SYSTEMATIC
PARIS REGION SYSTEMS & ICT CLUSTER

VERISICC
Deliverable L2.3
Generation and Optimization of Secure Code

CALL: FUI25

NAME OF THIS PROJECT: VERISICC

Leader of this deliverable

- Partner: Inria
- Contact name: Benjamin Grégoire
- Contact information: benjamin.gregoire@inria.fr

Leader of the project

- Company: CryptoExperts
- Contact name: Sonia Belaïd
- Contact information: sonia.belaid@cryptoexperts.com

Partners

- SMEs: CryptoExperts and NinjaLab
- Big Business: IDEMIA
- Public Institutions: INRIA, ANSSI, and Université du Luxembourg

Version : 1.0

Sommaire

1. Introduction

1.1. Context

A masking compiler transforms an unmasked circuit into a masked one. The main idea is that the compiler replaces basic operations, like addition and multiplication, by more sophisticate pieces of code (the core gadgets) operating on shares. Unfortunately the composition of secure core gadgets does not necessarily leads to a secure masked algorithm. Therefore, some extra checks are necessary.

The general idea is to do this within two stages. In the first stage, the compiler checks that the composition of core gadgets leads to a secure masked algorithm. This can be done using various techniques (e.g., typing for MaskComp, linear algebra for TORNADO). This step assumes that the core gadgets satisfy some properties (linear, NI, SNI). If the composition is not secure, then compiler can modify the base gadgets or insert some *refresh* gadgets to re-randomize the shares used between base gadgets. Once the first step is successfully completed, the compiler generates the code. This requires efficient implementations of the core gadgets satisfying the assumed security property.

1.2. Publications of the consortium

The consortium (co-)published two papers on the compiler tightPROVE/Tornado:

- **[BDMRW20]**: *Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations.* Sonia Belaïd, Pierre-Evariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Eurocrypt 2020
- **[BGR18]**: *Tight Private Circuits: Achieving Probing Security with the Least Refreshing.* Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Asiacrypt 2018,

two papers on the expanding compiler in the random probing model:

- **[BRT21]**: *On the Power of Expansion: More Efficient Constructions in the Random Probing Model.* Sonia Belaïd, Matthieu Rivain, and Abdul Rahman Taleb. Eurocrypt 2021
- **[BCPRT20]**: *Random Probing Security: Verification, Composition, Expansion and New Constructions.* Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Crypto 2020

and the following papers on the core gadgtes:

- **[CGLS21]**: *Hardware Private Circuits: From Trivial Composition to Full Verification*, Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, François-Xavier Standaert, IEEE Trans. Computers 70(10): 1677-1690 (2021)
- **[BGGOPP]**: *Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification*, Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, Lars Porth, IACR Trans. Cryptogr. Hardw. Embed. Syst. 2021(2): 189-228 (2021).
- **[BBDFGSS]**: *Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations*, Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert, Pierre-Yves Strub. J. Cryptogr. Eng. 10(1): 17-26 (2020).

1.3. Organization

We organize the deliverable according to two main contributions. In Section 2, we describe the Tornado compiler and we provide implementations of non-linear core gadgets (Refresh and Multiplication/And) in the probing model with the least randomness. In Section 3, we describe the expanding compiler and its base gadgets to build secure implementations in the random probing model.

2. Tornado in the Probing Model

2.1. Description of the Compiler

The method of Barthe et al. [BBD+16] allows one to safely compose t-NI and t-SNI gadgets and get probing security at any order. Nevertheless, it is not tight and makes use of more refresh gadgets than required. In many contexts, randomness generation is expensive and might be the bottleneck for masked implementations. For instance, Journault and Standaert describe an AES encryption shared at the order $d = 32$ for which up to 92% of the running time is spent on randomness generation [JS17]. In such a context, it is fundamental to figure out whether the number of t-SNI refresh gadgets inserted by Barthe et al.'s tool `maskComp` is actually minimal to achieve t-probing security. Belaïd, Goudarzi, and Rivain find out that it is not and provide a new method which *exactly* identifies the concrete probing attacks in a Boolean shared circuit [BGR18].

Let us take a simple example. We consider the small randomized circuit referred to as Circuit 1 and illustrated in Figure 1 with $[\oplus]$ a t-NI sharewise addition, $[\otimes]$ a t-SNI multiplication, and two Boolean sharings $[x_1]$ and $[x_2]$. Applying Barthe et al.'s tool `maskComp` on this circuit automatically inserts a t-SNI refresh gadget in the cycle formed by gates $[x_1]$, $[\oplus]$, and $[\otimes]$ as represented in Figure 2. However, it can be verified that for any masking order t, the initial circuit is t-probing secure without any additional refresh gadget. Therefore, the work of Belaïd, Goudarzi, and Rivain aims to refine the state-of-the-art method [BBD+16] to only insert refresh gadgets when absolutely mandatory for the t-probing security.

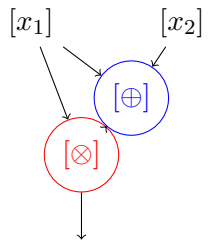

Figure 1: Graph representation of Circuit 1.

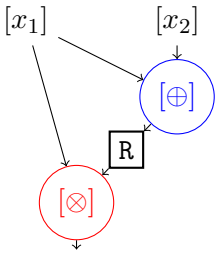

Figure 2: Graph representation of Circuit 1 after `maskComp`.

2.2. TIGHTPROVE

In a first attempt, Belaïd et al.'s built a tool referred to as `tightPROVE`. `tightPROVE` can be seen as a compiler for probing secure masked circuits. It takes as input a list of instructions that describes a shared circuit made of specific multiplication, addition and refresh *gadgets* and outputs either a probing security proof of these instructions –if implemented with carefully chosen masked gadgets–

or a probing attack. To that end, a security reduction is made through a sequence of four equivalent games. In each of them, an adversary $\mathcal{A}$ chooses a set of probes $\mathcal{P}$ (indices pointing to wires in the shared circuit) in the target circuit C, and a simulator $\mathcal{S}$ wins the game if it successfully simulates the distribution of the tuple of variables carried by the corresponding wires without knowledge of the secret inputs.

Game 0 corresponds to the t-probing security definition: the adversary can choose t probes in a $t+1$-shared circuit, on whichever wires she wishes. In Game 1, the adversary is restricted to only probe gadget inputs: one probe on an addition or refresh gadget becomes one probe on one input share, one probe on a multiplication gadget becomes one probe on each of the input sharings. In Game 2, the circuit C is replaced by another circuit C' that has a multiplicative depth of one, through a transformation called *Flatten*, illustrated in [BGR18]. In a nutshell, each output of a multiplication or refresh gadget in the original circuit gives rise to a new input with a fresh sharing in C'. Finally, in Game 3, the adversary is only allowed to probe pairs of inputs of multiplication gadgets. The transition between these games is mainly made possible by an important property of the selected refresh and multiplication gadgets: in addition to being t-probing secure, they are t-*strong non interfering* (t-SNI for short) [BBD+16]. Satisfying the latter means that t probed variables in their circuit description can be simulated with less than t_1 shares of each input, where $t_1 \leq t$ denotes the number of internal probes i.e., which are not placed on output shares.

Game 3 can be interpreted as a linear algebra problem. In the flattened circuit, the inputs of multiplication gadgets are linear combinations of the circuit inputs. These can be modelled as Boolean vectors referred to as *operand vectors*, with ones at indexes of involved inputs. From the definition of Game 3, the $2t$ probes made by the adversary all target these operand vectors for chosen shares. These probes can be distributed into $t+1$ matrices $M_0, \ldots, M_t$, where $t+1$ corresponds to the (tight) number of shares, such that for each probe targeting the share i of an operand vector $\mathbf{v}$, with i in $\{0, \ldots, t\}$, $\mathbf{v}$ is added as a row to matrix M_i. Deciding whether a circuit is t-probing secure can then be reduced to verifying whether $IM(M_0^T) \cap \cdots \cap IM(M_t^T) = \emptyset$ (where $IM(\cdot)$ denotes the column space of a matrix). The latter can be solved algorithmically with the following high-level algorithm for a circuit with m multiplications:

For each operand vector $\mathbf{w}$,

1. Create a set $\mathcal{G}_1$ with all the multiplications for which $\mathbf{w}$ is one of the operand vectors.
2. Create a set $\mathcal{O}_1$ with the co-operand vectors of $\mathbf{w}$ in the multiplications in $\mathcal{G}_1$.
3. Stop if $\mathbf{w} \in \langle \mathcal{O}_1 \rangle$ ($\mathcal{O}_1$'s linear span), that is if $\mathbf{w}$ can be written as a linear combination of Boolean vectors from $\mathcal{O}_1$.
4. For i from 2 to m, create new sets $\mathcal{G}_i$ and $\mathcal{O}_i$ by adding to $\mathcal{G}_{i-1}$ multiplications that involve an operand $\mathbf{w}'$ verifying $\mathbf{w}' \in (\mathbf{w} \oplus \langle \mathcal{O}_{i-1} \rangle)$, and adding to $\mathcal{O}_{i-1}$ the other operand vectors of these multiplications. Stop whenever $i = m$ or $\mathcal{G}_i = \mathcal{G}_{i-1}$ or $\mathbf{w} \in \langle \mathcal{O}_i \rangle$.

If this algorithm stops when $\mathbf{w} \in \langle \mathcal{O}_i \rangle$ for some i, then there is a probing attack on $\mathbf{w}$, i.e., for a certain t, the attacker can recover information on $\mathbf{x} \cdot \mathbf{w}$ (where $\mathbf{x}$ denote the vector of plain inputs), with only t probes on the $(t+1)$-shared circuit. In the other two scenarios, the circuit is proven to be t-probing secure for any value of t.

As an illustration, `tightPROVE` was applied to the efficient implementation of the AES s-box developed by Goudarzi and Rivain in [GR17]. Based on the previous state of the art, this s-box was

implemented using one SNI refresh gadget per multiplication gadget (to refresh one of the operands), hence requiring a total of 32 refresh gadgets (which was later on confirmed by the `maskComp` tool). The new method formally demonstrates that the same d-shared implementation is actually t-probing secure with *no* refresh gadget for any $d = t + 1$. This new implementation achieves an asymptotic gain up to 43%. The code is provided on the website

https://github.com/CryptoExperts/tightPROVE.

