + 2^{32l} c_{i+l}) 2^{32i}$, and using $2^n = 2^{32l-(32-d)} \equiv k \pmod{p}$, we obtain a first reduction of c modulo p:

$$r' = \sum_{0 \le i < l} (c_i + k2^{32-d} c_{i+l}) 2^{32i}.$$

Now, define $s = \sum_{0 \le i < l-2} (c_i + k2^{32-d} c_{i+l}) 2^{32i}$, $t = c_{l-2} + k2^{32-d} c_{2l-2}$ and $u = c_{l-1} + k2^{32-d} c_{2l-1}$ so that

$$r' = s + t2^{32(l-2)} + u2^{32(l-1)}.$$

Let $t = t_L + 2^{32} t_H$, with $t_L = t \bmod 2^{32}$ and $t_H = \lfloor t/2^{32} \rfloor$. Setting $v = t_H + u$, $v_L = v \pmod{2^d}$ and $v_H = \lfloor v/2^d \rfloor$ implies

$$t2^{32(l-2)} + u2^{32(l-1)} = t_L 2^{32(l-2)} + v_L 2^{32(l-1)} + 2^d v_H 2^{32(l-1)} = t_L 2^{32(l-2)} + v_L 2^{32(l-1)} + 2^n v_H.$$

Thus we can reduce further c using again $2^n \equiv k \pmod{p}$:

$$r = s + t_L 2^{32(l-2)} + v_L 2^{32(l-1)} + k v_H.$$

To show that $0 \le r < 2p$, we first note that $t_L 2^{32(l-2)} < 2^{32(l-1)}$ and $v_L 2^{32(l-1)} < 2^d 2^{32(l-1)} = 2^n$. Then, for $0 \le i < l$, we have $c_i + k2^{32-d} c_{i+l} \le (2^{32}-1) + (2^{31}-1)(2^{32}-1) = 2^{31}(2^{32}-1)$. Hence

$$s \le 2^{32(l-2)+31} - 2^{31} = 2^{32(l-1)-1} - 2^{31}.$$

Next, from $t \le 2^{31}(2^{32}-1) < 2^{63}$, we have $t_H < 2^{31}$ Then $v < 2^{31} + 2^{63} < 2^{64}$ implies $v_H < 2^{64-d}$. Therefore, observing from $k2^{32-d} < 2^{31}$ that $k < 2^{d-1}$, we obtain $k v_H < 2^{63}$. It follows that

$$\begin{aligned} 0 \le r &< 2^{32(l-1)-1} - 2^{31} + 2^{32(l-1)} + 2^n + 2^{34} < 2^{32(l-1)+1} - 2^{32(l-1)-1} + 2^{n+1} - 2^n + 2^{63} \\ &< 2^{n+1} - 2^{32(l-1)+d} + 2^{32(l-1)+1} < 2^{n+1} - 2^{32(l-1)+1} < 2p. \end{aligned}$$

To conclude, one can easily see that $r = (r_0, \ldots, r_{l-1})$, since each of the values $(c_0 + k2^{32-d} c_l + k v_H)$, $(r_{i-1}^H + c_i + k2^{32-d} c_{i+l})$ for $1 \le i < l$ and $(r_{l-3}^H + t_L)$ fits into two 32-bit limbs, and $(r_{l-2}^H + v_L)$ into one limb. Indeed, we have that $c_0 + k2^{32-d} c_l + k v_H < 2^{63} + 2^{34} < 2^{64}$, $r_{i-1}^H + c_i + k2^{32-d} c_{i+l} < 2^{32} + 2^{63} < 2^{64}$, $r_{l-3}^H + t_L < 2^{32} + 2^{32} = 2^{33} < 2^{64}$, and $r_{l-2}^H + v_L < 2 + 2^d < 2^{d+1} < 2^{32}$. □

References

[1] Marius Aardal, Gora Adj, Diego Aranha, Andrea Basso, Isaac Andrés Canales Martínez, Jorge Chavez-Saab, Maria Corte-Real Santos, Pierrick Dartois, Luca De Feo, Max Duparc, , Jonathan Komada Eriksen, Tako Boris Fouotsa, Décio Luiz Gazzoni Filho, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Luciano Maino, Michael Meyer, Kohei Nakagawa, Hiroshi Onuki, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Giacomo Pope, Damien Reijnders, Krijn Robert, Francisco Rodríguez-Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign: Algorithm specifications and supporting documentation, version 2.0. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/sqisign-spec-round2-web.pdf, 2025. National Institute of Standards and Technology.

