High-Order and Cortex-M4 First-Order implementations of Masked FrodoKEM

François Gérard[1] and Morgane Guerreau[2]

[1] University of Luxembourg
francois.gerard@uni.lu
[2] CryptoNext Security, Paris, France
morgane.guerreau@cryptonext-security.com

Abstract. The key encapsulation mechanism FrodoKEM is a post-quantum algorithm based on plain LWE. While it has not been selected by the NIST for standardization, FrodoKEM shares a lot of similarities with the lattice-based standard ML-KEM and offers strong security assumptions by relying on the unstructured version of the LWE problem. This leads FrodoKEM to be recommended by European agencies ANSSI and BSI as a possible choice to obtain post-quantum security. In this paper, we discuss the practical aspects of incorporating side-channel protections in FrodoKEM by describing a fully masked version of the scheme based on several previous works on LWE-based KEMs. Furthermore, we propose an arbitrary order C implementation based on the reference code and a Cortex-M4 implementation with gadgets specialized at order 1 in low level assembly code that incorporates bespoke modifications to thwart (micro-)architectural leakages. Finally, we validate our order 1 gadgets by performing TVLA on a ChipWhisperer.

Keywords: Frodo · SCA · KEM · Masking · Cortex-M4

1 Introduction

The NIST Post-Quantum Standardization process led to the selection of two Key Encapsulation Mechanism (KEM), Kyber [SAB+22] and HQC [AAB+22], the former being already standardized under the name ML-KEM [NIS24]. FrodoKEM was a KEM submitted to the competition which reached the third round as an alternate candidate, and despite not being selected by the NIST it is still particularly relevant. Indeed, it is currently in a standardization process at ISO and its use is recommended by several European agencies such as ANSSI (France) and BSI (Germany). These recommendations are driven by the security of FrodoKEM which relies on conservatives choices and unlike ML-KEM does not use structured lattices. FrodoKEM and ML-KEM are actually very related schemes as they both descend from a line of work aiming at designing "noisy" versions of Diffie-Hellman/ElGamal using the hardness of the learning with errors problem (LWE). Actually, tinkering with the parameters of Kyber to set the ring to dimension 1 would basically lead to FrodoKEM by making Kyber rely on plain LWE instead of MLWE. While relying on the unstructured case for lattice-based cryptosystem is certainly comforting security-wise, this comes with the price of larger and slower schemes. It is thus interesting to have a look at the practical aspects of such schemes to understand their limitations regarding real-life utilization. During the NIST standardization process, some works have studied the efficiency of the scheme, both in terms of speed [BFM+18] and compactness [BBC+23] to enable its use on embedded devices. We believe that the natural way to move forward toward real-life deployment is to discuss the possibility of incorporating generic side-channel protections such as masking. We take a first step in this direction by

proposing a fully masked implementation of the decapsulation procedure of FrodoKEM. Our plain C implementation supports an arbitrary masking order but is naturally subject to leakage since it does not target a specific architecture and thus does not incorporate specific adjustments to thwart micro-architectural effects. Hence, we propose in addition a Cortex-M4 specific order 1 implementation built upon the C implementation that incorporates handmade assembly code for the basic gadgets. These gadgets have been specifically crafted to offer resistance against a side-channel adversary. We validated their resiliency by performing TVLA using the popular ChipWhisperer platform.

Related works. The core technique used to protect FrodoKEM from side-channel attacks is masking [ISW03, PR13, BBD+15] and in particular the conversions between Boolean and arithmetic masked representations that are always required to mask lattice-based schemes [Gou01, CGV14]. The first attempt to mask a LWE-based PKE that can be seen as an ancestor of FrodoKEM can be found in [RRVV15], but it was not considering the CCA-secure version using the Fujisaki-Okamoto transform which is required for the KEM. This was addressed in [OSPG18] which proposes a masked implementation of the CCA-secure version of the RLWE encryption scheme that would later lead to NewHope and Kyber. Later, several works [SPOG19, VDK+20, BGR+21, CGMZ23] concretely masked the NIST candidates Kyber and Saber, and iteratively improved masking techniques for the individual components of the scheme such as the binomial sampler and the polynomial comparison for the FO transform. Since FrodoKEM can be seen as a plain LWE variant of those schemes, the discussions in those works are for the most part relevant to derive a masked version of the scheme. An exception is the sampling of the error, since FrodoKEM does not use a binomial distribution but instead uses a cumulative distribution table. Fortunately, sampling from such a table in a masked setting has already been studied in [GR20].

Contribution. In this paper, we take a first step toward enabling generic side-channel protections for FrodoKEM by proposing a masked version of the scheme. We discuss what has to be protected, what are the potential pain points and how to approach them with several techniques already described and well studied in the literature. Next, we provide a C implementation of the masked scheme based on the reference code that offers the flexibility to be compiled at any order. The code is written in a modular way such that it should be easy for anyone to extract the gadgets of FrodoKEM and to use them in another implementation if desired. Then, we specialize the basic gadgets (see Table 2) to order 1 in ARM assembly so that we have secure code for the Cortex-M4. The goal here is to provide security against (micro-)architectural leakage which is not something covered by the theoretical security proofs of the existing gadgets. Finally, we provide evidences that the goal has been reached by performing *fixed vs random* t-tests on the individual gadgets.

2 Preliminaries

Notations. We use bold lowercase letters for vectors and bold uppercase letters for matrices. For a matrix $\mathbf{B}$, we note $\mathbf{b}_i$ the i-th row of $\mathbf{B}$. We denote by $[\![x_i]\!]_{1\leq i\leq n}$ the sharing (arithmetic or boolean) into n shares of a value x. When the number of shares is not ambiguous (i.e. it remains identical during the whole execution of a described algorithm), we used the simplified notation $[\![x]\!]$.

2.1 FrodoKEM

FrodoKEM is a key encapsulation mechanism constructed from the key exchange protocol proposed in [BCD+16]. As the title of the paper suggests, the main idea was to construct a key exchange directly based on LWE and to eliminate the ring structure to increase confidence in the security of the scheme while still maintaining acceptable practical performances. To obtain a PKE, the KEX is made noninteractive by considering the public-key of the receiver as the first message of the protocol. This is very similar to the transition from the Diffie-Hellman key exchange to the ElGamal public-key encryption scheme. Then, going from PKE to KEM is achieved through a (variant of) the Fujisaki-Okamoto (FO) transform. Since the FO transform forces re-encryption to check validity of the ciphertext, our implementation of the masked `Decaps` algorithm actually encompasses both encapsulation and decapsulation.

As a plain LWE scheme, the main arithmetic operations of FrodoKEM are matrix operations over $\mathbb{Z}_q$ and the sampling of the error distribution χ. In particular, q is a power of two and χ is a discrete Gaussian. While the runtime of an unprotected implementation would be solely dominated by the cost of sampling matrices and performing arithmetic operations, in a masked setting, some less significant operations such as comparisons can present an unforeseen cost.

Because of the absence of structure, the public matrix $\mathbf{A}$ would be quite heavy to store and share. For this reason, a seed is instead included in the keypair and expanded at the beginning of each algorithm. This operation is denoted by the generic function Gen in Algorithms 1, 2 and 3. This function can be instantiated with two different algorithms: either AES128 or SHAKE128. We refer to the official specification for a detailed description of the Gen function.

Table 1: Security parameters of FrodoKEM

	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
D	15	15	16
q	32768	65536	65536
n	640	976	1344
$\bar{n}$	8	8	8
B	2	3	4
$len_{\mathbf{A}}$	128	128	128
len_{sec}	128	192	256
SHAKE	SHAKE128	SHAKE256	SHAKE256

2.1.1 Parameters sets

FrodoKEM proposes multiple sets of parameters depending on the targeted security level. Those can be found in Table 1, D is the bitsize of the arithmetic values used in the algorithm, it determines the modulus $q = 2^D$. The values n and $\bar{n}$ determine the size of the matrices used in the algorithm, in particular, n is the dimension of the underlying LWE problem. The value $len_{\mathbf{A}}$ is fixed to 128 and is the size of the seed used to generate $\mathbf{A}$ and len_{sec} is the size of the seeds used for sensitive values, it corresponds to the targeted bit-security level.

2.1.2 Key generation

The key generation algorithm of FrodoKEM is presented in Algorithm 1. It first samples a random bitstring containing the dummy key in case of failure s, the seed for the secret and error matrices $seed_{SE}$ and the (pre-)seed for the "public matrix" $\mathbf{A}$. It will then use those seeds to explicitly compute $\mathbf{A}$ and two $\bar{n}$ by n matrices $\mathbf{S}^T$ and $\mathbf{E}$ containing entries

sampled from χ. The matrix $\mathbf{S}^T$ is the main secret component of the secret key sk and the public-key pk is the LWE instance $(\mathbf{A}, \mathbf{AS} + \mathbf{E})$.