These results are extended to larger circuits by establishing new compositional properties on t-probing secure gadgets. In particular, these new composition properties well apply to the case of SPN-based block ciphers. They also apply to a wide range of Boolean circuits with common gadgets and input sets.

2.3. Tornado

Although nicely answering a relevant open issue, TIGHTPROVE still suffers two important limitations. First it only applies to Boolean circuits and does not straightforwardly generalize to software implementation processing ℓ-bit registers (for $\ell > 1$). Secondly, it does not provide a method to place the refresh whenever a probing attack is detected.

A parallel sequence of works has focused on the efficient implementation of masking schemes with possibly high orders. For software implementations, it was recently demonstrated in several works that the use of bitslicing makes it possible to achieve (very) aggressive performances. In the bitsliced higher-order masking paradigm, the ISW scheme is applied to secure bitwise `and` instructions which are significantly more efficient than their field-multiplication counterparts involved in the so-called polynomial schemes [GR17, JS17]. Moreover, the bitslice strategy allows to compute several instances of a cryptographic primitive in parallel, or alternatively all the s-boxes in parallel within an instance of the primitive. The former setting is simply called (full) bitslice in the deliverable while the latter setting is referred to as n*-slice*. In both settings, the high degree of parallelization inherited from the bitslice approach results in important efficiency gains. Verifying the probing security of full bitslice masked implementation is possible with TIGHTPROVE since the different bit slots (corresponding to different instances of the cryptographic primitive) are mutually independent. Therefore, probing an ℓ-bit register in the bitslice implementation is equivalent to probing the corresponding variable in ℓ independent Boolean circuits, and hence TIGHTPROVE straightforwardly applies. For n-slice implementations on the other hand, the different bit slots are mixed together at some point in the implementation which makes the verification beyond the scope of TIGHTPROVE. In practice, for masked software implementations, the *register probing model* makes much more sense than the *bit probing model* because a software implementation works on ℓ-bit registers containing several bits that leak all together.

Another limitation of TIGHTPROVE is that it simply verifies an implementation under the form of an abstract circuit but it does not output a secure implementation, nor provide a sound placing of refresh gadgets to make the implementation secure. In practice one could hope for an integrated tool that takes an input circuit in a simple syntax, determine where to place the refresh gadgets and compile the augmented circuit into a masked implementation, for a given masking order on a given computing platform. USUBA, introduced by Mercadier and Dagand in [MD19], is a high-level programming language for specifying symmetric block ciphers. It provides an optimizing compiler that produces efficient bitsliced implementations. On high-end Intel platforms, USUBA has demonstrated performance on par with several, publicly available cipher implementations. As part of its compilation

pipeline, USUBA features an intermediate representation, USUBAZ that shares many commonalities with the input language of TIGHTPROVE.

It was therefore natural to consider integrating both tools in a single programming environment. Belaïd, Dagand, Mercadier, Rivain, and Wintersdorff therefore aim at enabling cryptographers to prototype their algorithms in USUBA, letting TIGHTPROVE verify or repair its security and letting the USUBA back-end perform masked code generation [BDM+20]. Concretely, they provide the following contributions:

(1.) The authors tackle the limitations of TIGHTPROVE and propose an extended verification tool, that they call TIGHTPROVE+. This tool can verify the security of any masked bitslice implementation in the register probing model (which makes more sense than the bit probing model w.r.t. masked software implementations). Given a masked bitslice/n-slice implementation composed of standard gadgets for bitwise operations, TIGHTPROVE+ either produces a probing-security proof or exhibits a probing attack.

(2.) They present (and report on the development of) a new compiler TORNADO[1] which integrates USUBA and TIGHTPROVE+ in a global compiler producing masked bitsliced implementations proven secure in the bit/register probing model. This compiler takes as input a high-level, functional specification of a cryptographic primitive. If some probing attacks are detected by TIGHTPROVE+, the TORNADO compiler introduces refresh gadgets, following a sound heuristic, in order to thwart these attacks. Once a circuit has been identified as secure, TORNADO produces bitsliced C code achieving register probing security at a given input order. To account for the limited resources available on embedded systems, TORNADO exploits a generalization of bitslicing – implemented by USUBA – to reduce register pressure and implements several optimizations specifically tailored for Boolean masking code. The source code of TORNADO is available at:

https://github.com/CryptoExperts/Tornado

(3.) They evaluate TORNADO on 11 cryptographic primitives from the second round of the ongoing NIST lightweight cryptography standardization process.[2] The choice of cryptographic primitives has been made on the basis that they were self-identified as being amenable to masking. These implementation results give a benchmark of these different candidates with respect to masked software implementation for a number of shares ranging between 1 and 128. The obtained performances are pretty satisfying. For instance, the n-slice implementations of the tested primitives masked with 128 shares takes from 1 to a few dozen megacycles on an Cortex-M4 processor.

All the technical details can be found in the full version of the published paper [BDM+20]:

https://eprint.iacr.org/2020/506.pdf.

2.4. Implementation for core gadgets

In this section we provide an algorithmic description of important gadgets that need to be provided to be able to define a masking compiler. Those gadgets have been verified with `maskVerif` tool.

[1] TORNADO ambitions to be the work*horse* of those cryptographers that selflessly protect their ciphers through provably secure *mask*ing and precise bit*slicing*.

[2] https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates

2.4.1. Refresh

To reduce randomness utilization, we provide a full set of new optimized SNI refreshing gadgets for most practically-relevant orders (i.e. $d \in \{2, \ldots, 16\}$), which require less randomness. All gadgets were proven robust-SNI with the maskVerif tool. The randomness cost of the new refresh gadgets compares favorably to the state-of-the-art for both hardware and software implementations regarding randomness complexity (see Table 2). In particular, randomness gains of more than 30% are obtained for $d \in \{3, 4, 5, 7, 8\}$.

Table 1: Best known SNI refresh gadgets at some orders for both hardware (glitch-robust) and software implementations.

d	HW [**?**]	SW [**?**, **?**]	New refresh
2	1	1	1
3	3	3	2
4	6	4	4
5	10	8	5
6	15	12	7
7	21	13	9
8	28	16	11
9	36	18	13
10	45	20	15
11	55	22	17
12	66	24	20
13	78	26	26
14	91	28	28
15	105	30	30
16	120	32	32

The refresh algorithms are described using the maskVerif syntax. The + operator is the addition of the group (i.e. XOR for boolean masking or modular addition for arithmetic masking). Remark that maskVerif provides some short cut to describe this kind of operations:

- if `r` is a vector of shares (for example `r[0], r[1], r[2]`) then `r>>1` represents the rotation by 1 of the share (`r[2], r[0], r[1]`)
- if `r` and `R` are vectors of shares then `r + R` represents the vector composed by the sum of each share (`r[0] + R[0], r[1] + R[1], r[2] + R[2]`)

Order 1 ($d = 2$)

```
proc Refresh:
 inputs: a[0:1]
 outputs: c[0:1]
 randoms: t0;

 c[0] =![a[0]+t0];
 c[1] =![a[1]+t0];
end
```

Order 2 ($d = 3$)

```
proc Refresh:
 inputs: a[0:2]
 outputs: c[0:2]
 randoms: t0, t1;

 c[0] =![a[0]+t0];
 c[1] =![a[1]+t1];
 t2 =![t0+t1];
 c[2] =![a[2]+t2];
end
```

Order 3 ($d = 4$)

```
proc Refresh:
 inputs: a[0:3]
 outputs: c[0:3]
 shares: R[0:3]
 randoms: r[0:3];

 R =![r + (r>>1)];
 c =![a+R];
end
```

Order 4 ($d = 5$)

```
proc Refresh:
 inputs: a[0:4]
 outputs: c[0:4]
 shares: R[0:4]
 randoms: r[0:4];

 R =![r + (r>>1)];
 c =![a+R];
end
```

Order 5 ($d = 6$)

```
proc Refresh:
 inputs: a[0:5]
 outputs: c[0:5]
 shares: R[0:5]
 randoms: r[0:5], s0;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
 R[3] := R[3] + s0;
 c =![R];
```

```
 c =![a+c];
end
```

Order 6 ($d = 7$)

```
proc Refresh:
 inputs: a[0:6]
 outputs: c[0:6]
 shares: R[0:6]
 randoms: r[0:6], s0, s1;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
 R[2] := R[2] + s1;
 R[4] := R[4] + s0;
 R[6] := R[6] + s1;
 c =![R];
 c =![a+c];
end
```

Order 7 ($d = 8$)

```
proc Refresh:
 inputs: a[0:7]
 outputs: c[0:7]
 shares: R[0:7]
 randoms: r[0:7], s0, s1, s2;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
 R[1] := R[1] + s1;
 R[2] := R[2] + s2;
 R[4] := R[4] + s0;
 R[5] := R[5] + s1;
 R[6] := R[6] + s2;
 c =![R];
 c =![a+c];
end
```

Order 8 ($d = 9$)

```
proc Refresh:
 inputs: a[0:8]
 outputs: c[0:8]
 shares: R[0:8]
 randoms: r[0:8], s0, s1, s2;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
```

```
 R[1] := R[1] + s1;
 R[3] := R[3] + s2;
 R[4] := R[4] + s0;
 R[6] := R[6] + s1;
 R[7] := R[7] + s2;
 c =![R];
 c =![a+c];
end
```

Order 9 ($d = 10$)

```
proc Refresh:
 inputs: a[0:9]
 outputs: c[0:9]
 shares: R[0:9]
 randoms: r[0:9], s0, s1, s2, s3, s4;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
 R[1] := R[1] + s1;
 R[2] := R[2] + s2;
 R[3] := R[3] + s3;
 R[4] := R[4] + s4;
 R[5] := R[5] + s0;
 R[6] := R[6] + s1;
 R[7] := R[7] + s2;
 R[8] := R[8] + s3;
 R[9] := R[9] + s4;
 c =![R];
 c =![a+c];
end
```