[2] Marius A. Aardal, Gora Adj, Arwa Alblooshi, Diego F. Aranha, Isaac A. Canales-Martínez, Jorge Chávez-Saab, Décio Luiz Gazzoni Filho, Krijn Reijnders, and Francisco Rodríguez-Henríquez. Optimized One-Dimensional SQIsign Verification on Intel and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):497–522, Dec. 2024.

[3] Mehmet Adalier and Antara Teknik. Efficient and secure elliptic curve cryptography implementation of curve p-256. In *Workshop on elliptic curve cryptography standards*, volume 66, pages 2014–2017. NIST, 2015.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982, 2020.

[5] Mila Anastasova, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast strategies for the implementation of SIKE round 3 on ARM Cortex-M4. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(10):4129–4141, 2021.

[6] Mila Anastasova, Reza Azarderakhsh, Mehran Mozaffari Kermani, and Lubjana Beshaj. Time-efficient finite field microarchitecture design for Curve448 and Ed448 on Cortex-M4. In Seung-Hyun Seo and Hwajeong Seo, editors, *Information Security and Cryptology – ICISC 2022*, pages 292–314, Cham, 2023. Springer Nature Switzerland.

[7] Mila Anastasova, Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Compressed SIKE round 3 on ARM Cortex-M4. In Joaquin Garcia-Alfaro, Shujun Li, Radha Poovendran, Hervé Debar, and Moti Yung, editors, *Security and Privacy in Communication Networks*, pages 441–457, Cham, 2021. Springer International Publishing.

[8] Mila Anastasova, Mojtaba Bisheh-Niasar, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Efficient and side-channel resistant design of high-security Ed448 on ARM Cortex-M4. In *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 93–96, 2022.

[9] D. F. Aranha, C. P. L. Gouvea, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[10] Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, and Jefferson E. Ricardini. A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Paper 2013/647, 2013.

[11] Arm. Application binary interface for the arm® architecture. https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst, 2025.

[12] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 71–90, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] Andrea Basso, Pierrick Dartois, Luca De Feo, Antonin Leroux, Luciano Maino, Giacomo Pope, Damien Robert, and Benjamin Wesolowski. SQIsign2D-West: The Fast, the Small, and the Safer. In *Advances in Cryptology – ASIACRYPT 2024*, pages 339–370, Singapore, 2025. Springer Nature Singapore.

[14] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Proceedings of PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer Berlin Heidelberg, 2006.

[15] Daniel J. Bernstein and Tanja Lange. Safe curves for elliptic-curve cryptography. In Colin Boyd, Reihaneh Safavi-Naini, and Leonie Simpson, editors, *Information Security in a Connected World: Celebrating the Life and Work of Ed Dawson*, pages 124–191. Springer Nature Switzerland, Cham, 2025.

[16] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[17] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, 6(4):259–286, Nov 2016.

[18] Fabio Campos, Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. Optimizations and practicality of high-security CSIDH. *IACR Communications in Cryptology*, 1(1), 2024.

[19] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. The SQALE of CSIDH: sublinear Vélu quantum-resistant isogeny action with low exponents. *J. Cryptogr. Eng.*, 12(3):349–368, 2022.

[20] Jorge Chavez-Saab, Maria Corte-Real Santos, Luca De Feo, Jonathan Komada, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Michael Meyer, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Francisco Rodríguez-Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign: Algorithm specifications and supporting documentation, version 2.0. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/sqisign-spec-web.pdf, June 2023. National Institute of Standards and Technology.