Algorithm 1 FrodoKEM.KeyGen()

Input: /
Output: pk, sk

$s\|seed_{SE}\|z \leftarrow \$\{0,1\}^{len_{sec}+len_{SE}+len_A}$
$seed_A \leftarrow \mathsf{SHAKE}(z, len_A)$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$(r^0, r^1, \ldots, r^{2\bar{n}n+\bar{n}^2-1}) \leftarrow \mathsf{SHAKE}(\mathsf{0x5F}\|seed_{SE}, 32n\bar{n})$
$\mathbf{S}^\mathsf{T} \leftarrow \mathsf{SampleMatrix}((r^0, r^1, \ldots, r^{n\bar{n}-1}), \bar{n}, n)$
$\mathbf{E} \leftarrow \mathsf{SampleMatrix}((r^{n\bar{n}}, r^{n\bar{n}+1}, \ldots, r^{2n\bar{n}-1}), \bar{n}, n)$
$\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
$\mathbf{b} \leftarrow \mathsf{Pack}(\mathbf{B})$
$pkh \leftarrow \mathsf{SHAKE}(seed_A\|\mathbf{b}, len_{sec})$
return $pk = seed_A\|\mathbf{b},\ sk = s\|seed_A\|\mathbf{b}\|\mathbf{S}^\mathsf{T}\|pkh$

2.1.3 Encapsulation

The encapsulation procedure (described in Algorithm 2) uses the underlying LWE-based PKE to encrypt a randomly picked value u that is used to derive the shared secret ss. In a nutshell, two LWE instances $(\mathbf{A}, \mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}')$ and $(\mathbf{B}, \mathbf{V} = \mathbf{S}'\mathbf{B} + \mathbf{E})$ are computed and $\mathbf{V}$, which is indistinguishable from a random value under the LWE assumption, is used to mask u in a one-time pad fashion. The receiver in possession of the secret key will be able to recover u by computing $\mathbf{B}'\mathbf{S} \approx \mathbf{V}$ and subtract it from $\mathbf{C} = \mathbf{V} + u$ while the adversary will only see two uniformly random looking values $\mathbf{C}$ and $\mathbf{B}'$.

Algorithm 2 FrodoKEM.Encaps()

Input: $pk = seed_A\|b$
Output: $(c = c_1\|c_2\|salt,\ ss)$

$(salt,\ u) \leftarrow \$\{0,1\}^{len_{salt}} \times \{0,1\}^{len_{sec}}$
$pkh \leftarrow \mathsf{SHAKE}(pk, len_{sec})$
$seed_{SE}\|k \leftarrow \mathsf{SHAKE}(pkh\|u\|salt, len_{SE} + len_{sec})$
$(r^0, r^1, \ldots, r^{2\bar{n}n+\bar{n}^2-1}) \leftarrow \mathsf{SHAKE}(\mathsf{0x96}\|seed_{SE}, 16(2\bar{n}n + \bar{n}^2))$
$\mathbf{S}' \leftarrow \mathsf{SampleMatrix}((r^0, r^1, \ldots, r^{n\bar{n}-1}), \bar{n}, n)$
$\mathbf{E}' \leftarrow \mathsf{SampleMatrix}((r^{n\bar{n}}, r^{n\bar{n}+1}, \ldots, r^{2n\bar{n}-1}), \bar{n}, n)$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$c_1 \leftarrow \mathsf{Pack}(\mathbf{B}')$
$\mathbf{E}'' \leftarrow \mathsf{SampleMatrix}((r^{2\bar{n}n}, r^{2\bar{n}n+1}, \ldots, r^{2\bar{n}n+\bar{n}^2-1}), \bar{n}, \bar{n})$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C} \leftarrow \mathbf{V} + \mathsf{Encode}(u)$
$c_2 \leftarrow \mathsf{Pack}(\mathbf{C})$
$ss \leftarrow \mathsf{SHAKE}(c_1\|c_2\|salt\|k)$
return $(c = c_1\|c_2\|salt,\ ss)$

2.1.4 Decapsulation

While the decryption of the PKE simply basically boils down to the computation of $\mathbf{C} - \mathbf{B}'\mathbf{S}$, the FO transform used to ensure CCA2 security forces to re-encrypt the plaintext

to validate the ciphertext. This means that the bulk of the work of the decapsulation (Algorithm 3) consists in running again the encapsulation once the value u is recovered. Once the ciphertext is recomputed, it is compared with the initial ciphertext received and if the values match, the shared secret ss is derived and output. From the side-channel perspective, this algorithm is very sensitive since it is the heaviest and manipulates explicitly the secret key as well as the shared secret ss. Since it is basically a superset of the encapsulation procedure and since it contains all the operations of the key generation, we will focus on it for the rest of the paper.

Algorithm 3 FrodoKEM.Decaps ()

Input: $c = c_1 \| c_2 \| salt,\ sk = s \| seed_A \| b \| \mathbf{S}^T \| pkh$
Output: ss

$\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
$\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
$\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
$u' \leftarrow \mathsf{Decode}(\mathbf{M})$
$seed'_{SE} \| k' \leftarrow \mathsf{SHAKE}(pkh \| u' \| salt, len_{SE} + len_{sec})$
$\mathbf{S}' \leftarrow \mathsf{SampleMatrix}((r^0, r^1, \ldots, r^{n\bar{n}-1}), \bar{n}, n)$
$\mathbf{E}' \leftarrow \mathsf{SampleMatrix}((r^{n\bar{n}}, r^{n\bar{n}+1}, \ldots, r^{2n\bar{n}-1}), \bar{n}, n)$
$\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
$\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
$\mathbf{E}'' \leftarrow \mathsf{SampleMatrix}((r^{2\bar{n}n}, r^{2\bar{n}n+1}, \ldots, r^{2\bar{n}n+\bar{n}^2-1}), \bar{n}, \bar{n})$
$\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
$\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
$\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Encode}(u')$
$\bar{k} \leftarrow k'$ **if** $\mathbf{B}' \| \mathbf{C} = \mathbf{B}'' \| \mathbf{C}'$ **else** $\bar{k} \leftarrow s$ ▷ Constant-time check
$ss \leftarrow \mathsf{SHAKE}(c_1 \| c_2 \| salt \| \bar{k})$ **return** ss

2.2 Masking

A masked scheme aims at breaking the link between the side-channel leakage of intermediate values and the actual value of sensitive variables by splitting them into $n+1$ shares. The shares x_i of a secret value x are picked such that $x = x_0 \star x_1 \cdots \star x_n$. In the context of this work, $\star$ will be either the XOR operator ($\oplus$) — we call this Boolean masking — or the addition modulo q ($+$) — we call this arithmetic masking — and x will be an integer of fixed bitsize. In the usual t-probing model [ISW03], the security assumption is that the adversary is only able to learn at most n shares of a variable through the execution of the algorithm. The basic idea is that since sensitive values are split into $n+1$ shares, any subset of n shares should not reveal any information on the value itself. The parameter n is known as the *masking order*. In the framework of lattice-based cryptography, both Boolean and arithmetic masking are often required since some operations, such as matrix multiplications, are more efficient to perform over arithmetic shares while others, such as the hash function evaluation, require Boolean masking. This implies that conversions between the two representations are needed.

2.3 Micro-architectural leakage

Standard models for side-channel leakage define the adversary in a formal way which enables a methodical study of security regardless of any specific real-life framework. This is very convenient to design new masking schemes since the new gadgets can be proven secure on paper under the assumptions made on what the adversary can or cannot do. However, such models do not necessarily translate very well in terms of concrete deployment. While

circuit-based models can for example be fairly naturally translated to an actual electronic circuit, it is less clear to which extent it stays relevant when a piece of software is used to perform the corresponding computation on a CPU. In particular, the leakage can be specific to the ISA or even the underlying micro-architecture. The sources of leakage due to the architecture itself have been studied in the literature for a couple of years and even if some efforts have been made to model them [ZMM23], to characterize them [MPW22] or to automatically eliminate them [ZM24], it seems very hard to handle them efficiently besides by relying on ad-hoc handcrafted low level implementations of the gadgets.