Order 10 ($d = 11$)

```
proc Refresh:
 inputs: a[0:10]
 outputs: c[0:10]
 shares: R[0:10]
 randoms: r[0:10], s0, s1, s2, s3, s4, s5;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
 R[1] := R[1] + s1;
 R[2] := R[2] + s2;
 R[3] := R[3] + s3;
 R[4] := R[4] + s4;
 R[5] := R[5] + s0;
 R[6] := R[6] + s1;
```

```
 R[7] := R[7] + s2 + s5;
 R[8] := R[8] + s3;
 R[9] := R[9] + s4;
 R[10] := R[10] + s5;
 c =![R];
 c =![a+c];
end
```

Order 11 ($d = 12$)

```
proc Refresh:
 inputs: a[0:11]
 outputs: c[0:11]
 shares: R[0:11]
 randoms: r[0:11], s0, s1, s2, s3, s4, s5, s6, s7;

 R =![r + (r>>1)];
 R[0] := R[0] + s0;
 R[1] := R[1] + s1;
 R[2] := R[2] + s2+s6;
 R[3] := R[3] + s3;
 R[4] := R[4] + s4;
 R[5] := R[5] + s5+s6;
 R[6] := R[6] + s0;
 R[7] := R[7] + s1;
 R[8] := R[8] + s2+s7;
 R[9] := R[9] + s3;
 R[10] := R[10] + s4;
 R[11] := R[11] + s5+s7;
 c =![R];
 c =![a+c];
end
```

Order 12 ($d = 13$)

```
proc Refresh:
 inputs: a[0:12]
 outputs: c[0:12]
 shares: R[0:12], S[0:12]
 randoms: r[0:12], s[0:12];

 R =![r + (r>>1)];
 S =![s + (s>>3)];
 c =![R+S];
 c =![a+c];
end
```

Order 13 ($d = 14$)

```
proc Refresh:
 inputs: a[0:13]
 outputs: c[0:13]
 shares: R[0:13], S[0:13]
 randoms: r[0:13], s[0:13];

 R =![r + (r>>1)];
 S =![s + (s>>3)];
 c =![R+S];
 c =![a+c];
end
```

Order 14 ($d = 15$)

```
proc Refresh:
 inputs: a[0:14]
 outputs: c[0:14]
 shares: R[0:14], S[0:14]
 randoms: r[0:14], s[0:14];

 R =![r + (r>>1)];
 S =![s + (s>>3)];
 c =![R+S];
 c =![a+c];
end
```

Order 15 ($d = 16$)

```
proc Refresh:
 inputs: a[0:15]
 outputs: c[0:15]
 shares: R[0:15], S[0:15]
 randoms: r[0:15], s[0:15];

 R =![r + (r>>1)];
 S =![s + (s>>3)];
 c =![R+S];
 c =![a+c];
end
```

2.4.2. Multiplication

For the multiplication we recommand to use the DOM-AND [] algorithms or the PARA-AND [] algorithms. The randomness needed for multiplication (AND) algorithm can vary a lot depending on the security property we want to achive. Here we focus only on algorithms achiving the SNI property because they are more relevant in the context of a masking compiler (this simplify the composition).

Table 2: Best known SNI multiplication gadgets.

d	DOM	PARA
2	1	2
3	3	3
4	6	5
5	10	10

DOM-AND The DOM-AND algorithm works at any orders (in software it is relatively equivalent to the ISW-multiplication: same need for randomness, same number of AND and XOR). We provide the pseudo code.

```
proc AND:
  inputs: a[0:d], b[0:d]
  ouputs: c[0:d]
  random: r[0:(d+1)d/2]
  c := a * b;
  for i = 0 to d do
   for j = i + 1 to d do
     aux := a[i] * b[j];
     aux := ![aux + r[i*d + j-i-1]];
     c[i] := c[i] + aux;
     aux := a[j] * b[i];
     aux := ![aux + r[i*d + j-i-1]];
     c[j] := c[j] + aux;
   done
  done
```

PARA-AND, order 1 ($d = 2$)

```
proc AND:
  inputs: a[0:1], b[0:1]
  outputs: c[0:1]
  shares: ab[0:1]
  randoms: r[0:1];

  ab := a * b;
  c   = ![ab + r];
  ab := a * (b>>1);
  c  := c + ab;
  c  = ![c + (r>>1)];
end
```

PARA-AND, order 2 ($d = 3$)

```
proc AND:
  inputs: a[0:2], b[0:2]
  outputs: c[0:2]
```

```
  shares: ab[0:2]
  randoms: r[0:2];

  ab := a * b;
  c   = ![ab + r];
  ab := a * (b>>1);
  c   = ![c + ab];
  ab := (a>>1) * b;
  c   = ![c + ab];
  c   = ![c + (r>>1)];
end
```

PARA-AND, order 3 ($d = 4$)

```
proc AND:
  inputs: a[0:3], b[0:3]
  outputs: c[0:3]
  shares: ab[0:3], ba[0:3], sa[0:3], sb[0:3], sr[0:3]
  randoms: r[0:3],r';

  ab := a * b;
  c   = ![ab + r];
  sa := a >> 1;
  sb := b >> 1;
  ab := sa * b;
  c   = ![c + ab];
  ba := a * sb;
  c   = ![c + ba];
  sr := r >> 1;
  c   = ![c + sr];
  sb := b >> 2;
  ab := a * sb;
  c   = ![c + ab];
  c   = ![c + [r',r',r',r']];
end
```

PARA-AND, order 4 ($d = 5$)

```
proc AND:
  inputs: a[0:4], b[0:4]
  outputs: c[0:4]
  shares: ab[0:4]
  randoms: r[0:4], r'[0:4];

  ab := a * b;
  c   = ![ab + r];
  ab := a * (b>>1);
  c   = ![c + ab];
```

```
  ab := (a>>1) * b;
  c   = ![c + ab];
  c   = ![c + (r>>1)];
  ab := a * (b >> 2);
  c  = ![c + ab];
  ab := (a>>2) * b;
  c  := c + ab;
  c   = ![c + r'];
  c   = c+(r'>>1);
end
```

2.4.3. Boolean/Arithmetic conversion

In this section we consider the formal verification of the high-order Boolean to arithmetic conversion algorithm recently described at CHES 2017 [Cor17], with a t-SNI security proof for $n \geq t+1$. The algorithm can be seen as a generalization of Goubin's algorithm [Gou01] to any order, still with a complexity independent of the register size k. Although the algorithm has complexity $\mathcal{O}(2^n)$, instead of $\mathcal{O}(n^2 \cdot k)$ in [CGV14], for small values of n it is an order of magnitude more efficient. The algorithm takes as input n Boolean shares x_i such that

$$x = x_1 \oplus \cdots \oplus x_n$$

and using a recursive algorithm computes n arithmetic shares D_i such that

$$x = D_1 + \cdots + D_n \pmod{2^k}$$

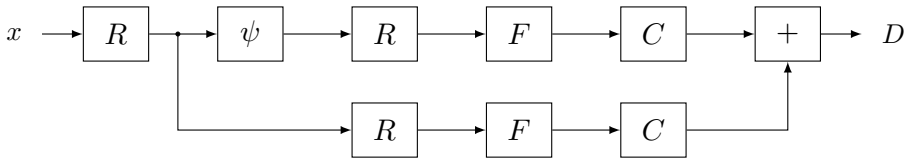

Figure 3: Sequence of operations in the Boolean to arithmetic conversion algorithm from [Cor17].

2.4.4. Boolean to arithmetic conversion.

The algorithm from [Cor17] is based on the affine property of the function $\Psi(x, r) := (x \oplus r) - r \pmod{2^k}$. As illustrated in Fig. 3 the algorithm is recursive and makes two recursive calls to the same algorithm C with $n-1$ inputs. For $n = 2$ one uses a t-SNI variant of Goubin's algorithm:

$$D_1 = \big((x_1 \oplus r_1) \oplus \Psi(x_1 \oplus r_1, r_2 \oplus (x_2 \oplus r_1))\big) \oplus \Psi(x_1 \oplus r_1, r_2) \quad (1)$$

$$D_2 = x_2 \oplus r_1 \quad (2)$$

For $n \geq 3$ the algorithm works as follows. One first performs a mask refreshing R, while expanding the x_i's to $n+1$ shares. One obtains, from the definition of the Ψ function:

$$\begin{aligned} x &= x_1 \oplus x_2 \oplus \cdots \oplus x_{n+1} \\ &= (x_1 \oplus \cdots \oplus x_{n+1} - x_2 \oplus \cdots \oplus x_{n+1}) + x_2 \oplus \cdots \oplus x_{n+1} \\ &= \Psi(x_1, x_2 \oplus \cdots \oplus x_{n+1}) + x_2 \oplus \cdots \oplus x_{n+1} \end{aligned}$$

From the affine property of the Ψ function, the left term can be decomposed into the xor of n shares $\Psi(x_1, x_i)$ for $2 \leq i \leq n+1$, where the first share is $(\overline{n \wedge 1}) \cdot x_1 \oplus \Psi(x_1, x_2)$:

$$x = (\overline{n \wedge 1}) \cdot x_1 \oplus \Psi(x_1, x_2) \oplus \Psi(x_1, x_3) \oplus \cdots \oplus \Psi(x_1, x_{n+1}) + x_2 \oplus \cdots \oplus x_{n+1}$$

We obtain that x is the arithmetic sum of two terms, each with n Boolean shares; this corresponds to the two branches in Fig. 3. One then performs a mask refreshing R on both branches, and then a compression function F that simply xors the last two shares, so there remains only $n-1$ shares on both branches. One can then apply the Boolean to arithmetic conversion C recursively on both branches, taking as input $n-1$ Boolean shares (instead of n), and outputting $n-1$ arithmetic shares; we obtain:

$$x = \big(A_1 + \cdots + A_{n-1}\big) + \big(B_1 + \cdots + B_{n-1}\big) \pmod{2^k}$$

Eventually it suffices to do some additive grouping to obtain n arithmetic shares as output, as required:

$$x = D_1 + \cdots + D_n \pmod{2^k}$$

We refer to [Cor17] for the details of the algorithm. The algorithm is proven t-SNI secure with $n \geq t+1$ shares in [Cor17].