[21] Maria Corte-Real Santos, Jonathan Komada Eriksen, Michael Meyer, and Krijn Reijnders. AprèsSQI: Extra Fast Verification for SQIsign Using Extension-Field Signing. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 63–93, Cham, 2024. Springer Nature Switzerland.

[22] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic. *Journal of Cryptographic Engineering*, 8(3):227–240, Sep 2018.

[23] Pierrick Dartois, Luciano Maino, Giacomo Pope, and Damien Robert. An algorithmic approach to (2, 2)-isogenies in the theta model and applications to isogeny-based cryptography. In *ASIACRYPT 2024, Part III*, volume 15369 of *Lecture Notes in Computer Science*, pages 304–338. Springer, 2024.

[24] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In *Advances in Cryptology – ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part I*, page 64–93, Berlin, Heidelberg, 2020. Springer-Verlag.

[25] Luca De Feo, Antonin Leroux, Patrick Longa, and Benjamin Wesolowski. New algorithms for the Deuring correspondence. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 659–690, Cham, 2023. Springer Nature Switzerland.

[26] Max Deuring. Die typen der multiplikatorenringe elliptischer funktionenkörper. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 14(1):197–272, Dec 1941.

[27] Max Duparc and Tako Boris Fouotsa. SQIPrime: A dimension 2 variant of SQISignHD with non-smooth challenge isogenies. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024*, pages 396–429, Singapore, 2025. Springer Nature Singapore.

[28] Kirsten Eisenträger, Sean Hallgren, Kristin Lauter, Travis Morrison, and Christophe Petit. Supersingular isogeny graphs and endomorphism rings: Reductions and solutions. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 329–368, Cham, 2018. Springer International Publishing.

[29] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[30] Armando Faz-Hernandez, Julio Lopez, Eduardo Ochoa-Jimenez, and Francisco Rodriguez-Henriquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, November 2018.

[31] Hayato Fujii and Diego F. Aranha. Curve25519 for the Cortex-M4 and beyond. In *LATINCRYPT*, volume 11368, pages 109–127. Springer, 2017.

[32] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 119–132, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[33] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):1–48, Feb. 2019.

[34] Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015.

[35] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 459–474, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[36] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2019/844, 2019.

[37] Sabyasachi Karati and Palash Sarkar. Kummer for genus one over prime order fields. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 3–32, Cham, 2017. Springer International Publishing.

[38] Kenneth MacKay. Micro-ecc. https://github.com/kmackay/micro-ecc, 2022. Acessed on June 2025.

[39] A. Langley, M. Hamburg, and S. Turner. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. https://tools.ietf.org/html/rfc7748, July 2005.

[40] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. https://tools.ietf.org/html/rfc7748, January 2016.

[41] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, 2016.

[42] Programming Languages and Verification Group at MIT CSAIL. Fiat-Crypto: Synthesizing correct-by-construction code for cryptographic primitives, 2023. https://github.com/mit-plv/fiat-crypto/tree/master.

[43] Younho Lee, Ill-Hee Kim, and Yongsu Park. Improved multi-precision squaring for low-end RISC microcontrollers. *J. Syst. Softw.*, 86(1):60–71, January 2013.

[44] Emil Lenngren. X25519 for ARM Cortex-M4 and other ARM processors. https://github.com/Emill/X25519-Cortex-M4, 2020.

[45] Emil Lenngren. Assembly optimised Curve25519 and Curve448 implementations for ARM Cortex-M4 and Cortex-M33. Cryptology ePrint Archive, Paper 2025/523, 2025.

[46] Zhe Liu, Hwajeong Seo, and Howon Kim. A synthesis of multi-precision multiplication and squaring techniques for 8-bit sensor nodes: State-of-the-art research and future challenges. *Journal of Computer Science and Technology*, 31(2):284–299, Mar 2016.

[47] Manuel Pégourié-Gonnard. p256-m. https://github.com/mpg/p256-m, 2022. Acessed on June 2025.