3 Masking FrodoKEM

While the masking of FrodoKEM has, to the best of our knowledge, not been considered previously, masking techniques for the sibling schemes based on RLWE/RLWR have appeared in the literature [RRVV15, OSPG18, BGR+21, KDvB+22, CGMZ23]. The techniques described in the recent papers on Kyber and Saber are mostly sufficient to derive a fully masked version of FrodoKEM. A notable exception is the sampling of the error distribution. While Saber and Kyber use a centered binomial distribution that is usually sampled by computing the difference between the hamming weight of two fixed size integers through a Boolean to arithmetic conversion, the distribution in Frodo is sampled through a cumulative distribution table (CDT). Fortunately, this is similar to the case of qTESLA [ABB+20] and a generic gadget to sample from a CDT has been previously proposed in [GR20].

3.1 Sensitive operations

Let us now describe the several operations that have to be masked in the decapsulation procedure of FrodoKEM.

3.1.1 Matrix operations

As a lattice-based key exchange based on LWE, FrodoKEM contains operations of the form $\mathbf{AS} + \mathbf{E}$ with $\mathbf{A}, \mathbf{S}$ and $\mathbf{E}$ matrices over $\mathbb{Z}_q$. These operations are usually heavy and one of the bottlenecks in terms of computation in an unmasked setting since, unlike the RLWE case, there is no additional algebraic structure that enables faster techniques such as NTT-based multiplications. However, using arithmetic masking, these operations will only scale *linearly* with the masking order. Indeed, for a masked matrix $\mathbf{S} = \mathbf{S}_0 + \mathbf{S}_1 + \cdots + \mathbf{S_n}$ and a public unmasked matrix $\mathbf{A}$, the shares of the product $\mathbf{AS}$ can be simply computed as $\mathbf{AS}_0, \mathbf{AS}_1, \ldots, \mathbf{AS}_n$. Actually, in an unmasked setting, the computation is sometimes performed from $(seed_{\mathbf{A}}, \mathbf{S})$ and the expansion of $\mathbf{A}$ from the seed accounts for a large part of the computational cost of the product. In the masked version, the expansion only has to be performed once (regardless of whether the full matrix $\mathbf{A}$ is stored in memory) and thus, the overhead of the masked computation is reduced.

3.1.2 Hashing

Since the CCA2 security of the scheme is obtained via a Fujisaki-Okamoto transform, the decapsulation procedure will actually re-encrypt the plaintext to verify the validity of the ciphertext. This re-encryption uses SHAKE to derive $\mathbf{E}, \mathbf{S}'$ and the shared key, which means that the hash function has also to be masked. Masking hash functions and symmetric-key primitives is mainly performed using Boolean masking, but the topic is out of scope of this paper.

3.1.3 Encoding and decoding

The Encode function is mapping the randomly drawn bit-string u to a matrix of size $\bar{n} \times \bar{n}$ over $\mathbb{Z}_q$ in order to use it as a plaintext for the underlying LWE encryption scheme. In short, it takes as input a bit-string of size $B \cdot \bar{n}^2$ and map each of the $\bar{n}^2$ B-bit substrings to a value in $\mathbb{Z}_q$ by multiplying their integer value by $q/2^B$. Those $\bar{n}^2$ values are the entries of the output matrix. The DECODE function is the natural inverse operation. It takes as input a matrix of dimension $\bar{n} \times \bar{n}$ over $\mathbb{Z}_q$ and decodes coefficient by coefficient. We refer to the official documentation of FrodoKEM and the accompanying reference implementation for the full details. Since those functions manipulate the sensitive value u, they have to be masked. It is also worth noting that the input and output will use different masking representations. While u is a Boolean masked bit-string, $\mathbf{M}$ is the output of an arithmetic operation and Encode and Decode boil down to conversions between arithmetic and Boolean masking. This is not a problem since the main operation, that is to say multiplication or division by $q/2^B$, is simply a shift (q is a power of two) and thus is trivial to perform in Boolean masked form.

3.1.4 Sampling

Sampling the error distribution in lattice-based schemes is a recurring problem that is often blown-up by adding a countermeasure such as masking on top of the scheme. Small uniform or binomial distributions are usually sampled using Boolean to arithmetic conversions, this is the case in e.g Dilithium, Saber or Kyber. In FrodoKEM, a table is used to encode (half of[1]) the distribution. To sample, one picks a random 15-bit integer s, finds the index of the last value in the table that is bigger than s and returns this index, augmented with a randomly picked sign, as the sample. A masked version of this algorithm has been presented in [GR20]. Since the whole table has to be read to avoid timing attacks, this technique scales poorly with the size of the table. Thankfully, FrodoKEM uses quite short tables, from 7 entries for FrodoKEM-1344 to 13 entries for FrodoKEM-640.

3.1.5 Comparison

The last step of the FO transform is to compare the received ciphertext $\mathbf{B}'||\mathbf{C}$ with the recomputed ciphertext $\mathbf{B}''||\mathbf{C}'$. The topic has been discussed extensively in [CGMZ23] in the context of comparing RLWE ciphertexts. However, the techniques described do not use the fact that the mathematical objects manipulated are polynomials since they interpret them as vectors of elements in $\mathbb{Z}_q$, so the discussion is relevant to FrodoKEM. Unfortunately, most of the techniques use the fact that, in Kyber, q is a prime. This means that to mask FrodoKEM, we have to rely on (a variant of) what they call PolyZeroTestAB, which is basically converting coefficients of the matrix one by one from arithmetic to Boolean masked form and then performing a masked logical OR between the bits to check if they all equal zero or not.

3.2 Gadgets

3.2.1 Basic gadgets

While the masking techniques used in our implementation are known in the literature, they appeared across multiple papers. Table 2 is summarizing the basic gadgets that we used in the implementation. The order 1 versions of those gadgets are the ones that have been rewritten in assembly for the Cortex-M4 implementation described in Section 4.2 in order to limit (micro)-architectural leakage.

[1] Since the distribution is centered, a sign is randomly picked once the absolute value of the sample is known.

3.2.2 Masked compare

Algorithm 4 provides a high-level description of the maskedCompare gadget. Its goal is to verify that the masked ciphertext recomputed in the FO transform is actually the same as the ciphertext received by the encapsulation procedure. The idea is to transform an equality test into a zero test by computing the difference between the two values and testing whether the result is zero. Since both $\mathbf{B}''$ and $\mathbf{C}$' are results of matrix operations, they are already in arithmetic masked form, which means that $\mathbf{B}'$ and $\mathbf{C}$ can be trivially (coefficient-wise) subtracted from the first share to obtain the difference. The gadget will then iterate over all the coefficients of $\mathbf{B}''$ and $\mathbf{C}'$ - that are now masking of 0 if the inputs are equal -, perform an arithmetic to Boolean conversion, negate the value and aggregate the result using SecAND. If every coefficient was a masking of 0, the logical AND of their negation is a -1. A zero-test on the negation of the result will yield whether or not all coefficients were actually zeroes. After this ultimate zero test, some cheap bitwise operations are performed on the result in order to obtain the desired output format, that is to say a masking of 0 if the inputs are equal, or a masking of -1 if they are different.

Algorithm 4 maskedCompare

Input: An arithmetic masking of the re-encryption $[\![\mathbf{B}''||\mathbf{C}'_i]\!]_{1\leq i\leq n}$ and the unmasked ciphertext $\mathbf{B}'||\mathbf{C}$

Output: Boolean masking of the selector $[\![s_i]\!]_{1\leq i\leq n}$ such that $s = 0$ if the re-encryption is equal to the ciphertext and $s = -1$ otherwise.

1: $[\![acc_i]\!]_{1\leq i\leq n} \leftarrow (1, 0, \ldots, 0)$
2: $\mathbf{B}''_0 \leftarrow \mathbf{B}''_0 - \mathbf{B}'$
3: $\mathbf{C}'_0 \leftarrow \mathbf{C}'_0 - \mathbf{C}$
4: **for** $j = 0$ to size($\mathbf{B}'$) **do**
5: $\quad [\![xa_i]\!]_{1\leq i\leq n} \leftarrow [\![\mathbf{B}''[j]_i]\!]_{1\leq i\leq n}$
6: $\quad [\![xb_i]\!]_{1\leq i\leq n} \leftarrow$ ArithmeticToBoolean$([\![xa_i]\!]_{1\leq i\leq n})$
7: $\quad [\![acc_i]\!]_{1\leq i\leq n} \leftarrow$ SecAND$([\![acc_i]\!]_{1\leq i\leq n}, \neg[\![xb_i]\!]_{1\leq i\leq n})$
8: **end for**
9: **for** $j = 0$ to size($\mathbf{C}$) **do**
10: $\quad [\![xa_i]\!]_{1\leq i\leq n} \leftarrow [\![\mathbf{C}'[j]_i]\!]_{1\leq i\leq n}$
11: $\quad [\![xb_i]\!]_{1\leq i\leq n} \leftarrow$ ArithmeticToBoolean$([\![xa_i]\!]_{1\leq i\leq n})$
12: $\quad [\![acc_i]\!]_{1\leq i\leq n} \leftarrow$ SecAND$([\![acc_i]\!]_{1\leq i\leq n}, \neg[\![xb_i]\!]_{1\leq i\leq n})$
13: **end for**
14: $[\![t_i]\!]_{1\leq i\leq n} \leftarrow$ SecZeroTest$(\neg[\![acc_i]\!]_{1\leq i\leq n})$
15: $[\![s_i]\!]_{1\leq i\leq n} \leftarrow -([\![t_i]\!]_{1\leq i\leq n} \wedge 1)$
16: $s_0 \leftarrow \neg s_0$
17: **return** $[\![s_i]\!]_{1\leq i\leq n}$

