Assembly optimised Curve25519 and Curve448 implementations for ARM Cortex-M4 and Cortex-M33

Emil Lenngren
emil.lenngren@gmail.com

Abstract

Since the introduction of TLS 1.3, which includes X25519 and X448 as key exchange algorithms, one could expect that high efficient implementations for these two algorithms become important as the need for power efficient and secure IoT devices increases. Assembly optimised X25519 implementations for low end processors such as Cortex-M4 have existed for some time but there has only been scarce progress on optimised X448 implementations for low end ARM processors such as Cortex-M4 and Cortex-M33. This work attempts to fill this gap by demonstrating how to design a constant time X448 implementation that runs in 2 273 479 cycles on Cortex-M4 and 2 170 710 cycles on Cortex-M33 with DSP. An X25519 implementation is also presented that runs in 441 116 cycles on Cortex-M4 and 411 061 cycles on Cortex-M33 with DSP.

1 Introduction

Curve25519 was introduced in 2005 by Daniel J. Bernstein [3] as a faster and safer alternative to the NIST curve P-256. Later in 2015, Mike Hamburg designed Curve448 [5] as a similar and direct replacement for Curve25519 when stronger security is required. Since both are Montgomery curves, the same high-level scalar multiplication algorithm can be used. The difference lies in the prime for the curve's field. Curve25519 uses $2^{255} - 19$, while Curve448 uses $2^{448} - 2^{224} - 1$. Curve25519 has a security level of 128 bits, while Curve448 has a security level of 224 bits. Both curves have been standardised for use with TLS. In particular, the X25519 key exchange function "SHOULD" be supported by TLS 1.3-compliant applications [9].

Curve25519 and Curve448 commonly refer to the particular elliptic Montgomery curve, while X25519 and X448 refer to Diffie Hellman functions using these curves. These two functions are specified in RFC 7748 [6]. Birationally equivalent (un)twisted Edwards curve variants are called edwards25519 and edwards448 (or Ed448-Goldilocks), respectively.

1.1 Related work

Multiple papers have been written about Curve25519 optimisations for Cortex-M4. One of the more recent that is of particular interest, due to its demonstrated performance, is written by Haase et al. [4]. Their implementation is a combination of code written in C with inline Assembly and Assembly and is reported to run in 609 779 cycles on STM32L476 at 16 MHz.

The earliest work found that targets Cortex-M4 for Curve448 is an implementation by Seo et al. [10]. It is reported to run in 6 218 135 cycles on an STM32F4 discovery board at 24 MHz. The implementation uses Edwards curve formulae, which results in a larger amount of field multiplications compared to the Montgomery ladder, that is otherwise commonly used. It uses a common field multiplication routine for both field multiplication and field squaring. There is thus potential for a more optimised approach that uses the Montgomery ladder with a specialised squaring routine.

Anastasova et al. proposed in 2023 [2] improved field arithmetics for Curve448 targeting Cortex-M4F (Cortex-M4 with FPU), resulting in a scalar multiplication implementation that is reported to run in 3 220 682 cycles on STM32F407VG at 24 MHz.

2 ARMv7 and the Thumb-2 instruction set

ARM Cortex-M4 and Cortex-M33 are ARMv7E-M and ARMv8-M designs, respectively, which implement the Thumb-2 instruction set, which is an extension to the Thumb instruction set.

A program written in ARM Assembly can be assembled into the standard ARM 32-bit encoding (fixed width). To get a more compact representation, an instruction set called Thumb has been developed, which uses a 16-bit instruction encoding (fixed width). This is a reduced instruction set where only the most common instructions can be represented. Thumb encoding also restricts what operands can be used. For most instructions, the destination register must be the same as the first source register. Additionally, most Thumb instructions can only operate on the lower half of the ARM register set (`r0-r7`). In most cases, only the flag-setting variant of the instruction is available, such as `adcs`.

The Thumb-2 instruction set was developed to get the code density benefits of the Thumb instruction set, while still being able to use (almost) all instructions in the ARM instruction set. This has been achieved by extending the Thumb instruction set with 32-bit encoded instructions for the instructions that cannot be represented in the 16-bit Thumb encoding. This means the Thumb-2 instruction set is variable in size, where an instruction uses either 16 bits or 32 bits.

A processor only supporting the Thumb instruction set is the ARM Cortex-M0 (with the exception of a few 32-bit Thumb-2 instructions).

When writing code targeting Thumb-2, it is important to know which instructions that can be represented in 16 bits, in order to optimise for code size. Generally, if there is both a flag-setting variant and a non-flag-setting variant of an instruction, picking the flag-setting variant and using registers from the lower half of the register set is usually desired, since that results in that the smaller 16-bit instruction is being emitted.

3 Cortex-M4

The Cortex-M4 uses a 3-stage pipeline. The three stages are fetch, decode, and execute. Almost all instructions can be executed in one cycle. Due to this design, there are generally no pipeline stalls when one could expect a data hazard (i.e. the result from one instruction is used in the next one). In practice, the only exceptions for non-branch instructions are load and store instructions, which execute in $1 + n$ cycles, where n is the number of words to transfer. The core performs the address calculation during the first cycle and transfers one word per successive cycle. A sequence of `ldr` instructions can be pipelined so that the data load of an instruction can be performed in parallel with the address calculation of the following instruction. This, however, means that if the address for an `ldr` instruction depends on data loaded from the previous `ldr` instruction, the cpu will stall for one cycle, i.e. the pipeline optimisation will not be used and thus two `ldr` instructions in sequence take four cycles. The `str` instruction in practice always executes in one cycle, due to the write buffer, which transparently buffers a write and executes it in the background during the next cycle. Multiword store instructions (`strd` and `stm`) can however not use this optimisation. One `str` instruction can also be pipelined after a sequence of one or more `ldr` instructions, which means the `str` instruction can be considered "free", assuming the write buffer is used. Thus, in general, `ldr` + `str` takes two cycles. The `str` instruction can even store what the `ldr` instruction loaded without any penalty. For some reason, a `nop` can also be pipelined after `ldr` in the same way. There are general data hazards for the address used in any load or store instruction. If the address, address base, or offset is generated during the previous instruction, the cpu will stall for one cycle. What has been stated in this paragraph (except pipelining of `nop`) is documented in the Cortex-M4 Technical Reference Manual [1].

The manual could have been even more precise about data hazards. It seems more correct to say that the data hazards for memory addresses apply when the address is generated during the previous *cycle*, rather than during the previous *instruction*. For example, a chain of `ldm` or `stm` with writeback, where the same address register is used for every instruction, will not result in additional stalls. If a core register destination for a four-operand `vmov` instruction is used as an address in a following memory instruction, there will be a stall if the second `vmov` destination register is used but not if the first destination register is used.

The core cannot fetch 16-bit instructions directly, nor can it fetch an unaligned 32-bit instruction directly from memory. Instead, it has a prefetch-unit FIFO of three 32-bit words, which reads from the memory system one 32-bit word per cycle. This unit can hence store up to six 16-bit instructions or three 32-bit instructions. The core can prefetch instructions ahead of execution and even speculatively from branch target addresses. What is not documented is how the alignment of 32-bit instructions can cause additional, unexpected, pipeline stalls. In particular, through experiments, it appears that a multi-cycle instruction followed by three unaligned 32-bit instructions (i.e. they start at addresses which are all 2 modulo 4), where at least the first two are single-cycle instructions, causes the core to stall for one cycle. In particular, this kind of sequence can easily occur when a memory load instruction is followed by three long multiplication instructions such as `umaal`. For this scenario, an `str` instruction can be the initial multi-cycle instruction even in situations the write buffer is used, i.e. takes one cycle. If instructions cannot be reordered to avoid this situation, an easy workaround is to add the `.w` suffix to

some 16-bit instruction to force the assembler to use the 32-bit variant of that instruction, to change the alignment of the following instructions.

