UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Extending MASKVERIF to Advanced Masking Schemes

Erweiterung von MASKVERIF *auf fortschrittliche Maskierungsschemata*

Masterarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Pajam Pauls

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth
mit Unterstützung von **Alexander Treff**

Lübeck, den 21. Mai 2023

Abstract

Side-channel attacks allow adversaries to gather information on cryptographic devices during execution. This information leads to increasingly more powerful attacks on cryptographic implementations. Consequently, side-channel countermeasures are an important topic of research, of which masking is the most researched and practically used choice. However, developing secure masked implementations can be complex and error-prone and therefore benefits from formal verification.

One state-of-the-art automated formal verification tool is MASKVERIF, which allows for verification based on formal security notions, even in the presence of physical defaults, like glitches and transitions. Many masked post-quantum cryptographic schemes use both Boolean and arithmetic operations and thus require secure conversion gadgets, which currently lead to false negatives when using MASKVERIF.

In this work, we extend MASKVERIF in a way that allows us to correctly verify first-order Boolean-to-arithmetic conversion gadgets in an automated fashion. We also provide a security proof for Goubin's seminal algorithm for Boolean-to-arithmetic conversion, showing that it achieves t–probing security. Additionally, we provide a number of MASKVERIF gadgets and explain the intricacies of interacting with the tool from a user's and programmer's perspective. Lastly, we provide a detailed explanation of MASKVERIF's codebase as well as in-code documentation, both of which allow for easier understanding and extension of the tool.

Zusammenfassung

Seitenkanal-Angriffe ermöglichen es Angreifern, während der Ausführung von kryptografischen Algorithmen Informationen zu sammeln. Diese zusätzlichen Informationen führen zu immer mächtigeren Angriffen auf kryptografische Implementierungen. Folglich sind Seitenkanal-Gegenmaßnahmen ein wichtiges Forschungsthema, von denen Maskierung die am meisten erforschte und genutzte Wahl ist. Die Entwicklung sicherer maskierter Implementierungen kann jedoch komplex und fehleranfällig sein und profitiert daher von formaler Verifikation.
Ein automatisiertes formales Verifikationstool ist MASKVERIF, das eine Verifikation auf der Grundlage formaler Sicherheitsbegriffe ermöglicht, selbst in Gegenwart von physikalischen Effekten wie Glitches und Übergängen. Viele maskierte post-quantum kryptographische Verfahren verwenden sowohl Boolsche als auch arithmetische Operationen und erfordern daher sichere Konvertierungsgadgets, die derzeit zu falsch negativen Ergebnissen führen, wenn MASKVERIF verwendet wird.
In dieser Arbeit erweitern wir MASKVERIF so, dass Boolsche-zu-arithmetische Konvertierungsgadgets der ersten Ordnung automatisch und korrekt überprüft werden können. Wir liefern auch einen Sicherheitsbeweis für Goubins bedeutenden Algorithmus aus Bereich Boolsche-zu-arithmetische Konvertierung und zeigen, dass er t–probing-Sicherheit erfüllt. Darüber hinaus stellen wir eine Reihe von MASKVERIF-Gadgets zur Verfügung und erläutern die Komplexitäten der Interaktion mit dem Tool von Seiten der Benutzer und Programmierer. Schließlich bieten wir eine detaillierte Erklärung der Codebasis von MASKVERIF sowie eine Dokumentation des Codes, die das Verständnis und die Erweiterung des Tools erleichtert.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 21. Mai 2023

Acknowledgements

First, I would like to thank professor Thomas Eisenbarth for giving me the opportunity to work on this interesting and challenging thesis and for his continued support.
I would like to express my gratitude towards Sebastian Berndt and Alexander Treff for their guidance, support and many interesting discussions over the course of this thesis. Your support has been invaluable, and I could not have asked for better advisors.
I am very grateful to Marc Gourjon, who taught me a lot about MASKVERIF and formal verification. Thank you for answering my tedious questions and having many insightful conversations with me over the course of this thesis.
I would also like to thank my friends Tim, Paula, Anna, and Johannes for the many years we have spent studying together. Pursuing this degree with all of you has made me a better student and scientist and allowed me to find my passion in this field of research. I want to thank you all for the interesting and thought-provoking discussions we have had and more importantly for all the jokes and laughs we have shared over the years. Without you all, I probably would not have pursued a master's degree and would never have discovered my genuine interest and curiosity for IT security and for that, I will always be grateful.
Last but not least I want to thank my family, who offered me unconditional emotional and financial support over the years and made sure that I had the ability to enjoy myself while studying.
Thank you all for your support and for making this journey as fun as it was, it would not have been the same without any of you.

Contents

1 Introduction

In a cryptographic key extraction scenario, an attacker tries to obtain secret information from a device while having restricted access to it. In the most basic scenario, referred to as the *Black-box model*, an attacker may only analyze the input and output behavior that they observe and use this information in their attempt to extract sensitive information.
This model, however, is oftentimes not sufficient to accurately assess the information the attacker has at their disposal. In addition to input and output behavior, information like power consumption [KJJ99], noise emission [GST14], fault injection [BS97], timing information [Koc96] and much more, are available to an attacker.
In reality, the attacker can thus also make use of this additional information, referred to as *side-channel information*, to perform *side-channel attacks*. This attack scenario is called the *Grey-box model* and permits much stronger attacks than the *Black-box model*, one of them being *differential power analysis* (DPA) introduced by Kocher et al. [KJJ99] in 1999. In a DPA attack, the power consumption is monitored and power traces are collected, which the attacker uses in a statistical analysis to extract the secret key. This attack turns out to be very powerful and has been used to break many symmetric cryptosystems, such as DES [KJJ99] and the AES candidates [CJRR99a, DR99], as well as public-key cryptosystems [Cor99, MDS99b].
Due to the potency of this and other side-channel attacks, research into countermeasures has become an increasing priority, leading to techniques like *masking* [CG00, CJRR99b]. The idea behind masking is to use *secret sharing schemes* to split the sensitive values into n *shares* such that the original value can only be reconstructed if one has all shares. A regular DPA attack is not able to extract the key anymore, since individual shares are not correlated to the sensitive values that are processed.
In order to assess the security of masked implementations formal security notions are necessary. One of the most used security notions is the *ISW model* [ISW03], which allows for practically relevant analysis with the simple assumption that an attacker can observe at most t intermediate values on the wires of a circuit [DDF14]. The ISW model has laid the foundation for more complex and sophisticated security notions like *non-interference* (t–NI), *strong non-interference* (t–SNI) [FGMDP+18], and *probe-isolating non-interference* (t–PINI) [CS20]. These notions offer different security guarantees and the development of algorithms satisfying them is an important topic of research. Since the development of masking schemes requires extensive effort and can be error-prone,

automated formal verification is often used to verify masked implementations regarding the previously mentioned security notions. In practice, formal verification tools like MASKVERIF [BBD+15, BBD+16, BBC+19], SILVER [KSM20], VERICA [RBFSG22], and others, allow researchers to automatically develop and verify more robust and secure countermeasures and offer guarantees regarding their security.