3.2.3 Masked Sampler

The explicit description of the maskedSampler procedure is given in Algorithm 5. The core part of the algorithm (based on [GR20]) consists in picking a random value r and iterating through the different entries of the table until a value smaller than r is found. The index of the current entry once the exploration stops corresponds to the sample. In a side-channel protected environment, one cannot stop directly after finding the value in the table. Even though this would not be problematic from the masking point of view, an early stop would enable an obvious timing attack. Instead, the algorithm always explores the whole table during the loop (Line 4) and update the output with the current index if the condition is met. In the core of the loop, the random value r of precision k (15 bits in the case of FrodoKEM) is subtracted to the current entry t and the sign bit b

of the result is extracted in order to determine whether $r > t$. A masked constant-time conditional statement of the form

$$x = (x \wedge b) \oplus (j \wedge \neg b)$$

is then used to select the index j or to keep the previous value depending on the result of the comparison. After the loop, x will actually contain the last index for which the condition is met and a random sign is added to get a symmetric distribution. In the fully masked description of the algorithm, maskedSampler is taking as input random values and outputting a matrix. Since we did not want to overload the notations, it is understood in this case that the random values correspond to pre-sampling of r and s and that the matrix is populated with a call to Algorithm 5 entry-wise.

Algorithm 5 maskedSampler

Input: Table T of probabilities of size L
Output: Arithmetic masking $[\![x]\!]$ of a value following the distribution encoded by T

1: $[\![x]\!] \leftarrow [\![0]\!]$
2: $[\![r]\!] \leftarrow \text{Rand}(k)$
3: $[\![s]\!] \leftarrow \text{Rand}(1)$
4: **for** $j = 0$ to $L - 1$ **do**
5: $\quad [\![t]\!] \leftarrow (-T[j] - 1, 0, \ldots, 0)$
6: $\quad [\![\delta]\!] \leftarrow \text{SecADD}([\![r]\!], [\![t]\!])$
7: $\quad [\![b]\!] \leftarrow [\![\delta]\!] >> 15$
8: $\quad [\![t]\!] \leftarrow (j + 1, 0, \ldots, 0)$
9: $\quad [\![\delta]\!] \leftarrow \text{SecAND}(\neg[\![b]\!], t)$
10: $\quad [\![b]\!] \leftarrow \text{SecAND}([\![b]\!], [\![x]\!])$
11: $\quad [\![x]\!] \leftarrow [\![\delta]\!] \oplus [\![b]\!]$
12: $\quad [\![x]\!] \leftarrow \text{Refresh}([\![x]\!])$
13: **end for**
14: $[\![x]\!] \leftarrow [\![x]\!] \oplus (-[\![s]\!])$
15: $[\![x]\!] \leftarrow \text{SecADD}([\![x]\!], [\![s]\!])$
16: **return** BooleanToArithmetic($[\![x]\!]$)

3.3 Masked decapsulation of FrodoKEM

The fully masked version of FrodoKEM.`Decaps` () is shown in Algorithm 6. We assume that the secret-key is received already in shared form. Line 1 and 2 compute the full arithmetic form of the received ciphertext. Since c_1 and c_2 are considered public values, this can be recomputed by the adversary and shall not be masked. The computation of $\mathbf{M}$ at line 3 is the first that has to be protected. The reason is twofold: first, the computation involves the secret-key and second, the matrix $\mathbf{M}$ allows to recover the plaintext u' from which it is trivial to compute the shared secret ss. From there, almost all the subsequent operations will be masked since knowing their input or their result would lead to either the knowledge of the secret or the result of the comparison (see [CGMZ23] for a discussion about the need to mask the comparison). The two notable exceptions are the generation of the public matrix $\mathbf{A}$ at line 8 and the unpacking of b at line 11 since those values are both computable by anyone from the public-key. All the needed gadgets have been described in the previous sections and we thus have all the ingredients to implement a fully masked decapsulation of FrodoKEM.

Table 2: Basic gadgets used in our implementation

	Order 1	Arbitrary Order
SecAND	Alg.12 [ISW03]	Alg.12 [ISW03]
SecADD	Alg.15 [Gou01]	Alg.8 [CGV14]
ArithmeticToBoolean	Alg.14 [Gou01]	Alg.10 [CGV14]
BooleanToArithmetic	Alg.13 [Gou01]	Alg.11 [CGV14]
SecZeroTest	Alg.7 [CGMZ23]	Alg.7 [CGMZ23]

3.3.1 Masked encapsulation and key generation

While the encapsulation and key generation of FrodoKEM have not been integrated into our implementation, they should be both fairly easy to derive from the masked decapsulation. The encapsulation procedure is basically a subset of the decapsulation since the FO transform is requiring a re-encryption of the ciphertext to ensure that it is well-formed and the key generation only uses procedures that have been masked for the decapsulation, that is to say, matrix operations, maskedSHAKE and maskedSampler.

Algorithm 6 MaskedFrodoKEM.Decaps ()

Input: $c = c_1 \| c_2 \| salt,\ sk = [\![s]\!] \| seed_A \| b \| [\![\mathbf{S}^T]\!] \| pkh$
Output: $[\![ss]\!]$

1: $\mathbf{B}' \leftarrow \mathsf{Unpack}(c_1, \bar{n}, n)$
2: $\mathbf{C} \leftarrow \mathsf{Unpack}(c_2, \bar{n}, \bar{n})$
3: $[\![\mathbf{M}]\!] \leftarrow \mathbf{C} - \mathbf{B}'[\![\mathbf{S}]\!]$
4: $[\![u']\!] \leftarrow \text{maskedDecode}([\![\mathbf{M}]\!])$
5: $[\![seed'_{SE} \| k']\!] \leftarrow \text{maskedSHAKE}(pkh \| [\![u']\!] \| salt, len_{SE} + len_{sec})$
6: $[\![\mathbf{S}']\!] \leftarrow \text{maskedSampler}([\![r^0, r^1, \ldots, r^{n\bar{n}-1}]\!], \bar{n}, n)$
7: $[\![\mathbf{E}']\!] \leftarrow \text{maskedSampler}([\![r^{n\bar{n}}, r^{n\bar{n}+1}, \ldots, r^{2n\bar{n}-1}]\!], \bar{n}, n)$
8: $\mathbf{A} \leftarrow \mathsf{Gen}(seed_A)$
9: $[\![\mathbf{B}'']\!] \leftarrow [\![\mathbf{S}']\!]\mathbf{A} + [\![\mathbf{E}']\!]$
10: $[\![\mathbf{E}'']\!] \leftarrow \text{maskedSampler}([\![r^{2\bar{n}n}, r^{2\bar{n}n+1}, \ldots, r^{2\bar{n}n+\bar{n}^2-1}]\!], \bar{n}, \bar{n})$
11: $\mathbf{B} \leftarrow \mathsf{Unpack}(b, n, \bar{n})$
12: $[\![\mathbf{V}]\!] \leftarrow [\![\mathbf{S}']\!]\mathbf{B} + [\![\mathbf{E}'']\!]$
13: $[\![\mathbf{C}']\!] \leftarrow [\![\mathbf{V}]\!] + \text{maskedEncode}([\![u']\!])$
14: $[\![\bar{k}]\!] \leftarrow [\![k']\!]$ **if** $\text{maskedCompare}(\mathbf{B}' \| \mathbf{C}, [\![\mathbf{B}'' \| \mathbf{C}']\!])$ **else** $\bar{k} \leftarrow [\![s]\!]$
15: $[\![ss]\!] \leftarrow \text{maskedSHAKE}(c_1 \| c_2 \| salt \| [\![\bar{k}]\!])$
16: **return** $[\![ss]\!]$

4 Implementation

4.1 High order C implementation

We provide a C implementation of the decapsulation following our masking strategy described in Section 3. This implementation builds upon the FrodoKEM reference implementation [ABD+] [2] and the masking gadgets are provided and compiled as a separated library. The whole implementation is available on the following public repository:

https://github.com/fragerar/masked_Frodo

[2]Note that to keep the API intact, we receive the secret key unmasked and artificially mask it at the start of the decapsulation. this should of course be modified in a real-life implementation.

