A performance study of X25519 on Cortex-M3 and M4

de Groot, W.

Award date:
2015

Master Thesis

A Performance Study of X25519 on Cortex-M3 and M4

Wouter de Groot

supervised by:
Prof. Dr. Tanja Lange
Rick van Galen, MSc (KPMG)
Eleonora Petridou, MSc (KPMG)

28 September 2015

Abstract

Embedded computing is becoming increasingly powerful and ubiquitous. With this increased use the need for security also increases. Freely available secure implementations of cryptographic primitives allow software designers to improve their products at little cost. In this thesis we provide a constant-time implementation of X25519 for the ARM Cortex-M4 architecture in assembler, using its hardware features and reduced refined Karatsuba to obtain a full scalar multiplication in 1 816 351 cycles with 628 bytes of runtime memory and 4140 bytes of ROM. In addition, we perform timing measurements on the related Cortex-M3 architecture to establish the suitability of its 32 bit multiplication instructions in order to port our implementation. Due to data-dependent optimisations these instructions are not suitable for a simple, secure implementation of X25519.

Contents

Chapter 1

Introduction

It will surprise no-one to note that technology's presence is pervading ever further into personal, professional and industrial areas. Computing devices are still shrinking while growing more powerful, and their applications become more and more intertwined with tasks of a private, safety-critical and/or legal liability nature. In order to carry out the manifold tasks entrusted to computing devices, certain measures must be taken to guarantee confidentiality, integrity, and availability of data and processing. One such measure is to employ strong cryptography. Designs and secure implementations of cryptosystems, as well as attacks against them, are subjects of active research. In this thesis an implementation of an elliptic curve Diffie-Hellman cryptosystem, X25519, for the 32 bit ARM Cortex-M4 microcontroller architecture will be discussed along with a series of experiments to ascertain the precise timing behavior of 32 bit multiplication instructions for the similar Cortex-M3 architecture.

The Diffie-Hellman (DH) function, first introduced in 1976 [11], allows parties to establish shared secrets using a common public generator and individual secret keys. In the original function the public key consists of a prime p and g, a primitive root modulo p, and the secret keys are arbitrary integers a and b for parties A and B. Parties compute $g^a \bmod p$ and $g^b \bmod p$ respectively and exchange these values. Upon reception of each other's intermediate value each party then raises it to their secret integer again modulo p, establishing the shared value $g^{ab} \bmod p$ for both. An eavesdropper observes only the intermediate values–although an active attacker might be able to perform a man-in-the-middle attack, pretending to be the legitimate correspondent to both parties and performing key exchange with both to become a conduit between them. The security of this scheme is related to the discrete logarithm problem: it is believed that solving for a from $g^a \bmod p$ has no efficient solution.

DH functions enable users who have had no prior communication to establish a shared secret, like in Transport Layer Security (TLS). The webserver and a user's browser have not had a prior opportunity to exchange secrets and so use DH to configure a symmetric key. Users who do have a long-term shared secret may still wish to use DH to establish session keys between them used for a short term only. This prevents an attacker who obtains the long-term secret at some later point from decrypting prior sessions since the secret used by both parties is never exposed to the communication channel. Care should be taken to protect the exchange of partial secrets since an attacker in a privileged position on the channel may intercept the partial secrets and establish their own secrets with either party, becoming a relay. In TLS the webserver uses certificates signed by a trusted third party, in some other systems the exchange is encrypted with the aforementioned long-term key. The symmetric keys generated by DH enable faster symmetric cryptographic primitives to be used for the actual encryption and decryption of data.

Daniel J. Bernstein introduced Curve25519 and X25519, respectively an elliptic curve and a Diffie-Hellman function using that curve, in 2006 [2]. The X25519 function allows users to exchange 32-byte public keys in order to establish a 32-byte shared secret. It features several practical advantages that make it suitable

for implementation in resource-constrained environments such as small code, small public and private keys, free key validation and a design which lends itself to timing-variance-free implementations. Open source and public domain implementations exist for several platforms (e.g., AVR[14], x86[5], ARM NEON[6]) and it is used by a large list of software including[1] Tor, Apple iOS, SSH and Google Android.

In this thesis we provide an implementation of X25519 for the Cortex-M4 which computes a shared secret in 1816341 cycles and avoids the use of conditional index operations or branches based on secret data. Our implementation has been placed in the public domain and is available online[2]. The code for this implementation fits in a little over 4KiB and uses 628 bytes of memory, making it suitable for inclusion on even the smallest M4 configurations. In order to perform the 256 bit multiplications of Curve25519 we employ reduced refined Karatsuba and reduced radix limbs to represent the field elements, modeled on an ARM NEON implementation of the larger Curve41417 [4]. For this cipher we opt to use the Karatsuba algorithm on the full operands while calculating the three half-size products Karatsuba requires using traditional schoolbook multiplication and combining them using additions and modular reduction. We also investigate the suitability of the less feature-rich, 'smaller' version of this architecture, the Cortex-M3. Its 32 bit multipliers are known to take a variable number of cycles depending on input. Our timing measurements establish credible hypotheses about the code paths and their associated cycle counts for each category of data. Based on this information we conclude that a secure implementation of X25519 for this architecture and using the long multipliers would require a sizeable engineering effort to obtain constant-time behavior, eradicating most or all benefits from using the multipliers.

The remainder of this thesis is organised as follows. Chapter 2 discusses the backgrounds of Curve25519 and X25519, side channel analysis, multiplication of large operands and the Cortex-M series in some detail. Then, Chapter 3 describes the M4 implementation of X25519 in detail. Our timing experiments on the Cortex-M3 are reported in Chapter 4. Performance results for our code are found in Chapter 5, along with comparisons and analysis. Finally, Chapter 6 provides closing remarks and suggestions for future work.

[1] X25519 deployment overview page.
[2] GitHub project page.

Chapter 2

Background

2.1 X25519

Elliptic curve cryptosystems were independently discovered by Koblitz and Miller in 1985 [17, 20]. Since then, a number of different systems have been developed, perhaps the most famous and most widely deployed of which is NSA's Suite B [16]. Many of these existing systems suffer from complexity where implementers must be careful to avoid edge cases, timing leaks and invalid inputs. In order to reduce the strain on implementers and the security risks associated with complex designs, as well as to set new speed records for security, Bernstein introduced Curve25519 [2] in 2006. This is a Montgomery form elliptic curve with equation

$$y^2 = x^3 + 486\,662x^2 + x$$

defined over the finite field $\mathbf{F}_{2^{255}-19}$. This curve serves as the mathematical foundation of the elliptic-curve-Diffie-Hellman key exchange protocol X25519 (originally both the curve and function were known as Curve25519, but following calls for naming clarity Bernstein suggested X25519 be used for the DH function). In this protocol, users generate a 32-byte secret key, apply X25519 with a fixed base point of 9 to obtain a 32-byte public key and finally apply X25519 again with the counter party's public key and their own secret key to establish a shared secret between them after a suitable one-way hashing function is applied to the result.

The protocol works as follows. A user selects a (potentially long-term) 32-byte secret by taking the output of a cryptographically secure random number generator. The only requirements on the secret key, aside from sufficient quality randomness, are that its top order bit (belonging to 2^{255}) and three lowest order bits are set to zero and its second-largest bit is set to one[1]. The requirement for the low order bits is effectively a multiplication by 8 which serves to defend against certain attacks as described below. The second largest bit being set provides a fixed leading one, preventing implementations from trying to optimise by skipping ladder steps until the leading one is encountered since this could lead to attacks–please see below. The user applies the X25519 scalar multiplication function on the base point 9 using the secret key she generated as the scalar. The resulting point is her 32-byte public key. When two users multiply each other's public keys with their own secret keys they establish a shared value which can be used, after applying a suitable key-generation function, as a symmetric shared secret.

The X25519 function takes any 32-byte point P, sets its top bit to zero (compliant implementations will ignore the value of the top bit), and multiplies it by the 32-byte secret scalar s, such that the result Q is $s \cdot P$. The multiplication itself is performed by applying the Montgomery ladder, a constant-time method for elliptic curve point multiplication developed in [22]. This method uses 255 ladder steps, one for each bit in

[1] Although this is formally a requirement on the user, implementations following the IETF draft standards should always ensure secret keys conform to this bit pattern.

the secret key, performing a doubling and a differential addition in each step. Upon completion of the ladder, the result is of the form x/z. The return value $x \cdot z^{p-2}$ is computed by raising z to the power $2^{255} - 21$ and multiplying it by x, yielding the 32-byte public key or shared secret depending on whether the base point or a public key was used as point P. An algorithmic overview of X25519 is recorded in Algorithm 1. In this listing `cswap` is a constant time conditional swap, returning either (a, b) when scalar bit s_t is cleared, or else (b, a). Please see Chapter 3 for implementation details. Algorithm 2 describes the Montgomery ladder in more conceptual terms. One might consider implementing the subscripts in this conceptualised algorithm using array indexes, which is the NIST-recommended way to do so. Please see the section on side channels below for arguments against this use.