Branch instructions, including instructions that modify `pc`, are multi-cycle instructions, assuming the branch is taken. The Technical Reference Manual does not describe every branch instruction's cycle count in detail. What has been observed are the following details:

- A load instruction that has `pc` as the last destination register takes three cycles longer than if the destination would have been an ordinary register.
- Unconditional and conditional branches to a label, including `bl`, take two cycles if the branch is taken (otherwise one cycle).
- The "`add pc, pc, rn`" operation takes four cycles.
- Branch instructions with a register containing the target, including "`mov pc, rn`", take three cycles, except if the register is `lr` (and `lr` is not updated in the previous instruction), in which case the instruction takes two cycles. The execution of a leaf function containing the only instruction "`bx lr`" that is called by "`bl label`", however, takes in total only four cycles (`bl label` and `bx lr`). In general, very small leaf functions seem to be optimised.

All these cycle counts correspond to when the target instruction is either a 16-bit instruction or an aligned 32-bit instruction, otherwise one extra cycle must always be added. As an example, "`pop {pc}`" will take six cycles if the target address is an unaligned 32-bit instruction. A one cycle faster alternative to "`pop {pc}`" is to pop to `lr` and then execute "`bx lr`", assuming some meaningful instruction can be inserted between those two.

Unless otherwise stated, cycle counts for various algorithms and operations described in this paper will refer to the number of cycles the operation takes on a Cortex-M4 processor with zero wait states.

3.1 Cortex-M4F

Cortex-M4F is a variant of Cortex-M4 that contains an FPU. The floating point instructions do not have much value for this application, but we can use the floating point registers as an alternative to using the stack for temporary data. The `vmov` instruction can move one or a pair of registers between the core register file and the floating point register file. The instruction takes one cycle per word to transfer and is thus one cycle faster than using `ldr`, `ldrd`, or `ldm` instructions to load the same amount of data from the stack, assuming the `ldr` instruction cannot be pipelined. There are 32 floating point registers, 32 bits each, that can be used.

4 Cortex-M33

The types of Cortex-M33 processors that will be studied are those that have the DSP option. The DSP option includes the `umaal` instruction. Cortex-M4 designs require DSP instructions to be available, but those are optional in Cortex-M33 designs.

Despite being an ARMv8 architecture, Cortex-M33 has many similarities with Cortex-M4. Cortex-M33 uses a 3-stage pipeline, just like Cortex-M4. Compared to the Cortex-M4 documentation, which documents the cycle count for every instruction and how to pipeline memory instructions in the most efficient way, little is documented about the precise timings for Cortex-M33. Experiments, however, show that most instructions execute in one cycle, just as on Cortex-M4. Memory instructions, on the other hand, appear to have been optimised to execute in only n cycles, where n is the number of words to transfer, which is an improvement of one cycle. There are therefore generally no benefits of using FPU registers as temporary storage on this processor.

There are also cases when the performance is worse on Cortex-M33 than on Cortex-M4: data hazards stalling the pipeline for one cycle, in particular. Examples are when the result of a `umaal` instruction is used by a following arithmetic instruction such as `adds` or `adcs`, or when the result of an `ldr` instruction is used as a multiplication operand by a `umaal` instruction.

5 Multiplication and squaring

To perform multiplication of arbitrarily big numbers, the instructions `umull RdLo, RdHi, Rn, Rm` and `umaal RdLo, RdHi, Rn, Rm` are used. For the `umull` instruction, `Rn` and `Rm` are multiplied and the 64-bit result is written

to the register pair `RdLo`, `RdHi`. The `umaal` instruction multiplies `Rn` by `Rm`, adds `RdLo`, adds `RdHi`, and writes the 64-bit result to the register pair `RdLo`, `RdHi`. Since all four source registers are 32-bit integers, the maximum result is $(2^{32}-1)^2 + 2(2^{32}-1) = 2^{64}-1$, which fits in 64 bits. This means that no overflows will occur. There is also a less useful `umlal` instruction that treats the two accumulator operands as a 64-bit value, which is added to the 64-bit product of the last two operands. This can easily overflow, and the carry flag is unfortunately not updated. Apart from memory instructions, the `umull` and `umaal` instructions are sufficient to calculate arbitrarily big numbers. The following diagram demonstrates "schoolbook multiplication" of two 160-bit integers, stored as little endian arrays a_0, a_1, a_2, a_3, a_4 and b_0, b_1, b_2, b_3, b_4.

```
                                  umull  umull  umull  umull  umull
                           umaal  umaal  umaal  umaal  umaal
                    umaal  umaal  umaal  umaal  umaal
             umaal  umaal  umaal  umaal  umaal
      umaal  umaal  umaal  umaal  umaal
```

From right to left, each row i represents multiplications of a_ib_0, a_ib_1, a_ib_2, a_ib_3 and a_ib_4. The standard approach is to perform the calculations row by row, which is commonly called operand-scanning. All b_j words can be kept in registers and do not need to be reloaded during the computation, while a new a_i word is loaded and discarded for every row. (Performing the calculations column by colmun, called product-scanning, would require more registers to hold the operands.) For each row, we execute the multiplication instructions from right to left. Note that after the first row, we have in total ten result registers containing the partial products. The lowest word of the first multiplication can however be stored in memory as the final result, but the other nine result words should stay in registers, if possible, so that they can be used as inputs to the `umaal` instructions on the next row without overhead of spilling to and restoring from the stack. With the described approach, when performing a `umaal`, there will always be exactly two available addends from the temporary registers. After each row is complete, the least significant register can be written to memory as the final result, and hence one register less is needed after each row. At the final row, the six registers are all written to memory as the final result. In total, ten words in memory make up the 320-bit result.

Note that there is no single instruction for performing a 32x32 $\rightarrow$ 64-bit multiplication with *one* 32-bit accumulate operand. Instead, we can first zero a register and then use it as one of the two accumulate operands to `umaal`. However, the zeroing of registers unfortunately costs additional instructions. A trick to zero two registers if we already know one third register contains zero is to use `umull` and multiply zero by zero.

This strategy can be used to create an algorithm requiring fewer temporary registers. After processing every row that is not the last, we will have five temporary registers and one result word:

```
                                 umaal*  umaal*  umaal*  umaal*  umull
                          umaal   umaal   umaal   umaal   umaal*
                   umaal  umaal   umaal   umaal   umaal*
            umaal  umaal  umaal   umaal   umaal*
     umaal  umaal  umaal  umaal   umaal*
```

Here, `umaal*` denotes that one accumulate operand is a new register containing zero.

We would however like to avoid all those additional instructions required to set registers to zero. A different approach that avoids zeroing registers is the following.

```
                                  umaal  umaal  umaal  umull  umull
                           umaal  umaal  umaal  umull  umaal
                    umaal  umaal  umaal  umull  umaal
             umaal  umaal  umaal  umull  umaal
      umaal  umaal  umaal  umaal  umaal
```

It is not possible to perform the above instructions strictly row by row. Whenever we encounter a `umaal` where only one addend is available, we first execute the multiplication on the next row, one column to the right. This is done recursively when needed. In total, six temporary registers are needed in this algorithm, and at most three a_j words need to stay in registers at the same time (if we want to avoid reloading them). However, at the time during the algorithm runs when as many as six temporary registers are needed, only one a_j word needs to stay in registers. To make it clear, the following diagram shows the order the multiplications are performed, with number of temporaries and a_j words needed in registers at most during the calculation of the corresponding partial product in parentheses. **Bold** indicates `umull`, rather than `umaal`.

				9 (3, 3)	6 (4, 2)	4 (2, 2)	**2 (3, 1)**	**1 (2, 1)**
			12 (5, 2)	11 (5, 2)	8 (3, 3)	**5 (4, 2)**	3 (3, 2)	
		16 (4, 2)	15 (4, 2)	14 (4, 2)	**10 (5, 2)**	7 (4, 3)		
	20 (6, 1)	19 (6, 1)	18 (6, 1)	**17 (6, 1)**	13 (5, 2)			
25 (2, 1)	24 (3, 1)	23 (4, 1)	22 (5, 1)	21 (6, 1)				

With this algorithm, we hence need in total 12 registers, including temporaries and space for the operands. ARM has 14 usable data registers, so we have two spare ones. The principle of this algorithm will be used as the basis for Curve25519 field multiplication, namely that `umaal` instructions are used in a column, from top to bottom, until there are two remaining, when a `umull` will be followed by a `umaal`.