Organization of the code. The code is split between two directories and aims to be modular and easy to iterate on in order to obtain additional masked schemes. The first one `/masking` contains all the non-trivial gadgets, from the basic ones like SecAND to the large ones like the sampler. Its goal is to act as an independent library that can be used to write a masked implementation of FrodoKEM or, more generally, of any other scheme (by reusing the basic gadgets). It also contains the ARM assembly gadgets for the Cortex-M4. The second one `/FrodoKEM` contains the reference implementation of FrodoKEM augmented by files implementing a masked version of the scheme. The objective was to make it somewhat oblivious to the implementation details of the gadgets in `/masking`. The file `masked_interface.c` and its associated header links the masking library to the code of masked FrodoKEM. We believe that this organization can facilitate code reuse in the future.

Performances. The performances of the C implementation is summarized in Tables 3, 4 and 5. The code has been compiled with GCC version `13.3.0` and optimization level `O3` and run on an `Intel(R) Core(TM) i5-6500 CPU`. Unless specified otherwise, all the benchmarks are running code that uses the AES variant for the generation of the matrix $\mathbf{A}$. The results for the gadgets in Table 4 use the parameters of FrodoKEM-640. The sampler has been run on matrix of dimension $n \times \bar{n}$. Since it has to be called twice during decapsulation and clearly appears as the largest gadget, finding more efficient way to sample from the distribution would significantly reduce the cost of masked FrodoKEM.

Randomness generation. We did not change the way the algorithm is sampling values used for the variables of the scheme. However, we decided to use a Xorshift PRNG called `Xoroshiro` to quickly extract the randomness required for the masks. Such a PRNG has been used in the past to speed-up FrodoKEM [BFM+18] by accelerating the sampling of $\mathbf{A}$. Discussing the quality of the randomness required for a masking scheme in practice is out of scope of this work but the PRNG is easily replaceable by a CSPRNG if needed.

Table 3: Benchmarks on x64 of the basic gadgets (in cycles).

Order	1	2	3	4	5	6	7
SecAND	77	95	110	173	249	318	379
SecADD	55	1039	1505	2379	3248	4383	5642
BooleanToArithmetic	20	2247	3363	6090	9303	12253	15322
ArithmeticToBoolean	38	1170	1687	3504	5555	7275	9056
SecZeroTest	140	504	760	1142	1609	2148	2712

Table 4: Benchmarks on x64 of the large gadgets used in FrodoKEM (in kilocycles).

Order	1	2	3	4	5	6	7
Key encode	2	152	219	392	596	788	994
Key decode	4	76	113	234	368	483	602
Compare	408	6312	9370	19358	31125	40369	49756
Sampler	9647	94967	144200	223058	318535	422430	531283

Table 5: Benchmarks on x64 of all versions of FrodoKEM (in kilocycles).

Order	1	2	3	4	5	6	7
FrodoKEM-640	57395	293703	498200	809345	1138708	1523870	2010644
FrodoKEM-976	93498	446945	781157	1258737	1844603	2490802	3143712
FrodoKEM-1344	117860	522497	903360	1550615	2215093	2962697	3874540

4.1.1 Randomness usage.

Table 6 shows the randomness usage of the decapsulation function in terms of calls to our random generation function `rand_u16()`, which generates randomness 16 bits at a time. An interesting observation is that the randomness consumption does not increase much between FrodoKEM-976 and FrodoKEM-640 (and is even smaller up to order 3). This is due to the fact that most of the randomness is consumed by maskedSampler, and the length of the CDT decreases as the security level of FrodoKEM increases, resulting in less randomness consumption for each coefficient sampling. This somewhat mitigates the increased number of coefficients to generate for the ephemeral matrices $\mathbf{S}'$, $\mathbf{E}'$ and $\mathbf{E}''$.

Table 6: Number of calls to `rand_u16()` for the `Decaps` function.

Order	1	2	3	4	5	6	7
FrodoKEM-640	855,481	9,388,635	18,632,934	31,628,890	47,673,783	66,720,765	88,816,684
FrodoKEM-976	1,113,553	12,524,699	24,861,150	42,336,282	63,884,495	89,434,749	119,058,084
FrodoKEM-1344	1,013,993	12,392,155	24,611,670	42,342,362	64,121,191	89,850,621	119,628,188

4.1.2 Memory usage.

Table 7 shows the memory footprint of the decapsulation in bytes. The main factor driving the memory usage at high order is the storage of the masked matrices $\mathbf{S}, \mathbf{S}', \mathbf{E}', \mathbf{B}''$ since they are of dimension $n \times \bar{n}$, have 2 bytes entries, and scale linearly with the masking order. The total size of a single share of those masked matrices ranges from 10240 to 21504 bytes, depending on the version of FrodoKEM used. However, at low order, the unmasked matrices also add some significant overhead. The public matrix $\mathbf{A}$ stored in full would alone take $2 \times 640 \times 640 = 819200$ bytes. Fortunately, the implementation never fully stores it and uses instead a submatrix of dimensions $n \times 8$, which accounts to a memory footprint equivalent to one share of the $n \times \bar{n}$ matrices. However, beside this basic well-needed optimization, the reference code that is the basis of our implementation does not make any effort at limiting the memory footprint of the algorithm, hence why the values in Table 7 are large. In [BBC+23] can be found an in-depth discussion about trade-offs to greatly reduce the memory usage of FrodoKEM. Applying those techniques to our implementation would be a great next step since side-channel countermeasures are mainly relevant on small devices that mostly have extremely limited memory.

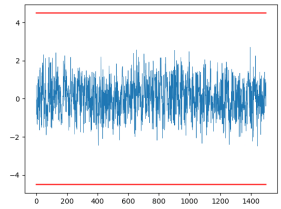
Table 7: Memory usage for the `Decaps` function (in bytes).

Order	1	2	3	4	5	6	7
FrodoKEM-640	196,368	268,232	340,112	412,008	483,880	555,760	627,632
FrodoKEM-976	300,016	410,320	520,560	630,856	741,064	851,344	961,616
FrodoKEM-1344	412,144	563,728	715,312	866,896	1,018,480	1,170,056	1,321,648

4.2 First order implementation for Cortex-M4

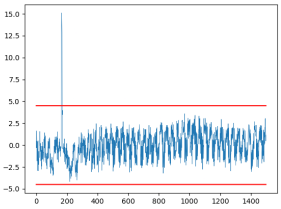
Masking gadgets implemented in plain C are known to exhibit micro-architectural leakage as this will be discussed later in Section 4.2.2. For this reason, we implemented five of our gadgets in assembly language (ARM) targeted for Cortex-M4. The assembly code can be compiled and run for any Cortex-M4 target but it has been specifically hardened and tested for STM32F303RCT7 target. The gadgets have been implemented only at order 1 and may be vulnerable to a higher order attacker. Due to the quite high resources consumption of unmasked FrodoKEM in terms of both memory and CPU, it is unlikely that a high order masked implementation would be suitable for a Cortex-M4 target and such implementation is thus out of scope of this work. Similarly to our C implementation

```
ldrh rx0, [px], #2
ldrh r, [pool]
ldrh rx1, [px], #2
```



```
ldrh rx0, [px], #2
ldrh r, [pool]
ldrh rx1, [px], #2

strh t1, [pool]
```



```
ldrh rx0, [px], #2
ldrh r, [pool]
ldrh rx1, [px], #2

eor t2, t2, t2
strh t1, [pool]
```

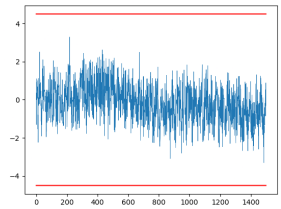


Figure 1: Leakage on load/store operations (2000 traces)

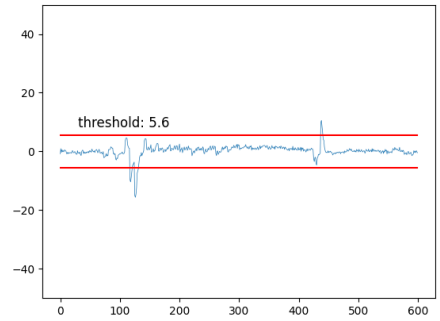
described in Section 4.1, all of our code is open source and can be found in the dedicated repository.