Algorithm 1 X25519. Scalar multiplication using Montgomery ladder and inversion to obtain x coordinate of sP. In the following, s_t is the t^{th} bit of the little-endian 255 bit secret scalar s, $p = 2^{255} - 19$ and $a24 = (486\,662 - 2)/4 = 121\,665$.

Input: scalar s, and point P expressed as x-coordinate
Output: sP, expressed as x-coordinate

$x_1 \leftarrow P$, $x_2 \leftarrow 0$, $z_2 \leftarrow 1$, $x_3 \leftarrow P$, $z_3 \leftarrow 1$
for $t = 254$ downto 0 **do**
  $(x_2, x_3) \leftarrow \texttt{cswap}(s_t, x_2, x_3)$
  $(z_2, z_3) \leftarrow \texttt{cswap}(s_t, z_2, z_3)$
  $A \leftarrow x_2 + z_2$
  $AA \leftarrow A^2$
  $B \leftarrow x_2 - z_2$
  $BB \leftarrow B^2$
  $E \leftarrow AA - BB$
  $C \leftarrow x_3 + z_3$
  $D \leftarrow x_3 - z_3$
  $DA \leftarrow D \cdot A$
  $CB \leftarrow C \cdot B$
  $x_3 \leftarrow (DA - CB)^2$
  $z_3 \leftarrow x_1 \cdot (DA - CB)^2$
  $x_2 \leftarrow AA * BB$
  $z_2 \leftarrow E \cdot (AA + a24 \cdot E)$
  $(x_2, x_3) \leftarrow \texttt{cswap}(s_t, x_2, x_3)$
  $(z_2, z_3) \leftarrow \texttt{cswap}(s_t, z_2, z_3)$
end for
return $x_2 \cdot z_2^{p-2}$

There are several advantages to using X25519 over the traditional Diffie-Hellman key-exchange function or other elliptic curves. The elliptic curve discrete logarithm problem is conjectured to be much harder to solve than the traditional discrete logarithm problem in the multiplicative group of a finite field and no known powerful attacks exist against elliptic curve cryptosystems. As a result, X25519 can use small keys to achieve high levels of security. Curve25519 is twist-secure. In Montgomery curves any x-coordinate not belonging to a point on the curve automatically belongs to a point on the quadratic twist of the curve. On either the original or the twisted curve there may be points of a small order b. Should an attacker offer a point with a small order on either curve, the victim may reveal information about their secret key s because the scalar multiplication will have been performed modulo this small order, leaving a small subset of possibilities in some resulting encrypted answer for the attacker to try. For the actual curve this attack fails because the order of the base point is prime therefore no small b exists. For Curve25519's twist, the only small values of b (i.e., below 2^{252}) are 1, 2, 4 and 8. Secret keys, by virtue of having their three low order bits cleared, are multiples of all of these, so that $s \mod b = 0$ for all secret keys s and orders b. In

Algorithm 2 Conceptual algorithm for modular multiplication using the Montgomery ladder.

Input: s, P
Output: sP
  $R_0 \leftarrow 0, R_1 \leftarrow P$
  for $i = 255$ downto 0 **do**
    if $s_i = 0$ **then**
      $R_1 \leftarrow R_0 + R_1$
      $R_0 \leftarrow 2R_0$
    else
      $R_0 \leftarrow R_0 + R_1$
      $R_1 \leftarrow 2R_1$
    end if
  end for
  return R_0

practice this means X25519 is not vulnerable to small-subgroup attacks on either Curve25519 or its "twist" group. As a result public keys do not need to be validated, all values may be accepted. Finally, X25519 does not need to perform any branching or array indexing based on secret data. The secret key is used bit by bit to determine the mask in the conditional swap only. This procedure can be carried out by performing arithmetic irrespective of the bit's value. The remainder of the algorithm is invariant under the key's value as well. This design eliminates entire classes of side channel attacks as described below.

2.2 Polynomial multiplication

Multiplying 256 bit operands is the chief operation in X25519, taking up by far the most code and time. In order to compute products of 256 bit operands we will use polynomial multiplication. In this method one splits an integer x into terms so that

$$x = \sum_{i=0}^{n-1} x_i \cdot r^i$$

where r is the radix of the terms, $0 \leq x_i < r$ (or $-\frac{r}{2} < x_i < \frac{r}{2}$ for signed terms) are the terms and n is the number of digits the integer requires in that radix. A traditional or schoolbook multiplication of two integers represented by polynomials takes the terms of each operand, multiplies them pairwise and sums up terms at matching powers of r. The results usually do not fit into integers of radix r, so a carry chain is needed to make them fit again. A multiplication of this form runs in $\mathrm{O}(n^2)$ time complexity. Here follows an example. In base 10, multiplying 1234 by 5678 would look like this. First the integer is split into terms:

$$(10^0 \cdot 4 + 10^1 \cdot 3 + 10^2 \cdot 2 + 10^3 \cdot 1)(10^0 \cdot 8 + 10^1 \cdot 7 + 10^2 \cdot 6 + 10^3 \cdot 5)$$

Then, the individual terms are multiplied:

$$\begin{aligned}
&10^0 \cdot 32 + 10^1 \cdot 28 + 10^2 \cdot 24 + 10^3 \cdot 20 + \\
&10^1 \cdot 24 + 10^2 \cdot 21 + 10^3 \cdot 18 + 10^4 \cdot 15 + \\
&10^2 \cdot 16 + 10^3 \cdot 14 + 10^4 \cdot 12 + 10^5 \cdot 10 + \\
&10^3 \cdot 08 + 10^4 \cdot 07 + 10^5 \cdot 06 + 10^6 \cdot 05
\end{aligned}$$

This yields the product, but in practice one stores the intermediate results in terms without the order of magnitude attached. Here the intermediate products are arranged, where the orders would be kept track of

implicitly:

$$\begin{aligned}
10^0 \cdot 32 &= 10^0 \cdot 32 \\
10^1 \cdot (28 + 24) &= 10^1 \cdot 52 \\
10^2 \cdot (24 + 21 + 16) &= 10^2 \cdot 61 \\
10^3 \cdot (20 + 18 + 14 + 8) &= 10^3 \cdot 60 \\
10^4 \cdot (15 + 12 + 7) &= 10^4 \cdot 34 \\
10^5 \cdot (10 + 6) &= 10^5 \cdot 16 \\
10^6 \cdot 5 &= 10^6 \cdot 5
\end{aligned}$$

The product is usable like this for additional computation in polynomial form, can be reduced mod p (when working in prime fields) to fit back into 4 terms, or recombined into a single integer. To do so, we split the terms on mod 10 boundaries. Since, e.g., $32 > 10$ we split it into 32 mod 10 = 2 and 32 div 10 = 3, carrying 3 over to the next integer to get $52 + 3 = 55$. Carrying the terms like this amounts to an aligned addition:

$$\begin{array}{r}
32 \\
52 \\
61 \\
{}_{1}60 \\
{}_{1}34 \\
16 \\
5 \\
\hline
7006652
\end{array}$$

Using Karatsuba's algorithm [15], asymptotic complexity may be reduced to $O(n^{\log_2 3})$. With Karatsuba an integer F of radix r with n digits, where n is assumed to be even, is split into two parts F_0 and $r^{n/2}F_1$ and multiplied as follows:

$$\begin{aligned}
&(F_0 + r^{n/2}F_1)(G_0 + r^{n/2}G_1) = \\
&\qquad F_0G_0 + r^{n/2}((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1) + r^n F_1G_1
\end{aligned}$$

In [3], Bernstein introduces refined Karatsuba

$$\begin{aligned}
&(F_0 + r^{n/2}F_1)(G_0 + r^{n/2}G_1) = \\
&\qquad (1 - r^{n/2})(F_0G_0 - r^{n/2}F_1G_1) + r^{n/2}(F_0 + F_1)(G_0 + G_1)
\end{aligned}$$

which still costs three multiplications but cuts down on the number of additions/subtractions. This form is the basis of our 256 bit multiplier. However, since the multiplications in X25519 are all computed modulo $2^{255} - 19$, we use "reduced refined Karatsuba" introduced in [4] to save on additions by reducing the size of partial products. Reductions remove multiples of p since any value larger than p is congruent to some value in the range $[0, p-1] \mod p$. Doing so means one cuts down on the size of intermediate and output integers. Chapter 3 contains a more thorough treatment of our implementation of reduced refined Karatsuba.

2.3 Side Channel Analysis

Security does not follow solely from the mathematical properties of a cryptosystem and the functional correctness of its implementations. Side channel analysis refers to a type of attack where the object is not to

break or subvert the mathematical properties of a cryptosystem, but rather to observe physical phenomena occurring as a side effect of computations in order to glean information about the secret data being computed. Various such side effects have been exploited over the years. In this section we discuss two categories of side channel attacks that are common and relevant to our implementation.