5.1 256-bit multiplication

For Curve25519, we will need a multiplier that multiplies two 256-bit values and outputs a 512-bit result. Eight words are needed for 256 bits, which is three more than in the algorithm shown above. It is easy to extend the algorithm for any sized multiplication (when a and b have the same size), but doing so either increases the number of required registers or introduces the need to spill to memory. The chosen approach is to split the calculation into two parts, one operating on the first five words of b and one operating on the last three words of b. This means every word in a will need to be loaded twice from memory but the words in b need only be loaded once. Note that only the lower result words of the first part can be stored immediately to the stack as the final result. The upper result words are kept in registers and will be used as addends to `umaal` in the second part. The register pressure will be smaller during the second part, since we only operate on three words of b, as well as result words can be written to the stack as they become complete. The increased register pressure during the first part, due to we cannot write the results yet to memory, will be a problem. To earlier finish the sixth column and be able to store its result word to memory, we calculate a_0b_5 during the first part instead of the second part (we load a_0 and b_5 exclusively for this multiplication). This saves two temporary registers that would otherwise need to be alive when we transition between the two parts. The final chosen order of the partial multiplications is shown below in Figure 1.

							42	**43**	**26**	9	7	4	**2**	**1**
						46	45	44	12	11	8	**5**	3	
					49	48	47	16	15	14	**10**	6		
				52	51	50	20	19	18	**17**	13			
			55	54	53	25	24	23	22	21				
		58	57	56	31	30	29	28	27					
	61	60	59	36	35	34	33	32						
64	63	62	41	40	39	38	**37**							

Figure 1: 256-bit multiplication order. **Bold** indicates `umull`, rather than `umaal`.

To save registers, the multiplication a_7b_0 (#37) overwrites the register where b_0 is stored and the multiplication a_0b_6 (#43) overwrites the register where a_0 is stored. The multiplication a_0b_5 (#26) overwrites both source registers. Except after the calculation of a_0b_6 (#43), where the multiplication also overwrites the address of a, there is always an available register for the address of a.

Note that in every column, the second last partial product calculation uses `umull`, except for the middle column, where this must be done earlier. Thanks to the early calculation of a_0b_5 (#26), we have an additional register to use as accumulate operand for the following column, which allows us to not need to perform a `umull` in this column during the first part. When we are about to calculate a_7b_0 (#37), we have only one accumulate operand, forcing us to use `umull` this early.

In total, 111 cycles are required for the 256-bit multiplication, assuming the addresses for a and b are already in registers, as well as spilled to the stack as copies. The 512-bit result will be stored as eight words on the stack at a fixed offset (lower words) and eight words in registers (upper words).

One could think that an approach splitting b into two equal-sized parts (four words each) would be good. It turns out to be worse, since that would require two more temporary registers when the first part is done, as well as the need to have b_4 stored in a register during the second part.

The use of `umull` instructions instead of `umaal` instructions with zeroing of registers constitutes most of the performance improvements over the method used in the work by Haase et al. [4].

5.2 480-bit multiplication

For Curve448 we will need a 480-bit multiplier. Why a 448-bit multiplier is not enough will be explained later. With such large operands, most of the working state will not fit in registers and we will be required to spill and restore temporaries to the stack, as well as load some operand words more than once. A 480-bit value is represented as 15 words. In total $15 \cdot 15 = 225$ partial products must be calculated, requiring 225 `umull`/`umaal` instructions and hence 225 cycles. The majority of the rest of the cycles will be spent in memory instructions, at least one cycle per transferred word. Depending on how optimised these memory instructions are, they will each add additionally either zero or one cycle spent during a pipeline stall. Recall that a chain of n `ldr` instructions require $1 + n$ cycles and each `str` instruction requires one cycle. An optimisation opportunity arises when we have one or more `ldr` instructions followed by one `str` instruction, which saves one cycle. These chains require n cycles, where n is the total number of words to transfer. Unfortunately, an algorithm where we have an `str` instruction followed by one or more `ldr` instructions can only be reordered if the register used in the `str` instruction is not among the set of loaded registers. In this case, we need an extra spare register and load to that one instead. If we do not reorder such a sequence of instructions, due to the lack of an available spare register, we will get an extra cycle while the pipeline is stalled during the last `ldr` instruction. If we minimise the number of memory transfers and always try to fill the end of each `ldr` chain with an `str` instruction, we will get the best performance.

Operand scanning, i.e. calculating row by row, operating on the whole operation at once, will result in bad performance. Since all 15 b_j words are used in every row, along with an additional 15 temporaries per row, which are accessed one by one from right to left, the least recently used words are to be used when we start the next row. Due to the limitation of only having 14 registers, most of the 30 values will need to be spilled and restored for every row, causing significant memory overhead. Product scanning, i.e. calculating column by column, seems hard to make something useful of, since we only have 64-bit accumulators with no carry flag when using `umlal`, and if we instead use `umaal`, we will produce one extra carry word per row that must be saved until the next column.

The proposed solution is to use operand scanning where the b operand is split into three equal-sized parts of five words each. Operand scanning with five words in b and any number of words in a can be performed efficiently. The technique is simple: the five words in b are all loaded to registers. We will then process the calculation iteratively row by row, producing one result word that will be stored to the stack per iteration. At the beginning, we set $t = 0$, where t is a 160-bit number stored in five registers, using one `mov` and two `umull` instructions. Each iteration loads the next word s from the stack (which contains one word of the calculated result so far at this position) as well as a_i into registers. Using a chain of five `umaal` instructions, the 192-bit value $a_i \cdot b_{j..j+4} + t + s$ is calculated. The upper 160 bits are the new t and the lower 32 bits are stored to the same stack word. After the last iteration, t is appended to the stack after the last stored word. Each part thus accumulates its partial products into the stack at the correct position. Before we start calculating the parts, the stack is initialised to 15 zero words (or alternatively, we could replace the loads from the stack in the first part with `mov` instructions, clearing the registers instead). Note how this algorithm uses eight instructions per iteration, five of which are "useful work", and the other three are memory overhead. We need five registers for storing the five b words, one register for a_i, five registers for t, one temporary register for s, and one for the address of a, which is 13 in total. The spare register is used to be able to delay the store until after the loads in the next iteration. Thus, the register used for s is alternating between each iteration. This means that these eight instructions will execute in eight cycles. The address for the next words in b is spilled to the stack after five words have been loaded (and the address has been incremented) and is restored when the next five words are to be loaded. Conveniently, the first iteration in a part calculation does not store anything to the stack after its `ldr` instructions, so this is where we place the spill of the b address. Similarly, the restore is performed just before the five t words are stored to the stack at the end of a part calculation. The last `str` of t for the second part is moved to after the two `ldr` instructions of the first iteration in the last part.

To avoid pipeline stalls, remember that any `ldr`/`str` that is followed by at least three unaligned 32-bit single-cycle instructions (such as `umaal`) requires an extra cycle. This situation arises in every inner loop, except in the first part where we have a 16-bit `movs` instruction before the multiplication instructions. We thus add the `.w` suffix to an `ldr`/`str` instruction when appropriate, to avoid this extra pipeline stall.

Two iterations of the inner loop are shown in Algorithm 1.

To optimise away eight cycles during the beginning of the first part, note that we use many `umaal` instructions with one or both accumulate operands being zero. We therefore directly replace the five first iterations with the 25 first partial product calculations from the 256-bit multiplication routine (which completes in 25 cycles), since both perform the same operation. In total, 398 cycles are needed for the calculation, assuming that the two addresses for a and b are already in registers and that the stack is allocated. At the end, the 960-bit result is placed on the stack.