4.2.1 Hardening strategy

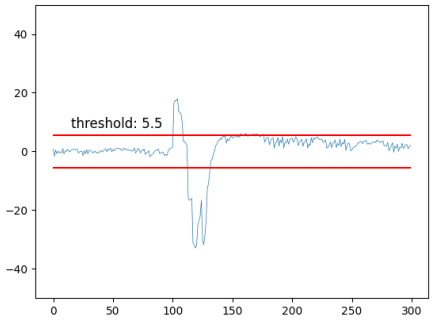
Hardened gadgets. We provide a hardened implementation of the five gadgets referenced in Table 2. We focused our implementation efforts on generic gadgets that can be reused for a high number of post-quantum schemes.

During the hardening process, we experimented many of the micro-architectural leakages demonstrated in [MPW22]. As our gadgets are not concerned by branching, we only discuss mitigation strategies for leakage caused by memory accesses and by consecutive operations in the pipeline.

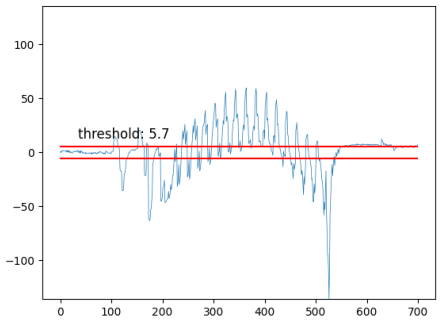
Memory accesses. In our experiments and similarly to [MPW22], loading or storing two different values induces leakage of their Hamming distance, even if those two memory accesses were separated by several `nop` or ALU instructions. In the case of gadgets operating on a single secret variable such as ArithmeticToBoolean or BooleanToArithmetic, this micro-architectural leakage was easy to mitigate by simply loading (resp. storing) a random value between the load (resp. store) of the two shares of the secret input (resp. output). This countermeasure is essentially free, as the loading of the random value is required for execution of the gadget. The additional store is however a dummy operation. For the gadgets operating on two secret variables such as SecAND or SecADD, it was surprisingly not enough to intertwine dummy memory accesses between the loading and storing of the shares. It was necessary to add dummy ALU and `nop` operations to prevent the leakage, and we could not find an explanation to this behavior. An example of



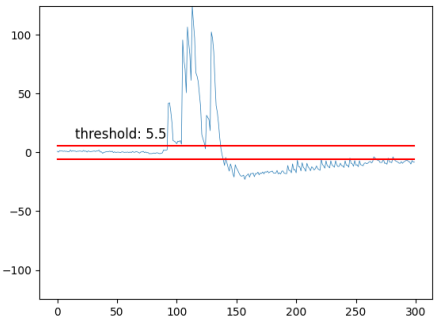
(a) ArithmeticToBoolean



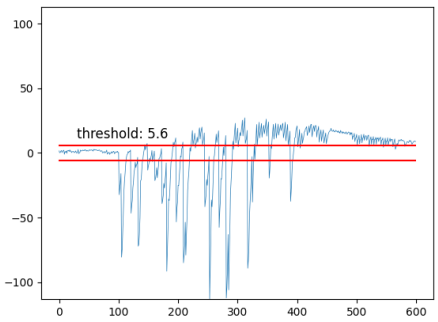
(b) BooleanToArithmetic



(c) SecAdd



(d) SecAnd



(e) ZeroTest

Figure 2: *t*-tests for gadgets with naive ASM implementation (5,000 traces)

unexpected leakage due to loads and stores is displayed in Figure 1. While the load of `r` interleaved between the loads of the shares `rx0` and `rx1` causes no leakage, adding an additional, seemingly unrelated, store worsens the situation. The leakage disappears again by performing a dummy ALU operation.

Consecutive ALU operations. It is well known that performing consecutive ALU operations on values that should not be recombined can lead to leakage. The experiments presented in [MPW22] as well as the discussion in [ZM24] illustrate the issue and how difficult it is to fix without incurring a large performance penalty. The micro-benchmarks

from [MPW22] certainly give good guidance but it appears that larger gadgets create more complex situations, with intricate interconnections between intermediate variables that depend on the shares or the randomness and are not strictly consecutive. Thus, a lot of the leakage removal boiled down to good practices, common sense and trial and error, interleaving a combination of dummy ALU, load/store, or nop operations when leakage is detected. Fortunately, as the basic gadgets were not too large at order 1, we were still able to greatly reduce leakage without adding a substantial overhead as would have automated tools such that the one presented in [ZM24].

4.2.2 Leakage assessment

A commonly used means to assess the resistance of masked implementations against side-channel attacks is to perform TVLA and compare the results with a fixed threshold. In this work, we perform a TVLA in the setting known as *fixed vs random*: we generate two sets of traces, one with fixed data being processed, while in the other set the data being processed are randomly chosen. We then perform a Welch t-test on each point of the trace, and we compare the result with a threshold computed from an error rate of $\alpha = 10^{-5}$ and taking into account the number of samples for each gadget as proposed in [DZD+18]. More precisely, the threshold TH is computed as follows:

$$\text{TH} = \text{CDF}^{-1}_{\mathcal{N}(0,1)}(1 - \sigma_{TH}/2),\ \sigma_{TH} = 1 - (1 - \sigma)^{(1/N_s)}$$

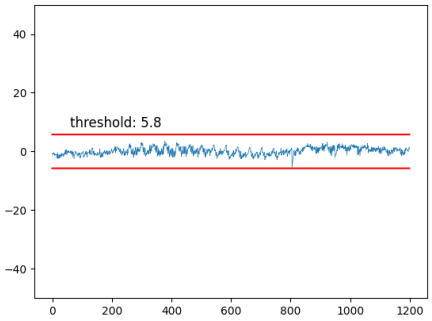
In Appendix B are shown the result of the TVLA performed on the gadgets output by the compiler from the naive C implementation. As expected, all are leaking. More interestingly, in Figure 2, the TVLA shows micro-architectural leakage happening with the "naive" ASM implementation with as few as 5,000 power traces. This means that bypassing potentially dangerous compiler optimizations by switching to handmade assembly code is not sufficient. Finally, the TVLA performed on the hardened gadgets that include the countermeasures described previously is displayed in Figure 3. We see no leakage even with 100,000 power traces. These versions of the gadget have been included in the Cortex-M4 version of the implementation. It is also possible for the reader to replicate our results on a ChipWhisperer-Lite by using the appropriate files available on the repository.

Table 8: Benchmarks on Cortex-M4 of the basic gadgets at order 1 (in cycles).

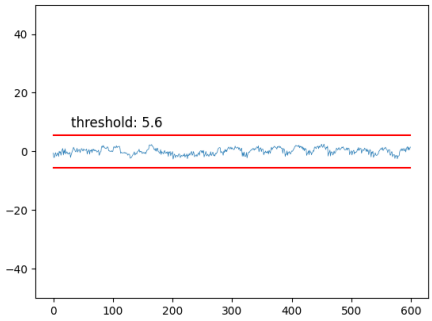
	C	ASM	ASM_h
SecAND	49	51	68 (+39%)
SecADD	206	149	248 (+20%)
BooleanToArithmetic	33	47	54 (+64%)
ArithmeticToBoolean	154	125	222 (+44%)
SecZeroTest	144	126	223 (+55%)

4.2.3 Performances

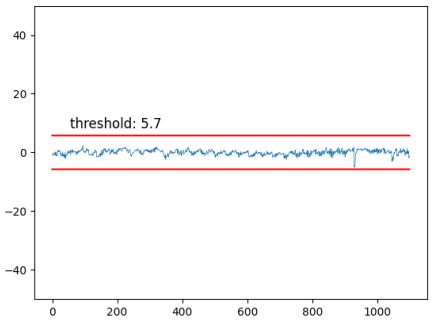
Since removing leakage often implies adding dummy operations to reduce the interaction between sensitive variables, it comes at a certain cost. Performances for our hardened gadgets are displayed in Table 8. The overhead of the leakage reduction compared to the C version is presented in the third column. For the record, the table also gives the number of cycles taken by the implementation of the gadgets before any leakage reduction effort. Since the final goal was to erase the leakage, the naive version of the gadgets have not been particularly optimized, even if some of them already show an improvement over the code output by the compiler. Although the overhead can be large, it is still very reasonable compared to what would be achieved by an automatic tool such as the one presented in [ZM24] (see their Table 1). Furthermore, if the randomness needed by the gadgets is



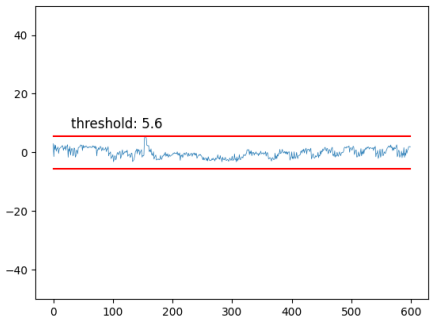
(a) ArithmeticToBoolean



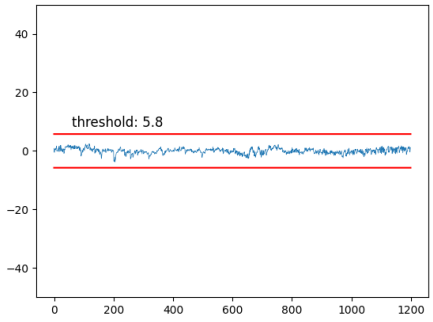
(b) BooleanToArithmetic



(c) SecAdd



(d) SecAnd



(e) ZeroTest

Figure 3: *t*-tests for hardened gadgets (100,000 traces)

sampled by an on-chip TRNG, it is possible to amortize the cost of the dummy operations by waiting for the random values to be available.

