



# Inspector Gadget

## A Toolbox for Fair Comparison of Masking Gadgets Application to Crystals-Kyber Compression

Camille Mutschler<sup>1,2</sup>, Laurent Imbert<sup>1</sup>  and Thomas Roche<sup>2</sup>

<sup>1</sup> LIRMM, CNRS, Univ. Montpellier, Montpellier, France

<sup>2</sup> NinjaLab, Montpellier, France

**Abstract.** We introduce `InspectorGadget`, an Open-Source Python-based software for assessing and comparing the complexity of masking gadgets. By providing a limited set of characteristics of a hardware platform, our tool allows to estimate the cost of a masking gadget in terms of cycle count equivalent and memory footprint. `InspectorGadget` is highly flexible. It enables the user to define her own estimation functions, as well as to expand the set of gadgets and predefined microcontrollers. As a case-study, we produce a fair comparison of several masked versions of Kyber compression function from the literature, together with novel alternatives automatically generated by our tool. Our results confirm that an interesting middle ground exists between theoretical performance measures (asymptotic complexity or operations count) and real implementations benchmarks (clock cycle accurate evaluations). `InspectorGadget` offers both simplicity and genericity while capturing the main performance-related parameters of a hardware platform.

**Keywords:** Masking gadgets · time and memory complexity · Kyber compression

## 1 Introduction

After three rounds and seven years of evaluation and analysis, the post-quantum standardization process initiated by the National Institute of Standards and Technology (NIST) led to the selection of four algorithms for general encryption and digital signature. These algorithms are intended to remain secure even against adversaries who possess a large scale quantum computer.

One of the major challenges in the upcoming deployment of these new standards is the protection of their implementations against side-channel attacks [PPM17, GJN20, ACLZ20, RRCB20, XPR<sup>+</sup>22, UXT<sup>+</sup>22]. A natural, yet acknowledged method to provide side-channel protection is masking. Whenever possible, *e.g.* for RSA or ECC, masking is provided thanks to algebraic properties with a reasonable overhead. However, in general, it is achieved through secret sharing schemes at much higher cost. The foundation of masking [CJRR99] is to split sensitive values into a predefined number of shares so that an attacker who has only access to a proper subset of these shares does not learn anything on the secret.

In their seminal work [ISW03], Ishai, Sahai and Wagner (ISW for short) introduced a generic compiler to convert any Boolean circuit into an equivalent one over shared variables. Roughly speaking, they built the first two masked gadgets: the Boolean *Xor* and *And* gates that takes shared variables and produce shared results. They showed that these gadgets could be composed securely and then create any Boolean circuit. The notion of gadget

---

E-mail: [camille.mutschler@lirmm.fr](mailto:camille.mutschler@lirmm.fr) (Camille Mutschler), [laurent.imbert@lirmm.fr](mailto:laurent.imbert@lirmm.fr) (Laurent Imbert), [thomas@ninjalab.io](mailto:thomas@ninjalab.io) (Thomas Roche)



and their composability was later formally introduced by Barthe *et al.* in [BBD<sup>+</sup>16]. While ISW generic compiler can be applied to any algorithm, it turns out to be impracticable for large circuits due to the resulting circuit size and randomness requirements. A dynamic avenue for research has been to build and prove more efficient masked gadgets, first for symmetric cryptography [RP10, CPRR14, Cor14, CGV14, CS20], more recently for post-quantum algorithms [RRVV15, RdCR<sup>+</sup>16, OSPG18, BBE<sup>+</sup>18, MGTf19, BDK<sup>+</sup>21, BGR<sup>+</sup>21, CGMZ22, BC22, CGMZ23, CGTZ23a, CGTZ23b].

Securing post-quantum Lattice-based algorithms are particularly challenging as they combine arithmetic operations over finite rings and non-algebraic functions which operate on bits or vectors of bits. This peculiarity imposes to consider masking schemes of very different nature together with mechanisms for converting between these different masking types. These new algorithm also offer new paradigms like the computation of threshold functions (*e.g.* the compression function of Kyber) or the equality test between polynomials.

The research community answered to these challenges by producing, these recent years, a continuously growing ecosystem of masked gadgets that can be composed to build secure post-quantum algorithms. This fast and fruitful research has several drawbacks:

1. It is hard to keep track with all the gadget proposals (let alone their specific advantages and limitations)
2. For two equivalent gadgets (*i.e.* implementing the same functionality), we have no common ground to compare them in terms of performances.

As a matter of fact, a new masked gadget complexity is usually given in terms of big-O notation (*w.r.t.* the number of shares and the cryptographic primitive parameters). While asymptotic complexity provides some insight about the gadget and allows to class them in different categories (cubic, quadratic, linear, etc.) it gives little if no information about the impact of the gadget in a real implementation (where the number of shares is hardly larger than 3 and the parameters of the cryptographic primitive are constants for the developers).

Alternatively, the cost of a masked gadget is sometimes given by the number of clock cycles taken by a specific implementation on a specific architecture, which is therefore difficult to generalize or compare.

One of the main challenges in assessing the performances of cryptographic primitive implementations is that they highly depend on the target microcontroller and its constraints (memory limitations, CPU architecture, T/P-RNG throughput, etc.). For instance some gadgets are interesting when randomness is cheap while being not competitive in other contexts. Ideally, we would need to have each gadget implemented on several target microcontrollers so that they can be benchmarked in various contexts. This would require a huge work and would not even be completely satisfactory: for fair comparison it would necessitate that all implementations, for each target microcontroller, have similar optimization level. Implementation of low level embedded software is an art where a skilled developer takes into account the specificity of the hardware (instruction set, number of registers, etc.).

In this paper, we introduce **InspectorGadget**, a highly configurable and flexible toolbox for estimating and comparing masking gadgets complexity. It includes cost functions for both speed and memory for a large number of existing masking gadgets. As a proof of concept, the initial version focuses on the variants that have been suggested over the years for masking Kyber compress function (see Section 2).

**InspectorGadget** is built around a simple abstract model of a microcontroller, fully parameterizable, that captures the most important features of a hardware family. This middle-grain precision level allows to smooth the very low-level optimizations out and to

retain the main characteristics. The goal is to compare gadgets with the same functionality, not to produce cycle count (or memory footprint) accurate figures.

By keeping the microcontroller modelling simple, our Python-based software allows the end-user to easily provide the main characteristics of his favorite architectures and add his own gadgets.

`InspectorGadget`'s second main innovation is the gadget library architecture, organized by grouping masking gadgets with equivalent functionality. This allows the estimation computation core to automatically create new gadgets by the composition of available sub-gadgets. This feature allows to investigate compositions that were not proposed in the literature but could be of practical interest. It must be emphasized that the newly created gadget compositions do not enjoy security guaranties (*e.g.* the composition of gadgets can break the  $t$ -probing security). They must be independently checked for security, for instance with an automated tool (see the related-work section below).

The tool was designed as a collaborative framework where developers and researchers can add new gadgets and new hardware abstract models in order to compare with the already implemented ones. The goal is to build a comprehensive library of masked gadgets that can be compared on a multitude of hardware models. `InspectorGadget` is distributed under the GPLv3 license and is available at <https://gite.lirmm.fr/crypto/inspector-gadget>.

After a short presentation of Kyber compress function (Section 2), the paper defines the main principles behind `InspectorGadget` and the high-level implementation choices in Section 3 (details about gadget integration in the library and how to write the cost functions – together with several well chosen examples – are given in Appendix A). The section 4 demonstrates the strength of our approach by applying `InspectorGadget` to the Kyber compress function. Indeed it allows to investigate simultaneously and on common ground all masked implementations (known to us at the time of writing this article), produce comparative figures for several abstract models of microcontrollers and introduce new compositions of sub-gadgets with competitive performances.

**Related Works:** Over the past few years, the number of academic tools related to masked gadgets/implementations has grown to a great extent [BBYS22]. The main objective of all of them is to assess a *security* level. These tools are thus complementary to `InspectorGadget` which does not consider security aspects, but rather study the efficiency of the masking gadgets. Although the objective are different, the masked gadgets security assessment tools share with `InspectorGadget` the *abstraction level dilemma*:

- The modeling of the hardware leakage can be very high-level with simple leakage rules. These tools usually do not work on an actual implementation but allow security proofs at high masking order, *e.g.* maskVerif [BBC<sup>+</sup>19], QMVerif [GZSW19], LeakageVerif [MPH23].
- An intermediate abstraction level exists where the user provides the actual implementation (or hardware design) and the tools provide security proofs based on an abstract (but already quite complex) leakage model, *e.g.* scVerif [BGG<sup>+</sup>20], SILVER [KSM20], PROLEAD [MM22] or PROLEAD\_SW [ZMM23].
- Finally, two sets of tools try to work on the finest leakage model. Some require the hardware design, like MAPS [CGD18], COCO [GHP<sup>+</sup>21] or RootCannal [KS22]. Others are based on a precise side-channel characterization of the devise (*e.g.* when the hardware design is not available), *e.g.* ASCOLD [PV17] and ELMO [MOW17]. The latter, more generic, is followed by GILES [EO19] that includes fault injection to

the side-channel simulations and ROSITA [SSB<sup>+</sup>21] dedicated to micro-architectural effects.

Another direction for automated security proofs is taken by TightPROVE [BGR18] (improving on maskComp [BBD<sup>+</sup>16]) where the *atomic* gadgets security is supposed to be known (inside the paradigm of (Strong-)Non-Interference probing security). The tool verifies that the composition of the gadgets ensures probing security (and then, for any masking order). `InspectorGadget` could quite simply be used in combination with TightPROVE so that the new compositions created by `InspectorGadget` are automatically checked by TightPROVE.

## 2 Background on Kyber

Kyber [SAB<sup>+</sup>20] is an IND-CCA2-secure key-encapsulation mechanisms (KEM) whose security depends on the hardness of solving the learning-with-errors problem over module lattices (MLWE [LS15]). Kyber KEM is based on the Fujisaki-Okamoto transform [FO99] built on top of an IND-CPA-secure public-key encryption scheme.

Let  $\mathbb{Z}_p$  denote the ring of integers modulo  $p$ ,  $\mathbb{Z}_p[X]$  the ring of univariate polynomials over  $\mathbb{Z}_p$ , and  $R_p$  the quotient ring  $\mathbb{Z}_p[X]/(X^k + 1)$ . In Kyber,  $k = 256$  and  $p = 2^8 \cdot 13 + 1 = 3329$  is prime.

Kyber requires arithmetic over  $R_p$  and small vector with elements in  $R_p$  as well as symmetric primitives (hash and pseudorandom functions), that manipulate arrays of bits. Therefore, the masking of Kyber requires two different masking types. In both cases, a masked quantity  $x$  (*e.g.* all the coefficients of a polynomial in  $R_p$ ) is represented with  $n$  shares as a tuple  $(x_1, \dots, x_n)$  such that  $x = x_1 + \dots + x_n \pmod p$  for the arithmetic masking, whereas  $x = x_1 \oplus \dots \oplus x_n$  in the Boolean masking (where  $\oplus$  denotes the Boolean XOR operation).

In this paper, we consider the  $\text{Compress}_p^d$  function of Kyber as a proof of concept of our software toolbox. In particular, the compress-to-1-bit function defined, for  $x \in \mathbb{Z}_p$ , as:

$$\text{Compress}_p^1(x) = \begin{cases} 1 & \text{if } p/4 \leq x < 3p/4 \\ 0 & \text{otherwise} \end{cases}$$

The  $\text{Compress}_p^1$  function is just a simple threshold function, however it is highly non-linear, which makes it a challenge to evaluate efficiently in a masked setting.