Algorithm representation. In [Cor18] we have described a formal verification of the security properties of RefreshMasks that are required for the security proof of the above Boolean to arithmetic conversion algorithm in [Cor17]. However this provides only a *partial* verification of the algorithm, since in that case the adversary is restricted to only probing the Boolean operations performed within the RefreshMasks. To obtain a *full* verification, we must consider an adversary who can probe any variable in the Boolean to arithmetic algorithm. In that case the formal verification becomes more complex as we must handle both Boolean and arithmetic operations.

Since in [Cor18] the nested list representation already uses the $+$ operator for the xor, we use the ADD keyword to denote the arithmetic sum. For example, the final additive grouping can be represented as:

```
> (additive-grouping '(A1 A2) '(B1 B2))
((ADD A1 B1) A2 B2)
```

which corresponds to the three arithmetic shares $D_1 = A_1 + B_1 \pmod{2^k}$, $D_2 = A_2$ and $D_3 = B_2$. We also use the PSI operator to denote the application of the Ψ function. For example, the Boolean to arithmetic conversion algorithm for $n = 2$ gives from (1) and (2):

```
> (convba '(X1 X2))
((+ (+ (+ X1 R1) (PSI (+ X1 R1) (+ R2 (+ X2 R1))))
    (PSI (+ X1 R1) R2))
 (+ X2 R1))
```

Simplification rules. Given a list of intermediate variables that must be simulated, as previously we must use a set of simplification rules to determine how many inputs x_i are required for the simulation. For the verification of Boolean circuits in the previous section, this was relatively straightforward as we had essentially a single simplification rule, namely replacing $x \oplus r$ by r when the random r appears only once in the intermediate variables. However when combining arithmetic and Boolean operations the formal verification becomes more complex and we used the following simplification rules.

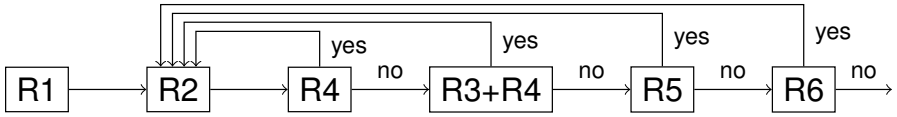

Figure 4: The rule application strategy for the formal verification of Boolean to arithmetic conversion.

- Rule 1: when $\omega = x_1 + x_2 \bmod 2^k$ must be simulated, simulate both x_1 and x_2.

```
> (prop-add '((ADD X1 X2)))
(X1 X2)
```

- Rule 2: from the affine property of the function Ψ, replace $\Psi(x, y) \oplus \Psi(x, z)$ by $x \oplus \Psi(x, y \oplus z)$.

```
> (replace-psi '(+ (PSI A B) (PSI A C)))
(+ A (PSI A (+ B C)))
```

- Rule 3: from the definition of Ψ, replace $\Psi(x, y)$ by $(x \oplus y) - y \bmod 2^k$; we denote by SUB the arithmetic subtraction.

```
> (replace-psi-sub '(PSI A B)
(SUB (+ A B) B)
```

- Rule 4: when a random r is used only once, replace $x \oplus r$ by r, and similarly for $x + r \bmod 2^k$ and $x - r \bmod 2^k$. This is an extension of the rule given by (**??**).

```
> (iter-simplify '((+ X1 R1) (ADD X2 R2) (SUB X3 R3)))
(R1 R2 R3)
```

- Rule 5: when a random r is not used in two intermediate variables e_1 and e_2, replace the simulation of $(e_1 \oplus r, e_2 \oplus r)$ by the simulation of $(r, (e_1 \oplus r) \oplus e_2)$; this corresponds to the change of variable $r' = e_1 \oplus r$.

```
> (simplify-x '((+ R1 X1) (+ R1 X2)))
(R1 (+ (+ R1 X1) X2))
```

- Rule 6: when $\Psi(x_1, x_2)$ must be simulated, simulate both x_1 and x_2.

```
> (prop-psi '((PSI A B)))
(A B)
```

We note that the order in which the rules are applied matters. For example, once Rule 3 has been applied, Rule 2 cannot be applied to the same expression, because the PSI operator has been replaced by SUB. One must therefore use the right strategy for the application of the rules; an overview is provided in Figure 4. In particular, we only apply Rule 3 if subsequently applying Rule 4 enables to eliminate the SUB operator, and Rule 6 is only applied as a last resort, when other rules have failed.

n	#variables	#tuples	Security	Time
2	11	11	✓	ε
3	48	1,128	✓	0.08 s
4	133	383,306	✓	85 s
5	312	387,278,970	✓	88 h

Table 3: Formal verification of the t-SNI property of the Boolean to arithmetic conversion algorithm from [Cor17].

Formal verification. In order to verify the t-SNI property of the Boolean to arithmetic algorithm, as previously we must check that for all possible $(n-1)$-tuples of intermediate variables (including the outputs D_i), the number of input variables x_i's that remain after the application of the above rules is always $\leq t_c$, where t_c is the number of non-output variables in the $(n-1)$-tuple.

We summarize in Table 3 the timings of formal verification for the algorithm in [Cor17]. Note that the Boolean to arithmetic conversion algorithm has complexity $\mathcal{O}(2^n)$, and therefore the number of possible $(n-1)$-tuples of intermediate variables is $\mathcal{O}(2^{n^2})$; that is why we could only perform the formal verification up to $n = 5$.

3. Expanding Compiler in the Random Probing Model

3.1. Description of the compiler

In [BCP+20], Belaïd et al. propose a method to compose different gadgets while achieving arbitrary levels of security in the random probing model. While the probing model considers a fixed number of wires to leak during an execution, the random probing model considers that each wire in the circuit leaks independently with fixed probability p. The advantage of the random probing model over the probing model is that the former is closer to the reality of physical leakage than the latter. In fact, the probing model has been challenged by the community, typically due to the fact that it does not capture horizontal side-channel attacks. A circuit is said to be secure in the so-called random probing model if there is a negligible probability ε that the leaking wires actually reveal any information about the secrets. We say that the circuit is (p, ε)-*random probing secure*. An equivalent simulation based definition is that there exists a simulator $\mathcal{S}$ which can successfully simulate the distribution of the tuple of variables carried by the corresponding leaking wires without knowledge of the secret inputs, and which fails to simulate them with probability at most ε.

When dealing with arbitrarily large circuits, it becomes difficult to provide their n-share random probing secure variants by considering the whole implementation at once. Instead, one starts with building block circuits called gadgets, and which are to be proved random probing secure. Then, providing a random probing secure version of any large circuit will amount to replacing each operation in the circuit by the corresponding n-share gadget and connecting the output shares of a gadget to the input shares of the next gadget in the circuit. This gives rise to the need of a composition notion in the random probing model. In [BCP+20], Belaïd et al. define the (t, p, ε)-*random probing composability* property, which is reminiscent of the t-SNI property. An n-share gadget is said to be (t, p, ε)-*random probing composable* for some $t \leq n-1$, if given any set of t output shares of the gadget and a set of internal observations constructed by adding each internal wire of the gadget to the set independently with probability p, then both sets can be perfectly simulated using at most

t shares of each of the gadget's input sharings, with a failure probability equal to ε. While the t-SNI property defines a fixed size for the set of output shares and the set of internal observations, the random probing composability defines the set of internal observations according to the random leakage definition and considers a failure event with a failure probability ε when the simulation is unsuccessful. Then, it is easy to prove that given a circuit C composed of $|C|$ gates (i.e operations), its n-share variant C' where each gate is replaced by the corresponding (t, p, ε)-random probing composable gadget is actually $(\mathbf{p}, |\mathbf{C}|.\varepsilon)$-*random probing secure*, considering that the probability of dependence on the secret inputs in C' is when the simulation is unsuccessful in any of the composed gadgets, which gives a negligible probability of $|C|.\varepsilon$. In a typical circuit, four types of gates or operations are considered: standard addition and multiplication, copy gates which produce two fresh copies of an input variable, and random gates which have no input and output a fresh random value. Then, provided with a (t, p, ε)-random probing composable n-share gadget for each of these base operations, one defines CC as the *circuit compiler* which outputs the n-share circuit C' out of the base circuit C (i.e replaces each gate by the corresponding gadget).

The security level of a gadget in the random probing model is defined according to the simulation failure event probability ε, in other words, it is defined as $-\log_2(\varepsilon)$. Intuitively, the higher the leakage probability p, the higher the failure probability ε, and for a fixed value p, a gadget can only achieve a certain security level depending on its structure and the number of shares n. In order to achieve arbitrary levels of security while considering fixed leakage probabilities, Belaïd et al. elaborate in [BCP+20] on the *expansion strategy* already introduced in a previous work [AIS18] but in a theoretical and impractical manner. While the construction of [AIS18] relies on a t-out-n secure MPC protocol in the passive security model, the advantage of Belaïd et al.'s strategy is that it can be achieved and/or verified by simple atomic gadgets leading to simple and efficient constructions. The basic principle of the gadget expansion strategy is as follows. Assume we have four n-share gadgets G_{add}, G_{mult}, G_{copy}, G_{rand} for each of the base operations and denote CC the circuit compiler for these base gadgets[3]. We can derive four new n^2-share gadgets by simply applying CC to each gadget: $G^{(2)}_{\mathsf{add}} = CC(G_{\mathsf{add}})$, $G^{(2)}_{\mathsf{mult}} = CC(G_{\mathsf{mult}})$, $G^{(2)}_{\mathsf{copy}} = CC(G_{\mathsf{copy}})$ and $G^{(2)}_{\mathsf{rand}} = CC(G_{\mathsf{rand}})$. This process simply consists in replacing each addition gate in the original gadget by G_{add}, each multiplication gate by G_{mult}, each copy gate by G_{copy} and each random gate by G_{rand}, and by replacing each wire by n wires carrying a sharing of the original wire. Doing so, we obtain n^2-share gadgets for the addition, multiplication, copy and random. This process can be iterated an arbitrary number of times, say k, to an input circuit C:

$$C \xrightarrow{CC} C_1 \xrightarrow{CC} \cdots \xrightarrow{CC} C_k \ .$$

The first output circuit C_1 is the original circuit in which each gate is replaced by a base gadget G_{add}, G_{mult}, G_{copy} or G_{rand}. The second output circuit C_2 is the original circuit C in which each gate is replaced by an n^2-share gadget $G^{(2)}_{\mathsf{add}}$, $G^{(2)}_{\mathsf{mult}}$, $G^{(2)}_{\mathsf{copy}}$ or $G^{(2)}_{\mathsf{rand}}$ as defined above. Equivalently, C_2 is the circuit C_1 in which each gate is replaced by a base gadget. In the end, the output circuit C_k is hence the original circuit C in which each gate has been replaced by a k-expanded gadget and each wire has been replaced by n^k wires carrying an (n^k)-linear sharing of the original wire. The underlying compiler is called *expanding circuit compiler*. In other words, given a circuit compiler CC with the four base gadgets and an expansion level k, the expanding circuit compiler takes as input a base circuit C and outputs the compiled circuit C_k as described above.