In total, $3 \cdot 15$ words from a, $3 \cdot 5$ words from b, and $2 \cdot 15$ words from the stack are loaded, and the b address

is restored two times, which means 92 words are loaded from memory. We store $3 \cdot 20$ words to the stack and spill the b address two times, which means that 62 words are stored to memory. $2 \cdot 3$ cycles are spent setting t to zero, and we have additionally 10 `mov` instructions for clearing registers during the last 10 iterations of the first part. For each of the three parts, one cycle is spent during stall when `ldm` is used to load the next five words from b (with post-increment), to save some code space instead of using five `ldr` instructions (which would remove these three cycles). Together with the 225 `umull`/`umaal` partial product calculations, we now have the full explanation for all 398 cycles. This result corresponds to a fully unrolled implementation. An implementation optimised for size where the same code is run for each of the three parts uses around 1/3 of the code size of the fully unrolled variant, but adds 50 cycles of execution time, mainly due to words containing zero having to be initially stored to the stack, which are then loaded during the first part. Between each part, the stack pointer is decreased by five words.

Note that the described algorithm is scalable and works for any sizes of a and b. If a is not divisible by five, either simply pad it with zeros after the most significant word or modify the last part to operate on the correct number of words. The algorithm uses $n^2 + (3n + 16)n/5 - 10$ cycles for any n by n-word multiplication, when n is divisible by 5 and $n > 5$. For any n by m-word multiplication, where n is the number of words in a and m is the number of words in b, the algorithm uses $n \cdot m + (3n + 3)\lceil m/5 \rceil + 3\lfloor m/5 \rfloor + \lfloor (m \mod 5) \cdot 0.76 \rceil + 2m - 10$ cycles when $n \geq 5$ and $m > 5$. The expression $\lfloor (m \mod 5) \cdot 0.76 \rceil$ corresponds to how many cycles it takes to set $m \mod 5$ registers to zero (0, 1, 2, 2 or 3 cycles).

On Cortex-M33, this general algorithm could be optimised further by using six words per part instead of five. This is possible because the last `ldr` instruction in an `ldr` chain does not have to be paired with an `str` instruction to avoid an extra cycle. Thus, the spare register can be used to hold the sixth operand. To free a register for use as the sixth word in t, we could discard the address register for a, which then requires us to copy a to a fixed location on the stack (if this is not already the case). This can, for example, be performed during the first part (we use five words in the first part) since we already load the entire a during the first part. If m is a multiple of six, however, it is more efficient to copy a to the stack before starting the algorithm, to avoid an extra part.

On both Cortex-M4F and Cortex-M33F, we can achieve six words per part without any pipeline stalls by having the a operand in FPU registers, if the sequence `ldr`, `str`, `vmov` is used between rows (the `ldr` instruction overwrites the register where the a operand word was previously used).

Using six words for a part can still be a useful strategy on Cortex-M4 (without FPU) to save around n cycles, when $m \equiv 1 \pmod 5$. In this case, use six words in the last part, but otherwise use five words per part. We avoid $3n$ cycles by avoiding the last part having only one word in b (spill, load operand word, restore), but instead add one cycle to store a copy of a_i in the first part and get one extra cycle when the cpu is stalled in the last part, per row.

In [11], Seo et al. propose a method they call "Refined Operand Caching", that requires $2\lceil n^2/(e+1) \rceil + 3\lfloor n/(e+1) \rfloor$ loads and $\lceil n^2/(e+1) \rceil + n$ stores, with the constant $e = 3$, in addition to the n^2 partial products. Their method is then used in their Curve448 implementation [10] as a 448-bit multiplier. It uses four registers for operand words in a, four registers for operand words in b, four registers for intermediate results, one temporal register, and one register for either the address of a or b. As this method does not have a spare register, it will suffer from many pipeline stalls at the end of `ldr` sequences. Additionally, we see that the formula of the total memory overhead is similar to our proposed method, with the important difference that $e + 1 = 4$ is used as divisor instead of 5. Our proposed method thus has lower memory overhead, which is directly translatable to fewer cycles. With our method, it is also possible to decide how much the loops should be unrolled, to save code space, thanks to the more regular pattern.

The work by Anastasova et al. [2] also uses five words in the inner loop. The inner loop consists of five `umaal` instructions, one `ldr` instruction and two `vmov` instructions. The `ldr` instruction loads the next operand word and the two `vmov` instructions load and store intermediate results from and to the floating point register set, respectively. This is very similar to our method, but uses floating point registers instead of the stack. However, the order of the instructions is different: each load is scheduled just before the loaded value is required, and the store is scheduled immediately after the value to store was produced. In any case, `ldr` instructions not followed by another `ldr` or `str` instruction suffer from pipeline stalls on Cortex-M4, which makes our method one cycle faster per inner loop. It is unclear if `ldr` is used rather than `ldr.w`. This difference would also cause an extra cycle, in case the following 32-bit instructions are not 32-bit aligned.

Algorithm 1 Multiplication, inner loop, two iterations, fully unrolled, 16 cycles

```
// start row 1 at col 10
ldr r0,[r1,#4*1]
ldr r2,[sp,#4*11]
str.w r3,[sp,#4*10]
umaal r2,r4,r0,r6
umaal r4,r8,r0,r7
umaal r8,r12,r0,r9
umaal r12,lr,r0,r10
umaal lr,r5,r0,r11
// start row 2 at col 10
ldr r0,[r1,#4*2]
ldr r3,[sp,#4*12]
str.w r2,[sp,#4*11]
umaal r3,r4,r0,r6
umaal r4,r8,r0,r7
umaal r8,r12,r0,r9
umaal r12,lr,r0,r10
umaal lr,r5,r0,r11
```

5.3 256-bit squaring

We use the fact that 28 of the 64 products are computed twice, since $a_i b_j = a_j b_i$ when $a = b$. Only products where $i = j$ are computed once. Two different algorithms will now be described.

5.3.1 Variant 1

Let <<, >> indicate left and right logical shift, respectively. The first algorithm is to calculate the result as $2X + Y$ where $X = \sum_{i=0}^{6} \sum_{j=i+1}^{7} (a_i a_j << 32(i+j))$ and $Y = \sum_{i=0}^{7} (a_i^2 << 64i)$. Since there is only one operand a, all its eight words can fit into registers during the entire calculation, as long as they are needed. When calculating X colmun by column from right to left, it happens six times that a column has the same number of products to calculate as the next column. When such a column has n products to calculate, n temporary registers, holding the upper 32 bits of each 64-bit product, will be produced. We cannot perform a chain of n `umaal` instructions in the next column with only n accumulate inputs. We need $n+1$ inputs, so therefore extra instructions are necessary to zero out a free register. Replacing `umaal` instructions by `umull` instructions will not help, as it will not change the parity of the number of needed inputs.

The diagram in Figure 2 shows an optimised ordering of the multiplications that allows us to keep as many result words in registers as possible, while not sacrificing cycles in other ways.

						15	13	9	7	**5**	3*	**2**	(**1**)	
					18	17	14	**11**	8*	6	(4)			
				22	20	**19**	16*	12	(10)					
			25	24	23	21	(26)							
		29	28	27*	(30)									
	32	31*	(33)											
34	(35)													
(36)														

Figure 2: Multiplication order. A star indicates a `umaal` instruction where one of the accumulator registers is zero, **bold** indicates `umull`, and parentheses that this product shall not be doubled. The strategy is, just as with the multiplication case, to calculate row by row as much as is possible, to free up the registers used for the inputs at the earliest possible time.

X is doubled using a chain of 15 `adds`/`adcs` instructions (since $2X = X + X$) and the result fits in 16 words (the first implicit word is always zero and thus never calculated). When calculating Y, we multiply-accumulate directly into $2X$ using a sequence of `umaal` instructions where we alternately multiply a_i by a_i and 0 by 0. The

latter zero-products are required to get the carry chain correct. We interleave the calculation of $2X$ and Y as shown in the diagram above. As the calculation proceeds, a_i operands start to become dead, and these registers can instead be used to hold result words to avoid storing them to the stack.

Assuming the a operand is already placed in registers, the calculation requires in total 69 cycles. Six result words will be stored on the stack at a fixed offset (lower words), and ten result words will be stored in registers (upper words). It is possible to store only the first five words on the stack and have eleven in registers, but that results in higher register pressure, resulting in a two cycle penalty of lost optimisation opportunities (using `umull` to zero two registers and keeping one register zeroed during the entire calculation).

5.3.2 Variant 2

A second, faster, algorithm will now be described, which has not been observed in any previous work on Curve25519 for Cortex-M4. The algorithm above uses 15 `adds`/`adcs` instructions for doubling. An additional eight instructions are required where 0 is multiplied by 0 to keep the carry chain correct. We would like an algorithm which reduces the overhead of these 23 cycles.