Contributions. Despite MASKVERIF being one of the leading automated verification tools, it currently does not correctly verify arithmetic-to-Boolean (A2B) and Boolean-to-arithmetic (B2A) conversion *gadgets* that are proven secure in the literature [SPOG19, CGV14]. The conversion algorithms play a crucial role in masked cryptographic implementations, e.g., masked versions of the post-quantum cryptographic schemes Kyber [BGR+21], Dilithium [MGTF19] , SABER [KDVB+22], and NTRU [CGTZ23]. Kyber and Dilithium are of special importance, since they were chosen for standardization by the National Institute of Standards and Technology[1]. Therefore, ensuring correct verification of conversion algorithms is vital to allow for formal verification of complex cryptographic schemes incorporating them. Our work in this thesis can be summarized in four contributions:

- The first contribution is the extension of MASKVERIF, in which we implement rules that solve the problems that occur during the verification of the first-order B2A conversion algorithms from Goubin [Gou01] and Coron [Cor17]. These rules find expressions which can lead MASKVERIF to incorrectly return a false negative and substitute these expressions by terms that are semantically equivalent and can be correctly verified.

- As our second step, we provide a concrete security proof for Goubin's B2A algorithm [Gou01]. More specifically, we show that the algorithm achieves t–probing security, but fails to achieve the stronger t–non–interference property. The rules we implemented, that aid MASKVERIF in its verification process of Goubin's B2A algorithm [Gou01], rely on this security proof to substitute expressions. This proof also allows researchers to easily understand which of the modern security properties Goubin's algorithm satisfies.

- Our third contribution is the documentation and explanation of MASKVERIF's codebase. The original MASKVERIF code[2] does not have any documentation, which makes it exceedingly difficult to understand how the verification is implemented and how it can be extended. This contribution should help researchers aiming to

[1] https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
[2] https://gitlab.com/benjgregoire/maskverif/

further improve the tool by offering explanations for parts of the codebase which are crucial for the verification.

- Last but not least, we implemented MASKVERIF gadgets for first-order B2A and A2B conversion for Goubin's algorithms [Gou01], first and second order gadgets for Coron's B2A algorithm [Cor17], mask refreshing gadgets for the algorithm by Rivain et al. [RP10] up to tenth-order and t–SNI secure bitwise AND, as well as gadgets for a t–SNI addition algorithm proposed by Coron et al. in Algorithm 1 and 2 of [CGV14] up to fifth order. All of these gadgets were implemented from scratch and can be incorporated into more complex gadgets.

Outline. We first go over preliminaries that are necessary for this work in Section 2. Then we present related work regarding masking conversion and formal verification tools in Section 3. Next, Section 4 contains an overview of MASKVERIF gadgets, verification algorithms and the gadgets for the t–SNI addition algorithm by Coron et al. [CGV14]. Section 5 goes into detail on how we fixed the verification of first-order B2A algorithms from Goubin [Gou01] and Coron [Cor17]. Lastly, our documentation and explanation regarding MASKVERIF's codebase can be found in Section 6.

2 Preliminaries

First, we briefly explain the basics of a *differential power analysis* attack and then go over the side-channel countermeasure *masking*. Afterward, we discuss different security notions used to categorize side-channel security and lastly discuss *formal verification* and its use cases for the design of side-channel protected algorithms.

2.1 Differential Power Analysis

Differential power analysis (DPA) was first introduced by Kocher et al. [KJJ99] and is a side-channel attack that analyzes the power consumption of the hardware running code of a target to obtain secret information. To perform a DPA attack, a target device containing secret information and running an unprotected cipher is needed. Furthermore, the attacker must be able to measure the power consumption of the device while running the cipher. One example of such target devices is smart cards, which Messerges et al. examined regarding power analysis attacks in [MDS99a]. To extract sensitive information from the device, we analyze the correlation between a hypothesis regarding key candidates and the actual power consumption. First, an attacker collects numerous power traces of the cryptographic device while it is running (known input) data. Using these traces, the attacker correlates the power consumption at any fixed moment in time. Each hypothesis simulates the power consumption of the given leaking function for the (known) input and guessed secret data. The actual attack and analysis can be separated into the following five steps:

1. **Choosing an Intermediate Result of the Executed Algorithm:** The first goal is to find an attack vector, e.g., a function call f that processes some known non-constant data d and a small part s of the secret key sk. This can be written as $f(d, s)$ where the only unknown is s, which we will later try to reveal.

2. **Measuring the Power Consumption:** The next step is to measure the power consumption to obtain power traces or in short just traces, consisting of q sample points of a run of the scheme. This procedure is repeated n times to receive n traces for different known values.

3. **Calculating Hypothetical Intermediate Values:** Next, we have to calculate the hypothetical output of the function f for each of the n runs. As the function input, we have the known inputs d for each of the n traces and all possible values for the key, denoted as key guess $s^* \in G$, where G are all possible key guesses. We obtain intermediate values which we use for our attack, by calculating $f(d, s^*)$, for all $s^* \in G$.

4. **Mapping Intermediate Values to Power Consumption Values:** The next step is to convert the hypothetical values from Step 3 to realistic power consumption values by using simulation techniques. Applying these simulation techniques, e.g., the Hamming weight or Hamming distance, gives us usable values for correlation with our power traces.

5. **Comparing the Hypothetical Power Consumption Values with the Power Traces:** The last step is to use a correlation function on the values from Step 4 with the measured power traces from Step 2. This gives us values indicating the correlation between our hypothesis regarding a key guess and the power traces, over the course of the execution. The highest correlation value (negative or positive depending on the practical setup) points to the key guess with the highest probability of being used during the execution on the device.

   If the highest value is either not the correct key or the difference in correlations is small, either more traces are needed to see a good result, or the hypothesis is not well-chosen and will not lead to a high correlation.

2.2 Masking

Masking is a *side-channel countermeasure* that hides a sensitive value x belonging to a group with operator $\circ$, depending on the circuit by splitting it into n individual shares, through the usage of a *secret sharing scheme*. For such a sharing, the following must hold [KLRBG22]:

$$x = x_0 \circ ... \circ x_{n-1}.$$

All but one of the shares are random, while the last value x_{n-1} is chosen such that the original value x is computed. The type of masking used and the circuit being shared dictate what group operator $\circ$ is used. Should $\oplus$ be chosen as the group operator this is called *Boolean masking*, in the case of an *addition* or *multiplication* we call this *arithmetic masking*.
To compute a sharing of a secret value x of k bit size the following procedure `share` can be used:

Algorithm 1: An algorithm that splits a secret x into n shares.

Input: A secret value x that is split into n shares.
Output: n independent shares $x_0, ..., x_{n-1}$ of secret x.

function `share(`x`)`**:**
    $x_1, ..., x_{n-1} \leftarrow_{\$} \{0,1\}^k$
    $x_0 \leftarrow x \circ x_1 \circ ... \circ x_{n-1}$
return $(x_0, ..., x_{n-1})$

For recombination of a secret value, the procedure `combine` in Algorithm 2 can be used:

Algorithm 2: An algorithm that combines n shares into the secret value x.