The earliest public attack using side channels is based on execution time, as explored originally by Kocher in 1996 [19]. With timing analysis, a cryptographic implementation is attacked that produces results in variable amounts of time or cycles depending on its input. When this timing variance depends on secret data, attackers are able to gain information about the secret data by measuring the time until a response is received. Timing attacks work against general implementations and even over networks against, e.g., TLS web servers [8]. More recent timing analysis also successfully attacks a vulnerable Montgomery ladder implementation because this ladder uses a loop to detect leading zeros in the scalar and adjusts its iterations accordingly [7]. A related concept is that of cache attacks. Seemingly constant-time operations can still take variable amounts of time based on the availability of required data in CPU caches. For instance, an array index operation is often assumed to be constant time but will complete more quickly when the desired element is in cache than when it must be retrieved from RAM. Bernstein puts it plainly in [1] where he introduces a cache-timing attack on AES: "Using secret data as an array index is a recipe for disaster." Defending against timing attacks is feasible when the design of the cryptosystem takes these problems into account. Some systems, like AES, are fundamentally difficult to implement securely due to their designs. Several suggestions exist [25, 26] but the main objections are always that performance suffers terribly and certain countermeasures are specific to the cryptographic primitive and/or hardware architecture. The best way to defend against these attacks is to design a cryptosystem that does not depend on table lookups or branching based on secret data, and not to implement the system to use these methods. X25519 is designed to avoid this class of attacks.

Power analysis, first explored by Kocher, Jaffe and Jun in 1999 [18], employs a hardware leakage model based on the content of data, rather than relying on the way code handles those data to leak information. The most basic attack, Simple Power Analysis (SPA), uses discernible differences in instantaneous power consumption of the computing device. When a conditional branch is taken, the power trace looks visibly different from a trace where the branch is not taken. Defence against SPA is again to avoid secret data-dependent code paths. More advanced is Differential Power Analysis (DPA). This attack assumes that the data content influences instantaneous power consumption when that data is under computation (Kocher's work uses a Hamming weight-based leakage model), and applies divide-and-conquer to statistically glean secret key data on a per-'S-box'[2] basis using many traces of different known plain- or ciphertexts. Data-based information leakage is a 'feature' of hardware. Attacks of this kind cannot be defended against by using a set of best practices like with timing-based analysis. Rather, to prevent adversaries from being able to correlate secret data to supplied inputs and observed physical characteristics of computation, the best practice is for implementations to introduce random elements when operating on secret data. An easy to implement method to randomise computation is to add some multiple k of the group order ℓ to the secret scalar s [10]. The scalar multiplication is unchanged mathematically, because multiplying P by the group order results in the neutral element, so $(s+k\cdot\ell)P = sP + k\cdot\ell P = sP + \mathcal{O} = sP$. The sequence of bits in the scalar does change, however, so a different sequence of computations is carried out each time. Research in the field of power analysis is ongoing. Quickly after the above method was invented researchers showed improved defenses and attacks, e.g. [23, 13]. Other attacks like the template attack have also been developed, where attackers work around iteration limits by first building a device profile offline and then targeting the victim with one or very few power traces to compare against the profile [9]. Since this type of attack uses hardware aspects of computation, X25519 cannot design against it. Our implementation does not implement any of the proposed DPA countermeasures for elliptic curve but could be adapted to do so.

[2]In AES and other block ciphers, the S-box is a function to add nonlinearity to a round function where m bits of input are replaced by n bits of linearly uncorrelated output, providing confusion [24]. S-boxes must be designed to resist cryptanalysis [21].

Name	Description	M3	M4
MUL	"Small" multiplier. Gives the least significant word of the product Available on M0 and higher.	1	1
MLA	"Small" multiply and accumulate. Least significant word of 32 bit products, added to another word.	2	2
SMULL	Signed 32 bit multiplier.	3-5	1
UMULL	Unsigned 32 bit multiplier.	3-5	1
SMLAL	Signed 32 bit multiplier with 64 bit accumulate.	4-7	1
UMLAL	Unsigned 32 bit multiplier with 64 bit accumulate.	4-7	1

Table 2.1: Multiplier instruction summary for M3 and M4. The columns **M3** and **M4** list cycles counts for the respective architectures as specified in their Technical Reference Manuals.

2.4 Cortex-M

Our implementation is designed for the ARM Cortex-M4. The Cortex-M range is a series of microcontroller designs, currently consisting of the M0, M0+, M1, M3, M4 and M7 ranging from the smallest, most low power and cheapest design to the most powerful one. All of them feature 14 general purpose registers not counting the Stack Pointer and Program Counter and their instruction sets expand going up in the range. At the low end, the M0 and M0+ (which is an optimised version of M0 with the same instruction set) lack most instructions that do not modify the status register (containing flags to indicate the result of an instruction produced a carry, was negative, etc.) and can only operate on half their registers. The M7 at the highest end has a longer pipeline, DSP instructions and fast hardware divide. Code is upwards binary compatible between the different M-series architectures.

At least one implementation of X25519 already exists for the M0 [12]. Due to the platform's limitations it is forced to use the only available multiplication instruction, which takes 32 bit inputs but returns only the low 32 bits of the result as a $16 \cdot 16 \rightarrow 32$ bit multiplier. It uses three levels of Karatsuba, splitting the 256 bit integers three times and using Karatsuba at every split, to work its way up to 256 bit multiplication. The Cortex M3 and higher feature a collection of 'long' multiplication instructions which yield the full 64 bit product of two 32 bit operands. It also has accumulating versions that take the existing value of the output registers and adds the product to them instead of overwriting their contents. Table 2.1 lists the instructions relevant to our implementation, along with the number of cycles they require. Several other instructions exist for extracting bytes and halfwords from words and multiplying them but those are not used in our implementation. Consult the Technical Reference Manuals (TRMs) for M3 and M4[3] for a complete listing.

Note the variable cycle counts on M3 for the multiplication instructions. The counts for M3 and M4 are copied directly from their respective TRMs. The Manual for M3 includes a note saying these instructions "use early termination depending on the size of the source values". No more specific information is provided. The Cortex-M has a load/store architecture. Each load/store instruction takes $1 + n$ cycles, where n is the number of words retrieved from memory. Instructions exist to fetch a single (`LDR` and `STR`), two (`LDRD` and `STRD`) or $1 \leq n \leq 15$ words at once (`LDM` and `STM`), including the Program Counter. Since the 1-cycle penalty for a load or store is paid per instruction rather than per transferred word, grouping loads and stores together as much as possible reduces the number of overhead cycles. Contrary to the M0(+), all registers may be accessed by instructions on the M3 and M4, providing greater flexibility to process data in batches.

[3] ARM® Cortex®-M3 Processor Technical Reference Manual, r2p1, table 3-1 and ARM® Cortex®-M4 Processor Technical Reference Manual, r0p0, table 3-1.

Chapter 3

Implementation

The Cortex-M4 features $32 \cdot 32 \rightarrow 64$ bit multiplication instructions guaranteed to complete in a single cycle. The presence of these fast, constant-time multiplication circuits means we should be able to obtain better performance characteristics than existing M0 code. Both unsigned and signed versions of the multipliers are provided. As such, we chose to implement the scalar multiplication using signed limbs, avoiding the need to add p before subtractions to guarantee nonnegative results. In this chapter we describe the algorithmic steps needed to perform a $256 \cdot 256$ multiplication modulo $2^{255} - 19$ as well as the architectural details of the implementation.

Since the Cortex-M has one load/store channel for both instructions and data and any data interaction with RAM, either heap or stack, incurs overhead of one cycle, we try and keep as much intermediate data in memory while batching the reads and writes as much as possible. The overhead applies per instruction, not per byte or word, so it is beneficial to use the `LDM` and `STM` instructions as much as possible for as many words at once as possible. Similarly the instructions `push` and `pop` can move some or all of the registers onto or off of the stack at once, although our code tends to interact with the stack using the stack pointer directly. We either modify the stack pointer directly to allocate and address memory or use the offset capabilities of the `STR/LDR` and `LDRD/STRD` instructions. These relative offsets can be specified as immediate values and incur no extra overhead to the load or store, allowing quick interaction with the stack. Interaction with the heap would entail either keeping a pointer in registers, increasing register pressure, or loading a pointer every time memory transfers are required, increasing code sizes and cycle counts. Some memory is kept on the heap, however. The main `scalarmult` function requires 312 bytes of temporary RAM storage for the input, both 'ladder points' and two temporary field elements, and `invert` uses an additional 160 bytes for temporary storage of four points. We opt to use the heap for this memory to keep the stack from growing too large. In order to offset the extra pointer analysis required we keep their pointers in registers throughout the functions and copy them into the argument registers for internal functions as needed.

One of the interesting features of the ARM architecture is its use of an inline barrel shifter. The second operand to many instructions can be shifted arithmetically or logically, or rotated as it enters the ALU. This is done by dedicated hardware which means there is no penalty for doing so. The barrel shifter proves useful when it comes to the carry chains compressing 64 bit limbs back into their reduced radix representation (see below) in the multiplication, squaring and multiplication by curve constant functions. They are also helpful when unpacking points from byte representation into limbs or packing them back into bytes. The existence of this inline barrel shifter not only saves cycles, it also reduces the need for extra temporary data registers in some cases and removes the need for some pre- or postprocessing instructions, shrinking the required code size.