In the algorithm above, the doubling in $2X$ is performed *after* the partial multiplications. The other approach is to double one of the two operands *before* the partial multiplications. Let us rewrite $2X$ as $2X = 2\sum_{j=1}^{7}\sum_{i=0}^{j-1}(a_i a_j << 32(i+j)) = \sum_{j=1}^{7}(2(\sum_{i=0}^{j-1}(a_i << 32i))a_j << 32j) = \sum_{j=1}^{7}((2a_{0..j-1})a_j << 32j)$.

The solution is therefore to initially calculate $2a$ and then multiply $(2a)_i$ by a_j for $i < j$ and $a_i a_j$ for $i = j$, i.e. 36 multiplications. We additionally need to multiply a_j by the top bit of $2a_{0..j-1}$, which can efficiently be calculated in one cycle using "`and Rd,Rn,Rm, asr #31`", where `Rn` is a_j and `Rm` is a_{j-1}. The result of each `and` operation is used as an accumulate operand to a following appropriate `umaal` instruction. It is otherwise sufficient to calculate the first seven words of $2a$, which is performed using a chain of `adds`/`adcs` instructions. Together with the seven `and` instructions, this algorithm has a computational overhead of only 14 cycles, which is 9 cycles less than the first algorithm variant which had 23 cycles of overhead.

We show the order the multiplications are performed to minimize register usage in Figure 3.

						14	12	10	7	5	3+	2*	**1**
					18	17	13+	11	8+	6*	4*		
				22	21	20*	16	15*	9*				
			26	25	24	23	19*						
		30	29	28	27+								
	33	32	31+										
35	34+												
36+													

Figure 3: Multiplication order. A star indicates a `umaal` instruction where one of the accumulator registers is zero, **bold** indicates `umull`, and a plus sign indicates that we here use the result of the `and` instruction as one of the accumulate operands.

As an example in the multiplication order above, the third multiplication operation multiplies $(2a)_0$ by a_2 using a `umaal` instruction having the high result from the second multiplication operation as the first accumulate operand, and the result of the multiplication $(a_0 >> 31)a_1$ as the second accumulate operand.

Just as in the first variant, we have seven accumulate operands that need to be zero. One `movs` and three `umull` instructions are used to produce these.

Note that the `adds`/`adcs` operation for a_i is performed after the calculation a_i^2. This way, we never have to keep both a_i and $(2a)_i$ in registers at the same time.

In total, the algorithm uses 59 cycles, assuming the a operand is already placed in registers. At the end, the five lower result words are located on the stack (whose `str` instructions account for five cycles) and the upper eleven are located in registers.

The work by Haase et al. [4] uses the approach in Variant 1 for optimising the $2X + Y$ calculation while this work uses Variant 2, which is what mainly improves the performance for the squaring operation compared to their work.

5.4 480-bit squaring

We use the same technique as in the 256-bit squaring case (the second variant), but with 15 input words this time. Due to the larger operand, we cannot keep both the full operand and temporaries in registers anymore.

We will instead split up the operation into three parts. The first part handles the first four rows, the second part handles the next five rows, and the last part handles the last six rows. For each part, we initially load the next four, five, or six words of the input. These words stay in registers during that part. Each part will then process one word at a time from the full a where one temporary word will be loaded and stored, as shown in the algorithm below.

Algorithm 2 480-bit squaring

INPUT: An input operand a, stored as an array of 32-bit words a_i.
OUTPUT: a^2.
$t_{0..14} = 0$
$d_0 = 0$
for $(start, end)$ in $(0, 3), (4, 8), (9, 14)$ **do**
  # Load $a_{start..end}$ to registers
  $v = 0$
  for $j = start$ to 14 **do**
    # Load a_j to register (if $j \notin [start..end]$)
    $v = v + t_j$ # v is stored using multiple words, and this addition is deferred until the next `umaal`
    for $i = start$ to end **do**
      if $i \leq j$ **then**
        $v = v + (a_i a_j << 32(i - start))$
        if $i = j$ **then**
          $v = v + (d_i << 32(i - start))$ # combined with previous addition in a `umaal`
          $d_{i+1} = (a_i >> 31)a_{i+1}$
          $a_i = 2a_i \mod 2^{32}$
        end if
      end if
    end for
    $t_j = v \mod 2^{32}$
    $v = v >> 32$
  end for
  $t_{15+start..15+end} = v$
end for
return $t_{0..29}$

The above algorithm is fully unrolled and is slightly adjusted. The exact order of the operations is shown in Figure 4, using the same notation as in the 256-bit case.

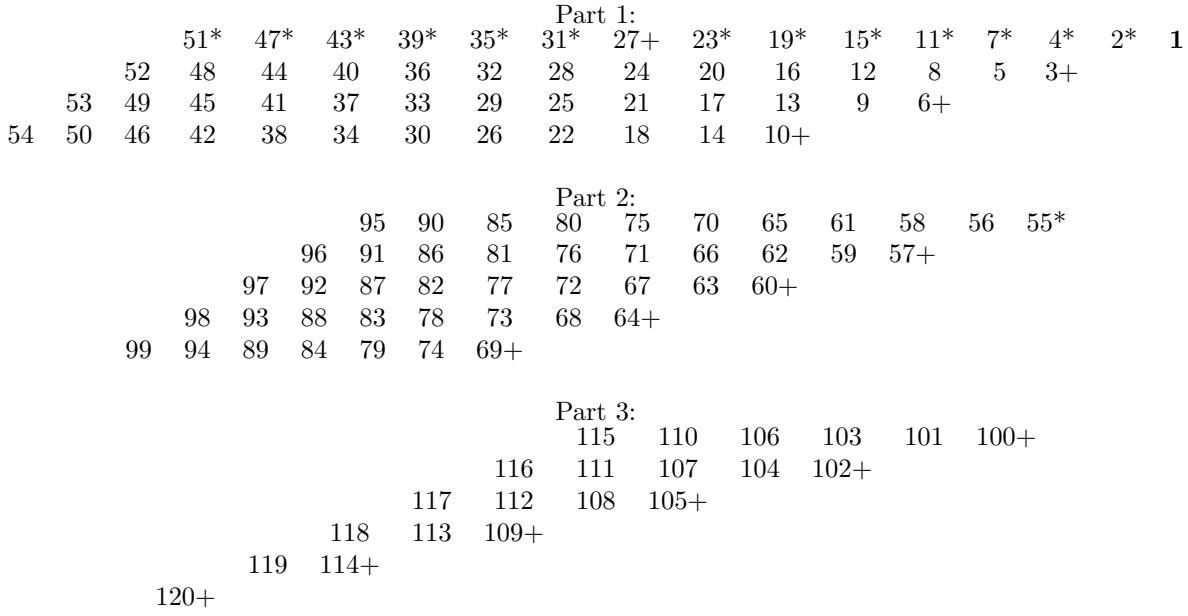

Figure 4: Multiplication order for 480-bit squaring. A star indicates a `umaal` instruction where one of the accumulator registers is zero, **bold** indicates `umull`, and a plus sign indicates that we here use the result of the `and` instruction as one of the accumulate operands.

A group of five rows is generally the maxmium we can process at once, due to the number of registers available. For the first part, we would like to have zero in a spare register so we can set two other registers to zero using `umull`. Therefore, for the best performance, we use four rows in this part. For the last part, we can process six rows at once, since there is no need to load a different operand register than the six belonging to this part. It turns out that the last part does not need any loads from the stack, and the second part does not need to spill anything to memory at its exit, since what is needed for subsequent calculations can fit in registers at this point.

In total, 250 cycles are needed, assuming the address of the a operand is present in a register. At the end, the 23 lower words of the result are stored on the stack, and the upper seven words are stored in registers.

6 Curve448 field arithmetic

A compact representation of a field element of 15 words is used. It would have been enough with 14 words, but an extra word has been added to simplify partial reductions. This will cost some extra time during multiplication and squaring operations, but will reduce the time performing field reductions. In particular, we skip field reductions altogether for addition and subtraction. Note that this puts constraints on how many additions or subtractions that can be performed in sequence before the value is consumed by a multiplication or squaring operation. This idea has not been observed in any previous work for Curve448 that targets Cortex-M processors.