The goal of the expansion strategy in the context of random probing security is to replace the

[3]Observe that an n-share gadget G_{rand} for a random gate can simply consist in n independent random gates. This trivial construction for G_{rand} is sufficient to achieve the desired properties since there are no intermediate operations.

leakage probability p of a wire in the original circuit by the failure event probability ϵ in the subsequent gadget simulation. If this simulation fails then one needs the full input sharing for the gadget simulation, which corresponds to leaking the corresponding wire value in the base case. The security is thus amplified by replacing the probability p in the base case by the probability ε (assuming that we have $\varepsilon < p$). If the failure event probability ε can be upper bounded by some function of the leakage probability: $\varepsilon < f(p)$ for every leakage probability $p \in [0, p_{\max}]$ for some $p_{\max} < 1$, then the expanding circuit compiler with expansion level k shall result in a security amplification as

$$p = \varepsilon_0 \xrightarrow{f} \epsilon_1 \xrightarrow{f} \cdots \xrightarrow{f} \varepsilon_k = f^{(k)}(p) \ ,$$

which for an adequate function f (eg $f : p \mapsto p^2$) provides exponential security.

In order to get such a security expansion, the gadgets must satisfy a stronger notion than the composability notion which is called (t, p, ε)-*random probing expandability*. In the evaluation of random probing composability, the failure event in the simulation of a gadget means that more than t shares from one of its input sharings are necessary to complete a perfect simulation. For a gadget to be expandable, slightly stronger notions than random probing composability are needed. As first requirement, a two-input gadget should have a failure probability which is independent for each input. This is because in the base case, each wire as input of a gate leaks independently. Typically, the probability that the simulator needs more than t shares of the first input (respectively second input) for the simulation would be ε_1 (respectively ε_2), and more ssimply with $\varepsilon_1 = \varepsilon_2$. On the other hand, during the expansion process of a gadget, in case of failure event in the child gadget (i.e the i-expanded gadget before being expanded into a $i + 1$-expanded gadget), the overall simulator should be able to produce a perfect simulation of the full output (that is the full input for which the failure occurs). To do so, the overall simulator is given the clear output (which is obtained from the simulation of the base case) plus any set of $n - 1$ output shares. Hence, to prove that an n-share gadget is (t, p, ε)-*random probing expandable* (with 2 inputs and 1 output for simplicity), there are two cases to consider:

- whenever the simulator is given a set of at most t output shares, the simulation process is similar to the case of random probing composability except for the failure events considered independently for each input.
- whenever the simulator is given a set of more than t output shares, he tries to replace it with any set of $n - 1$ output shares and produce a perfect simulation of these output shares along with the internal probes on the gadget, always considering a failure event probability ε independent for each input sharing.

If a circuit compiler CC is equiped with four n-share gadgets which are $(t, p, \varepsilon = f(p))$-random probing expandable, then it is proved in [BCP$^+$20] that the corresponding expanding compiler instantiated with CC and an expansion level k is equiped with k-expanded gadgets which are random probing expandable and achieve a security of $\varepsilon^k = f^{(k)}(p)$. Thus, on any input base circuit C, the expanding compiler produces a n^k-share circuit C_k with the same functionality as C and which is $(p, c.|C|.\varepsilon^k)$-random probing secure for some small constant c specified in the published work [BCP$^+$20], and for a fixed probability p.

The complexity of the expanding compiler can be expressed in terms of a desired security level κ. Specifically, if we consider that the failure probability ε is expressed as a function of the leakage probability $f(p) = c_d p^d + \mathcal{O}(p^{d+1})$ for some coefficient $c_d \in \mathbb{R}$ and $d \in \mathbb{R}$ (d can be called as the *amplification order* of f or of the gadget), then we need to achieve the security bound $f^{(k)}(p) \leq 2^{-\kappa}$ with the expansion level k. It can be verified using mathematical simplifications that given a base

circuit C, and a security level κ, the complexity of the resulting expanded n^k-share circuit $\hat{C}$ using the expanding compiler with the base failure function $f(p)$ can be expressed as

$$|\hat{C}| = \mathcal{O}\big(|C| \cdot \kappa^e\big) \quad \text{with} \quad e = \frac{\log N}{\log d} \,. \tag{3}$$

where d is the amplification order, and N is a parameter proportional to the complexity of the base gadgets of the expanding compiler in terms of number of gates or operations, i.e the complexities of G_{add}, G_{copy}, G_{mult} and G_{rand}. From the complexity formula, one can see that to get the best asymptotic complexity, one would need to achieve the best amplification order possible, while decreasing the complexity of the underlying gadgets.

A verification tool named VRAPS ((V)erifier of (RA)ndom (P)robing (S)ecurity) has been developed which is able to compute the failure function $f(p)$ of gadgets as well as determine the tolerated leakage probability by such gadgets. The tool can verify (t, p, ε)-random probing expandability, but also (p, ε)-random probing security and (t, p, ε)-random probing composability. The source code of VRAPS is available at:

https://github.com/CryptoExperts/VRAPS

An implementation of the expanding compiler has been done as well and is publicly available at:

https://github.com/CryptoExperts/poc-expanding-compiler

The expanding compiler takes as input the base n-share gadgets and the expansion level and outputs the k-expanded gadgets. A protected AES implementation is also included in the implementation as an illustration, which takes as input any n^k-share gadgets for $k \in \mathbb{N}$. All technical details on the expanding compiler can be found in the full version of the published paper [BCP+20]:

https://eprint.iacr.org/2020/786.pdf

The limitations of the expansion strategy are later studied by Belaïd et al. in [BRT21]. Namely, they show the amplification order d of the n-share gadgets of the expanding compiler is bounded by $d \leq \lfloor \frac{n+1}{2} \rfloor$, which gives a bound on the value that can be attained by the exponent e in equation (3) for a given number of shares. They also look for gadgets which can achieve this bound on the amplification order to have the best asymptotic complexity for the expansion strategy, while studying some well-known gadgets in the state-of-the-art like ISW applied to the expanding compiler. All the technical details on this study can be found in the full version of the published paper [BRT21]:

https://eprint.iacr.org/2021/434.pdf

3.2. Implementation of Core Gadgets

3.2.1. Constructions from [BCP+20]

In [BCP+20], Belaïd et al. provide a first instantiation of the expanding compiler with three 3-share base gadgets G_{add}, G_{mult}, G_{copy} (recall that for G_{rand} the trivial construction of 3 random gates is sufficient). The order of the operations in the following gadgets follows the classical order of operations:

$$\begin{aligned} G_{\mathsf{add}} : z_0 &\leftarrow x_0 + r_0 + r_4 + y_0 + r_1 + r_3 \\ z_1 &\leftarrow x_1 + r_1 + r_5 + y_1 + r_2 + r_4 \\ z_2 &\leftarrow x_2 + r_2 + r_3 + y_2 + r_0 + r_5 \end{aligned}$$

$$
\begin{aligned}
G_{\mathsf{copy}} : v_0 &\leftarrow u_0 + r_0 + r_1;\ w_0 \leftarrow u_0 + r_3 + r_4 \\
v_1 &\leftarrow u_1 + r_1 + r_2;\ w_1 \leftarrow u_1 + r_4 + r_5 \\
v_2 &\leftarrow u_2 + r_2 + r_0;\ w_2 \leftarrow u_2 + r_5 + r_3
\end{aligned}
$$

$$
\begin{aligned}
G_{\mathsf{mult}} : u_0 &\leftarrow x_0 + r_5 + r_6; \quad u_1 \leftarrow x_1 + r_6 + r_7; \quad u_2 \leftarrow x_2 + r_7 + r_5 \\
v_0 &\leftarrow y_0 + r_8 + r_9; \quad v_1 \leftarrow y_1 + r_9 + r_{10}; \quad v_2 \leftarrow y_2 + r_{10} + r_8
\end{aligned}
$$

$$
\begin{aligned}
z_0 &\leftarrow \big(u_0 \cdot v_0 + r_0\big) + \big(u_0 \cdot v_1 + r_1\big) + \big(u_0 \cdot v_2 + r_2\big) \\
z_1 &\leftarrow \big(u_1 \cdot v_0 + r_1\big) + \big(u_1 \cdot v_1 + r_4\big) + \big(u_1 \cdot v_2 + r_3\big) \\
z_2 &\leftarrow \big(u_2 \cdot v_0 + r_2\big) + \big(u_2 \cdot v_1 + r_3\big) + \big(u_2 \cdot v_2 + r_0\big) + r_4
\end{aligned}
$$

Using the verification tool VRAPS, one can check that the failure function $\varepsilon = f(p)$ of the expanding compiler instantiated with the above gadgets is expressed as $f(p) = \sqrt{83}p^{3/2} + \mathcal{O}(p^2)$ thus with an amplification order $d = 3/2$. The condition $\varepsilon < p$ which is necessary for the expansion strategy to be effective as discussed above, imposes that the maximum tolerated leakage probability by this compiler is $p_{\max} \approx 2^{-8}$. This means that for a given circuit, as long as the leakage probability in the random probing model does not exceed this value, the compiled circuit is considered secure after applying the expansion with the desired expanded failure event probability. Using these gadgets, the authors show that the corresponding expanding compiler has an asymptotic complexity formula from equation (3) equal to $\mathcal{O}\big(|C| \cdot \kappa^{7.5}\big)$ for any base circuit C to be expanded.