## 3 A powerful Gadget Library

In this section, we present the main concepts behind `InspectorGadget` and its architecture. We detail the internal representation of gadgets, the abstract model of microcontrollers and the specification of masking parameters.

### 3.1 Gadget representation

There exists a wide variety of masking gadgets in the literature. There may be several solutions for performing an operation in a masked way. Some gadgets work on Boolean masked values, while others work on arithmetic masked values or other types of masking. One example is the ISW multiplication [ISW03], which can be performed on various masking types, adapting the operations performed to the type of masking. There may also be different gadgets for performing the exact same operation but with different types of masking. For example, it is quite easy to build the naive gadget that performs the addition modulo a prime number  $p$  when the input and output masking types are arithmetic

(modulo  $p$ ). Whereas the same operation becomes much harder to perform when the input and output masking types are Boolean (a solution was first proposed in [BBE<sup>+</sup>18]).

To be able to clearly identify all the gadgets integrated into our estimator, with all their specific features, we have defined several parameters for defining a gadget. These parameters are as follows:

- gadget functionality
- input and output masking type
- gadget group
- gadget name
- gadget reference
- memory cost estimation function
- performance estimation function

One of the gadgets integrated into our estimator is the arithmetic to Boolean conversion (denoted A2B in the following) from [SPOG19]. We will use this gadget as an example throughout this section.

**Gadget functionality:** It denotes the type of operation the gadget performs. In order to clearly identify gadgets with the same functionality, we use a consistent naming scheme for all gadgets. In the case of our reference gadget, the operation performed is a secure A2B conversion with, as input, shares belonging to  $\mathbb{Z}/p\mathbb{Z}$  with  $p$  prime. We will refer to this functionality as SecA2BModp. Likewise, a gadget performing a secure A2B conversion over  $\mathbb{Z}/2^n\mathbb{Z}$  is denoted SecA2BPowerOf2. In this way, we can clearly identify the difference in use between two very similar gadgets. It is also important to distinguish gadgets that work with bit-sliced data from others. To do this, in our tool, we add the prefix "Bs" to the functionality of these gadgets, *e.g.* BsSecAdd [BC22].

**Masking types (input/output):** As previously stated, two gadgets can have the same functionality but different masking types for input and output values. In order to determine whether or not a gadget can be used in a given context, it is therefore important to specify the input and output masking types. In the case of our reference gadget, the input value is masked with arithmetic masking and the output value is masked with Boolean masking.

**Gadget group:** We defined a gadget group by gathering the functionality and the types of masking for input and output values of a gadget together. A gadget in a group can be replaced by any gadget in the same group, as it has the same characteristics. This notion of group allows to easily compare the complexity of all the gadgets in a given group, to know which one is the most efficient for a given cryptographic scheme and a given microcontroller. The group *ID* (or name) is formed as the concatenation of the functionality, and the types of input and output masking. By convention we denote the masking types as follows: S1 for Boolean, S2 for Arithmetic, etc. Using this convention, our reference gadget belongs to the group denoted SecA2BModpS2S1.

**Gadget name:** All the gadgets belonging to the same group have a unique gadget name. To make them easily identifiable, we chose a name referring to the paper into which they were first introduced. For our reference gadget, we chose the name SPOG19 (as a shortcut for SchneiderPOG19).

If we analyze our reference gadget, we may observe that it requires two other gadgets for performing a secured addition modulo  $p$  and a refresh operation on arithmetic masked values (see Algo 3 in Appendix A). Therefore, any gadget from the respective groups SecAddModpS1S1 and RefreshS1S1 can be used. In turn, the gadgets from these groups may also use sub-gadgets. Figure 1 illustrates this hierarchical representation. Gadget

groups are represented with boxed text. Dashed edges from a gadget group to gadget names symbolize all the gadgets from that group. Plain edges illustrate the fact that a given gadget requires one or more subgadgets belonging to one or more gadget group. In figure 1, the subtree in red represents the implementation choices made in [SPOG19].

It is interesting to note here that `InspectorGadget` does not ensure safe composition of gadgets. Looking at Figure 1, one can see that we have two options for the Refresh gadget. The original implementation of [SPOG19] uses the BDFGSZ16 version (from [BBD<sup>+</sup>16], Algo. 4b), which achieves strong non-interference (SNI) security (which implies some guaranties over the gadget composability). Switching to the RP10 version (from [RP10]) will improve the efficiency of the overall gadget, however it might break its security guaranties. Indeed RP10 refresh algorithm does not achieve SNI security.

The user must bear in mind that `InspectorGadget` can propose new paths in the gadget tree but does not ensure that these paths are safe. Interesting new paths will have to be proved independently.

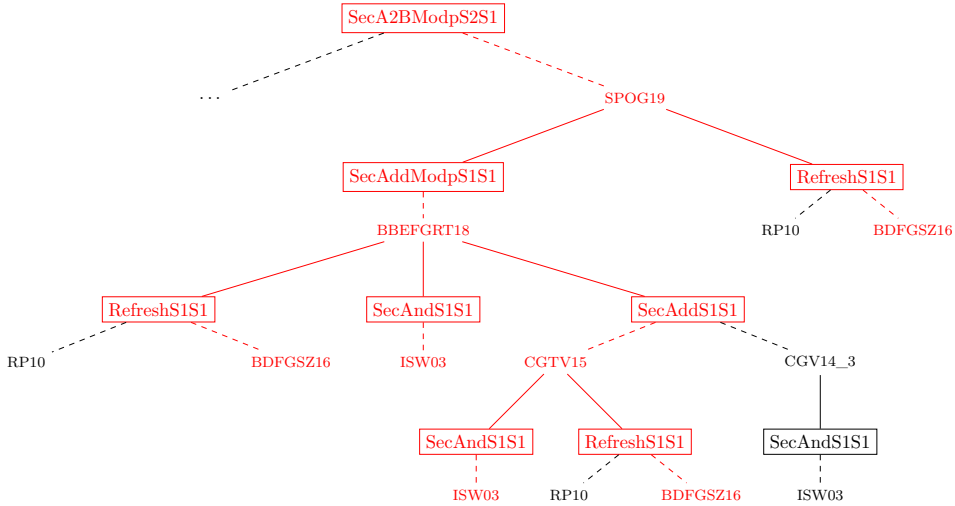


Figure 1: Tree-based representation of the alternative implementations of the gadget `SecA2BModpS2S1` from [SPOG19]

**Gadget reference:** In addition to the gadget group and name, which gives the main information about the gadget, we also associate a reference which contains a link to the paper where it was introduced.

**Estimation functions:** Each gadget is associated with a memory estimation function and a performance estimation function. More details on these functions will be given in section 3.4.

### 3.2 Modeling a Microcontroller

Our estimation tool uses a simplified, yet easily adjustable, model of microcontroller. The main characteristics of a microcontroller are stored in a configuration file. For a given microcontroller, this file should contain:

- the cost of each *atomic operation*
- the total size of the register space available (in bits)

- the total memory space available (in bits)

The idea is to not assume anything about these parameters and let the user decide, through a simple configuration file, the basic operation costs and storage dimension of the microcontroller.

**The atomic operations** gathers a predefined list of classical low level operations over small operands (here small means the microcontroller register size), like shift, addition, multiplication, load/store in memory, etc. that are usually handled by single assembly instructions. Few, more complex operations, are also considered *atomic*: integer division (*resp.* modular reduction) relative to a small dividend (*resp.* modulo) (specially useful for PQC algorithms) and, often useful in a masking scheme, the generation of a small random value.

As already mentioned, the goal of the estimation tool is not to produce cycle count accurate figures but provide estimates that allows to compare different gadgets in a peculiar microcontroller context. Hence, the cost of each *atomic operation* does not have to strictly correspond to a number of clock cycles on a specific microcontroller but translate relative costs between the different atomic operations. This allows to easily test specific hardware configurations where, *e.g.*, the access to memory (*i.e.* load/store operations) is particularly high or where the access to fresh random value is extremely cheap, etc.

The cost estimations of `InspectorGadget` is presented in unit of CCE (*Clock Cycle Equivalent*) to emphasize the fact that we are far from real microcontroller emulation.

For our preliminary tests as well as to provide examples and ease the use of the tool, we provide five hardware configuration files. Four of them do actually correspond to real microcontrollers, namely the Cortex M0+, M3 and M4 microcontrollers. The cost of the atomic operations have been set based on clock cycle figures provided by the simulator `μVision`<sup>1</sup>.

For the cost of more complex operations, it is important to take into account the actual implementation of the operation. For example, there exists many algorithms and implementations for the modular reduction operation, each with a different cost. Our cost reflects that of the the double Barrett reduction algorithm used in [HKL<sup>+</sup>22]. The cost of this specific operation can be changed through the parameter `modulo_Q_op` in the hardware configuration file.

Maybe even more subject to change from one context to the other is the choice of the random generation cost. As an example, we have chosen to measure the cost of generating a random value with the `rand()` C function. Yet, we reckon that practical implementation should base their random generation on a more secure PRNG (see Section 4.3).

We note that all these microcontrollers are very similar. This is due to the fact that they are almost all based on the same architecture. The costs of their atomic operations are the same, with the exception of the *CortexM0+* model, which differs in terms of load, store and division costs.

The last hardware configuration file is purely theoretical and tries to stay close to what can be found in the literature under the term *operation count*. The idea is to have an idealized context where all atomic operations, including random generation, cost 1 CCE. In the following, this configuration is referred to as the *OperationCount* model.

**The storage space** description might appear too simplistic but allows to capture what seems important while keeping the estimation functions simple and generic.

---

<sup>1</sup>Keil `μVision` Simulator: <https://www.keil.com/>

The core idea is to choose a size limit for data manipulation inside registers and inside memory (in the current version, no cache level is defined in `InspectorGadget`). On the one hand, for gadgets that can run inside the register size limit, the tool assumes that a good implementation would successfully limit the access to memory (*i.e.* the use of load and store operations when accessing data). On the other hand, gadgets for which the memory footprint estimation exceeds the memory size limit are deemed *impossible to run*.

The register geometry (their number and individual sizes) is not taken into account as it would force the gadget description to reach a description level too close to a real implementation level, a granularity we want to avoid to remain as generic as possible. This choice, while giving to `InspectorGadget` its strength, has many (bad) implications on the precision of our estimations that the user must understand before interpreting the results.

Say a gadget, for a number  $n$  of shares, is estimated to require around  $X$  bits of data storage (for intermediate variables, tables, etc.).

- If  $X$  is lower than the registers space size, the estimation tool will consider that the gadget can run without access to memory by keeping the  $X$  bits inside the registers space. Obviously, if  $X$  is too close to the register space size, the register geometry of a real device would make this assumption often impossible.
- Worse, if  $X$  is greater than the registers space size, the estimation tool will assume that part of the data is stored in memory. Due to the high level description of the gadgets, it is not possible for the estimator to automatically decide which part of the data will be stored in memory or not. However, this decision will have important impact on the performance cost (and is very much dependent on the low level implementation optimizations based on the register geometry).