Other implementations might use 2^{28} as radix, but due to the availability of `umaal`, which essentially performs two additions for free, and also carries bits over the 32-bit boundary, a radix of 2^{32} is a better choice on Cortex-M processors.

6.1 Subtraction

To subtract two values a and b modulo p, we calculate $(a + 2p) - b$. The result will always be non-negative, and thus correct, when b is at most $2p$. If also a is small enough, the result will fit in 15 words. To allow for two carry chains simultaneously and thus be able to perform the operation in one single pass, we use `umaal` to calculate $a+2p$ and `subs`/`sbcs` to then subtract b. No modular reduction is performed.

6.2 Addition and multiplication by small constant

To save code space, addition and "multiply by a small constant and add" uses the same code. We calculate $a + cb$, where a and b are the inputs and c is a constant that fits in one word. The constant is 1 for normal addition

and 39082 for multiplication by $(A+2)/4$, where A is the curve constant 156326. The operation is performed efficiently using `umaal` without performing any modular arithmetic. Given inputs that are small enough, the result will always fit in 15 words. It would require at least one extra pass if we would reduce fully modulo p, or reduce the result to fit in 448 bits.

6.3 Multiplication and squaring

Hamburg hints that Karatsuba multiplication can be used for field multiplication [5]. Let $\phi = 2^{224}$. Then

$$\begin{aligned}
&(a + b\phi) \cdot (c + d\phi) \\
&= ac + (ad + bc)\phi + bd\phi^2 \\
&\equiv (ac + bd) + (ad + bc + bd)\phi \pmod{p} \\
&= (ac + bd) + ((a + b)(c + d) - ac)\phi
\end{aligned}$$

This algorithm has been implemented and uses 540 cycles. It performs one 224-bit multiplication followed by two 256-bit multiplications, and finally a reduction part. It turned out this approach uses around 40 cycles more than the more simple operand scanning approach and was thus abandoned.

The 480-bit multiplication and squaring algorithms described earlier are used, resulting in 960-bit values. To slightly enhance the performance, we discard the upper 32 bits of the result and reduce the lower 928 bits modulo p. The input operands must hence be small enough (at most in total 928 significant bits) for the result to be correct. When squaring, the operand must be at most 464 bits. With $\phi = 2^{224}$, we will use the fact that $\phi^2 \equiv \phi + 1 \pmod{p}$, $\phi^3 \equiv 2\phi + 1 \pmod{p}$, and $\phi^4 \equiv 3\phi + 2 \pmod{p}$.

Then, with the 928-bit result written as $A + B\phi + C\phi^2 + D\phi^3 + E\phi^4$, where A, B, C, D are all 224 bits each, and E is 32 bits, we reduce as $(A + C + D + 2E) + (B + C + 2D + 3E)\phi$. To make the result fit in 449 bits, we start the reduction by adding the upper 32-bit words in B, C, D and D, resulting in 34 bits. The upper two bits F are removed and will instead be added as $F + F\phi$ to the result.

The additions are processed using two chains in parallel. First, word 0 and word 7 are produced by first loading the needed words from the stack and then using `umaal` followed by `umlal` instructions to add words (using the constant 1 or 2 as one of the multiplicands). Then, word 1 and word 8 are produced and so on. Words 0 to 6 can be stored immediately to the destination. When all words have been computed, we must use a sequence of `adds/adcs` to add and propagate the carry from word 6. Despite the instructions needed for this carry propagation, it is still cheaper than producing the words from lowest to highest in sequence, since C and D then need to be loaded twice. The result will fit in 449 bits and be less than $2p$. To save a few memory operations, some of the most significant words in the 928-bit multiplication result are never stored to the stack, but just kept in memory until they are to be used. The reduction code is shared and used for both multiplication and squaring and uses in total 90 cycles.

6.4 Inversion

Side-channel resistant field inversion is implemented using Fermat's little theorem $a^{-1} \equiv a^{p-2} \pmod{p}$. To perform the modular exponentiation, a sequence of 447 squarings and 13 multiplications is used.

6.5 Final reduction

Final reduction is performed after the last multiplication operation. We may thus reduce the top bit to make the result fit in 448 bits. To handle the case when the value is still larger than or equal to p, we subtract by p and then conditionally in constant time add p if the result becomes negative, or 0 otherwise. The reduction of the top bit and the subtraction of p is merged into a single pass.

7 Curve25519 field arithmetic

Similarly to Curve448, we use a compact representation with 8 words for a field element. The stored field element is an integer less than $2^{256} - 38$. This allows us to partially reduce either using modulo $p = 2^{255} - 19$ or $2p = 2^{256} - 38$.

7.1 Subtraction

First, the two operands are subtracted. If the result becomes negative, we add $2p$ to make it positive. To make the timing independent of the values being subtracted, a mask is created using the resulting borrow bit to either add 0 or $2p$.

7.2 Addition and multiplication by small constant

Just as for Curve448, we can see addition as a special case of "multiply by a small constant and add" thanks to the powerful `umaal`. The constants that will be used for Curve25519 are 1 and 121666.

First, we use `umaal` on the highest word in each operand, resulting in a value spanning two words. The lower 31 bits are saved for later use, and the upper bits are multiplied by 19, which will be used as carry-in for the rest of the addition, starting from the first word. After the addition of the seven lowest words, the resulting carry word will be added to the 31 bits we saved earlier, producing a 256-bit result less than $2^{256} - 38$.

7.3 Multiplication and squaring

The 256-bit multiplication and squaring algorithms return a 512-bit result. Since $2^{256} \equiv 38 \pmod{p}$, reduction can use the "multiply by a small constant and add" algorithm with the constant 38, where the operands are the upper half and the lower half of the 512-bit result, respectively. Conveniently, the multiplication and squaring algorithms we use leave the upper words in registers and can be consumed directly by the reduction algorithm without having to be spilled to and restored from the stack.

7.4 Inversion

Side-channel resistant field inversion is implemented using Fermat's little theorem $a^{-1} \equiv a^{p-2} \pmod{p}$. To perform the modular exponentiation, a sequence of 254 squarings and 11 multiplications is used.

7.5 Final reduction

The result is stored using 256 bits, modulo $2p = 2^{256} - 38$. We need to reduce this value modulo $p = 2^{255} - 19$. This is done by simply subtracting p from the value. If the result becomes negative, we add p. This is done using a mask by either adding 0 or p, to avoid branches.

8 Montgomery ladder

To perform elliptic curve scalar multiplication with a variable base point, the Montgomery ladder is used [7]. The Montgomery ladder is commonly used when it is desired to mitigate side-channel attacks, since the same operations are performed regardless of the secret scalar bits. Algorithm 3 shows the Montgomery ladder in its basic form.

Algorithm 3 Montgomery ladder

INPUT: A scalar $k \in [0, 2^m - 1]$ and a point P.
OUTPUT: kP.
$R_0 = 0$
$R_1 = P$
Treat bits in k as $b_0, b_1, \cdots, b_{m-1}$ from LSB to MSB
for $i = m - 1$ to 0, step -1 **do**
    if $b_i = 0$ **then**
        $(R_0, R_1) = (2R_0, R_0 + R_1)$
    else
        $(R_1, R_0) = (2R_1, R_0 + R_1)$
    end if
end for
return R_0

Each iteration will either double the current point R_0 if the bit was zero or double the point and add P if the bit was one, which thus constitutes the scalar multiplication kP.

Initially $R_1 - R_0 = P$. After every iteration, regardless of b_i, we can observe that this property always continue to hold.

We will use x/z representation of curve points and will not need the y component. This representation has efficient known operations for calculating point addition $P_0 + P_1$ when $P_1 - P_0$ is known, and for calculating point doubling $2P$. These operations, in particular the addition operation, work with the Montgomery ladder, since we know that $R_1 - R_0$ is always P. Note that on Montgomery curves, if $P = (x, y)$, then $-P = (x, -y)$. Thus, we can swap R_0 and R_1 in the addition formula and still get the same result, since $R_0 - R_1$ and $R_1 - R_0$ have the same representation when the y coordinate is not used.