The authors additionally compute the complexity in terms of number of gates for each of the compiled gadgets $G^{(k)}_{\mathsf{add}}$, $G^{(k)}_{\mathsf{copy}}$, $G^{(k)}_{\mathsf{mult}}$ for a given expansion level k. In Fig. 5, we plot the total number of gates in each of the compiled gadgets as a function of the level k. For instance, for level $k = 9$ the number of gates in the compiled gadgets is around 10^{12}. For the latter level and assuming a leakage probability of $p \approx 2^{-8}$ (which is the maximum this compiler can tolerate), a security of $\varepsilon \approx 2^{-76}$ can be achieved.

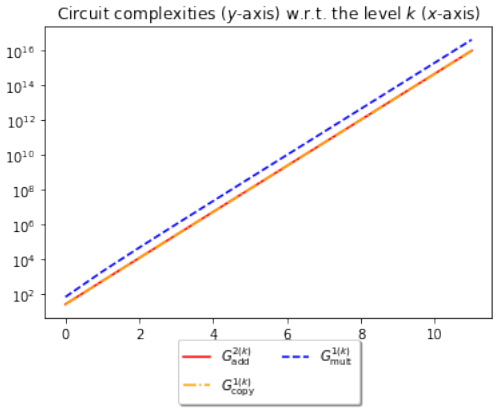

Figure 5: Number of gates for $G^{(k)}_{\mathsf{add}}$, $G^{(k)}_{\mathsf{copy}}$, $G^{(k)}_{\mathsf{mult}}$ circuits with respect to the level k.

Table 4 additionally shows the execution time in milliseconds of the compiled gadgets using the compiler with several expansion levels k, when run in C on randomly generated 8-bit integers. For

the generation of random variables, the authors consider that an efficient external random number generator is available in practice, and so they simply use the values of an incremented counter variable to simulate random gates. The compiled gadgets are the output of the open-source expanding compiler implemented in python, which takes as input the base gadgets and the desired expansion level, and outputs the corresponding k-expanded gadgets.

Table 4: Execution time (in ms, on an Intel i7-8550U CPU) for compiled gadgets $G^{(k)}_{\mathsf{add}}$, $G^{(k)}_{\mathsf{copy}}$, $G^{(k)}_{\mathsf{mult}}$ implemented in C.

k	# shares	Gadget	Execution time
1	3	$G^{(1)}_{\mathsf{add}}$	$1,69.10^{-4}$
		$G^{(1)}_{\mathsf{copy}}$	$1,67.10^{-4}$
		$G^{(1)}_{\mathsf{mult}}$	$5,67.10^{-4}$
2	9	$G^{(2)}_{\mathsf{add}}$	$2,21.10^{-3}$
		$G^{(2)}_{\mathsf{copy}}$	$2,07.10^{-3}$
		$G^{(2)}_{\mathsf{mult}}$	$9,91.10^{-3}$
3	27	$G^{(3)}_{\mathsf{add}}$	$9,29.10^{-2}$
		$G^{(3)}_{\mathsf{copy}}$	$9,84.10^{-2}$
		$G^{(3)}_{\mathsf{mult}}$	$3,67.10^{-1}$

It can be observed that both the complexity and running time grow by almost the same factor with the expansion level, with multiplication gadgets being the slowest as expected. Base gadgets with $k = 1$ roughly take 10^{-4} ms, while these gadgets expanded 2 times ($k = 3$) take between 10^{-2} and 10^{-1} ms. The difference between the linear cost of addition and copy gadgets, and the quadratic cost of multiplication gadgets can also be observed through the gadgets' complexities.

An n-share AES-128 implementation was developed in [BCP+20] in C as an illustration. The source code of the algorithm is provided at the same github link than the open-source expanding compiler. The same compiled gadgets above were used for base operations (addition, multiplication, copy). Applying the same complexity analysis done previously on the complexities of the individual gadgets,Fig. 6 display the total number of gates in the AES-128 encryption/decryption procedures as functions of the level k. For instance, for the same security level of 2^{-76} exhibited earlier in this section for the gadgets of Fig. 5, the AES-128 would have to be compiled at a level $k = 9$, and would count around 10^{16} gates.

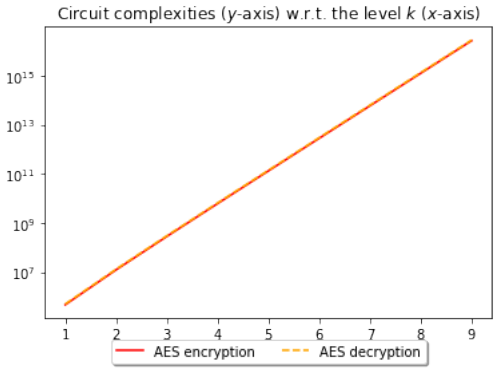

Figure 6: Number of gates after compilation of AES-128 encryption/decryption circuits with respect to the level k.

For the implementation, the authors chose the C 8-bit unsigned integer type, and considered operations in GF(256). Table 5 shows the AES-128 execution time on a 16-byte message with 10 pre-computed sub-keys, using compiled gadgets $G^{(k)}_{\mathsf{add}}$, $G^{(k)}_{\mathsf{copy}}$, $G^{(k)}_{\mathsf{mult}}$, with respect to the expansion level k and sharing order $n = 3^k$. It can be seen that the execution time increases with the expansion level with a similar growth as in Table 4. This is because the complexity of the AES circuit strongly depends on the gadgets that are used to replace each gate in the original arithmetic circuit. For example, with the above 3-share gadgets that tolerate a leakage probability of $p \approx 2^{-8}$, a 27-share $(k = 3)$ AES-128 takes almost 200 milliseconds to encrypt or decrypt a message.

Table 5: Standard and n-share AES-128 execution time (in ms, on an Intel i7-8550U CPU) using compiled gadgets $G^{(k)}_{\mathsf{add}}$, $G^{(k)}_{\mathsf{copy}}$, $G^{(k)}_{\mathsf{mult}}$.

AES Version	Execution Time (in ms)	
	Encryption	Decryption
Standard (no sharing)	0.06	0.05
3-share $(k = 1)$	1.08	1.07
9-share $(k = 2)$	11.71	10.26
27-share $(k = 3)$	200.29	197.70

3.2.2. Constructions from [BRT21]

While in [BCP+20] the authors try to find specific 3-share gadgets that have a good amplification order and low failure probability, the authors of [BRT21] elaborate on the construction of expanding compilers, specifically by looking for gadgets which achieve the optimal amplification order (i.e the bound $\lfloor \frac{n+1}{2} \rfloor$ for n-share gadgets specified at the end of section 3.1), while having good complexities in terms of number of operations. This would allow to have optimal constructions in terms of the exponent expressed in equation (3) for the expansion strategy in terms of the desired security level.

For linear gadgets G_{add} and G_{copy}, the authors first prove that constructing such gadgets can be done using an underlying refresh gadget G_{refresh}, and which security reduces to the security of the latter. The advantage of this approach is that it provides generic constructions for addition and copy gadgets which can achieve the maximum amplification order for any number of shares n. Also,

proving the security of a refresh gadget is usually easier than for more complicated addition and copy gadgets, typically using a verification tool like VRAPS. Algorithm 1 displays the generic construction for the copy gadget from a refresh gadget. It simply consists in refreshing the input sharing twice to obtain two fresh copies. And algorithm 2 displays the generic construction for the addition gadget from a refresh gadget. It simply consists in refreshing both input sharings before adding them.

Algorithm 1: Copy gadget G_{copy}

Input : $(a_1, \ldots, a_n)$ input sharing
Output: $(e_1, \ldots, e_n)$, $(f_1, \ldots, f_n)$ fresh copies of $(a_1, \ldots, a_n)$
$(e_1, \ldots, e_n) \leftarrow G_{\text{refresh}}(a_1, \ldots, a_n)$;
$(f_1, \ldots, f_n) \leftarrow G_{\text{refresh}}(a_1, \ldots, a_n)$;

Algorithm 2: Addition Gadget G_{add}

Input : $(a_1, \ldots, a_n), (b_1, \ldots, b_n)$ input sharings
Output: $(c_1, \ldots, c_n)$ sharing of $a + b$
$(e_1, \ldots, e_n) \leftarrow G_{\text{refresh}}(a_1, \ldots, a_n)$;
$(f_1, \ldots, f_n) \leftarrow G_{\text{refresh}}(b_1, \ldots, b_n)$;
$(c_1, \ldots, c_n) \leftarrow (e_1 + f_1, \ldots, e_n + f_n)$;

The authors are able to show that the addition and copy gadgets from these constructions achieve the same amplification order as the underlying refresh gadget. Thus, proving the security of the refresh gadget for the expansion strategy directly implies the security of the constructed gadgets. For instance, the authors also prove that the well-known ISW refresh gadget is actually random probing expandable and achieves the bound on the amplification order of $\lfloor \frac{n+1}{2} \rfloor$, and has a quadratic complexity in the number of shares.