For the former case, we decided to live with the issue, assuming that the error would not be huge in practice and would affect all gadgets similarly (one key goal of our tool is to be fair when comparing gadgets, before providing realistic figures). For the latter, more complicated, case we decided to delegate the choice to the user. This means that, when integrating a new gadget, the user should insert inside the cost estimation function, the list of load and store operations that will be applied to the data that seems deemed to be stored in memory when registers are overtaken. For instance, data that increase in size when  $n$  increase will typically need to be stored in memory at some point when  $n$  increases. A natural case appear in the table-based masked gadgets, an example is presented in Appendix A.5.

For all the *CortexM* models, the register space limit is set at 416 bits, which corresponds to the total register space available for a Cortex M0+, M3, and M4 microcontrollers. The memory space limit is set to:

- *CortexM0+* model:  $768 \times 10^3$  bits, *i.e.* to match with the FRDM-KL82Z chip (that can be found on the NXP Freedom Development Board for Kinetis Ultra-Low-Power KL82 MCUs).
- *CortexM3* model:  $1024 \times 10^3$  bits, *i.e.* to match with the STM32F215RET6 chip (that can be found on the ChipWhisperer NAE-CW308T-STM32F2HWC development board).
- *CortexM4* model:  $5120 \times 10^3$  bits, *i.e.* to match with the STM32L4R5ZIT6 chip (that can be found on the NUCLEO-L4R5ZI development board).

The *CortexM3* model, was our initial target, the corresponding chip can be found on the popular ChipWhisperer development board. Later, we added the *CortexM0+* and *CortexM4* models, in order to compare the estimations provided by `InspectorGadget` with the exact



costs of publicly available implementations. These comparisons are presented in Section 4.3.

In the idealized *OperationCount* model, the register and memory space are not taken into account. They are virtually set to infinity.

### 3.3 Masking parameters

Our tool only needs a very limited set of masking parameters, namely the number  $n$  of shares and their size (in bits). Examples of masking parameter configuration files are provided with our tool, including a configuration file for Kyber, where the bit size of masked shares is set to  $\lceil \log_2(p) \rceil = 12$  bits.

### 3.4 Estimation functions

#### Goal of Estimation functions

As we explained in section 3.1, each gadget is associated with a memory cost estimation function and a performance estimation function. The aim of the memory cost estimation function of a gadget is to take into account the critical memory path of the memory to estimate the maximum memory space required by the gadget. This memory space is given in bits. The aim of the gadget performance estimation function is to return an estimation of the cost of a gadget based on the number of CCE (for *Cycle Count Equivalent*) assigned to each atomic operation for a given microcontroller (see Section 3.2). Although these functions have two distinct purposes, they share similarities in their construction.

In Appendix A, we provide examples of such estimation functions. Note that our tool is flexible so that the user can define her own estimation function, for example by taking into account additional costs.

#### Choices of implementation

One of the aims of our tool is to keep it simple enough for a user to easily build these estimation functions. To achieve this, we ignored certain operations performed during the execution of a gadget.

For the memory cost estimation functions, we consider that the memory taken up by loop counters is negligible compared with the memory taken up by masked variables, so we do not take their memory cost into account. Similarly, we do not account for the memory cost of pointers. We also do not consider the memory taken up by the variables given as input to gadgets. If we did, some variables would be counted twice when adding the memory cost of an intermediate gadget to the memory cost of the main gadget.

For performance estimation functions, we also neglect loop counters manipulation. Concerning operations related to accessing and storing variables in memory, as explained in Section 3.2, we decided to count the cost of `load` and `store` operations (explicitly defined inside the cost function) when the overall required data storage of the gadget exceeds register space limit.

Also, we have decided to consider that a variable that contains a constant value (for a given masking parameter) is defined outside the gadgets. Thus, its memory and affectation cost is not taken into account. Even if such a variable value is explicitly computed (*e.g.* from the masking parameters), it is still considered to be calculated outside the gadget, and its CCE cost is not taken into account when calculating gadget performance. An typical example of such a variable is a binary mask defined to reduce values to the same number of bits as the size of the shares in the mask scheme.

When building estimation functions of a given masked gadget, the developer works from two different kind of material: the gadget source code when available (usually low level C or assembly code) and a pseudo-code (high level algorithmic description). As explained before, neither of these material perfectly fit with `InspectorGadget`'s description level. The developer needs to take a step back from the actual source code and remove architecture specific optimizations, whereas he should anticipate necessary additional costs when working from a high level pseudo code. Notably, he must foresee the use of temporary variables and hidden computations. For example, a large number of algorithms require the generation of a  $k$ -bit random value and we cannot assume that the random generator will provide that. It is then systematically assumed that a mask is applied at the output of the random generation to reduce the random value to  $k$  bits.

We emphasize again that the absolute results of the estimations are not the preliminary goal of `InspectorGadget`. Even if all the above mentioned implementation choices are arguably questionable and could be modified in the future, we believe that there is no perfect choices, and that the most important is to agree on a single implementation philosophy of the cost functions and stick with it.

For both types of function, we have provided (Appendix A), a few examples to show how they were built in practice in our tool and highlight very specific parts of the implementation philosophy we choose to follow:

In Section A.1, we begin by detailing the tools we use to construct our estimation functions for any gadget. In Section A.2 is presented an example of memory cost estimation function. For performance estimation functions several examples are provided. To explain the basic principles of our method, we present in details a first example in Section A.3 of a function based on a fairly simple gadget, which does not call other sub-gadgets. Next, we give an example in Section A.4 of a more complex gadget based on recursion. We explain the additional elements we had to take into account when building the performance estimation function of this gadget, compared to the previous example. Finally, our last example in Section A.5 presents a table-based gadget. We illustrate with it the case where storing variables in memory leads to additional performance costs.

### 3.5 Limitations and Possible improvements

In building our gadget estimation tool, we let various avenues of improvement for future work. The main directions are listed here.

**Modeling Strategies.** Choosing the right abstraction level for the hardware modelling is not an easy task and `InspectorGadget` works at a level of abstraction that obliterates most of the device architecture and then blinds us from many details that have important impacts on the real performances of an implementation. A more precise modelling would allow to capture finer grain of optimizations with respect to the hardware capabilities but will eventually require to work on concrete gadget implementations. In this sense, `InspectorGadget`'s level of abstraction is the last relevant step before real implementation comparisons and then is the last step where a comprehensive, homogeneous, comparison of masked gadgets is realistic (in terms of efforts from our community to work on a common platform).

Nevertheless, there is still some room for improvement on the modeling strategy. For instance, the register space information is mainly used to detect the necessity to store data in memory (and then use load/store operations when accessing it), while the memory space information is only here to set a limit for gadgets. Finer grain usage of these metrics can be imagined and we believe an interesting direction of work would be challenge our (maybe too simple) decisions while keeping the hardware modelling simple.

**Load, store and mov operations.** Taking into account load, store and mov operations has always been major source of questioning for us. These are very common operations, and it seems complicated to know whether an operation will require a load or a mov operation, without having the assembly code of the gadget. We have therefore chosen to rely on the developer of the costs functions (as mentioned in Sections 3.2 and 3.4). In the future, it may therefore be worthwhile to look at this question again to see if there is a simple way of taking these operations into account.

**Masking type control.** As described in Section 3.1, when we add a gadget to our tool, the type of masking of the inputs and outputs of this gadget must be specified to clearly identify which group the gadget belongs to. It might be interesting to keep track of the masking type through the gadget computation. Indeed, this type can change during the execution of a gadget (with masking type switching mechanisms) and the tool could automatically check that the input masking type of a sub-gadget call matches with the current masking type.

**Security in probing model.** All the gadgets integrated into `InspectorGadget` (and most of the recent proposals in the literature) are proven secure in a security model. The security models are not equivalent (*e.g.* NI, SNI [BBD<sup>+</sup>16] or PINI [CS20] provide different security guaranties with respect to composability). A useful improvement to `InspectorGadget` would be to store the security model of all gadgets. This would make possible to constraint the sub-gadgets security model when automatically composing new gadgets (*e.g.* require a SNI refresh gadget when this is required by parent gadget to stay secure). Another interesting avenue of improvement would then be to plug Tightprove [BGR18] tool so that the probing security of the various gadget compositions could be automatically verified.

**New Hardware models.** New hardware models can be easily created and added to `InspectorGadget`. Right now only ARM-CortexM micro-controllers are available, next step would be to work on non-ARM-based models, *e.g.* RISC-V.

## 4 Comparison of Masked Kyber 1-bit Compression Gadgets

In this section, we will demonstrate the main advantages of our tool, namely (1) aggregate all the (formally proved) masking schemes of the literature in a coherent library of masked gadgets, (2) compare them on fair grounds (*i.e.* go beyond the asymptotic complexity and include hardware specificity) and (3) automatically generate new variants based on a growing library of masked gadgets.

We focus on the Kyber key encapsulation scheme as a case study and more precisely on the 1-bit compression function<sup>2</sup> (see Section 2 for a description of the function). Indeed several proposals for masking this function can be found in the recent literature and finding an efficient solution is still an active research topic.

---

<sup>2</sup>In Kyber the 1-bit compression function is applied independently to all coefficients of polynomials of degree 256. Some implementations take into account that several coefficients are compressed to improve efficiency, we hence compare the 1-bit compression function applied to a full polynomial rather than on a single coefficient.

The masked versions of Kyber 1-bit compression integrated into our tool are listed in Table 1 with their asymptotic complexities (as a function of the number of shares  $n$ , Kyber prime modulus  $p$  and its bit-size  $k = \lceil \log_2 p \rceil$ ).

Table 1: Complexity of masked 1-bit compression implementations, with  $n$  the number of shares and a  $k$ -bit prime modulus  $p$  (for Kyber,  $p = 3329$  and  $k = 12$ ).

1-bit Compression Gadget		Complexity
CoronGMZ23	[CGMZ23]	$\mathcal{O}(n^2(\log_2 p + \log_2 n))$
BronchainC22	[BC22]	$\mathcal{O}(n^2(\log_2 p + \log_2 n))$
CoronGMZ22_13	[CGMZ22]	$\mathcal{O}(n^2)$
BosGRSV21	[BGR <sup>+</sup> 21]	$\mathcal{O}(n^2 \log_2 k)$
OderSPG18	[OSPG18]	NA

Each of these implementations has its own specificity. OderSPG18 compression is a special case, as it is an implementation that only works for 2 shares. The CoronGMZ23 and BronchainC22 implementations, on the other hand, can perform  $d$ -bit compression, for  $d \geq 1$ , and not just  $d = 1$ . The BronchainC22 implementation uses bitslicing to optimize calculations. Finally, the CoronGMZ22\_13 implementation is table-based, and takes advantage of the use of registers (when available) to be more efficient.

Moreover, the compression function will usually require the use of generic sub-gadgets. For instance, one way to implement the compression is by first converting the input from arithmetic (mod  $p$ ) masking to Boolean masking (an *A2B* conversion for short) and then implement the compression function on the Boolean shares. A priori any A2B conversion can be used. Our tool allows to either force a specific implementation or give the possibility to use any available A2B conversion in the gadget library. An A2B conversion is usually also based on various generic sub-gadgets (*e.g.* the *SecADDModp* gadget that allows to add (mod  $p$ ) two inputs while the inputs and outputs are masked with Boolean masking), and so on.

Hence, for each of the 1-bit compression gadgets, several variants are possible (if the tool finds two or more equivalent versions of a sub-gadget). Table 2, in appendix, keeps track of all five implementations from Table 1 together with their variants (the published variants are highlighted in red).