References

[AAB+22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[ABB+20] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Juliane Krämer, Patrick Longa, and Jefferson E. Ricardini. The Lattice-Based Digital Signature Scheme qTESLA. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20International Conference on Applied Cryptography and Network Security, Part I*, volume 12146 of *LNCS*, pages 441–460. Springer, Cham, October 2020. doi:10.1007/978-3-030-57808-4_22.

[ABD+] Erdem Alkim, Joppe W. Bos, Léo Ducas, Karen Easterbrook, Patrick Longa, Brian LaMacchia, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Ste-bila. FrodoKEM. https://frodokem.org/.

[BBC+23] Joppe W. Bos, Olivier Bronchain, Frank Custers, Joost Renes, Denise Verbakel, and Christine van Vredendaal. Enabling FrodoKEM on Embedded Devices. *IACR TCHES*, 2023(3):74–96, 2023. doi:10.46586/tches.v2023.i3.74-96.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified Proofs of Higher-Order Masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Berlin, Heidelberg, April 2015. doi:10.1007/978-3-662-46800-5_18.

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1006–1018. ACM Press, October 2016. doi:10.1145/2976749.2978425.

[BFM+18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! Faster Frodo for the ARM Cortex-M4. Cryptology ePrint Archive, Report 2018/1116, 2018. URL: https://eprint.iacr.org/2018/1116.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and Higher-Order Implementations. *IACR TCHES*, 2021(4):173–214, 2021. URL: https://tches.iacr.org/index.php/TCHES/article/view/9064, doi:10.46586/tches.v2021.i4.173-214.

[CGMZ23] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order Polynomial Comparison and Masking Lattice-based Encryption. *IACR TCHES*, 2023(1):153–192, 2023. doi:10.46586/tches.v2023.i1.153-192.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure Conversion between Boolean and Arithmetic Masking of Any Order.

In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, Berlin, Heidelberg, September 2014. doi:10.1007/978-3-662-44709-3_11.

[DZD+18] A. Adam Ding, Liwei Zhang, Francois Durvaux, Francois-Xavier Standaert, and Yunsi Fei. Towards Sound and Optimal Leakage Detection Procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications*, pages 105–122, Cham, 2018. Springer International Publishing.

[Gou01] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES 2001*, volume 2162 of *LNCS*, pages 3–15. Springer, Berlin, Heidelberg, May 2001. doi:10.1007/3-540-44709-1_2.

[GR20] François Gérard and Mélissa Rossi. An Efficient and Provable Masked Implementation of qTESLA. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications*, pages 74–91. Springer International Publishing, 2020.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Berlin, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_27.

[KDvB+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-Order Masked Saber. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 93–116. Springer, Cham, September 2022. doi:10.1007/978-3-031-14791-3_5.

[MPW22] Ben Marshall, Dan Page, and James Webb. MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices. *IACR TCHES*, 2022(1):175–220, 2022. doi:10.46586/tches.v2022.i1.175-220.

[NIS24] NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication, NIST FIPS 203, 2024. https://doi.org/10.6028/NIST.FIPS.203.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure Masked Ring-LWE Implementations. *IACR TCHES*, 2018(1):142–174, 2018. URL: https://tches.iacr.org/index.php/TCHES/article/view/836, doi:10.13154/tches.v2018.i1.142-174.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against Side-Channel Attacks: A Formal Security Proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Berlin, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_9.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A Masked Ring-LWE Implementation. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 683–702. Springer, Berlin, Heidelberg, September 2015. doi:10.1007/978-3-662-48324-4_34.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 534–564. Springer, Cham, April 2019. doi:10.1007/978-3-030-17259-6_18.

[VDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A Side-Channel Resistant Implementation of SABER. Cryptology ePrint Archive, Report 2020/733, 2020. URL: https://eprint.iacr.org/2020/733.

[ZM24] Jannik Zeitschner and Amir Moradi. PoMMES: Prevention of Micro-architectural Leakages in Masked Embedded Software. *IACR TCHES*, 2024(3):342–376, 2024. doi:10.46586/tches.v2024.i3.342-376.

[ZMM23] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. PROLEAD_SW Probing-Based Software Leakage Detection for ARM Binaries. *IACR TCHES*, 2023(3):391–421, 2023. doi:10.46586/tches.v2023.i3.391-421.

A Basics gadgets

Algorithm 7 SecZeroTest

Input: $[\![x_i]\!]_{1\leq i\leq n} \in \{0,1\}^k$

Output: $b \in \{0,1\}$ with $b = 1$ if $\bigoplus_{i=1}^{n} x_i = 0$ and $b = 0$ otherwise

1: $m \leftarrow \lceil \log_2 k \rceil$
2: $y_1 \leftarrow \overline{x_1}$ or $(2^{2^m} - 2^k)$
3: **for** $i = 2$ to n **do** $y_i \leftarrow x_i$
4: **for** $j = 0$ to $m - 1$ **do**
5: $\quad [\![z_i]\!]_{1\leq i\leq n} \leftarrow \text{Refresh}(y_1 \gg 2^i, \ldots, y_n \gg 2^i)$
6: $\quad [\![y_i]\!]_{1\leq i\leq n} \leftarrow \text{SecAND}(m, [\![y_i]\!]_{1\leq i\leq n}, [\![z_i]\!]_{1\leq i\leq n})$
7: **end for**
8: **return** RecombineShares$(y_1 \,\&\, 1, \ldots, y_n \,\&\, 1)$

Algorithm 8 SecADD

Input: $[\![x_i]\!]_{1\le i\le n}$ and $[\![y_i]\!]_{1\le i\le n}$

Output: $[\![z_i]\!]_{1\le i\le n}$ such that $\bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} z_i + \bigoplus_{i=1}^{n} y_i$

$[\![w_i]\!]_{1\le i\le n} \leftarrow \text{SecAND}([\![x_i]\!]_{1\le i\le n}, [\![y_i]\!]_{1\le i\le n})$
$[\![u_i]\!]_{1\le i\le n} \leftarrow 0$
$[\![a_i]\!]_{1\le i\le n} \leftarrow [\![x_i]\!]_{1\le i\le n} \oplus [\![y_i]\!]_{1\le i\le n}$
for $j = 1$ to $k-1$ **do**
    $[\![ua_i]\!]_{1\le i\le n} \leftarrow \text{SecAND}([\![u_i]\!]_{1\le i\le n}, [\![a_i]\!]_{1\le i\le n})$
    $[\![u_i]\!]_{1\le i\le n} \leftarrow [\![ua_i]\!]_{1\le i\le n} \oplus [\![w_i]\!]_{1\le i\le n}$
    $[\![u_i]\!]_{1\le i\le n} \leftarrow 2[\![u_i]\!]_{1\le i\le n}$
end for
$[\![z_i]\!]_{1\le i\le n} \leftarrow [\![x_i]\!]_{1\le i\le n} \oplus [\![y_i]\!]_{1\le i\le n} \oplus [\![u_i]\!]_{1\le i\le n}$
return $[\![z_i]\!]_{1\le i\le n}$

Algorithm 9 Expand

Input: $[\![x_i]\!]_{1\le i\le n}$

Output: $[\![y_i]\!]_{1\le i\le 2n}$ such that $\bigoplus_{i=1}^{2n} y_i = \sum_{i=1}^{n} x_i$

$[\![r_i]\!]_{1\le i\le n} \leftarrow \mathsf{Rand}(k)$
$[\![y_{2i}]\!]_{1\le i\le n} \leftarrow [\![x_i \oplus r_i]\!]_{1\le i\le n}$
$[\![y_{2i+1}]\!]_{1\le i\le n} \leftarrow [\![r_i]\!]_{1\le i\le n}$
return $[\![y_i]\!]_{1\le i\le 2n}$