As for the multiplication gadget G_{mult}, the authors start by showing that the ISW multiplication gadget is random probing expandable but does not achieve the maximal amplification order. In fact, it achieves an order of $\lfloor \frac{n+1}{4} \rfloor$. Using the ISW-based addition and copy gadgets and the ISW multiplication gadget thus yields an expanding compiler achieving a quadratic complexity with an overall amplification order of $\lfloor \frac{n+1}{4} \rfloor$. Figure 7 shows the evolution of the complexity exponent from equation (3) with the ISW-based expanding compiler (blue curve) and an expanding compiler which uses the ISW-based addition and copy gadgets but assumes a multiplication gadget which reaches the maximal amplification order (orange curve). The value of e clearly decreases as the number of shares grows, and this decrease is faster for a small number of shares ($5 \leq n \leq 10$). It is also clear that the decrease of the exponent with respect to the number of shares is faster with the expanding compiler achieveing the maximal amplification order.

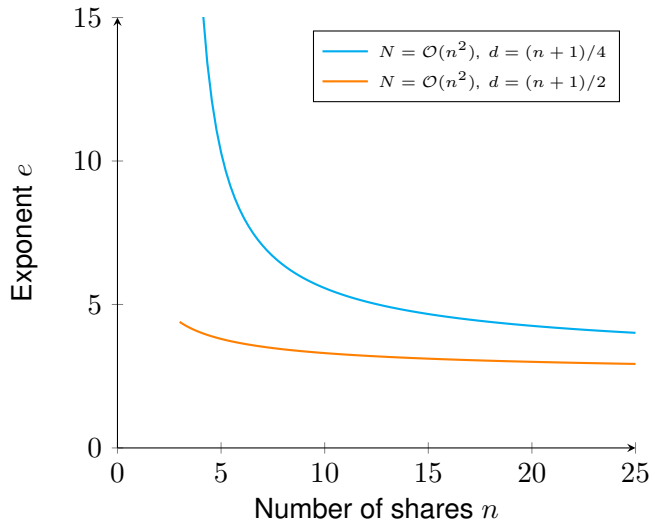

Figure 7: Evolution of the complexity exponent $e = \log(N)/\log(d)$ from equation (3) with respect to the number of shares n. The blue curve matches the instantiation with the ISW-based gadgets; the orange curve assumes the optimal amplification order (i.e an improvement of the multiplication gadget) with the same asymptotic complexity.

Hence, to reach the optimal amplification order for a generic number of shares n, one would need a different multiplication gadget than ISW. The authors then introduce a new multiplication gadget which starts by refreshing the input sharings before performing the product of shares. The gadget's algorithm is depicted in Algorithm 3.

Algorithm 3: New multiplication gadget from [BRT21]

Input : $(a_1, \ldots, a_n)$,$(b_1, \ldots, b_n)$ input sharings, $\{r_{ij}\}_{1 \leq i \leq n, 1 \leq j \leq n}$ random values, refresh gadget G_{refresh}
Output: $(c_1, \ldots, c_n)$ sharing of $a \cdot b$
for $i \leftarrow 1$ **to** n **do**
  $(b_1^{(i)}, \ldots, b_n^{(i)}) \leftarrow G_{\text{refresh}}(b_1, \ldots, b_n)$;
end
$(a'_1, \ldots, a'_n) \leftarrow G_{\text{refresh}}(a_1, \ldots, a_n)$;
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $p_{i,j} \leftarrow a'_i \times b_j^{(i)} + r_{i,j}$;
  end
end
$(v_1, \ldots, v_n) \leftarrow (0, \ldots, 0)$;
$(x_1, \ldots, x_n) \leftarrow (0, \ldots, 0)$;
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $v_i \leftarrow v_i + p_{i,j}$;
    $x_i \leftarrow x_i + r_{i,j}$;
  end
end
for $i \leftarrow 1$ **to** n **do**
  $c_i \leftarrow v_i + x_i$;
end
return $(c_1, \ldots, c_n)$;

The authors show that if the underlying G_{refresh} gadget from the above algorithm satisfies certain properties similar to the ones needed for G_{refresh} in the generic addition and copy gadgets constructions, then the multiplication gadget G_{mult} described in Algorithm 3 is random probing expandable and achieves the maximum amplification order $\lfloor \frac{n+1}{2} \rfloor$, while having a quadratic asymptotic complexity (if we assume for instance the ISW-based refresh gadget or any refresh gadget which has a quadratic complexity). Using the above multiplication gadget with the ISW-based generic addition and copy gadgets leads to an expanding compiler which complexity blowup from equation (3) corresponds to the orange curve in Figure 7.

The authors finally provide 3-share and 5-share instantiations of expanding compilers which achieve the optimal amplification orders and tolerate competitive leakage probabilities. Their 3-share instantiation is based on the following refresh gadget which uses only two random values, better than the 3-share ISW refresh gadget:

$$\begin{aligned} G_{\text{refresh}} : c_1 &\leftarrow r_1 + a_1 \\ c_2 &\leftarrow r_2 + a_2 \\ c_3 &\leftarrow (r_1 + r_2) + a_3. \end{aligned}$$

This refresh gadget leads to the 3-share addition gadget:

$$\begin{aligned} G_{\mathsf{add}} : c_1 &\leftarrow (r_1 + a_1) + (r_3 + b_1) \\ c_2 &\leftarrow (r_2 + a_2) + (r_4 + b_2) \\ c_3 &\leftarrow \big((r_1 + r_2) + a_3\big) + \big((r_3 + r_4) + b_3\big) \end{aligned}$$

and the following 3-share copy gadget:

$$\begin{aligned} G_{\mathsf{copy}} : c_1 &\leftarrow r_1 + a_1; \qquad d_1 \leftarrow r_3 + a_1 \\ c_2 &\leftarrow r_2 + a_2; \qquad d_2 \leftarrow r_4 + a_2 \\ c_3 &\leftarrow (r_1 + r_2) + a_3; \quad d_3 \leftarrow (r_3 + r_4) + a_3. \end{aligned}$$

For the 3-share multiplication, the authors use the gadget from Algorithm 3 instantiated with the above 3-share refresh gadget:

$$\begin{aligned} G_{\mathsf{mult}} : i_{1,1} &\leftarrow r_1 + b_1; & i_{1,2} &\leftarrow r_2 + b_2; & i_{1,3} &\leftarrow (r_1 + r_2) + b_3 \\ i_{2,1} &\leftarrow r_3 + b_1; & i_{2,2} &\leftarrow r_4 + b_2; & i_{2,3} &\leftarrow (r_3 + r_4) + b_3 \\ i_{3,1} &\leftarrow r_5 + b_1; & i_{3,2} &\leftarrow r_6 + b_2; & i_{3,3} &\leftarrow (r_5 + r_6) + b_3 \\ a'_1 &\leftarrow r_7 + a_1; & a'_2 &\leftarrow r_8 + a_2; & a'_3 &\leftarrow (r_7 + r_8) + a_3 \end{aligned}$$

$$\begin{aligned} c_1 &\leftarrow (a'_1 \cdot i_{1,1} + r_{1,1}) + (a'_1 \cdot i_{1,2} + r_{1,2}) + (a'_1 \cdot i_{1,3} + r_{1,3}) + (r_{1,1} + r_{2,1} + r_{3,1}) \\ c_2 &\leftarrow (a'_2 \cdot i_{2,1} + r_{2,1}) + (a'_2 \cdot i_{2,2} + r_{2,2}) + (a'_2 \cdot i_{2,3} + r_{2,3}) + (r_{1,2} + r_{2,2} + r_{3,2}) \\ c_3 &\leftarrow (a'_3 \cdot i_{3,1} + r_{3,1}) + (a'_3 \cdot i_{3,2} + r_{3,2}) + (a'_3 \cdot i_{3,3} + r_{3,3}) + (r_{1,3} + r_{2,3} + r_{3,3}). \end{aligned}$$

Table 6 displays the results for the above gadgets obtained through the VRAPS tool. The second column gives the complexity, where N_a, N_c, N_m, N_r stand for the number of addition gates, copy gates, multiplication gates and random gates respectively in each gadget. The third column provides the amplification order of the gadget. And the last column gives the maximum tolerated leakage probability. The last row gives the global complexity, amplification order, and maximum tolerated leakage probability for the expanding compiler using these three gadgets from the results provided in [BCP+20].

Table 6: Results for the 3-share gadgets achieving the bound on the amplification order.

Gadget	**Complexity** (N_a, N_c, N_m, N_r)	**Amplification order**	$\log_2$ **of maximum tolerated proba**
G_{refresh}	$(4, 2, 0, 2)$	2	-5.14
G_{add}	$(11, 4, 0, 4)$	2	-4.75
G_{copy}	$(8, 7, 0, 4)$	2	-7.50
G_{mult}	$(40, 29, 9, 17)$	2	-7.41
Compiler	$\mathcal{O}(\lvert C\rvert \cdot \kappa^{\mathbf{3.9}})$	$\mathbf{2}$	$\mathbf{-7.50}$

As for the 5-share construction, the authors use the following 5-share refresh gadget which re-

quires five uniform random values, instead of ISW-refresh:

$$\begin{aligned} G_{\text{refresh}} : c_1 &\leftarrow (r_1 + r_2) + a_1 \\ c_2 &\leftarrow (r_2 + r_3) + a_2 \\ c_3 &\leftarrow (r_3 + r_4) + a_3 \\ c_4 &\leftarrow (r_4 + r_5) + a_4 \\ c_5 &\leftarrow (r_5 + r_1) + a_5. \end{aligned}$$