It is important to bear in mind, however, that the variants created by the tool do not enjoy a security proof. Then, if those variants are deemed of interest based on their efficiency and/or memory footprint, the security must be assessed separately.

For our tests, we used hardware parameters that models an ARM Cortex M3 chip and set the memory size to  $1024 \times 10^3$  bits, *i.e.* to match with the STM32F215RET6 chip (that can be found on the ChipWhisperer NAE-CW308T-STM32F2HWC development board). Another parameter that will be important is the cost of random generation, which we have set at 32 CCE (Cycle Count Equivalent) based on the cost in CPU cycles of a call to the C function `rand`<sup>3</sup>.

This chip model (referred to as *CortexM3* in the following) will be compared to the *OperationCount* model where all atomic operations (including the random generation) have same cost (1 CCE) and register size is considered infinite.

First, we will look at the performance results before investigating the memory footprint

<sup>3</sup>We used *μVision* (<https://www.keil.com/>) to estimate the average cost of a call to `rand` on a ARM Cortex M3 chip.

estimations.

## 4.1 Performance Comparisons

### 4.1.1 Realistic Number of Shares

Figure 2 shows the CCE cost of each gadget variants, for a number of shares  $n \in \{2, 3, 4, 5\}$  (*i.e.* a realistic context). The legend identifies all the variants (all details are given in Table 2, Appendix B) for the 5 masked compressions. For each masked compression, the original implementation (proved in the original paper) has a plain line while automatically generated variants have dashed lines.

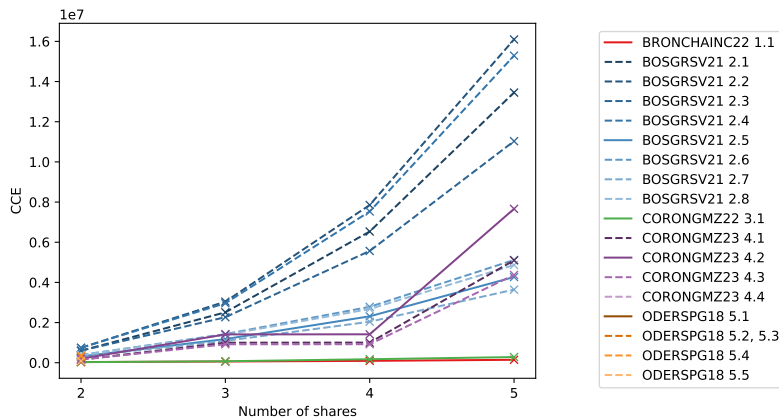


Figure 2: Performances – CortexM3 model

First of all, we can see 4 blue curves that stand out from the rest because of their high cost. These curves represent the compression of BosGRSV21, which uses an A2B conversion modulo a prime  $p$  of [BBE<sup>+</sup>18], which has cubic complexity ( $\mathcal{O}(n^3 \log k)$ ). The other 4 blue curves represent variants of BosGRSV21 compression using the A2B conversion of [SPOG19] (Algo.3), which has quadratic complexity ( $\mathcal{O}(n^2 \log k)$ ).

The purple curves represent CoronGMZ23 compression variants. A kind of plateau can be observed for  $n = 3$  and 4 for the first three variants (the fourth one – 4.4 – only works for  $n = 2$ ). These three variants use CoronGV14 A2B (modulo a power of 2) conversion [CGV14], which requires the number of shares of the masked values to be a power of two<sup>4</sup>. Interestingly, the published variant (from [CGMZ23]) appears to be the most expensive of the three. It uses the original CoronGV14 A2B conversion, based on the *SecADD* gadget from [CGV14]. The other two variants use the more efficient CoronGTV15 *SecADD* gadget from [CGTV15]<sup>5</sup>.

Comparing the published variants of BosGRSV21 (2.5) and CoronGMZ23 (4.2), our tool shows quite similar performance results (with slightly better results for CoronGMZ23 when  $n$  is a power of 2). This is clearly not something one would have expected when only

<sup>4</sup>When this is not the case,  $n$  is brought to the next higher power of 2. This results in an additional cost when the number of shares is not a power of 2.

<sup>5</sup>In [CGMZ23], the authors explain that they chose to keep the original CoronGV14 A2B conversion because, for a small number of shares, using CoronGTV15 *SecADD* gadget would not have a noticeable impact on the performance of their conversion. Their comparison of the two *SecADD*s was made using their cost in number of operations. However, in Figure 2, we see a performance gain when using CoronGTV15 *SecADD* for their conversion, as soon as more than 2 shares are used for masking.

looking at their asymptotic complexity (recalled in Table 1).

CoronGMZ22\_13 (green) and BronchainC22 (red) are the most effective according to our estimates. Here again, we can see the limits of asymptotic complexity, which placed BronchainC22 compression at the same level as CoronGMZ23 complexity. In their paper introducing this compression, Bronchain and Cassiers showed, by comparing the performance of C implementations, that their bitsliced compression is fairly equivalent to that of CoronGMZ22\_13 for  $n = 3$ , although very quickly, as the number of shares increased, it becomes much more efficient. The results we obtained with our tool clearly confirms this result. We compare these two compressions in more details below.

Finally, some compression variants only work for  $n = 2$  shares. These are the compression variants of OderSPG18, one of the first masked compressions, and CoronGMZ23 4.4, which uses Goubin01 A2B modulo to a power of 2 conversion [Gou01], which only works for  $n = 2$  shares. Interestingly enough, these variants do not seem to beat BronchainC22 and CoronGMZ22\_13 even though these implementations work for any number of shares (this will appear clearly on Figure 3).

**A closer look** To investigate which compression variant gives the best results, Figure 3a depicts the same results but with a zoom on the y-axis. For comparison purpose, Figure 3 also includes Figure 3b which displays the results when the cost of random generation is divided by 2 and Figure 3c where all atomic operations (including the random generation) cost 1 CCE.

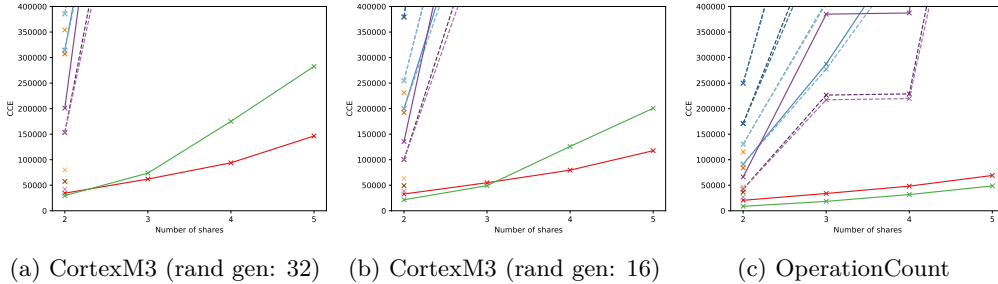


Figure 3: Performances – Small Number of Shares

Looking at the zoomed-in results in Figure 3a, we can see that for  $n = 2$  shares, BronchainC22 and CoronGMZ22\_13 compressions have similar costs. We can see that other gadgets are also very similar in cost: 2 variants of OderSPG18 compression, including the original OderSPG18 5.1 implementation and a variant using Goubin A2B conversion. There is also the CoronGMZ23 variant using the same A2B conversion. As our tool gives only approximate results, we can consider that the cost difference between these compression variants is negligible. Or, in other words, it is hard to tell from here which variant would beat the others considering real implementations on a real device.

From Figure 3a one can also see that, as the number of shares increases, BronchainC22 compression (red) quickly becomes more interesting than CoronGMZ22\_13 compression (green), which is coherent with [BC22] results.

A large number of parameters come into play when estimating the cost of a gadget. Given that most compression gadgets need to generate random values, it is natural to have a good estimation of this cost. However, there is no correct estimation in general since every real world scenario will have a different way to generate randoms and its cost can be

very different (depending on the availability of a hardware generator, the implementation of an efficient PRNG, the use of a slow TRNG, etc.). As a purpose of illustration, Figure 3b shows the results when the cost is divided by 2 (compared to our original model).

We can see that this change has a positive impact on the cost of compressing CoronGMZ22\_13 compared to BronchainC22 compression. It shows that somehow CoronGMZ22\_13 is more dependent on the RNG cost than BronchainC22.

With Figure 3c we go one step further. In this model the random generation costs no more than any atomic operation (*e.g.* an XOR operation) and the register size is considered infinite, *i.e.* there is not additional cost to access and store data in memory (which is then beneficial for table-based schemes like CoronGMZ22\_13). This so-called operation count estimation of the complexity of a masking scheme is pretty common in the literature (see *e.g.* [CGMZ22, CGMZ23]). In this setting, CoronGMZ22\_13 indeed becomes the most efficient scheme.

In conclusion, in terms of performance, CoronGMZ22\_13 and BronchainC22 are both highly effective masked compressions. Their costs remain very similar for a small number of shares, and they have the advantage of working for any number of shares  $n \geq 2$ .

Let us now look at what can be learned on these schemes when considering large number of shares.

#### 4.1.2 Large Number of Shares

Figure 4a shows all the compression variants for  $n$  ranging from 2 to 256 while Figure 4b slightly zooms on the y-axis.

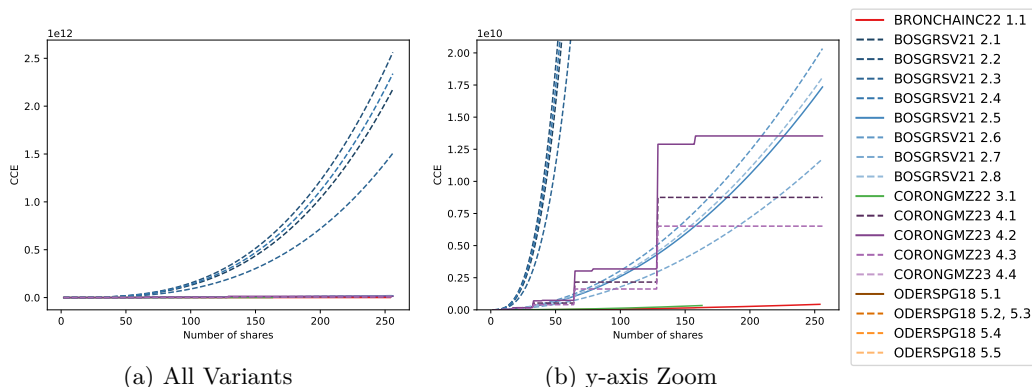


Figure 4: Performances – CortexM3 model

As expected, it appears clearly on Figure 4a that BosGRSV21 variants using the A2B conversion with cubic complexity have an exploding CCE cost. To distinguish the other compression variants, Figure 4b displays a first level of zoom.

The performance curves obtained are consistent with those obtained for small numbers of shares. BosGRSV21 compression variants using A2B conversion of quadratic complexity and CoronGMZ23 compression variants remain within the same orders of complexity. CoronGMZ22\_13 and BronchainC22 remain the most efficient compression implementations. However, we can see that the curve for CoronGMZ22\_13 compression stops at around  $n = 160$ . This is because the memory footprint of this compression exceeds the total memory space of the modeled microcontroller<sup>6</sup>. Similarly for BronchainC22 compression,

<sup>6</sup>The memory space is limited to  $1024 \times 10^3$  bits and is considered (quite conservatively) to be fully available for the execution of the masked compression.

the results stop slightly before reaching the 256 shares. We’ll look at this in more detail in the next section, when we investigate the memory cost estimates.