Algorithm 10 ArithmeticToBoolean

Input: $[\![A_i]\!]_{1\le i\le n}$

Output: $[\![z_i]\!]_{1\le i\le n}$ such that $\bigoplus_{i=1}^{n} z_i = \sum_{i=1}^{n} A_i$

if $n = 1$ **then**
    return A_1
end if
$[\![x_i]\!]_{1\le i\le n/2} \leftarrow \text{ArithmeticToBoolean}([\![A_i]\!]_{1\le i\le n/2})$
$[\![x'_i]\!]_{1\le i\le n} \leftarrow \text{Expand}([\![x_i]\!]_{1\le i\le n/2})$
$[\![y_i]\!]_{1\le i\le n/2} \leftarrow \text{ArithmeticToBoolean}([\![A_i]\!]_{n/2+1\le i\le n})$
$[\![y'_i]\!]_{1\le i\le n} \leftarrow \text{Expand}([\![y_i]\!]_{1\le i\le n/2})$
$[\![z_i]\!]_{1\le i\le n} \leftarrow \text{SecADD}([\![x'_i]\!]_{1\le i\le n}, [\![y'_i]\!]_{1\le i\le n})$
return $[\![z_i]\!]_{1\le i\le n}$

Algorithm 11 BooleanToArithmetic

Input: $[\![x_i]\!]_{1\leq i\leq n}$

Output: $[\![A_i]\!]_{1\leq i\leq n}$ such that $\sum_{i=1}^{n} A_i = \bigoplus_{i=1}^{n} x_i$

1: $[\![A_i]\!]_{1\leq i\leq n-1} \leftarrow \mathsf{Rand}(k)$
2: $[\![A'_i]\!]_{1\leq i\leq n-1} \leftarrow [\![-A_i]\!]_{1\leq i\leq n-1}, A'_n \leftarrow 0$
3: $[\![y_i]\!]_{1\leq i\leq n} \leftarrow \textsc{ArithmeticToBoolean}([\![A'_i]\!]_{1\leq i\leq n})$
4: $[\![z_i]\!]_{1\leq i\leq n} \leftarrow \textsc{SecADD}([\![x_i]\!]_{1\leq i\leq n}, [\![y_i]\!]_{1\leq i\leq n})$
5: $[\![z_i]\!]_{1\leq i\leq n} \leftarrow \textsc{Refresh}([\![z_i]\!]_{1\leq i\leq n})$
6: $A_n \leftarrow \textsc{Xor}([\![z_i]\!]_{1\leq i\leq n})$
7: **return** $[\![A_i]\!]_{1\leq i\leq n}$

Algorithm 12 SecAND

Require: $[\![x_i]\!]_{1\leq i\leq n} \in \{0,1\}^k$, $[\![y_i]\!]_{1\leq i\leq n} \in \{0,1\}^k$

Ensure: $[\![z_i]\!]_{1\leq i\leq n} \in \{0,1\}^k$, with $\bigoplus_{i=1}^{n} z_i = (\bigoplus_{i=1}^{n} x_i) \wedge (\bigoplus_{i=1}^{n} y_i)$

1: for $i = 1$ **to** n **do** $z_i \leftarrow x_i \wedge y_i$
2: **for** $i = 1$ to n **do**
3: **for** $j = i + 1$ to n **do**
4: $r \leftarrow \{0,1\}^k$
5: $r' \leftarrow (r \oplus (x_i \wedge y_j)) \oplus (x_j \wedge y_i)$
6: $z_i \leftarrow z_i \oplus r$
7: $z_j \leftarrow z_j \oplus r'$
8: **end for**
9: **end for**
10: **return** $[\![z_i]\!]_{1\leq i\leq n}$

Algorithm 13 BooleanToArithmetic (Order 1)

Input: x_1, x_2

Output: y_1, y_2 such that $y_1 + y_2 = x_1 \oplus x_2$

1: $G \leftarrow \textsc{Rand}(k)$
2: $T \leftarrow x_1 \oplus G$
3: $T \leftarrow T - G$
4: $T \leftarrow T \oplus x_1$
5: $G \leftarrow G \oplus x_2$
6: $y_1 \leftarrow x_1 \oplus G$
7: $y_1 \leftarrow y_1 - G$
8: $y_1 \leftarrow y_1 \oplus T$
9: $y_2 \leftarrow x_2$
10: **return** y_1, y_2

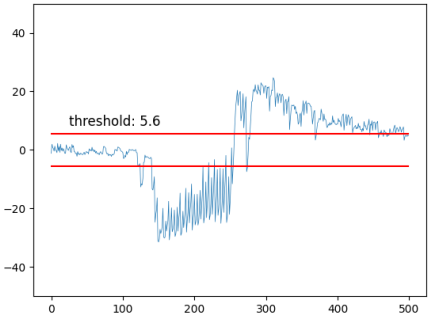
Algorithm 14 ArithmeticToBoolean (Order 1)

Input: x_1, x_2
Output: y_1, y_2 such that $y_1 \oplus y_2 = x_1 + x_2$
1: $G \leftarrow \text{Rand}(k)$
2: $T \leftarrow 2G$
3: $y_1 \leftarrow G \oplus x_2$
4: $O \leftarrow G \wedge y_1$
5: $y_1 \leftarrow T \oplus x_1$
6: $G \leftarrow G \oplus y_1$
7: $G \leftarrow G \wedge x_2$
8: $O \leftarrow O \oplus G$
9: $G \leftarrow T \wedge x_1$
10: $O \leftarrow O \oplus G$
11: **for** $k = 1$ to 15 **do**
12: $\quad G \leftarrow T \wedge x_2$
13: $\quad G \leftarrow G \oplus O$
14: $\quad T \leftarrow T \wedge x_1$
15: $\quad G \leftarrow G \oplus T$
16: $\quad T \leftarrow 2G$
17: **end for**
18: $y_1 = y_1 \oplus T$
19: $y_2 = x_2$
20: **return** y_1, y_2

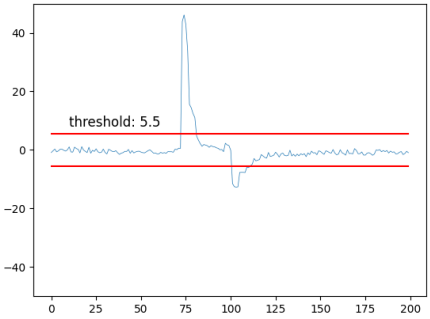
Algorithm 15 SecADD (Order 1)

Input: x_1, x_2, y_1, y_2
Output: z_1, z_2 such that $z_1 \oplus z_2 = x_1 \oplus x_2 + y_1 \oplus y_2$
1: $xa_1, xa_2 \leftarrow \text{BooleanToArithmetic}(x_1, x_2)$
2: $ya_1, ya_2 \leftarrow \text{BooleanToArithmetic}(y_1, y_2)$
3: $za_1 = xa_1 + ya_1$
4: $za_2 = xa_2 + ya_2$
5: $z_1, z_2 \leftarrow \text{ArithmeticToBoolean}$
6: **return** z_1, z_2

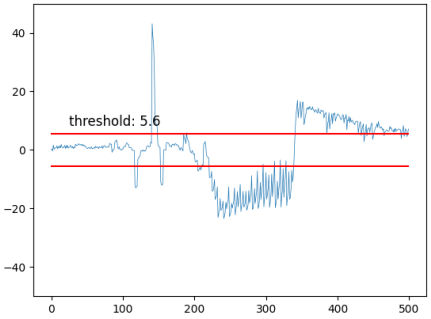
B Leakage Assessment of C code



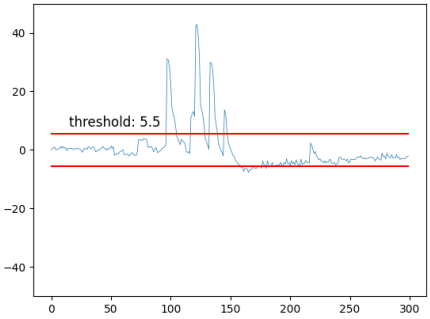
(a) ArithmeticToBoolean



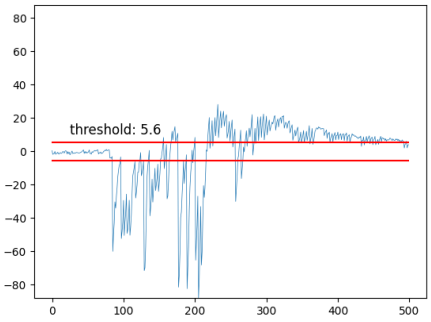
(b) BooleanToArithmetic



(c) SecAdd



(d) SecAnd



(e) ZeroTest

Figure 4: t-tests for gadgets implemented in C (5,000 traces)