[48] Vitor Satoru Matsumine, Félix Rodrigues, Décio Gazzoni Filho, Caio Teixeira, Julio López, and Ricardo Dahab. Evaluation of modular multiplication techniques for Supersingular Isogeny Schemes on ARMv8 cores. In *Anais do XXII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 29–42, Porto Alegre, RS, Brasil, 2022. SBC.

[49] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[50] Kohei Nakagawa, Hiroshi Onuki, Wouter Castryck, Mingjie Chen, Riccardo Invernizzi, Gioella Lorenzon, and Frederik Vercauteren. SQIsign2D-East: A New Signature Scheme Using 2-Dimensional Isogenies. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024*, pages 272–303, Singapore, 2025. Springer Nature Singapore.

[51] NIST. Digital Signature Standard. https://csrc.nist.gov/pubs/fips/186-5/final, February 2023.

[52] Thomaz Oliveira, Julio López, Hüseyin Hisil, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder - improving the performance of X25519 and X448. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, volume 10719 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2017.

[53] Kenny Patterson. Formal request from TLS WG to CFRG for new elliptic curves. Crypto Forum Research Group archives, 2015. https://mailarchive.ietf.org/arch/msg/cfrg/Hvihr_yQhVB_Qdl-mtwTdVbHGiU.

[54] T. Pornin. crrl: Rust library for cryptographic research, 2023. https://github.com/pornin/crrl.

[55] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, 2017. Internet-Draft. https://tools.ietf.org/html/draft-ietf-tls-tls13-19.

[56] Maria Corte-Real Santos, Jonathan Komada Eriksen, Michael Meyer, and Francisco Rodríguez-Henríquez. Finding practical parameters for isogeny-based cryptography. *IACR Communications in Cryptology*, 1(3), 2024.

[57] Michael Scott. Elliptic curve cryptography for the masses: Simple and fast finite field arithmetic. Cryptology ePrint Archive, Paper 2024/779, 2024.

[58] Hwa-Jeong Seo. Consecutive operand-caching method for multiprecision multiplication, revisited. *Journal of Information and Communication Convergence Engineering*, 13(1):27–35, mar 2015.

[59] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (SIKE) round 2 on ARM Cortex-M4. *IEEE Transactions on Computers*, 70(10):1705–1718, 2021.

[60] Hwajeong Seo and Reza Azarderakhsh. Curve448 on 32-bit ARM Cortex-M4. In *Information Security and Cryptology – ICISC 2020: 23rd International Conference, Seoul, South Korea, December 2–4, 2020, Proceedings*, page 125–139, Berlin, Heidelberg, 2020. Springer-Verlag.

[61] Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. SIKE round 2 speed record on ARM Cortex-M4. In Yi Mu, Robert H. Deng, and Xinyi Huang, editors, *Cryptology and Network Security*, pages 39–60, Cham, 2019. Springer International Publishing.

[62] Hwajeong Seo and Howon Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In Dong Hoon Lee and Moti Yung, editors, *Information Security Applications*, pages 55–67, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[63] Hwajeong Seo, Zhe Liu, Jongseok Choi, and Howon Kim. Multi-precision squaring for public-key cryptography on embedded microprocessors. In Goutam Paul and Serge Vaudenay, editors, *Progress in Cryptology – INDOCRYPT 2013*, pages 227–243, Cham, 2013. Springer International Publishing.

[64] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):1–20, Aug. 2018.

[65] Hwajeong Seo, Taehwan Park, Shinwook Heo, Gyuwon Seo, Bongjin Bae, Lu Zhou, and Howon Kim. Multi-precision squaring for public-key cryptography on embedded microprocessors, a step forward. In Dooho Choi and Sylvain Guilley, editors, *Information Security Applications*, pages 331–340, Cham, 2017. Springer International Publishing.

[66] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR-39, Dept. of C&O, University of Waterloo, 1999.

[67] Trusted Firmware. Mbed-TLS. https://github.com/Mbed-TLS/mbedtls, 2025. Acessed on June 2025.

[68] Benjamin Wesolowski. The supersingular isogeny path and endomorphism ring problems are equivalent. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1100–1111, 2022.