Input: n independent shares $x_0, ..., x_{n-1}$ of secret x.
Output: The secret value x.

function `combine(`$x_0, ..., x_{n-1}$`)`**:**
    $x \leftarrow x_0 \circ ... \circ x_{n-1}$
return x

A function f that processes the intermediate value x also needs to have a sharing that fulfills the following equality [KLRBG22]

$$f(x) = \texttt{combine}(f(x_0), ..., f(x_{n-1})),$$

meaning that the function now operates on shares that, when recombined, are equivalent to the original function computed on the secret input.
The type of masking depends on the operations used in a function, when masked cryptographic implementations use both Boolean and arithmetic operations, a conversion between both masking types becomes necessary. Examples of such implementations are masked versions of Kyber [BGR+21], Dilithium [MGTF19], SABER [KDVB+22], and NTRU [CGTZ23].

2.3 Security Notions

To assess the security of cryptographic implementations we need formal models. These allow us to verify a masking scheme and make security guarantees regarding specific adversarial capabilities.
One security model of particular interest is the *ISW model* introduced by Ishai et al. [ISW03], which allows an attacker to read up to t intermediate values on the wires of the circuit. Protecting a circuit in this model can be done by splitting a sensitive input value into at least $t + 1$ independent shares, such that no subset of shares leaks anything about the

secret value, using an appropriate *secret-sharing scheme*.

In the following, we will go over the most used side-channel security notions from the literature.

Definition 1 t–probing security [ISW03, FGMDP$^+$18]: A circuit gadget **G** is t–probing secure iff every $t-$tuple of its intermediate variables is independent of any sensitive variable.

Definition 2 t–non–interference (t–NI) [FGMDP$^+$18]: A gadget **G** is t–non–interference iff for any set of t_1 probes on its intermediate values and every set of t_2 probes on its output shares with $t_1 + t_2 \leq t$, the totality of the probes can be simulated with $t_1 + t_2$ shares of each input.

This means that for a circuit using a sharing of at least $t + 1$ independent values, a distinguisher is unable to differentiate between an attacker's view that has an arbitrary position of probes and a simulator's view generated only from input shares. Before the computation of a circuit, a sharing of the secret value x into $t + 1$ values is generated in a way that any combination of t shares is independent and identically distributed. To generate such a sharing, Algorithm 1 can be used. Therefore, all intermediate values that the attacker obtains can be simulated from t input shares and since $t + 1$ shares are necessary to obtain the secret input, the attacker obtains no sensitive information.

We can conclude from this, that a t–probing attacker can not learn anything about a $(t + 1)$-shared circuit that is t–NI. However, when multiple t–NI circuits are composed, the resulting circuit is not guaranteed to be t–NI [FGMDP$^+$18]. For this purpose, the property of t–strong non–interference was defined, which guarantees that the composition of multiple t–SNI circuits is also t–SNI.

Definition 3 t–strong non–interference (t–SNI) [FGMDP$^+$18]: A gadget **G** is t–SNI iff for any set of t_1 probes on its intermediate values and every set of t_2 probes on its output shares with $t_1 + t_2 \leq t$, the totality of the probes can be simulated with t_1 shares of each input.

The t–SNI property tightens the t–NI property further by only allowing the simulator to use an input share for each intermediate probe. This means that in the absence of intermediate probes, the distribution of outputs will be uniform and independent of the input values. However, in the presence of intermediate probes, the distribution of outputs is entirely dependent on the probed internal wires. According to Barthe et al., the much-desired composability of t–SNI can be achieved by using mask refresh gadgets at specific

positions between t–NI gadgets [BBD+16]. An alternative security notion that also offers composability is called t–probe–isolating non–interference, which defines restrictions on the positions of probes.

Definition 4 t–probe–isolating non–interference (t–PINI) [CS20]: Let **G** be a gadget over n shares and P a set of t_1 probes on wires of **G** (called internal probes). Let A be a set of t_2 share indices. The set $P \cup y^G_{A,*}$ denotes the combination of internal wires P and probes on the share indices A of the outputs y belonging to gadget **G**. Additionally, the set $x^G_{A\cup B,*}$ denotes the set of inputs x belonging to the share indices $A \cup B$ of gadget **G**. The gadget **G** is t–probe–isolating non–interference iff for all P and A such that $t_1 + t_2 \leq t$, there exists a set B of at most t_1 share indices such that probes on the set $P \cup y^G_{A,*}$, can be simulated with the wires $x^G_{A\cup B,*}$.

The general idea is that a gadget is t–PINI if a set B of at most t_1 share indices exists, such that the union of any t_1 internal probes in set P and any outputs with t_2 share indices in set A, can be simulated from the inputs with share indices $A \cup B$. This means that the restriction is now on the positions of the probes, meaning the indices of the shares, instead of the number of probes, like in the t–SNI property.

Cassiers et al. have also proven that any t–PINI gadget (with a number of shares $n > t$) is t–probing secure [CS20]. Interestingly, t–PINI offers a very simple composition property, where a gadget composed only of t–PINI gadgets is also t–PINI. Notably, this avoids the expensive mask refresh gadgets that the t–SNI property requires and can offer a performance increase when building masked circuits if many refresh gadgets would be necessary.

While we have taken a look at the ISW model and different security notions based on it, physical defaults occurring on concrete implementations can cause masking schemes which are proven secure in the ISW model to be vulnerable against side-channel attacks [MPG05]. Two of these physical defaults that the MASKVERIF tool considers are *glitches* and *transitions*.

Glitches describe the behavior of information not propagating simultaneously during an execution, leading to additional dependencies between an instruction and its predecessors [MPG05, BGG+15]. Transitions occur when a value in a register is overwritten by another value and the resulting leakage then depends on both values [BGG+15].

Now that we have taken a look at the relevant security notions we can talk about formal verification and its benefits.

2.4 Formal Verification

Developing side-channel secure implementations is an important topic of research, but the process is tedious and error-prone. Oftentimes many cycles of design and verification are necessary to develop such protected implementations.
To assist researchers and offer a fast and reliable method of identifying vulnerabilities in masked implementations, tools for formal verification were developed. Barthe et al. [BBD+15] published the first formal verification tool that was able to verify gadgets in the ISW model. Over the years, this topic was researched extensively and many more tools with different capabilities for very specific use cases were developed [RBSS+21, KSM20, RBFSG22, BMRT22, BBD+16, BBC+19, Cor18]. However, a bottleneck of these formal verification tools is that they use algorithms to verify t probe positions for which the number of t-tuples grows exponentially. Therefore, current approaches for formal verification face inherent performance limitations and every tool has different strategies to allow for relatively efficient verification. Consequently, the correct tool for a respective verification needs to be chosen appropriately, since many tools have a trade-off between completeness and performance. This means that developing new tools that allow for faster, complete verification of newly published security notions is an important topic of research.