We would like to avoid the branches in the algorithm above. The bit b_i decides which of R_0 and R_1 to use for point doubling as well as in what order to store the results. For the point addition, it does not matter whether the operands are swapped or not. We can rewrite the algorithm to always store the results of the point doubling and the point addition to the same locations if we swap the operands at the beginning of the next iteration. Therefore, we need to save the bit for the next iteration. Since the point doubling uses only one of the operands, the conditional swap can be optimised to a conditional select operation. If the current bit and the previous bit are equal, R_0 is chosen, otherwise R_1 is chosen as the input for point doubling. Algorithm 4 shows this amended variant. The CSEL function takes a condition as the first parameter. If the condition is true, the second parameter is returned. If the condition is false, the third parameter is returned.

Algorithm 4 Side-channel resistant Montgomery ladder

INPUT: A scalar $k \in [0, 2^m - 1]$ and a point P.
OUTPUT: kP.
$R_0 = 0$
$R_1 = P$
$lastbit = 0$
Treat bits in k as $b_0, b_1, \cdots, b_{m-1}$ from LSB to MSB
for $i = m - 1$ to 0, step -1 **do**
  $T = CSEL(b_i \neq lastbit, R_1, R_0)$
  $(R_0, R_1) = (2T, R_0 + R_1)$
  $lastbit = b_i$
end for
return $CSEL(lastbit = 1, R_1, R_0)$

Note that for X25519 and X448, the least significant bit is always 0, due to clamping of the scalar, so the last CSEL can be optimised away so that R_0 is always returned.

The steps for performing point addition and point doubling are shown below in Algorithms 5 and 6, respectively.

Algorithm 5 Point addition

INPUT: Points $(X2 : Z2)$, $(X3 : Z3)$, $(X1 : 1)$, where $(X1 : 1) = (X2 : Z2) - (X3 : Z3)$
OUTPUT: The point $(X5 : Z5) = (X2 : Z2) + (X3 : Z3)$
$A = X2 + Z2$
$B = X2 - Z2$
$C = X3 + Z3$
$D = X3 - Z3$
$DA = D \cdot A$
$CB = C \cdot B$
$T1 = DA - CB$
$T3 = T1^2$
$T2 = DA + CB$
$X5 = T2^2$
$Z5 = X1 \cdot T3$

Algorithm 6 Point doubling

INPUT: Point $(X2 : Z2)$
OUTPUT: The point $(X4 : Z4) = 2(X2 : Z2)$
$A = X2 + Z2$
$B = X2 - Z2$
$AA = A^2$
$BB = B^2$
$E = AA - BB$
$T = BB + a24 \cdot E$
$X4 = AA \cdot BB$
$Z4 = E \cdot T$

After inlining these operations into the Montgomery ladder, we get Algorithm 7, which also uses the minimal number of temporary field elements.

Algorithm 7 Side-channel resistant Montgomery ladder

INPUT: An even scalar $k \in [0, 2^m - 1]$ and a point $X1$.
OUTPUT: $kX1$.
$X2 = 1$
$Z2 = 0$
$X3 = X1$
$Z3 = 1$
Treat bits in k as $b_0, b_1, \cdots, b_{m-1}$ from LSB to MSB
$lastbit = 0$
for $i = m - 1$ to 0, step -1 **do**
  $A = X2 + Z2$
  $B = X2 - Z2$
  $C = X3 + Z3$
  $D = X3 - Z3$
  $DA = D \cdot A$
  $AA = CSEL(b_i \neq lastbit, C, A)^2$
  $CB = C \cdot B$
  $BB = CSEL(b_i \neq lastbit, D, B)^2$
  $T1 = DA - CB$
  $T3 = T1^2$
  $T2 = DA + CB$
  $X5 = T2^2$
  $Z5 = X1 \cdot T3$
  $E = AA - BB$
  $T = BB + a24 \cdot E$
  $X4 = AA \cdot BB$
  $Z4 = E \cdot T$
  $Z2 = Z4$
  $X2 = X4$
  $Z3 = Z5$
  $X3 = X5$
  $lastbit = b_i$
end for
return $(X2, Z2)$

Table 1 shows what variables occupy the same memory space:

0	A, AA
1	B, E
2	X2, C, BB, X4
3	Z2, D, T1, T3, T, Z4
4	X3, CB, T2, X5
5	Z3, DA, Z5
6	X1

Table 1: Register allocation for field registers

It is important to note that the CSEL operation can either be performed by selecting what pointer to use as input to the subsequent squaring function, or alternatively, by loading both values to registers, conditionally selecting words, writing the results to memory, and then using that memory location as input to the subsequent squaring routine. On processors with data caches for the RAM (not flash data caches), the latter is usually suggested since it avoids cache timing leakage. If the processor does not have a data cache, however, it is not only faster with the former approach, but also potentially exhibits less leakage [8]. Since Cortex-M4 and Cortex-M33 devices do not have data caches, this former approach is chosen.

This work aims to be resistant against timing side-channel attacks, which is the same type of attacks that the original Curve25519 reference implementation claims to be resistant against, as mentioned in the original paper [3]. Note that it might in some cases be required to have an implementation that is resistant to attacks using other side-channels than timing. The work by Nascimento et al. [8] uses power analysis to attack the conditional move of either the entire field value or the pointer that contains the address of the field value to load. The authors claim that typical countermeasures such as adding a random multiple of the curve's order to the scalar are inefficient to their attack, since they only need one trace to recover the secret scalar, and that randomising the Z coordinate for the initial base point is also inefficient to their attack, since they extract the secret scalar using the leakage from the conditional moves only. Instead, they propose other countermeasures, such as randomly swapping addresses for the field registers in the inner loop of the Montgomery ladder. What are the best and most efficient countermeasures are most likely dependent on the target system in question, and such a study is out of scope for this work.

9 Implementations

The X25519 and X448 functions have been implemented using the techniques above, targeting both Cortex-M4 and Cortex-M33. Separate implementations have not been created for the two processors. Instead, the implementations are optimised primarily for Cortex-M4 and secondarily for Cortex-M33. The optimisations for Cortex-M33 have mainly been done by reordering instructions to avoid data hazards while never decreasing the performance for Cortex-M4.

The implementations are written in 100% handwritten Assembly language, except for most of the 480-bit multiplier, which could be generated by a script due to its regular pattern. One could suggest to implement the field operations in Assembly while implementing the high-level Montgomery ladder in a higher-level language like C. The main reason for having everything in Assembly is to be able to use custom ABIs for every function.

The standard ABI used in C requires the callee to save registers `r4` to `r11` at entry and restore them at exit. If the function uses all these registers, this will incur an overhead of at least 16 cycles per function invocation, which would decrease performance by around 15% for X25519. Additionally, for X25519, field operations return the result in registers, rather than storing it at an address passed as an argument to the function. Specifically for the squaring function, the eight input words are also passed in registers, rather than passing a pointer. This will, under the right circumstances, avoid extra memory instructions for loading from and storing to the stack.

To save code space, the sequence of multiplication and square operations that inverts a field element is not stored as a sequence of individual function calls. Instead, a table containing all those operations has been written, using one 16-bit word per operation. A tiny interpreter then processes this table to perform the operations. Since only one inversion is performed per scalar multiplication and the number of operations is relatively small, this has negligible impact on performance.

For each curve, "speedopt" and "sizeopt" variants have been created. The sizeopt variant is intended to save code space while sacrificing some performance. This is mainly done by omitting the squaring function. The multiplication function is then used for both squaring and multiplication. For the Curve448 field multiplication, the "inner loop" is not unrolled in the sizeopt variant but is executed three times.

For X25519 on Cortex-M4F, a variant that makes use of the FPU has also been created. In this variant, some memory instructions are replaced by `vmov` instructions in the multiplication and squaring functions. This variant is pointless on Cortex-M33 since memory instructions are optimised to only use one cycle per word.

9.1 Performance

The work has been tested on nRF52840 (Cortex-M4F), nRF5340 (Cortex-M33 with FPU and DSP extensions) and STM32G431KBT6 (Cortex-M4F).