A benefit of using assembly language over raw opcodes (besides programmer productivity) is that the assembler is able to take care of most of the memory layout issues for the programmer. One does not need

to know where exactly functions are stored or similar complications. This benefit should extend to code and data alignment issues, too. The processor is more efficient when reading and writing memory that is aligned to word or doubleword boundaries than when transferring unaligned memory. After performing a small optimisation in the multiplier we noticed, however, that cycle counts were not affected and discovered the need to explicitly instruct the assembler to align the instructions on a doubleword boundary. Alternatively, inserting an instruction that performs no operation, `NOP`, in a specific location takes care of the issue without costing cycles itself. The assembler should be able to choose instruction encodings and code locations to ensure the best timing, but hand-optimising can yield–in this case comparatively minuscule–benefits.

For our implementation of the 256 bit multiplier we chose to use a reduced-radix representation to avoid having to perform most carries. Instead of representing 256 bit field elements in 8 limbs using all 32 bits of each processor word we use 10 limbs, each containing 26 or 25 bits. Using all bits in a word for each limb would mean that additions of 32 bit limbs (or, after a multiplication of two limbs, 64 bit limbs) would cause overflows which would need to be carried over to the next limb, which again might cause an overflow to be carried, and so on. With a reduced-radix representation the spare bits can be used as buffer space for these additions. Multiplication of field elements uses many additions so this saves significant overhead–albeit at the cost of having to process two more limbs per element. An element is split as follows:

$$F = f_0 + 2^{26} f_1 + 2^{52} f_2 + 2^{77} f_3 + 2^{103} f_4 + 2^{128} f_5 + 2^{154} f_6 + 2^{180} f_7 + 2^{205} f_8 + 2^{231} f_9$$

Note that limbs 0, 1, 3, 5, 6 and 8 hold 26 bits whereas limbs 2, 4, 7 and 9 hold 25. Upon these 10 limbs we apply reduced refined Karatsuba. To do so, we must split a pair of operands F and G into $(F_0 + 2^{128} F_1)$ and $(G_0 + 2^{128} G_1)$ and apply the function listed in Section 2.2. The refined Karatsuba function can be thought of as three separate sub-operations. First, compute $F_0 G_0 - 2^{128} F_1 G_1$. Then multiply this term with $(1 - 2^{128})$, i.e., compute $(F_0 G_0 - 2^{128} F_1 G_1) - 2^{128}(F_0 G_0 - 2^{128} F_1 G_1)$. The respective limbs of these terms align, so that this computation comes down to a series of 64 bit subtractions. The final step is to add $2^{128}(F_0 + F_1)(G_0 + G_1)$ to the just-computed term. This again is achieved by a series of 64 bit subtractions and additions. Reductions are performed after the first and third step and the result is then completely reduced mod $2^{255} - 19$. For reductions the limbs higher than the 2^{256} boundary are added to the low limbs. Because calculations are performed modulo p, these high limbs can be reinterpreted:

$$2^{256} f_{10} = 2^1 2^{255} f_{10} = 2^1 \cdot 19 f_{10} = 38 f_{10}$$

and similarly for f_{11} through f_{14}. Due to this reinterpretation it is clear that adding these high limbs multiplied by 38 to their aligned low limbs achieves the reduction. At this point the limbs are still contained in 64 bit doublewords. A complete reduction not only folds these 'extra' high limbs onto their corresponding low limbs but additionally compresses the limbs back into 26 or 25 bits using a carry chain.

The sub-products $F_0 G_0$, $F_1 G_1$ and $(F_0 + F_1)(G_0 + G_1)$ are computed using the schoolbook method as outlined in Section 2.2. For $F_0 G_0$ the five limbs of each operands are loaded into the ten lowest registers, leaving four registers to compute two of the nine 64 bit products. After two limbs `STM` is used to free the registers. The pattern holds for the second and third pair at which point limbs 0 through 5 of the product have been written to the stack. Some of the source operands are not needed to multiply the last three limbs. This frees up two registers and allows us to compute these last three in one batch that can be written out to memory with a single `STM` instruction. For $F_1 G_1$ we do not repeat the above procedure but combine it with the first Karatsuba step. The first ten registers are again occupied with the source operands but instead of using the remaining registers to compute two limbs at once a single product is computed. Then the matching limb of $F_0 G_0$ is loaded and the fresh product is subtracted from it. For the limbs 5 through 8 the limbs are first multiplied by 38 before subtracting to achieve the reduction inline. Limb 4 of the shifted term needs to be subtracted from zero. Instead of carrying this out explicitly the operation is done implicitly and subsequent additions and subtractions involving this limb are changed accordingly, saving a bit of work.

The stack now contains the intermediate result $F_0G_0 - 2^{128}F_1G_1$. Subtracting the 128 bit shifted version of this from itself is quick, being not much more than a series of loads, subtractions and stores. As per the general strategy, this sequence is batched as much as possible. Computing the third schoolbook term is combined with the last Karatsuba step, similarly to before. First the terms $F_0 + F_1$ and $G_0 + G_1$ are computed by 32 bit addition. The schoolbook products are then computed and immediately processed into the intermediate result limb by limb. As before, reducing the intermediate result is also included in this processing step. In RAM is now the partially reduced product of the full 256 bit multiplication. It is then completely reduced into 26 or 25 bits limbs by means of a carry chain. This chain takes the bits outside the radix from each limb starting at the lowest order limb, shifts them down and adds them to the next higher limb. For the top limb the excess is multiplied by 38 and added it to the bottom limb and finally the bottom limb's excess is carried into the second limb. The full chain is from $limb_0$ to $limb_1$ to ... to $limb_9$, from $limb_9$ to $limb_0$ with multiplication by 38, and finally from $limb_0$ to $limb_1$. Although the value now fits inside the limbs it is not a unique representation since all values up to $2^{256} - 1$ can be expressed, not only up to p. When the limbs are rearranged into a packed 32 byte representation at the end of the computation the result is uniquely reduced modulo p.

- *Squaring* (`sqr`) follows the same strategy as multiplication except it benefits from reduced register pressure. There are fewer source limbs to compute allowing more data to be processed before memory transfers are required.
- *Multiplication by the curve constant* (`mul121665`) is easier to implement. Its input consists of a single element of 10 limbs of 26 or 25 bits. Along with the curve constant and temporary storage words these fit in the processor's registers. The only memory interactions are to load the element at the start of multiplication and to store it at the end. The actual multiplication is interleaved with the carry chain in order not to have to keep track of an intermediate field element consisting of 64 bit limbs.
- *Addition* (`add`) and *subtraction* (`sub`) are some of the smallest functions. Both operands' limbs are loaded into registers, added and written to output in two batches.
- *Reading packed bytes* (`from_bytes`) and rearranging them into reduced radix elements is similarly uncomplicated, stretching 8 words of 32 bits into 10 limbs of 26 or 25 bits using the barrel shifter.
- *Writing to packed bytes* (`to_bytes`) after scalar multiplication is a bit more involved than unpacking as it carries out the final reduction before packing the reduced radix limbs back into bytes. This reduction involves moving a carry across all of the limbs to collect any excess bits. The carry is multiplied by 38 and added to the bottom limb. After that a final carry chain is performed which is simpler than the other chains because it carries from 32 bit words instead of 64 bit ones. Further, packing limbs into bytes discards excess bits altogether, obviating the need to explicitly clear them. As such, this carry chain is nothing more than a series of additions making use of the inline barrel shifter to shift down the excess bits before adding them to the higher limbs.

Finally, we will outline the *conditional swap* (`cswap`). The Montgomery ladder performs a differential addition and a point doubling for every bit in the scalar, as listed in Algorithm 2. It differentiates based on the value of each bit, either doubling the first point and performing an addition of both points for the second or performing the addition for the first and doubling the second. This differentiated computation must avoid branching, index or other timing-sensitive operations. To do so, the ladder uses a function to conditionally swap the two points, allowing the point addition and doubling code path not to have to consider the two cases. Two acceptable methods are suggested in an early draft standard for X25519[1] but the current draft[2] only allows a single method. The first uses the difference between the two points, multiplies it with the

[1] X25519 'draft-turner' at GitHub
[2] ietf.org

secret bit and finally subtracts the difference from one point while adding it to the other. This method is displayed in Algorithm 3. The second method, which is the remaining method in the later draft, uses bitwise operations and is the one we implement. In this method the secret bit is negated, creating a mask of either all zeros or all ones. This mask is then ANDed with the XOR of the two points A and B. The resulting dummy is then XORed with each point in turn. If the bit is set the dummy is $A \oplus B$, otherwise it is zero. After XORing point A with the dummy it is either $A \oplus (A \oplus B) = B$ or $A \oplus 0 = A$. This method is also listed as Algorithm 4. The swap operation depends fully on the value of the secret bit but the series of bitwise operations used to implement it does not. Note that both algorithms XOR the current scalar bit with the previous one. This is done so that the actual swap is only performed when the bit is different from the previous one, preserving swap state between ladder steps. Not doing so would mean having to 'reset' the points to their unswapped state after computation, necessitating an extra conditional swap per ladder step. Note that Algorithms 3 and 4 differ from the function in 1 by including s_{t-1}. In the actual implementation the XOR between s_t and s_{t-1} is handled by the top-level `scalarmult` function which means `cswap` itself still works with only one bit.