3 Related Work

In the first part of this section, we discuss related work in the field of Boolean-to-arithmetic and arithmetic-to-Boolean conversion and offer an overview of algorithms used in the literature. Then we talk about numerous formal verification tools, explain some of their capabilities regarding the different security notions, and discuss what some of their use cases are.

3.1 Boolean to Arithmetic Conversion (B2A)

In 2001, Goubin published algorithms to convert between arithmetic and Boolean masking and has proven his algorithms to be secure against DPA [Gou01]. First, we need to look at Goubin's definition of Boolean and arithmetic masking, so that we can take a deeper look at his conversion algorithms:

$$\begin{aligned} &\text{Boolean masking: } x = x_0 \oplus x_1, \\ &\text{Arithmetic masking: } x = A_0 + A_1 \bmod 2^k. \end{aligned}$$

Here, x is the sensitive value, x_1 and A_1 respectively are sampled uniformly at random, and x_0 or A_0 respectively are chosen such that they fulfill the equation of the respective masking type. All arithmetic operations we perform are considered mod 2^k, where a processor is assumed to have k-bit registers (in practice the values for k are usually 8, 16, 32 or 64).

3.1.1 Goubin's B2A Algorithm

The B2A algorithm uses the function $B2A(x_0, x_1) : \mathbb{F}_{2^k} \times \mathbb{F}_{2^k} \mapsto \mathbb{F}_{2^k}$ to generate an arithmetic sharing, such that:

$$B2A(x_0, x_1) = (x_0 \oplus x_1) - x_1 \bmod 2^k.$$

Theorem 1 (Goubin [Gou01]) *The function $B2A(x_0, x_1)$ is affine with respect to x_1 over $\mathbb{F}_2$.*

Goubin uses this property to devise an algorithm (Algorithm 1 in [Gou01]), that performs B2A conversion secure against first-order attacks with a constant runtime of $\mathcal{O}(1)$ independent of input size. Due to this affine property, an arithmetic sharing can also be generated securely by using a random masking value. Given two shares x_0, x_1, which fulfill the condition $x = x_0 \oplus x_1$, and r, a randomly sampled value of $\mathbb{F}_{2^k}$, we calculate an arithmetic sharing that fulfills $x = A_0 + A_1 \bmod 2^k$. Since all the arithmetic operations we talk about in this thesis are performed mod 2^k, we will omit the mod 2^k from this point forward.

$$\begin{aligned} A_0 &= (x_0 \oplus x_1) - x_1 = B2A(x_0, x_1) = B2A(x_0, x_1 \oplus r \oplus r) \\ &= B2A(x_0, r \oplus (x_1 \oplus r)) = B2A(x_0, r) \oplus B2A(x_0, x_1 \oplus r) \oplus B2A(x_0, 0) \\ &= ((x_0 \oplus r) - r) \oplus x_0 \oplus ((x_0 \oplus x_1 \oplus r) - (x_1 \oplus r)) \end{aligned}$$

This calculation can be written as an algorithm that can be seen in Algorithm 3.

Algorithm 3: **Goub B2A** rewritten version of Goubin's Algorithm 1 in Section 3.2 [Gou01]

Input: (x_0, x_1) such that $x = x_0 \oplus x_1$.
Output: (y_0, y_1) such that $x = y_0 + y_1$.

1 $r \leftarrow_{\$} \{0,1\}^k$
2 $t_1 \leftarrow x_0 \oplus r$
3 $t_2 \leftarrow t_1 - r$
4 $t_3 \leftarrow t_2 \oplus x_0$
5 $t_4 \leftarrow x_1 \oplus r$
6 $t_5 \leftarrow t_4 \oplus x_0$
7 $t_6 \leftarrow t_5 - t_4$
8 $t_7 \leftarrow t_3 \oplus t_6$
9 $y_0 \leftarrow t_7$
10 $y_1 \leftarrow x_1$
11 **return** (y_0, y_1)

In Section 5, we will examine Algorithm 3 further and evaluate its security. However, as we will later, see this approach only achieves t–probing security.

Since Goubin's approach only works for first-order masking, Coron et al. extended this approach to allow for the secure conversion of B2A masking of any order [CGV14]. They present a B2A algorithm (Algorithm 6 in [CGV14]) with complexity $\mathcal{O}(n^2k)$, where n and k are the number of shares and the register size, respectively. The previously listed algorithms, however, only work for arithmetic moduli of 2^k. Thus, more generic algorithms were developed [BBE+18, SPOG19] to allow for conversion even when arithmetic moduli

Table 3.1: Operation count for B2A conversions mod 2^k for (**A**) Goubin [Gou01], (**B**) Coron [Cor17], (**C**) Bettale et al. [BCZ18] and (**D**) Schneider et al. [SPOG19]. Here n is the number of shares and k is the register size.

	n	2	3	4	5	6	7	9	11
$\forall k$	(**A**)	7	-	-	-	-	-	-	-
$\forall k$	(**B**)	9	55	155	367	803	1 687	7 039	28 519
$\forall k$	(**C**)	9	49	123	277	591	1 225	5 053	20 401
$k = 1$	(**D**)	12	33	63	102	150	207	348	525
$k = 8$	(**D**)	156	354	624	966	1 380	1 866	3 054	4 530
$k = 32$	(**D**)	636	1 434	2 520	3 894	5 556	7 506	12 270	18 186

are used that are not of a power of two. Additionally, Coron et al. published another conversion algorithm that has complexity $\mathcal{O}(2^n)$ and is therefore independent of register size and very efficient for small n [Cor17]. Coron's algorithm was further improved upon by Bettale et al. [BCZ18] requiring about 50% less random values and achieving a lower operation count, despite having the same runtime complexity of $\mathcal{O}(2^n)$. This means that choosing a suitable conversion algorithm depends on the specific order n and register size k of the larger scheme. In Table 3.1, we compare the operation count of algorithms from Coron [Cor17], Bettale et al. [BCZ18], and Schneider et al. [SPOG19].

As Table 3.1 shows, the algorithm by Schneider et al. performs very well for small k and increasingly larger n. However, as k grows even small orders require a lot more operations. The advantage of Coron's and Bettale et al.'s improvement of Coron's algorithm is that they are independent of k, making it very efficient for large k and small orders n. A downside is that the complexity of both algorithms is exponential in n and the performance suffers substantially as the number of shares increases.

3.1.2 Coron's B2A Algorithm

In 2017, Coron [Cor17] published an algorithm that uses Goubin's previously described algorithm as a subroutine. By applying additional randomness, Coron creates a version of Goubin's algorithm that achieves the stronger t–SNI property for $t = 1$. The general idea behind the higher-order conversion is to recursively apply the t–SNI Goubin subroutine to turn Boolean shares into arithmetic ones. A crucial component of the higher-order conversion is *mask refreshing*, which means applying new randomness to each share such that $x = x_0 \oplus ... \oplus x_{n-1}$ still holds. We will walk through the higher-order conversion algorithm by first examining the t–SNI Goubin subroutine, then looking at an insecure version of the higher-order conversion algorithm that highlights the idea and finally com-

bining this with applications of the mask refreshing to achieve a t–SNI higher-order B2A conversion algorithm.