Similarly as for the small number of shares, let us look at the best performance results when we deviate from our original hardware model.

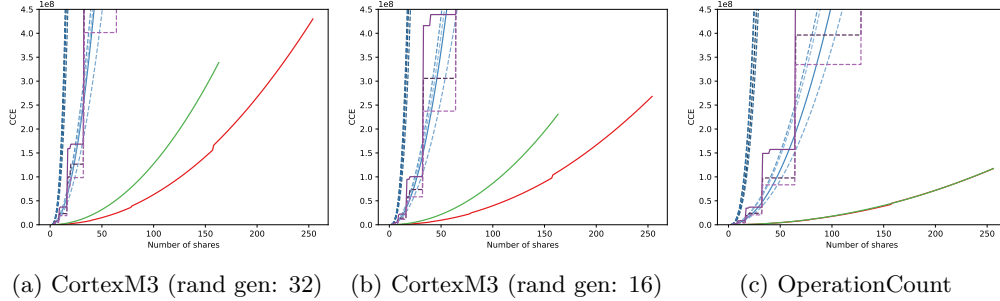


Figure 5: Performances – Large Number of Shares

Figure 5a shows the previous results, zoomed in on the costs of CoronGMZ22\_13 and BronchainC22 compressions. Figure 5b shows the results obtained using the same microcontroller model, except that the cost of random generation has been divided by 2. Finally, Figure 5c shows the results obtained using the OperationCount model.

We can observe that, in the more realistic models BronchainC22 shows a clear performance improvement compared to CoronGMZ22\_13, while in the OperationCount model, the compression performance curves of CoronGMZ22\_13 and BronchainC22 overlap. This confirms that the main difference (in performance) between CoronGMZ22\_13 and BronchainC22 lies inside the random generation cost and the cost to access and store data in memory.

Let us now turn our attention to the memory footprint.

## 4.2 Memory Footprint Comparisons

### 4.2.1 Realistic Number of Shares

Figure 6 shows the memory costs for the different compression variants for a number of shares  $n \in \{2, 3, 4, 5\}$ . Also shown in black is the register space, in our CortexM3 model this limit is set at 416 bits, which corresponds to the total register space available for a Cortex M3 microcontroller. In the estimation tool, when the required memory reaches this limit, the `load` and `store` operations start adding performance cost, below this limit these operations are considered free. This might seem a bit simplistic but it allows to capture (in a conservative way) the (often hidden) complexity of data manipulation.

It can be seen in this figure that some memory cost curves for compression variants have been merged. As with the performance results, some intermediate gadget configurations are equivalent in terms of memory cost.

We can see that all BosGRSV21 variants have overlapping results, *i.e.* very similar memory footprints, even though they showed very different performances.

When looking at all variants of OderSPG18 (only for 2 shares), we can observe that the original (published) version (5.1) is much more greedy in memory than the auto-generated variants. Which makes the latter quite competitive for  $n = 2$  (as we have seen they demonstrate good performances too). However, these variants should be first checked for security flaw before being seriously considered.



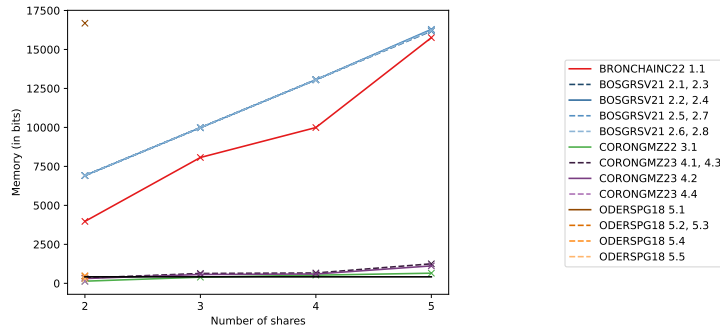


Figure 6: Memory footprint – CortexM3 model – Small number of shares

CoronGMZ23 variants have a low memory footprint but does not beat CoronGMZ22\_13 compression which shows (for these small number of shares) to be the best choice for memory constraint devices, whereas BronchainC22 has an important memory usage, due to data bitslicing. Although CoronGMZ22\_13 compression is table-based, the fact that it performs an A2A (Arithmetic modulo a prime  $p$  sharing to Arithmetic modulo a power of 2 sharing) transformation to a modulo  $2^k$  with a quite small  $k$ , and expect a single bit output, allows to keep fairly small tables.

As we will see for large values of  $n$ , the tables will eventually grow to excessive sizes but all in all CoronGMZ22\_13 implementation seems a very interesting choice for reasonable number of shares both in performances and memory footprint.

#### 4.2.2 Large Number of Shares

Figure 7 shows the results obtained for  $n$  ranging from 2 to 256. This time, the black horizontal line defines the memory space limit, it is set to  $1024 \times 10^3$  bits in our CortexM3 model.

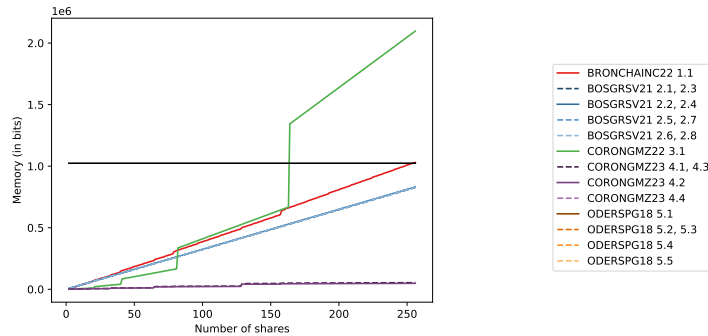


Figure 7: Memory footprint – CortexM3 model – Large number of shares

CoronGMZ23 variants keep a very low memory cost. This is mostly explained by the sequential treatment of the polynomial coefficients, one compression at a time, unlike most other implementations which perform the compression on 32 or 256 coefficients simultaneously (for performances gain).

We can see that for masking with a large number of shares, BronchainC22 compression becomes more expensive than BosGRSV21 compression variants. In fact, around  $n = 250$  shares, the memory cost of BronchainC22 compression is higher than the total memory space of the modeled microcontroller.

One can also observe small increments in the memory cost of BronchainC22 compression. These are due to the use of an A2A conversion to a power of 2, whose exponent increases with the number of shares, and therefore implies having larger variables.

In the case of CoronGMZ22\_13 compression, one can observe similar increments, but on a much larger scale. This implementation also uses A2A conversion to a power of 2, which increases with the number of shares. However, in this case it implies an exponential increase in the table size used for the A2B conversion, and then drastically increases its memory footprint. As a result, CoronGMZ22\_13 compression becomes the most expensive in memory from around  $n = 80$  shares, and is no longer usable after around  $n = 160$  shares for the modeled microcontroller.

### 4.3 Comparisons with Real Implementations

In this section we compare `InspectorGadget`'s estimates with the real costs of two different implementations, namely the compressions to 1 bit proposed by Bronchain and Cassiers in [BC22], denoted BronchainC22, and that of Kundu *et al.* [KDB<sup>+</sup>22], referred to as KunduDBKV22, which is a compression used in the Saber's scheme [DKRV18].

We chose these implementations because they satisfy three important points for our comparisons: 1) they work for high-order masking, 2) the respective costs in number of clock cycles are provided in the associated papers, and 3) they have the advantage of being open-source<sup>7,8</sup>. For BronchainC22 [BC22], the cost is given for a number of shares ranging from 2 to 16 shares. For KunduDBKV22 [KDB<sup>+</sup>22], it is limited to 2, 3 and 4 shares.

Both implementations are evaluated on Cortex M4 microcontrollers. For BronchainC22, the authors estimated the cycle counts on a NUCLEOL4R5ZI development board. For KunduDBKV22, the authors used the STM32F407-DISCOVERY.

The two implementations generate random values differently, which was taken into account in our estimates. For BronchainC22, thanks to the authors, we learned that for 16 shares, the cost of random generation was about 80% of the total cost. Using this information, we estimated that the random generation cost should be set to about 74 CCE in `InspectorGadget` to obtain a similar cost ratio (for 16 shares). For KunduDBKV22, the authors explain in their paper that their implementation uses the on-chip TRNG to generate randoms in 20 CPU cycles.

In Figure 8, we present a comparison between these two masked compression implementations against the CCE cost estimates obtained with `InspectorGadget` using the hardware modelling of the corresponding microcontrollers. Solid lines represent reference implementations, while dotted lines represent `InspectorGadget`'s estimates.

For BronchainC22, we can see that `InspectorGadget`'s estimation results (in dark red) are close to the actual costs of the reference implementation (in light red). As the number of shares increase, the two curves follow the same shape.

For KunduDBKV22, `InspectorGadget`'s estimation results are shown in dark blue, while the actual costs of this implementation are shown in light blue. For 2 shares, `InspectorGadget`'s estimate and the actual cost are very close. For 3 and 4 shares, the gap between the two curves is of the same order of the gap observed for Bronchain22 curves.

Although we did not have cycle counts for KunduDBVK22 for more than 4 shares, we estimated the cost for up to  $n = 16$  using `InspectorGadget`. It can be seen that as the number of shares increases, Saber's compression becomes more efficient than that of Kyber, which is coherent with the respective costs of the random generation (i.e. 74 CCE vs 20 CCE).

<sup>7</sup>BronchainC22: [https://github.com/uclcrypto/pqm4\\_masked](https://github.com/uclcrypto/pqm4_masked)

<sup>8</sup>KunduDBKV22: <https://github.com/KULeuven-COSIC/Higher-order-masked-Saber>

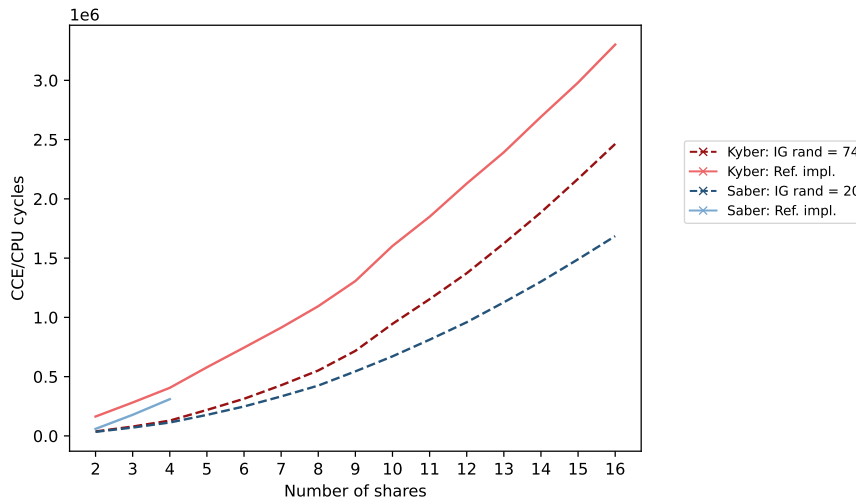


Figure 8: Comparisons between `InspectorGadget`'s estimates and actual implementation costs for the 1-bit Compression functions of Kyber (BronchainC22) and Saber (KunduD-BKV22)

As explained in Section 3.4, the discrepancies between `InspectorGadget`'s estimation results and the actual implementation costs come from the fact that the cost of some operations are not taken into consideration in our estimates.