This gadget only uses n randoms for an n-share construction, and while it does not achieve enough security in the generic case (unless the refresh block is iterated on the input a certain number of times [BBD$^+$20, BDF$^+$17]), it proves to be more than enough to achieve the necessary amplification order for the 5-share constructions. The authors use a variant of the original version (also suggested in [BBD$^+$20]): they choose to sum the random values first (thus obtaining a sharing of 0) before adding them to the input shares. The idea is to avoid using the input shares in any of the intermediate variables, so that input shares only appear in the input variables $\{a_i\}_{1\leq i\leq n}$ and the final output variables $\{c_i\}_{1\leq i\leq n}$. Intuitively, this trick allows to have less failure tuples in the gadget because there are less variables that could leak information about the input. This is validated experimentally where better results are obtained in terms of amplification order and tolerated leakage probability for small gadgets. From this circular refresh, the authors obtain an addition gadget with ten random values which reaches the upper bound on the amplification order:

$$\begin{aligned} G_{\text{add}} : c_1 &\leftarrow \big((r_1 + r_2) + a_1\big) + \big((r_6 + r_7) + b_1\big) \\ c_2 &\leftarrow \big((r_2 + r_3) + a_2\big) + \big((r_7 + r_8) + b_2\big) \\ c_3 &\leftarrow \big((r_3 + r_4) + a_3\big) + \big((r_8 + r_9) + b_3\big) \\ c_4 &\leftarrow \big((r_4 + r_5) + a_4\big) + \big((r_9 + r_{10}) + b_4\big) \\ c_5 &\leftarrow \big((r_5 + r_1) + a_5\big) + \big((r_{10} + r_6) + b_5\big) \end{aligned}$$

and a copy gadget with also ten random values and which also reaches the upper bound on the amplification order:

$$\begin{aligned} G_{\text{copy}} : c_1 &\leftarrow (r_1 + r_2) + a_1; \qquad d_1 \leftarrow (r_6 + r_7) + a_1 \\ c_2 &\leftarrow (r_2 + r_3) + a_2; \qquad d_2 \leftarrow (r_7 + r_8) + a_2 \\ c_3 &\leftarrow (r_3 + r_4) + a_3; \qquad d_3 \leftarrow (r_8 + r_9) + a_3 \\ c_4 &\leftarrow (r_4 + r_5) + a_4; \qquad d_4 \leftarrow (r_9 + r_{10}) + a_4 \\ c_5 &\leftarrow (r_5 + r_1) + a_5; \qquad d_5 \leftarrow (r_{10} + r_6) + a_5. \end{aligned}$$

The following construction is a 5-share instantiation of the multiplication gadget described in Algorithm 3. For the input refreshing, the 5-share circular refresh gadget described above is used. The

gadget advantageously achieves the optimal amplification order with 55 random values:

$$
\begin{aligned}
G_{\mathsf{mult}} : i_{1,1} &\leftarrow (r_1 + r_2) + b_1;\ \ i_{1,2} \leftarrow (r_2 + r_3) + b_2;\ \ i_{1,3} \leftarrow (r_3 + r_4) + b_3;\\
i_{1,4} &\leftarrow (r_4 + r_5) + b_4;\ \ i_{1,5} \leftarrow (r_5 + r_1) + b_5\\
\\
i_{2,1} &\leftarrow (r_6 + r_7) + b_1;\ \ i_{2,2} \leftarrow (r_7 + r_8) + b_2;\ \ i_{2,3} \leftarrow (r_8 + r_9) + b_3;\\
i_{2,4} &\leftarrow (r_9 + r_{10}) + b_4;\ \ i_{2,5} \leftarrow (r_{10} + r_6) + b_5\\
\\
i_{3,1} &\leftarrow (r_{11} + r_{12}) + b_1;\ \ i_{3,2} \leftarrow (r_{12} + r_{13}) + b_2;\ \ i_{3,3} \leftarrow (r_{13} + r_{14}) + b_3;\\
i_{3,4} &\leftarrow (r_{14} + r_{15}) + b_4;\ \ i_{3,5} \leftarrow (r_{15} + r_{11}) + b_5\\
\\
i_{4,1} &\leftarrow (r_{16} + r_{17}) + b_1;\ \ i_{4,2} \leftarrow (r_{17} + r_{18}) + b_2;\ \ i_{4,3} \leftarrow (r_{18} + r_{19}) + b_3;\\
i_{4,4} &\leftarrow (r_{19} + r_{20}) + b_4;\ \ i_{4,5} \leftarrow (r_{20} + r_{16}) + b_5\\
\\
i_{5,1} &\leftarrow (r_{21} + r_{22}) + b_1;\ \ i_{5,2} \leftarrow (r_{22} + r_{23}) + b_2;\ \ i_{5,3} \leftarrow (r_{23} + r_{24}) + b_3;\\
i_{5,4} &\leftarrow (r_{24} + r_{25}) + b_4;\ \ i_{5,5} \leftarrow (r_{25} + r_{21}) + b_5\\
\\
a'_1 &\leftarrow (r_{26} + r_{27}) + a_1;\ \ a'_2 \leftarrow (r_{27} + r_{28}) + a_2;\ \ a'_3 \leftarrow (r_{28} + r_{29}) + a_3;\\
a'_4 &\leftarrow (r_{29} + r_{30}) + a_4;\ \ a'_5 \leftarrow (r_{30} + r_{26}) + a_5
\end{aligned}
$$

$$
\begin{aligned}
c_1 \leftarrow\ & (a'_1 \cdot i_{1,1} + r_{1,1}) + (a'_1 \cdot i_{1,2} + r_{1,2}) + (a'_1 \cdot i_{1,3} + r_{1,3}) + (a'_1 \cdot i_{1,4} + r_{1,4})\\
& + (a'_1 \cdot i_{1,5} + r_{1,5}) + (r_{1,1} + r_{2,1} + r_{3,1} + r_{4,1} + r_{5,1})\\
c_2 \leftarrow\ & (a'_2 \cdot i_{2,1} + r_{2,1}) + (a'_2 \cdot i_{2,2} + r_{2,2}) + (a'_2 \cdot i_{2,3} + r_{2,3}) + (a'_2 \cdot i_{2,4} + r_{2,4})\\
& + (a'_2 \cdot i_{2,5} + r_{2,5}) + (r_{1,2} + r_{2,2} + r_{3,2} + r_{4,2} + r_{5,2})\\
c_3 \leftarrow\ & (a'_3 \cdot i_{3,1} + r_{3,1}) + (a'_3 \cdot i_{3,2} + r_{3,2}) + (a'_3 \cdot i_{3,3} + r_{3,3}) + (a'_3 \cdot i_{3,4} + r_{3,4})\\
& + (a'_3 \cdot i_{3,5} + r_{3,5}) + (r_{1,3} + r_{2,3} + r_{3,3} + r_{4,3} + r_{5,3})\\
c_4 \leftarrow\ & (a'_4 \cdot i_{4,1} + r_{4,1}) + (a'_4 \cdot i_{4,2} + r_{4,2}) + (a'_4 \cdot i_{4,3} + r_{4,3}) + (a'_4 \cdot i_{4,4} + r_{4,4})\\
& + (a'_4 \cdot i_{4,5} + r_{4,5}) + (r_{1,4} + r_{2,4} + r_{3,4} + r_{4,4} + r_{5,4})\\
c_5 \leftarrow\ & (a'_5 \cdot i_{5,1} + r_{5,1}) + (a'_5 \cdot i_{5,2} + r_{5,2}) + (a'_5 \cdot i_{5,3} + r_{5,3}) + (a'_5 \cdot i_{5,4} + r_{5,4})\\
& + (a'_5 \cdot i_{5,5} + r_{5,5}) + (r_{1,5} + r_{2,5} + r_{3,5} + r_{4,5} + r_{5,5}).
\end{aligned}
$$

Table 7 gives the results for the above gadgets obtained through the VRAPS tool.

Table 7: Results for the 3-share gadgets achieving the bound on the amplification order.

Gadget	**Complexity** (N_a, N_c, N_m, N_r)	**Amplification order**	$\log_2$ **of maximum tolerated proba**
G_{refresh}	$(10, 5, 0, 5)$	3	-4.83
G_{add}	$(25, 10, 0, 10)$	3	$[-6.43, -3.79]$
G_{copy}	$(20, 15, 0, 10)$	3	$[-6.43, -5.78]$
G_{mult}	$(130, 95, 25, 55)$	3	$[-12.00, -6.03]$
Compiler	$\mathcal{O}(\lvert C\rvert \cdot \kappa^{\mathbf{3.23}})$	**3**	$[\mathbf{-12.00}, \mathbf{-6.03}]$

From Tables 6 and 7, it can be seen that the asymptotic complexity is better for the instantiation based on 5-share gadgets as they provide a better amplification order with limited overhead. While this result can seem to be counterintuitive, it actually comes from the fact that each gadget will be expended less in the second scenario. The authors could only obtain an interval $[2^{-12}, 2^{-6}]$ for the tolerated leakage probability because it was computationally too expensive to obtain a tighter interval from the VRAPS tool. Meanwhile, one can consider that the best complexity $\mathcal{O}(|C| \cdot \kappa^{3.2})$ comes at the price of a lower tolerated leakage probability of 2^{-12} (5-share gadgets) compared to the $\mathcal{O}(|C| \cdot \kappa^{3.9})$ complexity and $2^{-7.5}$ tolerated leakage probability obtained for the 3-share instantiation. In comparison, the previous instantiation of the expanding compiler [BCP$^+$20] could only achieve a complexity of $\mathcal{O}(|C| \cdot \kappa^{7.5})$ for maximum tolerated probabilities of 2^{-8}.

Références bibliographiques

[AIS18] Prabhanjan Ananth, Yuval Ishai, and Amit Sahai. Private circuits: A modular approach. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 427–455. Springer, Heidelberg, August 2018.

[BBD+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.

[BBD+20] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations. *Journal of Cryptographic Engineering*, 10(1):17–26, April 2020.

[BCP+20] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Random probing security: Verification, composition, expansion and new constructions. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 339–368. Springer, Heidelberg, August 2020.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 535–566. Springer, Heidelberg, April / May 2017.

[BDM+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 311–341. Springer, Heidelberg, May 2020.

[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.

[BRT21] Sonia Belaïd, Matthieu Rivain, and Abdul Rahman Taleb. On the power of expansion: More efficient constructions in the random probing model. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 313–343. Springer, 2021.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pages 188–205, 2014.

[Cor17] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 93–114, 2017.

[Cor18] Jean-Sébastien Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, pages 65–82, 2018.

[Gou01] Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In *CHES*, pages 3–15, 2001.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 567–597. Springer, Heidelberg, April / May 2017.

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 623–643. Springer, Heidelberg, September 2017.

[MD19] Darius Mercadier and Pierre-Évariste Dagand. Usuba: high-throughput and constant-time ciphers, by construction. In *PLDI*, pages 157–173, 2019.