Algorithm 3 Conditional swap using addition and subtraction.

Input: Points A and B, secret bits s_t and s_{t-1}.
Output: (B, A) if $s_t = 1$, else (A, B)
dummy $= (s_t \oplus s_{t-1}) \cdot (A - B)$
$A = A -$ dummy
$B = B +$ dummy

Algorithm 4 Conditional swap using bitwise operations.

Input: Points A and B, secret bits s_t and s_{t-1}.
Output: (B, A) if $s_t = 1$, else (A, B)
mask $= 0 - (s_t \oplus s_{t-1})$
dummy $=$ mask AND $(A \oplus B)$
$A = A \oplus$ dummy
$B = B \oplus$ dummy

As referenced in the Background (Section 2.3), care must be taken to defend cryptographic implementations against a variety of side channel attacks. X25519 is designed with these attacks in mind, making it easy for implementators to construct timing vulnerability-free code. Correspondingly, our implementation is free of branches or indices based on secret data. The secret scalar is read in its entirety to a local copy where its top and bottom bits are set to their requisite values using bitwise operations. All accesses to secret data are predictable. Each ladderstep accesses one of the bits in succession from the most to the least significant bit. Intermediate data is multiplied, squared, added or subtracted independent of the value of these secret bits and the conditional swap, as outlined above, similarly performs its code path independent of its secret swap bit. Timing attacks are therefore not possible against our implementation. By contrast, no efforts are made to defend against power analysis or any other emanations analysis. While this opens the implementation to a broad range of attacks, it also means no randomness in the implementation is required. This implementation can be extended without much difficulty to add a random multiple of p to the public point before multiplying it by the secret scalar in order to defend against power analysis, among several possibilities [10]. However, this is only possible in the presence of a random number generator.

Even though 26 or 25 bits are easily contained in a 32 bit integer and the products of two such limbs similarly fit within 64 bit integers, the issue of overflow must be addressed. The 256 bit multiplier performs several additions and multiplications on limbs of finite width so the risk exists that eventually one or more limbs overflow. In order to investigate this risk we created a Python facsimile of the squaring function which

uses the same algorithm as the multiplier but lets us work with fewer operands. It performs all of the steps the squaring does, but adds printouts of the base 2 logs of the limbs. We manually supply the content of operands. Python integers 'have unlimited precision'[3] making them suitable to check whether their contents exceed 64 bits. In our tests we use 26.5 bit limbs instead of 25.5 bits because the operands to the squaring or multiplication are sometimes sums of two 25.5 bit field elements. The second limb would be 27 bits but we set it to 28 since the carry chain deposits an extra bit in this limb making 28 bits the largest possible bit length. We tested both with all of the bits up to the bit size set or only the top bit. In all cases we observe that during the entire multiplication no limbs grow larger than 63 bits, leaving one bit of buffer space. The algorithm uses only multiplications and additions where maximal outputs are obtained from maximal inputs (or minimal outputs for minimal inputs), so the 63 bits per limb are an upper bound. We prove that the carry chain does not overflow either by reasoning with the 63 bit upper bound rather than the observations from the script which are lower for most limbs (63 for the first limb, 62 for the next five, then 61, 60, 59 and 58). The carry chain takes the excess bits in a limb, shifts them down to be the low order bits and adds the shifted bits to the next limb. They are then shifted back and subtracted from the former limb. This series of operations reduces the width of the current word to 26 or 25 bits. The limb to which the bits are added becomes at most 64 bits since we observed that it contains at most 63 and at most 64 - 25 = 39 bits are added to it. The limb from which the excess bits are subtracted is now at most 26 bits. The top limb's excess bits are multiplied by 38 and added to the bottom limb. This means that 45 bits are added to the bottom limb ($\lceil \log_2(38) \rceil = 6$), making the latter 46 bits. The final carry takes the top 20 bits and adds them to the second limb. It already contained 26 bits, making the product 27. At this point the element is contained in 25.5 bit limbs with the second limb at most one bit larger. Since neither during the multiplication nor during the carry chain the intermediate results grow too large to be contained in 64 bit doublewords we conclude that overflows do not occur.

[3]Python 3.4 built-in types documentation page.

Chapter 4

Cortex-M3 instruction timing

Our experiments focus on `UMULL` and `UMLAL`, the $32 \cdot 32 \rightarrow 64$ bit unsigned multiplier and multiply-accumulate instructions, as well as `SMULL` and `SMLAL`, their signed-integer counterparts. We will first describe the experimental setup and then discuss the results for the unsigned and signed results, respectively.

4.1 Experimental setup

In order to understand how the unsigned multiplication instructions behave we created a small on-device program to receive two operands via one of the device's hardware UARTs, carry out the instruction under test and return the observed cycle counts for that instruction. The testing function in the program consists of 100 invocations of the instruction, followed by another 100 invocations where the source and destination registers are the same interleaved with move instructions to reset the operands to their UART-received values. These 200 instructions and the accompanying move instructions are placed inside a loop, to be carried out 100 times. Cycle counts are obtained using the Cortex-M's onboard cycle counter `SysTick`[1] which is accessible by code running on the device. Both the multiplier and multiply-accumulate instructions are tested, one after the other, for the same two operands. As such, two measurements are returned per pair of operands.

The operands in question are generated in a script running on a computer connected to the devices via UART. This Python script generates two random numbers per iteration using a wrapper[2] to Intel's RdRand[3]. It then sends them over to both devices and collects the returned measurements. The script iterates through all bit lengths for either operand, testing several hundred instances per combination of operand lengths. In addition, a separate script is used to test the cases where either or both operands are an exact power of two.

The signed and unsigned versions of the experiments are largely the same. On the device the code is changed to use the `S` or `U` versions of the multipliers. For the computer side a small change is made to properly account for the bitsize of negative numbers. We record the number of bits in a negative integer by taking the base two log of its absolute value, rather than the value itself. Not only is the logarithm of negative numbers undefined, but the multiplier also differentiates between negative numbers of 32 bits and negative numbers requiring less space, even though their top bit is set per definition.

[1] ARM Keil support page with `SysTick` primer.
[2] Python Package Index page for 'rdrand' wrapper.
[3] Intel Digital Random Number Generator (DRNG) Software Implementation Guide. Please note that cryptographic use of this DRNG is not recommended.

For the timing experiment we used two devices. The first is the STMicroelectronics STM32L100C-DISCOVERY[4], a low-cost development and prototyping board based on the STM32L100RCT6 microcontroller, a device with 256kB of storage, 16kB of RAM and running at 32MHz. The other is the Arduino Due[5], a third-party development board targeted primarily at hobbyists and based on Atmel's SAM3X8E microcontroller. This controller has 512kB of storage space and 96kB of RAM, supporting an operating frequency of 84MHz. For programming the first board we use Keil's μVision IDE. We programmed the Arduino using its default IDE, implementing the above test function with inline assembly.

4.2 Unsigned multiply and multiply-accumulate

The first observation is that both devices performed identically for every pair of operands tested. In addition, the experiments show that `UMLAL` does not take 4 to 7 cycles as the TRM reports, but 3 to 7 cycles. Further, the results from the testing scripts demonstrate that behavior is not dependent only on operand size but has special conditions for powers of two. Zero is another special case: whenever one or both of the operands is zero, the instruction takes the lowest rated number of cycles. In essence, there are five categories into which one can divide operands. Large (i.e., $\geq 2^{16}$), large power of two, small ($< 2^{16}$), small power of two and zero. In addition, operands which would otherwise be considered large but which only have bits set in the upper 16 bits region behave like powers of two. We call these 'topheavy', and combine them with powers of two into a 'special' category.

The flow charts to determine the expected number of cycles to execute an instruction, based on our observations, are drawn in Figures 4.1 and 4.2 for `UMULL` and `UMLAL`, respectively. There are two main flows, the first of which is the main case. Here, the instruction takes its base number of cycles and adds one or two cycles, depending on the instruction, for each operand that exceeds 16 bits. The other flow consists of 'special' integers, complicating the overall picture significantly. Moreover, `UMULL` and `UMLAL` differ in the way they deal with the presence of the class of special integers. We do not observe any asymmetry in the handling of categories: The pair of operands may be supplied to the instruction in either order. Only the combination of categories matters. In addition, the cycle counts in our observations are fully deterministic, there has not been a single observation of an operation with a particular set of operands requiring more of fewer cycles in different iterations. Alternatively, differences in cycle counts could all happen to cancel out over all the iterations in the test, but our pilot setup using a single instruction as testing function did not suggest this to be the case. Note that these charts are based on our observation and are not based on ARM's design documents. We have tried to obtain information from ARM but were informed that for this kind of architectural information we would require a hardware license from the company which we could not acquire for the purposes of our research.