By neglecting costs such as loop counters manipulation or load/store operations, we significantly reduce the estimation cost of an implementation. As explained in Section 3.5, there is still room for improvements, in order to get closer to the real cost of an implementation. It is nevertheless important to remember that the aim of `InspectorGadget` is not to give the exact cost of a precise implementation on a specific microcontroller, but rather to propose an estimate of this cost, lying between the asymptotic complexity of the gadget and the real cost of its implementation. Our goal was to have a simple, yet fair way to compare masking gadgets. We assumed that the neglected operations, *e.g.* load/store, have a similar impact (in first approximation) on all gadgets with equivalent level of optimization.

## 5 Conclusions

For low-level functions of relatively low-complexity, it is often difficult to provide fair metrics for estimating the practical cost on some specific hardware. For that purpose, we designed `InspectorGadget` as a flexible toolbox that allows to compare masking gadgets on relatively fair grounds. By providing some basic characteristics of the hardware onto which the gadget is supposed to run, our software can output an efficiency measure in cycle count equivalent together with a memory footprint in bits.

We warn the reader that the results provided by `InspectorGadget` should not be taken as absolute values. They should mainly be used for comparison purposes. One important feature of `InspectorGadget` is its ability to automatically combine compatible gadgets together. Yet, the produced gadgets may require extra security proofs.

As a case-study, `InspectorGadget` is applied to the Kyber compression function. Through this example, we demonstrate the need to compare performance figures of the literature proposals on common grounds.

But `InspectorGadget` is not limited to Kyber or PQC algorithms, it is meant to contain a comprehensive library of masking gadgets. `InspectorGadget` is open-source. We encourage the community to use it for their own purposes and to make it grow with new gadgets and microcontrollers.

## References

- [ACLZ20] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. Defeating NewHope with a single trace. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 189–205. Springer, Heidelberg, 2020. doi:[10.1007/978-3-030-44223-1\\_11](https://doi.org/10.1007/978-3-030-44223-1_11).
- [BBC<sup>+</sup>19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *ESORICS 2019, Part I*, volume 11735 of *LNCS*, pages 300–318. Springer, Heidelberg, September 2019. doi:[10.1007/978-3-030-29959-0\\_15](https://doi.org/10.1007/978-3-030-29959-0_15).
- [BBD<sup>+</sup>16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016. doi:[10.1145/2976749.2978427](https://doi.org/10.1145/2976749.2978427).
- [BBE<sup>+</sup>18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Heidelberg, April / May 2018. doi:[10.1007/978-3-319-78375-8\\_12](https://doi.org/10.1007/978-3-319-78375-8_12).
- [BBYS22] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. SoK: Design tools for side-channel-aware implementations. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 756–770. ACM Press, May / June 2022. doi:[10.1145/3488932.3517415](https://doi.org/10.1145/3488932.3517415).
- [BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR TCHES*, 2022(4):553–588, 2022. doi:[10.46586/tches.v2022.i4.553-588](https://doi.org/10.46586/tches.v2022.i4.553-588).
- [BDK<sup>+</sup>21] Michiel Van Beirendonck, Jan-Pieter D’Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *ACM J. Emerg. Technol. Comput. Syst.*, 17(2):10:1–10:26, 2021. doi:[10.1145/3429983](https://doi.org/10.1145/3429983).
- [BGG<sup>+</sup>20] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Open source publication of complete leakage model, implementations and the verification tool, 2020. URL: <https://github.com/scverif/scverif>, <https://github.com/scverif/gadgets>.

- [BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight private circuits: Achieving probing security with the least refreshing. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 343–372. Springer, Heidelberg, December 2018. doi:10.1007/978-3-030-03329-3\_12.
- [BGR<sup>+</sup>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. <https://tches.iacr.org/index.php/TCHES/article/view/9064>. doi:10.46586/tches.v2021.i4.173-214.
- [CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-M3 processors. In Junfeng Fan and Benedikt Gierlichs, editors, *COSADE 2018*, volume 10815 of *LNCS*, pages 82–98. Springer, Heidelberg, April 2018. doi:10.1007/978-3-319-89641-0\_5.
- [CGMZ22] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order table-based conversion algorithms and masking lattice-based encryption. *IACR TCHES*, 2022(2):1–40, 2022. doi:10.46586/tches.v2022.i2.1-40.
- [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.
- [CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Heidelberg, March 2015. doi:10.1007/978-3-662-48116-5\_7.
- [CGTZ23a] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of NTRU. *IACR TCHES*, 2023(2):180–211, 2023. doi:10.46586/tches.v2023.i2.180-211.
- [CGTZ23b] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of dilithium. *IACR TCHES*, 2023(4):110–145, 2023. doi:10.46586/tches.v2023.i4.110-145.
- [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, Heidelberg, September 2014. doi:10.1007/978-3-662-44709-3\_11.
- [CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO’99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999. doi:10.1007/3-540-48405-1\_26.
- [Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, Heidelberg, May 2014. doi:10.1007/978-3-642-55220-5\_25.

- [CPRR14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shihō Moriai, editor, *FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, Heidelberg, March 2014. doi:10.1007/978-3-662-43933-3\_21.
- [CS18] Gaëtan Cassiers and François-Xavier Standaert. Improved bitslice masking: from optimized non-interference to probe isolation. Cryptology ePrint Archive, Report 2018/438, 2018. <https://eprint.iacr.org/2018/438>.
- [CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, PP:1–1, 02 2020. doi:10.1109/TIFS.2020.2971153.
- [DKRV18] Jan-Pieter D’Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 282–305. Springer, Heidelberg, May 2018. doi:10.1007/978-3-319-89339-6\_16.
- [EO19] Scott Egerton and Elisabeth Oswald. GILES, 2019. URL: <https://github.com/sca-research/GILES>.
- [FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO’99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999. doi:10.1007/3-540-48405-1\_34.
- [GHP<sup>+</sup>21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on CPUs. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1469–1468. USENIX Association, August 2021.
- [GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 359–386. Springer, Heidelberg, August 2020. doi:10.1007/978-3-030-56880-1\_13.
- [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, Heidelberg, May 2001. doi:10.1007/3-540-44709-1\_2.
- [GZSW19] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. Verifying and quantifying side-channel resistance of masked software implementations. *ACM Trans. Softw. Eng. Methodol.*, 28(3), jul 2019. doi:10.1145/3330392.
- [HKL<sup>+</sup>22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked kyber on ARM cortex-M4. Cryptology ePrint Archive, Report 2022/058, 2022. <https://eprint.iacr.org/2022/058>.
- [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, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4\_27.

- [KDB<sup>+</sup>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, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 93–116. Springer, 2022. doi:10.1007/978-3-031-14791-3\_5.
- [KS22] Pantea Kiaei and Patrick Schaumont. SoC root canal! Root cause analysis of power side-channel leakage in system-on-chip designs. *IACR TCHES*, 2022(4):751–773, 2022. doi:10.46586/tches.v2022.i4.751-773.
- [KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 787–816. Springer, Heidelberg, December 2020. doi:10.1007/978-3-030-64837-4\_26.
- [LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015. doi:10.1007/s10623-014-9938-4.
- [MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 344–362. Springer, Heidelberg, June 2019. doi:10.1007/978-3-030-21568-2\_17.
- [MM22] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR TCHES*, 2022(4):311–348, 2022. doi:10.46586/tches.v2022.i4.311-348.
- [MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: ‘grey box’ modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 199–216. USENIX Association, August 2017.
- [MPH23] Quentin L. Meunier, Etienne Pons, and Karine Heydemann. Leakageverif: Efficient and scalable formal verification of leakage in symbolic expressions. *IEEE Trans. Softw. Eng.*, 49(6):3359–3375, jun 2023. doi:10.1109/TSE.2023.3252671.
- [OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical cca2-secure and masked ring-lwe implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):142–174, Feb. 2018. URL: <https://tches.iacr.org/index.php/TCHES/article/view/836>, doi:10.13154/tches.v2018.i1.142-174.
- [PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 513–533. Springer, Heidelberg, September 2017. doi:10.1007/978-3-319-66787-4\_25.
- [PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *COSADE 2017*, volume 10348 of *LNCS*, pages 282–297. Springer, Heidelberg, April 2017. doi:10.1007/978-3-319-64647-3\_17.

- [RdCR<sup>+</sup>16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-LWE masking. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 233–244. Springer, Heidelberg, 2016. doi:[10.1007/978-3-319-29360-8\\_15](https://doi.org/10.1007/978-3-319-29360-8_15).
- [RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Heidelberg, August 2010. doi:[10.1007/978-3-642-15031-9\\_28](https://doi.org/10.1007/978-3-642-15031-9_28).
- [RRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR TCHES*, 2020(3):307–335, 2020. <https://tches.iacr.org/index.php/TCHES/article/view/8592>. doi:[10.13154/tches.v2020.i3.307-335](https://doi.org/10.13154/tches.v2020.i3.307-335).
- [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, Heidelberg, September 2015. doi:[10.1007/978-3-662-48324-4\\_34](https://doi.org/10.1007/978-3-662-48324-4_34).
- [SAB<sup>+</sup>20] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>.
- [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, Heidelberg, April 2019. doi:[10.1007/978-3-030-17259-6\\_18](https://doi.org/10.1007/978-3-030-17259-6_18).
- [SSB<sup>+</sup>21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS 2021*. The Internet Society, February 2021.
- [UXT<sup>+</sup>22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. *IACR TCHES*, 2022(1):296–322, 2022. doi:[10.46586/tches.v2022.i1.296-322](https://doi.org/10.46586/tches.v2022.i1.296-322).
- [VDV21] Michiel Van Beirendonck, Jan-Pieter D’Anvers, and Ingrid Verbauwhede. Analysis and comparison of table-based arithmetic to boolean masking. *IACR TCHES*, 2021(3):275–297, 2021. <https://tches.iacr.org/index.php/TCHES/article/view/8975>. doi:[10.46586/tches.v2021.i3.275-297](https://doi.org/10.46586/tches.v2021.i3.275-297).
- [XPR<sup>+</sup>22] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, David F. Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Trans. Computers*, 71(9):2163–2176, 2022. doi:[10.1109/TC.2021.3122997](https://doi.org/10.1109/TC.2021.3122997).



- 
- [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](https://doi.org/10.46586/tches.v2023.i3.391-421).

# Appendices

## A Examples

In this section, we go into the details of how we constructed the performance and memory cost estimation functions for the gadgets we integrated into our estimation tool. We then present a few examples of how we constructed these functions for certain gadgets, to give a better understanding of what a user should take into account when integrating new gadgets into our tool.

### A.1 Tools to construct estimation functions

To retrieve the costs of the various operations performed in a gadget, or the cost in memory of all the variables used in this gadget, it is necessary to retrieve certain parameters that define these costs.

**Retrieving essential parameters.** Each estimation function needs to retrieve essential parameters to calculate its result. For all these functions, it is necessary to retrieve the value of certain masking parameters (see Section 3.3). For the performance estimation function, additional parameters must be retrieved. In order to calculate its cost in CCE, it is necessary to retrieve the cost in CPU cycles of all operations performed by the gadget. An example is given in Section A.3.