First-Order SNI Algorithm

To extend Goubin's algorithm, Coron adds randomness to both input shares x_0 and x_1 before computing the rest of Goubin's algorithm. This modification can be written as the following algorithm:

Algorithm 4: SNI Conv rewritten version of Coron's Algorithm 1 in Section 3.2 [Cor17]

Input: (x_0, x_1) such that $x = x_0 \oplus x_1$.
Output: (A_0, A_1) such that $x = A_0 + A_1$.

1 $r, s \leftarrow_{\$} \{0,1\}^k$
2 $a_0 \leftarrow x_0 \oplus s$
3 $a_1 \leftarrow x_1 \oplus s$
4 $t_1 \leftarrow a_0 \oplus r$
5 $t_2 \leftarrow t_1 - r$
6 $t_3 \leftarrow t_2 \oplus a_0$
7 $t_4 \leftarrow a_1 \oplus r$
8 $t_5 \leftarrow t_4 \oplus a_0$
9 $t_6 \leftarrow t_5 - t_4$
10 $t_7 \leftarrow t_3 \oplus t_6$
11 $A_0 \leftarrow t_7$
12 $A_1 \leftarrow a_1$
13 **return** (A_0, A_1)

The intermediate variables can be simulated with the knowledge of either x_0 or x_1, similar to Goubin's conversion algorithm. Additionally, we need to show that both outputs A_0, A_1 can be perfectly simulated without the use of any input shares. The value $A_1 = x_1 \oplus s$ is uniformly distributed over $\{0,1\}^k$ since an $\oplus$ operation between the original share and a uniformly distributed random variable is performed.

The converted share $A_0 = (a_0 \oplus a_1) - a_1$ can also be expressed as $A_0 = (x_0 \oplus s \oplus x_1 \oplus s) - (x_1 \oplus s) = (x_0 \oplus x_1) - (x_1 - s)$. Since $x_1 \oplus s$ is uniformly distributed, subtracting it from $x_0 \oplus x_1 = x$ also results in a uniformly distributed value, because the subtraction is a bijection. Therefore, both outputs can be simulated without the usage of any input shares, meaning that Coron's algorithm fulfills the t–SNI property.

Insecure Higher-Order Conversion Algorithm

This insecure version of a higher-order conversion algorithm is described by Coron in Section 4.1 of [Cor17] to illustrate the basic idea behind the higher-order t–SNI algorithm. Assuming we have n Boolean shares $x_0, ..., x_{n-1}$ such that

$$x = x_0 \oplus ... \oplus x_{n-1},$$

we want to convert them to n arithmetic shares $A_0, ..., A_{n-1}$, such that

$$x = A_0 + ... + A_{n-1}.$$

Coron now recursively splits off Boolean shares into arithmetic shares by computing the $B2A$ function on the share being split off and all remaining Boolean shares. An example for the share x_0 looks like this:

$$x = x_1 \oplus ... \oplus x_{n-1} + ((x_0 \oplus x_1 \oplus ... \oplus x_{n-1}) - (x_1 \oplus ... \oplus x_{n-1})).$$

Using the notation we established for Goubin's conversion we can write:

$$x = x_1 \oplus ... \oplus x_{n-1} + B2A(x_0, x_1 \oplus ... \oplus x_{n-1}).$$

Since the B2A function is affine, we can split up the right side of the addition and obtain:

$$x = x_1 \oplus ... \oplus x_{n-1} + (n \wedge 1) \cdot x_0 \oplus B2A(x_0, x_1) \oplus ... \oplus B2A(x_0, x_{n-1}).$$

At this point, we have $n-1$ terms on both sides of the addition and keep converting these by recursively using the same algorithm on both sub-terms to convert a single share at a time. This gives us $2n-2$ arithmetic shares of the following form:

$$x = A_0 + ... + A_{n-2} + B_0 + ... + B_{n-2}.$$

Since Coron's goal is to obtain n arithmetic shares, he now adds up some of these terms to reduce the amount of shares.

Let $D_i \leftarrow A_i + B_i$ for $0 \leq i \leq n-3$ and $D_{n-2} \leftarrow A_{n-2}$ and $D_{n-1} \leftarrow B_{n-2}$. This results in n arithmetic shares, such that

$$x = D_0 + ... + D_{n-1}.$$

However, as Coron states, this algorithm is insecure. The value $B2A(x_0, x_1 \oplus ... \oplus x_{n-1})$ can be obtained by placing $n-1$ probes into the values of the term $(n \wedge 1) \cdot x_0 \oplus B2A(x_0, x_1) \oplus \ ... \oplus \ B2A(x_0, x_{n-1})$.
Now the adversary can check the distribution of $B2A(x_0, x_1 \oplus ... \oplus x_{n-1})$ and $B2A(x_0, x_{n-1})$, which depends on x. Furthermore, this attack can even be applied with a single probe, by targeting the recursive part of the conversion. A recursive call with $n-1$ values leaks information about its variable with $n-2$ probes. By attacking the deepest point in the recursion, the attacker can use a single probe to obtain a value that depends on the secret value x.

Secure Conversion from Boolean to Arithmetic Masking

To secure this algorithm, Coron uses *mask refreshing* at specific points in the algorithm to remove the dependency within the recursive calls. The refresh algorithm that Coron uses was published by Rivain et al. [RP10].

Algorithm 5: RefreshMasks algorithm used by Coron

Input: $x_0, ..., x_{n-1}$ such that $x = x_0 \oplus ... \oplus x_{n-1}$.
Output: $y_0, ..., y_{n-1}$ such that $x = y_0 \oplus ... \oplus y_{n-1}$.

$y_{n-1} \leftarrow x_{n-1}$
for $i = 0$ *to* $n-2$ **do**
    $r_i \leftarrow_{\$} \{0,1\}^k$
    $y_i \leftarrow x_i \oplus r_i$
    $y_{n-1} \leftarrow y_{n-1} \oplus r_i$
end
return $(y_0, ...y_{n-1})$

The **RefreshMasks** algorithm shown in Algorithm 5 generates n shares, where every combination of at most $n-1$ shares is independent of the original input shares $x_0, ..., x_{n-1}$. The security of Coron's higher-order conversion algorithm hinges on the security properties of **RefreshMasks**. Coron describes his algorithm $\mathcal{C}_n$ recursively in the following fashion: Initially, we have n Boolean shares $x = x_0 \oplus ... \oplus x_{n-1}$. In the case $n = 2$, the t–SNI version of Goubin's algorithm described in Algorithm 4 is used. Otherwise, we do the following:

1. Perform a **RefreshMasks** on the input shares with an additional value $x_n = 0$ to get $n + 1$ shares:

$$a_0, ..., a_n \leftarrow \textbf{RefreshMasks}(x_0, ..., x_n).$$

This maintains the equality $x = a_0 \oplus ... \oplus a_n$, therefore, we can now use these shares for conversion. Just as in the insecure example, we now split off the first share and convert it using Goubin's B2A formula:

$$\begin{aligned} x &= a_1 \oplus ... \oplus a_n + ((a_0 \oplus a_1 \oplus ... \oplus a_n) - (a_1 \oplus ... \oplus a_n)) \\ &= a_1 \oplus ... \oplus a_n + B2A(a_0, a_1 \oplus ... \oplus a_n). \end{aligned}$$

2. Since $B2A$ is affine we can also write this as:

$$x = a_1 \oplus ... \oplus a_n + \overline{(n \wedge 1)} \cdot a_0 \oplus B2A(a_0, a_1) \oplus ... \oplus B2A(a_0, a_n).$$

For readability, let $b_0 \leftarrow \overline{(n \wedge 1)} \cdot a_0 \oplus B2A(a_0, a_1)$ and let $b_i \leftarrow B2A(a_0, a_{i+1})$, where $1 \leq i \leq n - 1$. Using this notation, we get:

$$x = a_1 \oplus ... \oplus a_n + b_0 \oplus ... \oplus b_{n-1}.$$

3. Now a **RefreshMasks** of all a_i and b_i is performed:

$$\begin{aligned} c_0, ..., c_{n-1} &\leftarrow \textbf{RefreshMasks}(a_1, ..., a_n), \\ d_0, ..., d_{n-1} &\leftarrow \textbf{RefreshMasks}(b_0, ..., b_{n-1}). \end{aligned}$$

4. Next the two last shares of the c_i's and d_i's are xored together to compress the number of shares down to $n - 1$. Let $e_i \leftarrow c_i$ and $f_i \leftarrow d_i$ for $0 \leq i \leq n - 3$ and $e_{n-2} \leftarrow c_{n-2} \oplus c_{n-1}$ and $f_{n-2} \leftarrow d_{n-2} \oplus d_{n-1}$. We now have $2(n - 1)$ shares that fulfill the equation:

$$x = e_0 \oplus ... \oplus e_{n-2} + f_0 \oplus ... \oplus f_{n-2}.$$

5. At this point, two recursive calls to $\mathcal{C}_{n-1}$ are performed to convert the rest of the shares:

$$A_0, ..., A_{n-2} \leftarrow \mathcal{C}_{n-1}(e_0, ..., e_{n-2}),$$
$$B_0, ..., B_{n-2} \leftarrow \mathcal{C}_{n-1}(f_0, ..., f_{n-2}).$$

   This results in $2(n-1)$ arithmetic shares such that:

$$x = A_0 + ... + A_{n-2} + B_0 + ... + B_{n-2}.$$

6. Finally, we reduce these $2(n-1)$ into n shares. Let $D_i \leftarrow A_i + B_i$ for $0 \leq i \leq n-3$ and $D_{n-2} \leftarrow A_{n-2}$ and $D_{n-1} \leftarrow B_{n-2}$. Our final result is an arithmetic sharing with n shares for which the following equality holds:

$$x = D_0 + ... + D_{n-1}.$$

As Coron shows in Section 4, Theorem 3 of [Cor17], this algorithm achieves the t–SNI property.

3.2 Arithmetic to Boolean Conversion (A2B)

Goubin also developed an A2B algorithm to convert a first-order arithmetic sharing into a valid Boolean sharing. Theorem 2 describes the recursion formula that can be used to convert arithmetic to Boolean shares in an unmasked fashion.

Theorem 2 (Goubin [Gou01]) *If we denote $x_0 = (A_0 + A_1) \oplus A_1$, we also have $x_0 = A_0 \oplus u_{K-1}$, where u_{K-1} is obtained from the following recursion formula:*

$$\begin{cases} u_0 = 0 \\ \forall k \geq 0,\ u_{k+1} = 2[u_k \wedge (A_0 \oplus A_1) \oplus (A_0 \wedge A_1)]. \end{cases}$$

To secure this algorithm Goubin slightly altered this theorem into a version that uses a random value to mask the intermediate values. Goubin describes this in the following corollary:

Corollary 2.1 (Goubin [Gou01]) *For any value r, if we denote $x_0 = (A_0 + A_1) \oplus A_1$, we also have $x_0 = A_0 \oplus 2r \oplus t_{K-1}$, where t_{K-1} is obtained from the following recursion formula:*

$$\begin{cases} t_0 = 2r \\ \forall k \geq 0,\ t_{k+1} = 2[t_k \wedge (A_0 \oplus A_1) \oplus w], \end{cases}$$

in which $w = r \oplus (2r) \wedge (A_0 \oplus A_1) \oplus A_0 \wedge A_1$.
Using this formula, Goubin created an A2B algorithm (Algorithm 2 in [Gou01]) that can be described as following:

Algorithm 6: **Goub_A2B** rewritten version of Goubin's Algorithm 2 in Section 4.2 [Gou01]

Input: (A_0, A_1) such that $x = A_0 + A_1$.
Result: (x_0, x_1) such that $x = x_0 \oplus x_1$.

$r \leftarrow_{\$} \{0,1\}^k$
$t \leftarrow 2r$
$a \leftarrow A_1 \oplus r$
$b \leftarrow a \wedge r$
$a \leftarrow t \oplus A_0$
$c \leftarrow a \oplus r$
$c \leftarrow c \wedge A_1$
$b \leftarrow b \oplus c$
$c \leftarrow t \wedge A_0$
$b \leftarrow b \oplus c$
for $i = 1$ *to* $k - 1$
do
    $c = t \wedge A_1$
    $c \leftarrow c \oplus b$
    $t \leftarrow t \oplus A_0$
    $c \leftarrow c \oplus t$
    $t \leftarrow 2c$
end
$x_0 \leftarrow a \oplus t$
$x_1 \leftarrow A_1$
return (x_0, x_1)

In Section 4.3 of his work, Goubin presents a security proof for this algorithm and shows that all intermediate values depend on at most one secret share. As with the B2A algorithm, Goubin did not include the outputs of his algorithm in his security proof. However,

since $x_0 = (A_0 + A_1) \oplus A_1$ does not contain any randomness to hide either share, the distribution of x_0 depends on both shares. This means that the algorithm does not fulfill the t–NI property and is only t–probing secure, since $x_0 = (A_0 + A_1) \oplus A_1 = x \oplus A_1$ is uniformly distributed if A_1 is a uniformly sampled random value.

This approach does not work for higher-order conversion and was also extended to arbitrary orders by Coron et al. [CGV14]. The first iteration of their A2B algorithm has complexity $\mathcal{O}(n^3k)$, where n and k are the number of shares and the register size, respectively. After introducing some small changes, the complexity is reduced to $\mathcal{O}(n^2k)$. However, again these conversion algorithms only work for arithmetic moduli 2^k. The more generic approach by Barthe et al. [BBE+18] can be used for arithmetic moduli that are not a power of two.