It is possible, and not unreasonable to speculate, that a multiplication of two 'small' operands would cause the instructions to fall back to `MUL` since its result would fit in the low 32 bits of the result. If so, and given that the small multiplier requires a single cycle, this could mean that the wide multipliers spend their first two cycles determining the categories of their operands and selecting the appropriate circuits for computation. The nonstandard cycle counts for powers of two, especially `UMULL`s low count for two large powers of two, could indicate operands are transformed into a more efficient internal representation for quicker multiplication. The observation that topheavy integers behave like powers of two indicates that operands are indeed transformed into a more efficient representation wherever possible to process them more quickly.

[4]STMicroelectronics STM32L100C-DISCOVERY product page.
[5]Arduino Due product page.

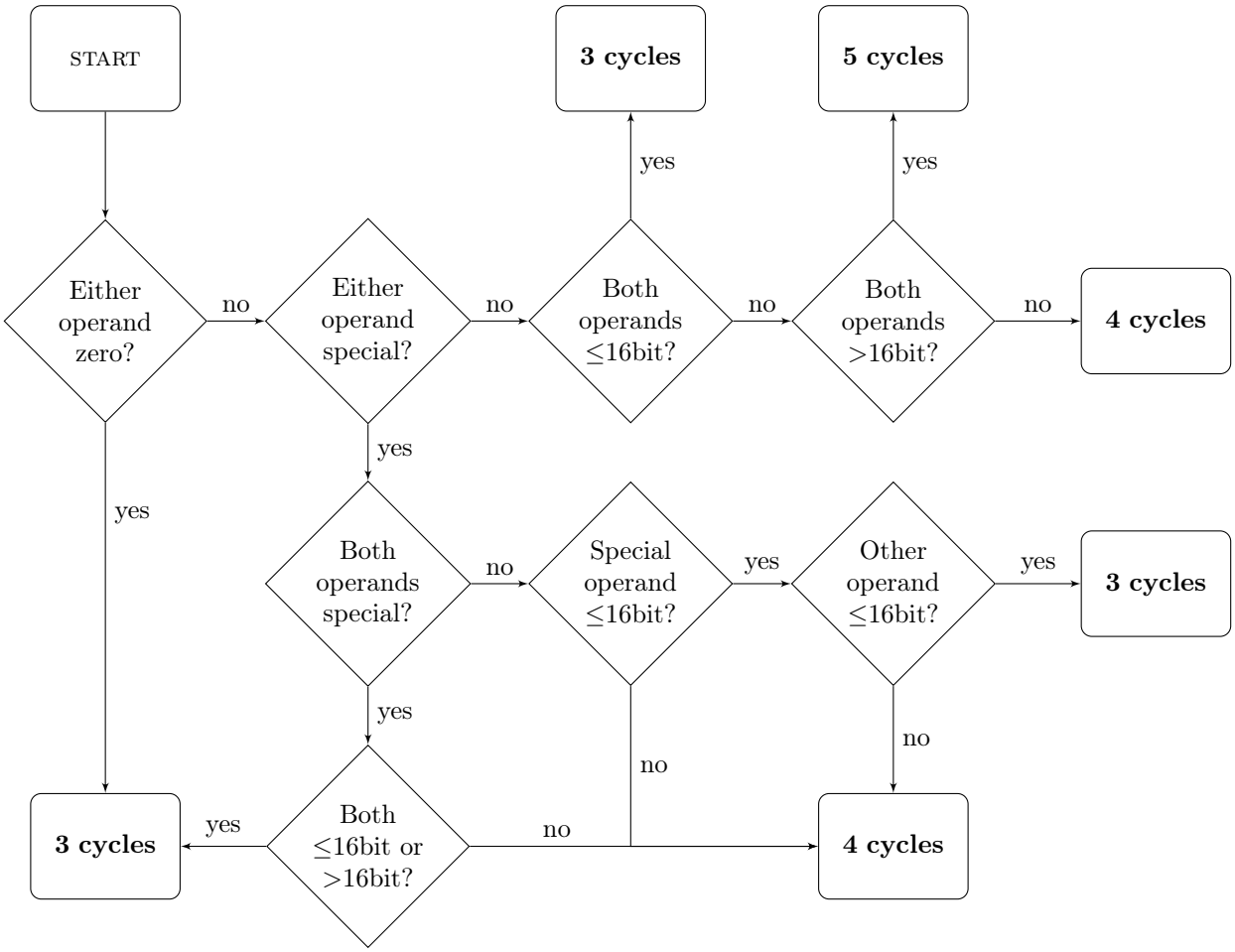

Figure 4.1: Flow chart describing the number of cycles required for `UMULL` execution. 'Either' means at least one of the multiplied operands satisfies the condition without left or right hand side distinction. 'Special' refers to power of two and topheavy operands.

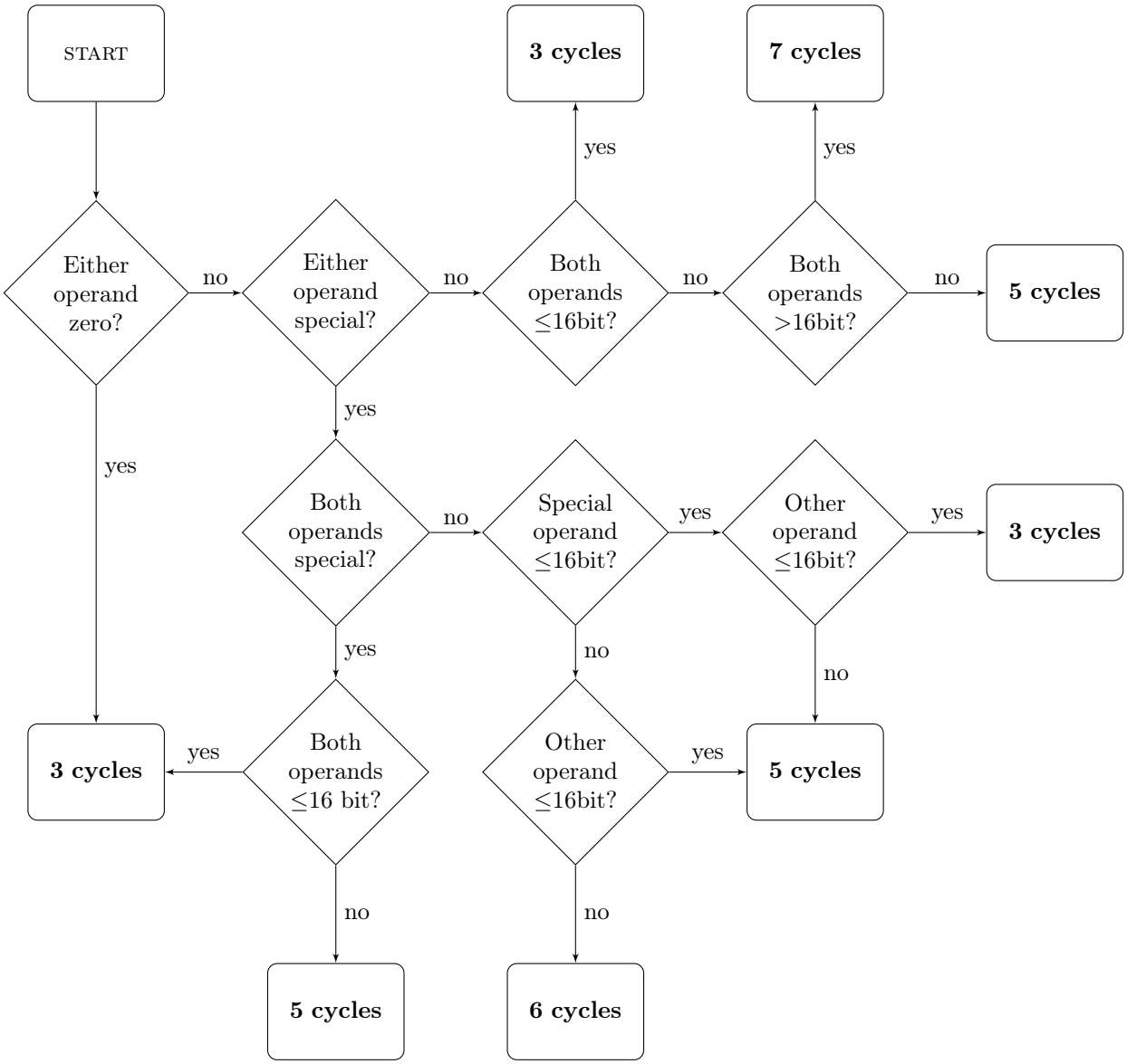

Figure 4.2: Flow chart detailing the behavior of the `UMLAL` instruction. 'Either' means at least one of the multiplied operands satisfies the condition without left or right hand side distinction. 'Special' refers to power of two and topheavy operands.