**Retrieving intermediate gadget estimates.** Some gadgets themselves use other gadgets of a specific group during their execution. For each type of function, it is necessary to retrieve information about these intermediate gadgets. For the memory cost estimation function, we need to retrieve the memory cost of each of the intermediate gadgets to find the critical path of the memory. For the performance estimation function, we need to retrieve the CCE cost of each intermediate gadget to factor it into the total cost of the main gadget. There are two ways to retrieve this information. The first is to choose to retrieve the information of a specific gadget, specifying the name of the gadget we want to use, in addition to information about its group. In this case, users can be sure that only one specific intermediary gadget will be used for their gadget. The second option is to select a gadget from the required group, specifying the group of the gadget to be used without specifying a name. In this second case, the gadget whose information will be taken will not be fixed. It can be chosen by the user when he launches his estimates, or even to test all the gadgets in the group to determine which is the most interesting for his use. It is important to note that the method used to retrieve this information must be the same for the memory cost and performance estimation functions linked to the same gadget.

**Masking parameter changes.** In the course of running a gadget, the masking parameters may change. It may happen, for example, that the shares of a masked value change of field and thus size in number of bits. It may also happen that an intermediate gadget is executed on a masked value with a different number of shares. When such changes occur, the way we have designed our estimation tool means that it is up to the user to specify the masking changes that are made in his estimation functions. He can either directly modify the masking parameters given as input to his functions, or create a new set of parameters, when these are only modified for the call to an intermediate gadget. An example of a performance estimation function we have integrated into our estimation tool, for a gadget with masking parameters that change during gadget execution, is available in Section A.4.

With all these values in hand, it is relatively easy to construct estimating functions, as we will see in the following examples.

## A.2 Example of memory cost estimation function

To give an example of a memory cost estimation function, we shall explain in detail how we chose to build this function for the SecAddModpS1S1\_BBFGRT18 gadget from [BBE<sup>+</sup>18]. This gadget belongs to the group SecAddModpS1S1 and was considered in the secure A2B conversion example from Figure 1. This gadget is described in Algorithm 1 (where  $\tilde{c}_i$  in lines 6 and 8 means bit  $c_i$  expanded from 1 to  $w$  bits).

---

### Algorithm 1: Mod-p addition of Boolean maskings (SecAddModp) [BBE<sup>+</sup>18]

---

**Data:** Boolean maskings  $(x_i)_{1 \leq i \leq n}$ ,  $(y_i)_{1 \leq i \leq n}$  of integers  $x, y$ ; the bit size  $w$  of the masks (with  $2^w > 2p$ )

**Result:** A Boolean masking  $(z_i)_{1 \leq i \leq n}$  of  $x + y \bmod p$

- 1  $(p_i)_{1 \leq i \leq n} \leftarrow (2^w - p, 0, \dots, 0)$
  - 2  $(s_i)_{1 \leq i \leq n} \leftarrow \text{SecAdd}((x_i)_{1 \leq i \leq n}, (y_i)_{1 \leq i \leq n}, w)$
  - 3  $(s'_i)_{1 \leq i \leq n} \leftarrow \text{SecAdd}((s_i)_{1 \leq i \leq n}, (p_i)_{1 \leq i \leq n}, w)$
  - 4  $(b_i)_{1 \leq i \leq n} \leftarrow (s'_i e \gg (w - 1))_{1 \leq i \leq n}$
  - 5  $(c_i)_{1 \leq i \leq n} \leftarrow \text{Refresh}((b_i)_{1 \leq i \leq n}, w)$
  - 6  $(z_i)_{1 \leq i \leq n} \leftarrow \text{SecAnd}((s_i)_{1 \leq i \leq n}, (\tilde{c}_i)_{1 \leq i \leq n}, w)$
  - 7  $(c_i)_{1 \leq i \leq n} \leftarrow \text{Refresh}((b_i)_{1 \leq i \leq n}, w)$
  - 8  $(z_i)_{1 \leq i \leq n} \leftarrow (z_i)_{0 \leq i \leq d} \oplus \text{SecAnd}((s'_i)_{1 \leq i \leq n}, (\neg \tilde{c}_i)_{1 \leq i \leq n}, w)$
  - 9 **return**  $(z_i)_{1 \leq i \leq n}$
- 

We built this function based on an implementation given in [BC22] that we show below in Listing 1. The memory cost function code that we have defined for this gadget is described in Listing 2.

First, we start by retrieving the masking parameters. In Listing 2, in lines 2 and 3 we retrieve the number of shares  $n$  and the size of the shares of masked values `bitsize_shares`, which will be useful in the following. In lines 5 to 7, we also retrieve the estimated memory cost of all intermediate gadgets: Refresh, SecAnd and SecAdd. For these 3 gadgets, input and output values are Boolean masked values. We choose not to specify a gadget name, to leave the choice of which gadgets to use to the user, when launching his estimates.

With all the important values retrieved, we can count all the variables used during the execution of the gadget. Referring to the lines 7 to 13 of Listing 1, there are 6 variables used in this implementation:  $p$ ,  $sp$ ,  $s$ ,  $b$ ,  $cp$  and  $zp$ . We do not count  $c$ , as it is a pointer, or loop counters, as explained in Section 3.4. All the variables have  $n$  shares of size  $w$  bits. In line 10 of Listing 2, we defined the memory cost as the sum of the respective footprint for these 6 variables.

Finally, to complete the function, in line 11, we add to this cost, the memory cost of the intermediate gadget which is the most expensive. This gives us the maximum memory required to run the gadget.

## A.3 Example of basic performance estimation function

We illustrate the building of our performance estimation functions on a first simple example, with a gadget that does not require the use of other gadgets. This gadget is RefreshS1S1\_BDFGSZ16. We built the performance estimation directly from Algorithm 2.

The estimation function is described in Listing 3.

```

1 void SecAddModq(uint32_t *x,
2   uint32_t *y,
3   uint32_t *z,
4   uint32_t d,
5   uint32_t q,
6   uint32_t k){
7   uint32_t p[d];
8   uint32_t sp[d];
9   uint32_t s[d];
10  uint32_t b[d];
11  uint32_t *c;
12  uint32_t cp[d];
13  uint32_t zp[d];
14  uint32_t mask = (1<<k)-1;
15  for(uint32_t i=1;i<d;i++){
16    p[i] = 0;
17  }
18  p[0] = (1<<k) - q;
19  SecAdd(x,y,s,d,k);
20  SecAdd(s,p,sp,d,k);
21  for(uint32_t i=0;i<d;i++){
22    b[i] = sp[i] >> (k-1);
23  }
24  c = b;
25  RefreshXOR(c,d);
26  for(uint32_t i=0;i<d;i++){
27    cp[i] = ((1<<k) - (c[i]&0x1))&mask;
28  }
29  SecAnd(s,cp,z,d);
30  RefreshXOR(c,d);
31  c[0] ^= 1;
32  for(uint32_t i=0;i<d;i++){
33    cp[i] = ((1<<k) - (c[i]&0x1))&mask;
34  }
35  SecAnd(sp,cp,zp,d);
36  for(uint32_t i=0; i<d;i++){
37    z[i] ^= zp[i];
38  }
39 }

```

Listing 1: Implementation of the gadget SecAddModpS1S1\_BBFGRT18 from [BC22]

First, as before, we retrieve the masking parameters that will be useful for the function. The number  $n$  of shares of the masked values is used all the time in these functions. It is retrieved on line 3 of the function.

Once this has been done, we list all the atomic operations performed during the execution of the gadget and retrieve their cost. In lines 6 to 8, we therefore recover the cost of the generation of a random number, an `xor` and an `and` operation, which we will use in the following. Since this gadget does not call any intermediate gadgets, we move straight on to calculating the CCE cost of the gadget.

To do this, we go through each line of the algorithm describing the gadget and add to the result the cost of all operations performed on that line, with the exception of operations listed in Section 3.4, like operations on loop counters. For example, we do not count operations performed on lines 1 and 2. We therefore go straight to the contents of the second loop, lines 5 to 7. In line 5, we first generate a  $w$ -bit random number. If we anticipate the additional costs of implementing this algorithm, we can see that we will have to use a mask after generating a random integer with the command `rand()` in C for example, to reduce this integer to  $w$  bits. Since the same mask is always used to generate a random of  $w$  bits, we consider it to be an immediate constant value defined in

```

1 def SecAddModpS1S1_BBEFGRT18_memory(mask_params):
2     n = mask_params.n
3     bitsize_shares = mask_params.bitsize_shares
4
5     Refresh_memory = gadgets_dict.get_memory(mask_params, "Refresh",
6         sharing_type.Boolean, sharing_type.Boolean, name = None)
7     SecAnd_memory = gadgets_dict.get_memory(mask_params, "SecAnd",
8         sharing_type.Boolean, sharing_type.Boolean, name = None)
9     SecAdd_memory = gadgets_dict.get_memory(mask_params, "SecAddModPowerOf2",
10        sharing_type.Boolean, sharing_type.Boolean, name = None)
11
12    # Computation of the memory taken by the variables
13    # Masked values of w bits: p, sp, s, b, cp, zp
14    memory = 6*n*bitsize_shares
15    memory += max(Refresh_memory, SecAnd_memory, SecAdd_memory)
16    return memory

```

Listing 2: Memory cost function of the gadget SecAddModpS1S1\_BBEFGRT18

**Algorithm 2:** Mask Refreshing Gadgets (Refresh) [BBD<sup>+</sup>16]

---

**Data:**  $\mathbf{x} = (x_i)_{1 \leq i \leq n} \in (\mathbb{F}_2)^w$  such that  $\bigoplus_i x_i = x$   
**Result:**  $\mathbf{z} = (z_i)_{1 \leq i \leq n} \in (\mathbb{F}_2)^w$  such that  $\bigoplus_i z_i = x$

- 1 **for**  $i = 1$  **to**  $n$  **do**
- 2      $z_i \leftarrow x_i$
- 3 **for**  $i = 1$  **to**  $n$  **do**
- 4     **for**  $j = i + 1$  **to**  $n$  **do**
- 5          $r \xleftarrow{\$} (\mathbb{F}_2)^w$
- 6          $z_i \leftarrow z_i \oplus r$
- 7          $z_j \leftarrow z_j \oplus r$
- 8 **return**  $(z_i)_{1 \leq i \leq n}$

---

the code and therefore does not need to be generated. To generate a  $w$ -bit random, all we have to do is generate a random value, then perform an **and** operation between this random and the mask to reduce it to  $w$  bits. On lines 6 and 7, a simple **xor** operation is performed. So, in line 11 of our performance function, we define the CCE cost of the RefreshS1S1\_BDFGSZ16 gadget as the sum of all these operations, multiplied by the number of times they are executed through the two nested loops.

#### A.4 Example of a more complex performance estimation function

Some gadgets require more complex performance estimation functions. This is particularly true of the SecA2BModp\_SPOG19 gadget. To build the performance estimation function of this gadget, we also based it on the algorithm. Algorithm 3, which implements this gadget is recursive. In order to best calculate its performance cost, we decided to build a recursive estimation function described in Listing 4.