3.3 Verification Tools

After talking about security notions, masking, and conversion between Boolean and arithmetic masking, we now examine some tools used to formally verify masked implementations in practice. Generally, three types of tools exist. The first are tools focused on passive attacks, called *side-channel analysis* (SCA) that consider what information an attacker can perceive by obtaining intermediate values. The different security notions highlighted in Section 2.3 fall under this category and are verified by a multitude of tools. The second type of tools are focused on active attacks like *fault injection analysis* (FIA) and explore what information an attacker can obtain by introducing faults into a target. It is important to note that different security notions exist for FIA [FRBSG22], however, we will not go over them here since MASKVERIF, the tool we focus on, does not include those notions and only checks security regarding passive attackers.

3.3.1 Side-Channel Security Tools

MASKVERIF

The security of higher-order masked implementations can be verified using MASKVERIF in an automated fashion, even in the presence of physical defaults (glitches and transitions). This tool was developed in three iterations. The first one was published in 2015 [BBD+15], the second one in 2016 [BBD+16], and the latest one in 2019 [BBC+19]. The first version of the tool was only able to verify t–probing security and the second version added t–NI and t–SNI verification. The latest version added verification of glitches and transitions, is able to parse Verilog implementations, and has improved performance.

Barthe et al. use language-based approaches with a divide-and-conquer approach, embodied in two algorithms. The first one checks if leakage is independent of secrets for fixed sets of observations and the second one explores admissible observation sets and calls the first algorithm on them.
The tool makes use of a relevant model from the literature, the ISW model, as well as the versions of the model including glitches and transitions, which are supposed to model additional leakage in hardware or software implementations. Additionally, MASKVERIF evaluates implementations regarding the three security properties t–probing security, t–NI, and t–SNI. The order at which MASKVERIF can verify gadgets depends strongly on the complexity of the gadget.
Due to its wide range of features and its efficiency, MASKVERIF is a state-of-the-art tool to verify masked implementations regarding side-channel security. Its main flaw, however, is the occurrence of false negatives, declaring some implementations not secure (despite the algorithms being proven secure), which can be solved by introducing more randomness that in reality is unnecessary. An example of this was shown by Knichel et al. [KSM20] for a shared version of the 4-bit bijection quadratic class $\mathcal{Q}_{12}^4$, as described by Bilgin et al. [BNN+15]. The version used for verification has two shares per input, as presented in the Appendix of [RBN+15], and is classified as not being t–probing secure by MASKVERIF, although all possible probes are statistically independent of secrets. Knichel et al. also show a modified example of this design that has additional randomness added to it and is correctly verified by MASKVERIF. These false positives occur mostly when MASKVERIF verifies non-linear designs and are a downside of the non-complete, but very performant, approach that MASKVERIF uses for verification.
Currently, MASKVERIF does not support arithmetic-to-Boolean masking conversion or vice versa, which is an important functionality in various masked cryptographic implementations, specifically post-quantum cryptographic schemes like masked versions of Kyber [BGR+21], Dilithium [MGTF19], SABER [KDVB+22], and NTRU [CGTZ23].

CHECKMASKS

This tool was published by Coron in 2018 [Cor18] and uses an approach similar to the first version of MASKVERIF. One of the differences, however, is the representation of the underlying circuit that Coron uses. Possible observations are represented as nested lists in CHECKMASKS, where MASKVERIF uses recursively defined expressions to build its state of observations.
Coron adds a number of rules to his tool that allow him to also verify Boolean-to-arithmetic masking for the algorithms proposed in [Gou01, Cor17]. The verification works for orders up until $n = 6$ at which point it does not finish since the higher-order

algorithm has complexity $\mathcal{O}(2^n)$ and the amount of possible $(n-1)$ tuples of intermediate variables is $\mathcal{O}(2^{n^2})$ according to Coron. Additionally, Coron has added rules that improve the verification of mask-refreshing gadgets. This tool does not see very much use in the literature and is generally not compared much to state-of-the-art verification tools in the literature. However, it was used for the verification of an improved version of Coron's B2A algorithm by Bettale et al. [BCZ18] and the improved high-order masking of look-up tables countermeasure by Coron et al. [CRZ17]. Overall this tool is an alternative to MASKVERIF when B2A conversion is part of the tested circuit.

IRONMASK

IRONMASK is a verification tool developed by Belaïd et al. [BMRT22] for implementations protecting against passive attacks. One of IRONMASK's core features is that it can verify gadgets in the random probing model while guaranteeing completeness. The *random probing model* states that each wire in a circuit leaks its value independently with a probability p during its evaluation [BCP+20]. This model is interesting since it bridges the gap from the *noisy leakage model* to the *t-probing model* [DDF14]. The noisy leakage model best describes the physical reality of side channels but is very cumbersome to evaluate security proofs in. The random probing model offers a compromise between the neat and theoretical t–probing model and the noisy probing model. For standard probing security notions (t–NI, t–SNI) IRONMASK outperforms SILVER, another formal verification tool also offering completeness, by several orders of magnitude. When compared to MASKVERIF the speed depends on the use case. IRONMASK outperforms MASKVERIF on multiplication gadgets but is much slower on mask refreshing gadgets. To establish how viable IRONMASK is as an alternative to MASKVERIF for standard probing security a more detailed comparison of a variety of gadget types would be necessary.

SILVER

SILVER was developed by Knichel et al. [KSM20] and uses exhaustive analysis of probability distributions to verify the security of masked circuits and implementations. The tool receives a Verilog implementation or instruction list and can verify it in the ISW model even when glitches are present. Security properties from the literature like t–probing, t–NI, t–SNI, and t–PINI can be verified. All of this is possible without the occurrence of false negatives since SILVER offers completeness at the cost of speed. According to the authors, this makes it a viable competitor to MASKVERIF, despite being slower and less efficient for larger designs.

3.3.2 Fault Injection Security Tools

FIVER

FIVER aims to detect vulnerabilities produced by an active attacker through FIA. This tool was developed by Richter-Brockmann et al. [RBSS+21] in 2021 and used as a building block for the combined security analysis that VERICA performs. The protected cryptographic algorithm is given in the form of a gate-level netlist to create a model of the underlying logic. The verification uses symbolic fault injection to model all possible fault events that can occur for a given logic circuit, meaning that FIVER fulfills completeness. Since this tool was published Richter-Brockmann et al. refined and integrated it into VERICA [RBFSG22], however, as the first tool for verification of countermeasures against FIA achieving completeness it deserves a mention.

3.3.3 Combined Security Tools

VERICA

VERICA [RBFSG22] is an automated framework for the formal verification of hardware circuits under *Combined Analysis* (CA). Other tools consider security threats from SCA or FIA in isolation. VERICA, however, is the first tool to consider a combination of both. To implement this, the SCA analysis and verification is based on the SILVER tool [KSM20], and the FIA verification is based on the FIVER tool [RBSS+21]. A limitation of the tool is that with an increasing number of gates, CA becomes very hard to verify. This is due to a drastic increase in valid fault injections, which each need to be verified for side-channel security, meaning that this tool is most useful for implementations with a low order of masking and a small number of gates.