4.3 Signed multiply and multiply-accumulate

Negative numbers on the Cortex-M are represented using two's complement notation. In signed integer interpretation, the top bit represents the value $-(2^{32}-1)$ while all other bits retain their positive value. Thus, any negative number has its top bit set. An initial hypothesis might be that therefore, all negative numbers are treated as large operands by the signed multipliers. In general the timing characteristics follow the same pattern as the unsigned instructions: 3-5 cycles for `SMULL` and 3-7 cycles for `SMLAL`.

Measurements suggest that our initial hypothesis is incorrect. First, we observe an increase in the number of categories of numbers: 32 bit negative, <32 bit negative, small positive, large < 32 bit positive, 32 bit positive. As with the `U`-versions, topheavy operands are treated as large powers of two. Negative numbers can also be topheavy and are still treated as powers of two in terms of cycle counts. This expansion of categories in part flows naturally from the extension to negative numbers, but note that numbers that actually require 32 bits are treated differently. This can be seen in both negative numbers and positive numbers as well as the specific integer 2^{31}. Sometimes this "32 bitness" results in saving a cycle ("positive small" times "negative 32 bit" takes 5 cycles for `SMLAL` as opposed to "positive small" times "<32 bit negative" taking 6 cycles), and other times costs an extra cycle. In addition to the increase in categories of integers, an extra complication is introduced. The cycle count is not determined solely by the combination of categories but by the order of categories as well. For example, "negative" times "small" on `SMLAL` takes 5 cycles, "small" times "negative" costs 6.

Rather than presenting a flow chart, we aggregate the observations for the signed instructions in Table 4.1. Combinations of categories are grouped by their associated cycle count and combinations sharing categories are grouped in rows. The table could be represented with fewer categories listed by combining them wherever possible (e.g., the category for small positive integers contains the one for small powers of two). We have opted not to do so in order to make it easier to distinguish visually where categories do and do not overlap. The general pattern of a base number of cycles expanding as operands grow holds true for these instructions although there are more complications introduced.

SMULL			SMLAL		
Cycles	Left	Right	Cycles	Left	Right
3	PS, XS	PS, XS	3	PS, XS	PS, XS
	0, N, P	0		0, N, P	0
	0	0, N, P		0	0, N, P
	XLU	XLU	5	N, PL, XL	PS, XS
4	NT, PU, XS	XLU		PS, XS	NT, PLU
	N, PL, XS	XS		PS, XU	XLU
	N, PL	PS	6	PS, XS	NU, PT
	PS, XS	PLU,NT		NT, PLU	XLU
	XL	N, P		PS, X	XT
5	NT, PL, XS	XT		XL	N, PL
	N, P	N		XT	XL
	N, PL	PL	7	N, PL	N, PL, XT
	P, XS	PT		NU, PT	XLU
	NU	XL			
	PT	XLU			
	XS	NU			

Legend

0	The constant zero	S	$\leq$ 16 bits
N	Negative integer	L	$>$ 16 bits
P	Positive integer	T	$=$ 32 bits
X	Special	U	$<$ 32 bits

Table 4.1: Timing behavior for `SMULL` and `SMLAL`. The Special category refers to power of two and topheavy operands. The righthand column of the legend defines modifiers to the lefthand side, so the symbol NT defines negative integers that require 32 bits of space and PLU means positive integers larger than 16 bits but smaller than 32 bits. The broadest definition makes the category so e.g., X consists of XS and XL. We do not combine overlapping cagtegories of P and X. Please not that the grouping of categories is not unique.

Chapter 5

Results and analysis

5.1 Implementation and performance on M4

Measuring the implementation's characteristics is done in two ways. Code size and memory use are derived from the output generated by μVision. As part of building the binary file to flash onto the target device, Keil's toolchain outputs a human-readable file with size statistics broken down per file. Our implementation uses the 'one function per file' model, allowing us to easily obtain these characteristics. To measure timing we use the aforementioned SysTick interface. Device-specific driver libraries written by manufacturers provide C functions to use this timer but since we require as much accuracy as possible we implement the timing function in assembly to gain greater control.

Our main function is written in C and sets up the UART to receive scalars and public points, and to transmit cycle counts. Once everything is set up the measuring function is called. This function initialises and starts the cycle counter, calls the function under tests and reads SysTick's counter. The running value of the counter is accessed by reading a special memory location with `LDR`, so the code under measurement is surrounded by two `LDR` calls. Because reading memory takes two cycles the observed difference between start and end is inflated by two—not four since memory reads happen at the same point for both `LDR` instructions—which the function corrects for. The function call and return are measured as part of the function, but setting up the arguments in the appropriate registers is not since doing so is implementation-dependent.

The main characteristics of our M4 implementations are listed in Table 5.1. We note that the implementation for M0(+) in [12] completes a scalar multiplication in 3 589 850 cycles using 7900 bytes of ROM and 548 bytes of RAM. The use of single-cycle $32 \cdot 32$ bit multiplication as well as the ability to use all of the core's registers for instructions yields a cycle count reduction of 49.40%. The column denoted overhead lists the numbers of cycles required for memory management as opposed to being used for arithmetic per se. To calculate these numbers we tabulate all of the loads, stores, pushes, pops and movs, compute their required cycle counts where needed and add these up. From this sum we subtract the cycles required for function prologue and epilogue (i.e., saving and restoring registers) and an idealised version of data loading and storing. The idealised load and store are taken to be a single 10-word transfer instruction per field element to be read or written. As an example, `mul`'s overhead is reduced by 33 cycles to account for loading two field elements and storing a third, and another 20 cycles are subtracted for pushing and popping caller context. Note that while the overhead cycles are computed by tallying the instructions, the overall cycle counts are observed from execution and include unconditional jumps to and from the respective function. As such, 1 to 3 cycles per jump are extra in the observed cycle count depending on the jump distance and alignment. Within a function the overhead counts represent the number of cycles to be saved if the M4 had unlimited registers, or at least enough never to require interactions with RAM. For the `scalarmult` and `invert` instructions the overhead is not listed because these functions are almost entirely procedural rather than arithmetic, consisting for the most part of calls to arithmetic functions. Simple functions (`add`, `sub`)

Function	ROM	Heap	Stack	Cycles	Overhead
`add`	44	0	0	73	3
`cswap`	212	0	0	149	28
`from_bytes`	84	0	0	65	0
`invert`	388	160	0	151 997	n/a
`mul`	1284	0	152	631	284
`mul121665`	300	0	4	129	4
`scalarmult`	432	312	4	1 816 351	n/a
`sqr`	1168	0	128	563	232
`sub`	64	0	0	77	3
`to_bytes`	164	0	0	85	0
Total	**4140**	**472**	**156**	**1 816 351**	n/a

Table 5.1: X25519 per-function code and memory sizes in bytes as well as cycle counts. Note that all names refer to the functions operating on field elements, e.g., `add` refers to the addition of two 10-limb elements contained in 32 bit words as one functional unit. ROM total is the sum of all functions. Overhead lists the number of cycles that are estimated to be used for memory interaction rather than arithmetic. Totals for heap and stack use are based on maximum use at any time, this is `scalarmult` plus `invert` plus `mul`. Running time total is for the complete X25519 scalar multiplication.

or functions which operate on a single field element (`from_bytes`, `to_bytes`, `mul121665`) are able to exploit the 14 registers with very little overhead. As soon as more data is introduced along with more complicated processing of the data (`cswap` already has more overhead because of the need to compute and keep track of a dummy) memory interactions have a serious impact on performance. For the squaring and multiplier, 41% and 45% of the cycles, respectively, is taken up by memory interactions. These interactions comprise storing and loading intermediate results as well as retrieving pointers that are pushed out of registers by other computations. In the complete scalar multiplication there are 1286 iterations of `mul` and 1274 iterations of `sqr`. In total, they require 811 466 cycles and 717 262 cycles, respectively, for a combined total of 1 528 728 cycles. These two functions represent 84% of the total number of cycles. In the idealized scenario they would require 365 224 and 295 568 cycles, totalling 660 792. Extrapolating, if 660 792 is 84% of the total runtime the idealized version of the multiplier would require around 800 000 cycles. One can safely say that the overhead from memory interactions has a large impact on the runtime of this implementation. The impact on code size is much smaller, on the other hand. Most arithmetic instructions require one or two cycles whereas the quickest load or store requires two and the slowest can take 15. The hypothetical reduction in code size would therefore not be commensurate with the reduction in cycles. Squaring would require 56 fewer instructions, multiplication 76. Depending on their precise encoding (which is dependent on the use of pointer offset features and on instruction alignment) these instructions require two or four bytes each, saving at most 224 and 304 bytes for both, far less than the 45% of cycles these instructions occupy.