The nRF52840 uses a constant CPU frequency of 64 MHz, features a 2 kB instruction cache (I-Cache) and can also run code from "Code RAM", which is an alias of the system RAM mapped at 0x08000000 for the ICode bus. Running code from Code RAM will result in zero wait states. Running code from flash will result in instructions being fetched with zero wait states, assuming the cache is turned on and there is a cache hit.

The nRF5340 can be configured to a CPU frequency of either 64 or 128 MHz. Compared to nRF52840, it has a larger 8 kB, two-way set associative instruction/data cache (CACHE) that caches flash accesses. This cache is large enough for each of our implementations to get a negligible amount of cache misses.

The STM32G431KBT6 has a configurable CPU frequency up to 170 MHz. It features a 1 kB flash instruction cache and a 256 byte flash data cache. It is equipped with a proprietary Adaptive real-time (ART) memory accelerator that implements an instruction prefetch queue and branch cache which can increase execution speed when running from flash. The instruction prefetch queue can accelerate sequential flash reads when cache misses occur by speculatively reading the next word from flash before the CPU requests it. This block must be manually enabled after boot. The number of wait states added when running from flash is configurable, but the minimum allowed value depends on the configured CPU frequency and voltage level. The reference manual contains a table that shows all the allowed combinations. In the boost mode (voltage range 1), zero wait states can be configured for up to 34 MHz, two wait states can be configured for up to 102 MHz, and four wait states can be configured for up to 170 MHz (the maximum possible speed). The device also has a dedicated block of 10 kB RAM called CCM SRAM that can be accessed on the I-bus, that allows zero wait states execution even at 170 MHz.

9.1.1 Results

Here, execution times are shown, measured in clock cycles. Code size and stack usage are measured in bytes. For X25519, "fpu" here indicates that optimisations that utilise the FPU register file are used and "dcache" indicates that the code is side-channel protected for devices having a data cache for RAM, i.e. the RAM access pattern is not dependent on the secret scalar. The "dcache" implementations are primarily made for performance comparison and are not intended to be used in practice when targeting Cortex-M4 and Cortex-M33 processors for the reasons discussed in Section 8.

Implementation	CPU	mul	sqr	add	sub
X25519	nRF52840 Code RAM	147	91	39	41
X25519 + fpu	nRF52840 Code RAM	140	88	39	41
X25519	nRF5340 CACHE	137	89	39	39
X448 speedopt	nRF52840 Code RAM	501	356	81	104
X448 sizeopt	nRF52840 Code RAM	554	554	81	104
X448 speedopt	nRF5340 CACHE	492	339	65	88
X448 sizeopt	nRF5340 CACHE	545	545	67	90

Table 2: Field arithmetic cycle counts

Table 2 shows cycle counts for various field operations, including function call overhead. For X25519, the cycle counts do not include the time needed to set the input registers nor the time needed to store the result to memory, which must be performed by the caller. For multiplication, addition and subtraction, this corresponds to setting two pointers as inputs and storing eight result words to memory. For squaring, this corresponds to loading eight words from memory and storing eight words to memory. For X448, the cycle counts include the time needed to set pointers for the input and output.

Table 3 shows the performance of the full X25519 or X448 function.

The FPU optimisations result in some speedups on Cortex-M4, but as explained in earlier sections, do not increase the performance on Cortex-M33. The performance drop that instead occurs is not due to `vmov` being slower than memory instructions, but that the multiplication routine is modified to initially copy the full a operand into FPU registers, so that we can then load the same 32-bit values multiple times in one cycle each (on Cortex-M4).

Implementation	Code size	Stack usage	nRF52840 Code RAM	nRF52840 I-Cache	nRF5340 CACHE
X25519 speedopt	1 472	348	441 116	441 332	411 061
X25519 speedopt + fpu	1 484	312	428 283	428 470	418 769
X25519 speedopt + dcache	1 544	348	457 434	457 615	425 085
X25519 speedopt + fpu + dcache	1 556	312	444 603	444 914	432 794
X25519 sizeopt	1 132	348	520 750	520 874	479 243
X448 speedopt	3 074	644	2 273 479	2 551 377	2 170 710
X448 sizeopt	1 548	664	2 836 304	2 836 673	2 744 689

Implementation	STM32G431 CCM SRAM 170 MHz	STM32G431 Cache 170 MHz	STM32G431 Cache/Prefetch 170 MHz	STM32G431 Cache/Prefetch 102 MHz	STM32G431 Cache/Prefetch 32 MHz
X25519 speedopt	441 401	496 255	481 121	450 108	441 117
X25519 speedopt + fpu	428 570	483 189	472 901	439 836	428 286
X25519 speedopt + dcache	457 721	555 666	526 007	472 543	457 438
X25519 speedopt + fpu + dcache	444 890	550 511	519 322	462 535	444 606
X25519 sizeopt	521 045	521 112	521 032	520 824	520 751
X448 speedopt	2 273 520	4 943 967	4 172 126	2 691 806	2 273 479
X448 sizeopt	2 836 342	3 407 184	3 229 748	2 923 710	2 836 305

Table 3: Full scalarmult code size, stack usage, and cycle counts

The total cost, despite more transfer operations compared to the non-FPU based approach, is less on Cortex-M4, but more on Cortex-M33.

The effect on cache size and code size is particularly noticeable on the STM32G431. The performance when running code from flash with cache will decrease as the CPU frequency is increased, in terms of number of cycles. But the performance will also decrease as the code size increases (selecting speedopt instead of sizeopt) — even though the algorithm optimised for speed should be faster — due to an increased amount of cache misses. The X448 sizeopt implementation is significantly faster than the speedopt implementation at 170 MHz when running from flash. For the X25519 sizeopt implementation, the main loop apparently fits in the 1 kB instruction cache, resulting in about the same number of cycles regardless of CPU frequency, while the speedopt implementations get slower as the CPU frequency is increased. On nRF52840, the only implementation that has a negative impact when running from flash with the instruction cache enabled instead of from Code RAM is the X448 speedopt implementation, which is larger than the 2 kB cache size. The results thus show the importance of optimising not only for "performance", but also for size.

10 Conclusion

This work has shown that accurate knowledge about the Cortex-M4 and Cortex-M33 processors is of great importance when designing highly efficient implementations for cryptographic algorithms, as well as well-thought optimisations at every possible level. The implementations presented should be close to optimal performance as is theoretically possible within the code size and stack usage budget used, at least for Cortex-M4.

It is expected that the presented multiplication and squaring algorithms could be useful in other applications as well that require fast big integer arithmetic on Cortex-M4 and Cortex-M33. The findings about cycle timings not found in ARM's manuals should be of interest for an even further range of applications.

This work only evaluates performance for variable base scalar multiplication. Using the same field arithmetic functions and multiplication algorithms, it should be possible to implement other high-level algorithms as well, e.g. fixed base point scalar multiplication for Ed25519, or implementations for the NIST curves.

11 Acknowledgements

Thanks to Björn Haase for giving useful feedback on this paper.

References

[1] Arm® Cortex®-M4 Processor Technical Reference Manual. https://developer.arm.com/documentation/100166/0001.

[2] Mila Anastasova, Reza Azarderakhsh, Mehran Mozaffari Kermani, and Lubjana Beshaj. Time-Efficient Finite Field Microarchitecture Design for Curve448 and Ed448 on Cortex-M4. Cryptology ePrint Archive, Paper 2023/168, 2023. https://ia.cr/2023/168.

[3] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. Cryptology ePrint Archive, Report 2018/286, 2018. https://ia.cr/2018/286.

[5] Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. https://ia.cr/2015/625.

[6] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. RFC 7748, RFC Editor, January 2016.

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

[8] Erick Nascimento, Lukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ECC implementations through cmov side channels. Cryptology ePrint Archive, Report 2016/923, 2016. https://ia.cr/2016/923.

[9] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, August 2018.

[10] 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.

[11] Hwajeong Soe, Amir Jalali, and Reza Azarderakhsh. SIKE Round 2 Speed Record on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/535, 2019. https://ia.cr/2019/535.