In addition to this particularity, we can also see that this function is called each time with masking parameters different from those given as input to the initial call. The input values always have the same share size, but the recursion takes place with a number of shares  $n$  divided by 2, in lines 3 and 5 of the algorithm. As explained above in Section A.1, we had to take this change in parameters into account in the function. In lines 10 to 13 of the performance function, which represent line 3 of the algorithm, we can see that we choose to define another set of masking parameters, setting the number of shares to  $\lfloor n/2 \rfloor$ , before recovering the cost of the recursion function on these new parameters.

```

1 def RefreshS1S1_BBDFGSZ16_cost(mask_params):
2     # Retrieval of the number of shares of masked values in gadget input
3     n = mask_params.n
4
5     # Retrieving the cost of all types of operations performed in our code
6     rand_generation = atomic_operations.get("rand_generation").get("cost")
7     xor_op = atomic_operations.get("xor_op").get("cost")
8     and_op = atomic_operations.get("and_op").get("cost")
9
10    # Randomization of each part of the masked variable done in two nested for
11    loops
12    cost = ((n*(n-1))/2)*(rand_generation+and_op+(2*xor_op))
13    return cost

```

Listing 3: Performance function of the gadget RefreshS1S1\_BBDFGSZ16

**Algorithm 3:** Secure A2B conversion [SPOG19]

---

**Data:**  $\mathbf{A} = (A_i)_{1 \leq i \leq n} \in \mathbb{F}_p$  such that  $\sum_i A_i = x \bmod p \in \mathbb{F}_p$   
**Result:**  $\mathbf{x} = (x_i)_{1 \leq i \leq n} \in (\mathbb{F}_2)^w$  with  $2^w > 2p$  such that  $\bigoplus_i x_i = x$

- 1 if  $n = 1$  then
- 2 |  $x_1 \leftarrow A_1$
- 3  $(y_i)_{1 \leq i \leq \lfloor n/2 \rfloor} \leftarrow \text{SecA2BModp}((A_i)_{1 \leq i \leq \lfloor n/2 \rfloor})$
- 4  $(y_i)_{1 \leq i \leq n} \leftarrow \text{Refresh}((y_1, \dots, y_{\lfloor n/2 \rfloor}, 0, \dots, 0), w)$
- 5  $(z_i)_{1 \leq i \leq \lfloor n/2 \rfloor} \leftarrow \text{SecA2BModp}((A_i)_{\lfloor n/2 \rfloor + 1 \leq i \leq n})$
- 6  $(z_i)_{1 \leq i \leq n} \leftarrow \text{Refresh}((z_1, \dots, z_{\lfloor n/2 \rfloor}, 0, \dots, 0), w)$
- 7  $\mathbf{x} \leftarrow \text{SecAddModp}(\mathbf{y}, \mathbf{z})$
- 8 return  $\mathbf{x}$

---

## A.5 Example of performance estimation function of a table-based gadget

The final special case for gadget performance estimation functions is the additional cost of data stored in memory rather than in register. The SecA2B1bitS2S1\_CoronGMZ22\_12 gadget from [CGMZ22] is a register-based gadget. Its optimization is based on the use of the `ror` assembly instruction, which allows register rotation to the left. It is presented in Algorithm 4. For this gadget, we based the performance estimation function on the implementation given by [CGMZ22]. In Listing 5, we shown the implementation for the case where  $l = w = 4$  and in Listing 6, we present the performance estimation function associated to this gadget.

The algorithm shows that the data to be stored in registers are the arrays  $(R_i)_{1 \leq i \leq n}$ . In the gadget implementation, these registers are represented by the array  $T$ , defined on line 2. To explain the addition of `load` and `store` costs, we can take as an example lines 12 to 16 of the implementation of the gadget, which are represented by lines 20 to 22 in the estimation function. Line 20 of the function represents the generation of  $n - 1$  random values. Line 22 represents the cost of the two `xor` operations performed on the registers. To perform these operations, the  $T[j]$  and  $T[\text{MASKING\_ORDER}]$  values must first be loaded from the memory. After applying the `xor` operations, the result must be store in memory. So this gives us the line 22 of the function, which is multiplied by  $n - 1$  because of the loop.

Note that in the implementation of this gadget, the authors preferred to use the calculation made on line 10, rather than the `ror` operation, as on line 5 of the algorithm. For the function, we decided to take into account the operations performed in the authors' implementation, but it would have been possible to consider that only `ror` operations

```

1 def SecA2BModpS2S1_SPOG19_cost(mask_params):
2     n = mask_params.n
3     if(n == 1):
4         return 0
5     else:
6         rightshift_op = atomic_operations.get("rightshift_op").get("cost")
7         Refresh_cost = gadgets_dict.get_cost(mask_params, "Refresh",
8         sharing_type.Boolean, sharing_type.Boolean, name = None)
9         SecAddModp_cost = gadgets_dict.get_cost(mask_params, "SecAddModp",
10        sharing_type.Boolean, sharing_type.Boolean, name = None)
11        cost = rightshift_op
12        mask_params_copy = deepcopy(mask_params)
13        half_order = floor(n/2)
14        mask_params_copy.n = half_order
15        cost += SecA2BModpS2S1_SchneiderPOG19_cost(mask_params_copy)
16        cost += Refresh_cost
17        mask_params_copy2 = deepcopy(mask_params)
18        mask_params_copy2.n = n-half_order
19        cost += SecA2BModpS2S1_SchneiderPOG19_cost(mask_params_copy2)
20        cost += Refresh_cost
21        cost += SecAddModp_cost
22    return cost

```

Listing 4: Performance function of the gadget SecA2BModp\_SPOG19

**Algorithm 4:** Secure A2B-1 bit conversion [CGMZ22]**Data:**  $x_1, \dots, x_n \in \mathbb{Z}_{2^w}$ **Result:**  $y_1, \dots, y_n \in \{0, 1\}$  such that  $y_1 \oplus \dots \oplus y_n = f(x_1 + \dots + x_n \bmod 2^w)$ 

```

1 forall  $u \in \mathbb{Z}_{2^w}$  do  $R_1[u] \leftarrow f(u)$ 
2 forall  $2 \leq j \leq n$  do  $R_j \leftarrow 0$ 
3 for  $i = 1$  to  $n - 1$  do
4     for  $j = 1$  to  $n$  do
5          $R_j \leftarrow \text{ROR}[x_i](R_j)$ 
6     for  $j = 1$  to  $n - 1$  do
7          $r \leftarrow \{0, 1\}^{2^w}$ ,  $R_j \leftarrow R_j \oplus r$ ,  $R_n \leftarrow R_n \oplus r$ 
8  $(y_1, \dots, y_n) \leftarrow \text{Refresh}_{\{0,1\}}(R_1[x_n], \dots, R_n[x_n])$ 
9 return  $y_1, \dots, y_n$ 

```

```

1 void SecA2B1bitS2S1_CGMZ22_12_1_4(Masked* x, Masked* b, unsigned l){
2     uint16_t T[MASKING_ORDER+1];
3     uint16_t r;
4     T[0] = 0x0FF0;
5     for(int i=1; i < MASKING_ORDER+1; ++i){
6         T[i] = 0;
7     }
8     for(int i=0; i < MASKING_ORDER; ++i){
9         for(int j=0; j < MASKING_ORDER+1; ++j){
10            T[j] = (T[j] << (16-x->shares[i])) + (T[j]>>(x->shares[i]));
11        }
12        for(int j=0; j < MASKING_ORDER; ++j){
13            r = rand16();
14            T[j] ^= r;
15            T[MASKING_ORDER] ^= r;
16        }
17    }
18    for(int i=0; i < MASKING_ORDER+1; ++i) b->shares[i] = (T[i]>>(x->shares[
19        MASKING_ORDER]))&1;
20    for(int j=0; j < MASKING_ORDER; ++j){
21        r = rand16();
22        b->shares[j] ^= r;
23        b->shares[MASKING_ORDER] ^= r;
24    }
}

```

Listing 5: Implementation of the gadget SecA2B1bitS2S1\_CGMZ22\_12 for  $l = 4$  [CGMZ22]

```

1 def SecA2B1bitS2S1_CoronGMZ22_12_cost(mask_params):
2     n = mask_params.n
3     rand_generation = atomic_operations.get("rand_generation").get("cost")
4     load_op = atomic_operations.get("load_op").get("cost")
5     store_op = atomic_operations.get("store_op").get("cost")
6     leftshift_op = atomic_operations.get("leftshift_op").get("cost")
7     rightshift_op = atomic_operations.get("rightshift_op").get("cost")
8     xor_op = atomic_operations.get("xor_op").get("cost")
9     add_op = atomic_operations.get("add_op").get("cost")
10    and_op = atomic_operations.get("and_op").get("cost")
11    sub_op = atomic_operations.get("sub_op").get("cost")
12
13    # Generation table
14    cost = store_op
15    cost += (n-1)*store_op
16    for i in range(n-1):
17        # Cyclic rotation of the registers to the right
18        cost += n*(load_op+sub_op+leftshift_op+rightshift_op+add_op+store_op)
19        # Random generation
20        cost += (n-1)*rand_generation
21        # Randomization of all the shares
22        cost += (n-1)*(2*(load_op+xor_op+store_op))
23    cost += n*(load_op+rightshift_op+and_op)
24    # Final Refresh
25    cost += (n-1)*(2*(xor_op+store_op)+rand_generation)
26    return cost

```

Listing 6: Performance function of the gadget SecA2B1bitS2S1\_CGMZ22\_12



had been performed. To show that taking these additional costs into account makes it possible to see the impact of storing in memory the data of a gadget based on instructions optimized for register (see Figure 9).

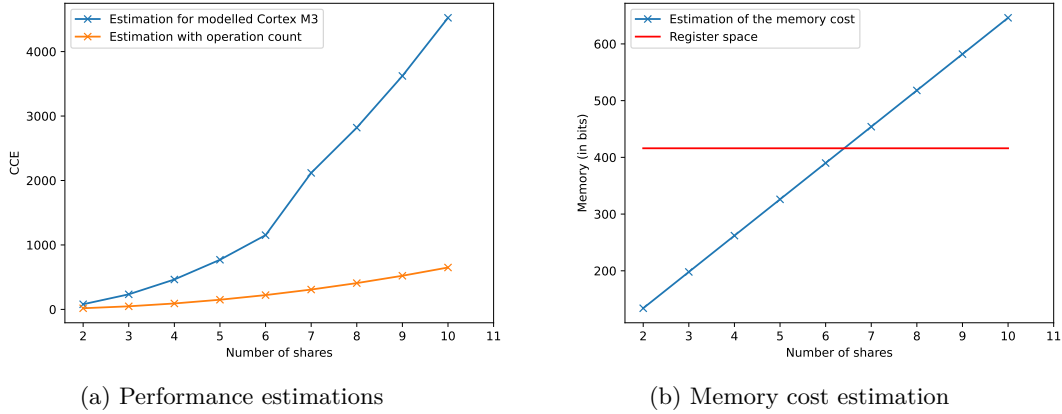


Figure 9: Impact of memory storage for data whose calculations are optimized in registers

Figure 9a shows the evolution of the CCE cost of the `SecA2B1bit_CoronGMZ22_12` gadget, with an input of 6 bits, in relation to the number of shares of masked values, according to the model chosen for the calculations. The blue curve represents the estimates obtained by running our calculations on a model of a Cortex M3 processor. The yellow curve represents the estimates obtained by counting the number of operations performed in the gadget, *i.e.* considering that all operations are worth 1 CPU cycle and that register and memory space is infinite. Figure 9b shows the estimated memory cost of this gadget, with the register space available for a Cortex M3 processor in red. It can be seen in this figure that, starting from a masking with  $n = 7$  shares, the gadget requires too much data to be stored in registers. Then, from  $n = 7$ , all data is stored in memory. This data storage in memory does have an impact on the gadget performance. When the calculations are performed on a model of a Cortex M3 processor, we can see that when we go to  $n = 7$  shares, the gadget CCE cost increases much more than before.