In order to understand the performance differences between hand-optimised assembly versus compiler-optimised C code we include figures from the C version compiled for the same M4 device. We used ARM Compiler, integrated in the KEIL μVision MDK version 5.14[1], for this purpose. This C version of the scalar multiplier was originally developed as a correctness baseline, first performing the sub-multiplications in full before combining them by applying the reduced refined Karatsuba algorithm. We updated the code for this comparison to resemble the assembler version as much as possible. Results of two compilation settings are provided, once using the *-O0* option instructing the compiler not to optimise at all and once with *-O3 -Otime* to optimise as much as possible and favoring running time over code size. The results are listed in Table 5.2. A clear difference in performance can be observed. First, the optimisation flags the compiler offers provide substantial improvement over unoptimised C, all metrics improve when doing so. Still, both C versions perform significantly worse than the hand-written assembler version on all fronts. Running time

[1] ARM Compiler page.

is almost halved, ROM size is more than halved and even memory consumption is down significantly. For performance-sensitive code, especially of a write-once-run-often nature, investing in fine-tuned code could be well worth the time and effort.

	unoptimised C	**optimised C**	**assembly**	**Difference**
cycles scalarmult	4686147	3552657	1816351	-48.9%
cycles multiplier	1841	1465	631	-56.9%
codesize in bytes	11630	9016	4140	-54.1%
RAM use in bytes	832	776	628	-19.1%

Table 5.2: Code size, memory use and running time for the assembler version and compiled C versions without and with maximal optimisation. The difference compares optimised C to assembly.

5.2 Performance on Cortex-M3

Running the code on an M3–using the same DISCOVERY board we used for the timing tests in Chapter 4 yields an average running time for the full scalar multiplication of 2 661 659 cycles ($n = 10000, sd = 289.0$), and 1048.0 cycles ($n = 10000, sd = 0.49$) for the multiplier alone. These measurements were taken by supplying a random point and scalar for each sample individually. No modifications were made to the X25519 code in any way to run it on M3. Even without any additional work to guarantee constant time execution, we observe that the number of cycles required for a multiplication increases sharply compared to M4. The M0+ implementation in [12] requires 1469 cycles for a multiplication and reduction which is still significantly more than this implementation on M3, but the M0 version has to cope with additional limitations not present on M3. Moreover, from unpublished work by Peter Schwabe we know that the M0+ implementation, when modified to make use of the unrestricted register file, performs the 256 bit multiplication and reduction in 1280 cycles. Our implementation would still be faster than this, but the difference is shrinking and our code is not be constant time on M3. If `UMULL` and `UMLAL` were forced to always take 5 or 7 cycles, respectively, the 256 bit multiplier would take 1067 cycles. This does not account for the additional code required to enforce this behavior. If we assume that this behavior can be enforced by modifying the operands before multiplication and removing the modification after using a one-cycle instruction preceding a multiplication or multiply-and-accumulate and another one following it, 170 cycles would be added per multiplication instruction. The total cost would then be 1237 cycles for a full multiplication, almost matching the modified M0+ code. At this point our code would also be more complex. Furthermore, the behavior of the signed long multipliers is more complicated than their unsigned counterparts, casting doubt on the estimation of two additional cycles per multiplication. It also assumes no other design changes would be needed to ensure the modified operands would continue to fit.

Chapter 6

Discussion

6.1 Future work

We see several avenues to further explore X25519 on these architectures. This implementation performs scalar multiplications for arbitrary points. If one were only interested in scalar multiplications on the fixed base point, optimisations would be possible by precomputing multiples of the base point. For verifying signatures one can even disregard considerations for constant time behavior. Another possible optimisation is not to use reduced refined Karatsuba for the squaring but to use the schoolbook method instead. We have noted that memory interactions are responsible for a large percentage of the total running time of squarings. Our preliminary calculations indicate it is doable to compute a squaring by keeping the source element in registers and calculating the output limbs one by one using schoolbook multiplication. The source limbs would take 10 registers, leaving two for the 64 bit limb being calculated and two for scratch memory, for instance when multiplying limbs by 4 or 38. Constants like 38 would still need to be placed in registers multiple times but the stack pointer can be used directly with offsets without having to place it in a (different) register first. For the multiplication function there are two many source limbs to keep them all in registers, making this approach infeasible. Finally, this implementation can be adapted in the presence of a random number generator to randomise its secret scalar in defense against power analysis attacks.

In its current state the implementation is vulnerable to timing attacks on M3. Without specifically choosing public points to generate timing differences we observe a range of between 2 657 369 and 2 661 870 cycles, inclusive. One should be able to choose specific points so as to generate an intermediate state where especially few or many bits are set for one of the two points. In this chosen state, the point doubling will cost more or less depending on the exact setup, revealing whether the corresponding scalar bit was set or not. Choosing such intermediate states for progressive bits of the scalar given successive hypotheses on previously analysed bits will eventually yield the entire secret scalar. The difficulty in this attack is determining two sets of suitable points to feed the scalar multiplier to test both values of the second scalar bit. After the initial ladder step where `cswap` is guaranteed to swap the points (since the top bit of the scalar is always set per the algorithm) one set of points must be cheaper to compute when the swap is performed, while the other set of points is cheaper when it is not. As such, attackers must reason back from these special points to an input point to deliver to the victim given that the steps leading to the special case are known. This attack is entirely feasible to carry out. The requirements are that the attacker has an M3 device and our implementation in order to construct the sets of points, and that she is able to time the multiplication performed by the victim. Once she has deduced sets of points offline, the attacker would be able to have the victim process points successively until the scalar is deduced. This attack treats the M3 as a black box with a fixed private key like in a lunchtime attack. Instead of attacking, another avenue for further work is to construct a constant time implementation for M3 using the long multiplication instructions. As we outline in Section 5.2 we believe the performance benefits compared to a modified version of the M0 version that makes use only of the `MUL` instruction would be small or non-existent, but we cannot know for sure.

6.2 Conclusion

In this thesis we have presented an implementation of X25519 for a new processor architecture. Compared to the existing M0 implementation our code benefits from more usable processor registers and fast 32 bit multipliers with accumulate functionality. Combining more powerful hardware with hand-optimised assembly code implementing efficient algorithms we are able to reduce the running time by almost 50%. In addition, due to curve design and our implementation choices this implementation is hardened against timing attacks. We have also performed detailed timing measurements to establish the usefulness of the same multiplication instructions on the Cortex-M3. Due to the higher and variable number of cycles for them on this platform we conclude that our current implementation is insecure on it. The required modifications would make the implementation complicated and probably not yield meaningful performance benefits in return.

As mentioned in the introduction, Diffie-Hellman functions enable the use of session keys or for parties to establish a shared secret without having had prior contact. Our implementation provides this functionality for users of the Cortex-M4 using a state-of-the-art cryptographic primitive and hand optimised assembly. We hope it proves useful. At the same time, this work should serve as a warning that cryptographic functions are still difficult to get right. Running this implementation on the otherwise similar Cortex-M3 would leave the resulting product vulnerable to feasible timing attacks. Designers working with the M3 who have a security need suitable for X25519 should instead consider the constant time M0 code, or at any rate refrain from using anything but the `MUL` instruction. Although very useful for general computation, the long multipliers on M3 are not suitable for simple yet secure cryptographic software.

Bibliography

[1] Daniel J Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[2] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.

[3] Daniel J Bernstein. Batch binary Edwards. In *Advances in Cryptology-CRYPTO 2009*, volume 5677 of *LNCS*, pages 317–336. Springer, 2009.

[4] Daniel J Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, volume 8731 of *LNCS*, pages 316–334. Springer, 2014.

[5] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 124–142. Springer, 2011.

[6] Daniel J Bernstein and Peter Schwabe. NEON crypto. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, 2012.

[7] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011.

[8] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[9] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002*, volume 2523 of *LNCS*, pages 13–28. Springer, 2003.

[10] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.

[11] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[12] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, pages 1–22, 2015.

[13] Louis Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In *Public key cryptography—PKC 2003*, volume 2567 of *LNCS*, pages 199–211. Springer, 2002.

[14] Michael Hutter and Peter Schwabe. NaCl on 8-Bit AVR Microcontrollers. In *AFRICACRYPT*, volume 7918 of *LNCS*, pages 156–172. Springer, 2013.

[15] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[16] Cameron F. Kerry and Charles Romine. FIPS Pub 186-4 Federal Information Processing Standards Publication Digital Signature Standard (DSS), 2013.

[17] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, volume 1966 of *LNCS*, pages 388–397. Springer, 1999.

[19] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[20] Victor Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology—CRYPTO'85 Proceedings*, volume 218 of *LNCS*, pages 417–426. Springer, 1986.

[21] Serge Mister and Carlisle Adams. Practical S-box design. In *Workshop on Selected Areas in Cryptography, SAC*, volume 96, pages 61–76. Citeseer, 1996.

[22] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[23] Katsuyuki Okeya and Kouichi Sakurai. Power analysis breaks elliptic curve cryptosystems even secure against the timing attack. In *Progress in Cryptology—INDOCRYPT 2000*, volume 1977 of *LNCS*, pages 178–190. Springer, 2000.

[24] Claude E Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

[25] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[26] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.