4 Introduction to MASKVERIF

This section offers further insight into some of MASKVERIF's inner workings and introduces the reader to the tool. Two different types of files can be verified using MASKVERIF. The first are MASKVERIF gadgets written as `.mv` files and the second are Verilog implementations. Verilog is a hardware description language to model electronic circuits and is often in the design and verification of digital circuits. Verification of Verilog implementations was added with MASKVERIF version three and can be useful to verify an already existing design quickly, without having to write a separate gadget. The benefit of MASKVERIF gadgets is that the user has more control over specific assignments and instructions. We provide a detailed overview of MASKVERIF gadgets and the functions used to verify them.

4.1 MASKVERIF Gadgets

A gadget starts with `proc` to indicate its start and is declared finished with `end`. The two gadgets displayed in Figure 4.1 serve as a very simple example of what MASKVERIF input files can look like. We will now examine different features and notations to comprehensively show how MASKVERIF gadgets are used.

Gadget Parameters

A gadget can have a multitude of parameters that are declared before the actual instructions are written. These can be seen in lines 11-13 and 27-30 of Figure 4.1. Note that these parameters need to be declared in this exact order, otherwise, the gadget will not be parsed correctly. All gadget parameters are summarized in Table 4.1.

Table 4.1: Parameters of a MASKVERIF gadget.

Parameters		Description
`inputs:`	`a[0:1], b[0:1]`	Number of gadget inputs and input shares
`outputs:`	`c[0:1]`	Number of gadget outputs and output shares
`shares:`	`d[0:1]`	Intermediate variables, necessary for compact assignment notation
`randoms:`	`r`	Fresh randomness for assignments

```
(* Comments look like this in maskVerif. *)

(* #or is a custom-defined operator, define number of arguments,
results and variable size in bits.
Can also be declared as "bij" in case it is a bijective operator,
which allows maskVerif to apply additional rules. *)

op #or : w1, w1 -> w1

proc example1:
    inputs: a[0:1], b[0:1]
    outputs: d[0:1]
    shares: c[0:1];

    (* c = ~a is a shortcut for c[0] = ~a[0] and
    c[1] = ~a[1] in a single assignment.
    This shortcut only works if the left side of the assignment is
    declared as a share in the inputs section. *)

    c = ~a;
    d = #or(c,b);
end

Probing example1

proc example2:
    inputs: a[0:1]
    outputs: c[0:1]
    shares: b[0:1]
    randoms: r;

    b = example1(a,a);
    c[0] = b[0] + r;
    c[1] := b[1] + r;
end

(* verbose is a debugging parameter and offers additional information.
Can be configured from 1, offering the least information, to 4,
offering maximum debug output. *)

verbose 4

(* The following indicates to maskVerif
what type of security notion is verified. *)

Probing example2
NI example2
noglitch SNI example2
SNI example2
```

Figure 4.1: Two example MASKVERIF gadgets

Inputs, outputs, and randoms are fairly straightforward. Shares, however, are an optional parameter that can be used to make gadgets easier to implement and read. Most notably, shares allow the user to write the compact assignments $c = a + b$ that MASKVERIF offers, which performs the $\oplus$ operation for all shares of a and b and saves the result into the corresponding shares of c.
A user can also use intermediate variables which are not declared as shares. However, then the compact notation will not work since MASKVERIF interprets this as a mismatch between a single variable and a shared variable. Therefore, it is helpful to carefully identify which intermediate variables of an algorithm are best declared as shares and which ones can simply be used as temporary variables.
By default, all the above parameters are 1-bit values. Should a user wish to use different values these also need to be declared. The following short gadget displays how $\oplus$ is used for 8-bit values.

```
1 proc xor_8bit:
2     inputs: w8 a[0:3], w8 b[0:3]
3     outputs: w8 c[0:3];
4
5     c := a ^w8 b;
6 end
```

Operators

In its current version MASKVERIF only supports the Boolean operators $\oplus$, $\wedge$, and $\neg$. These are available for 1, 8, 16, 32, and 64-bit values, which can be seen in Table 4.2. Additionally, we have added subtraction mod 2^k as an operator, where k is 1, 8, 16, 32, or 64.

Table 4.2: Operators in a MASKVERIF gadget.

1-bit operator	8-64 bit operators	Description
`+`	`^w8, ^w16, ^w32, ^w64`	$\oplus$ operator
`*`	`&w8, &w16, &w32, &w64`	$\wedge$ operator
$\sim$	$\sim$ `w8,` $\sim$ `w16,` $\sim$ `w32,` $\sim$ `w64`	Bitwise complement operator
`-`	`-w8, -w16, -w32, -w64`	Subtraction mod 2^k

In cases where these operators do not suffice, a user can declare custom operators. An example of this can be seen in line 8 of Figure 4.1 where we declare the $\vee$ operator for a

1-bit value. There are a couple of things to consider when declaring custom operators: they start with an optional parameter `bij`, which should be added if the custom operator is bijective. This is crucial since MASKVERIF is only able to apply one of its most important simplification rules, the *optimistic sampling rule*, which we go over in Section 4.4.1, on bijective operators. Then, we declare a custom operator with `op`, followed by the operator's name with # at the start. Additionally, the number of input parameters and their size in bits are declared, as well as the number of outputs and their size. A common mistake that can occur when writing gadgets is to have mismatches between the sizes of variables or operators, so we recommend to pay special attention when using custom operators.

Assignment Types

When writing a gadget it is very important to consider which type of assignment to use, since this greatly impacts the possible observations or, in some cases, leads to parsing errors in gadgets if used incorrectly. Five types of assignments with different leakages are supported by MASKVERIF and summarized in Table 4.3.

Table 4.3: Assignments in a MASKVERIF gadget.

Assignment	Description
`var := expr`	Hardware instruction, leaks sub-expressions, causes glitches
`var = ![expr]`	Hardware register, leaks sub-expressions, stops glitches
`var = expr`	Software instruction, leaks sub-expressions, stops glitches
`var = {expr}`	Computation result, does not leak sub-expressions, stops glitches
`var <- expr`	Instruction with no observable leakage does not stop glitches

Hardware instruction assignment The first assignment type is `var := expr`, which represents an assignment in the glitch extended probing model [FGMDP+18]. This means that the expression leaks the joint distribution of all sub-expressions that have not been written to a register yet.

An example of this is the assignment $y_0 := x_0 \oplus x_1 \oplus r$, where y_0 is an observable intermediate variable, x_0 and x_1 are a sharing of the secret x, and r is a uniformly distributed sampled value. When the attacker probes y_0 the leakage has the joint distribution (x_0, x_1, r), meaning that the attacker can retrieve the first and second components and compute $x = x_0 \oplus x_1$, recovering the secret value.

Hardware register assignment To stop the propagation of glitches a register assignment of the form `var = ![expr]` can be used. We will now split the previous assignment into two assignments to prevent the attacker from recovering the secret. The

first assignment is $t = ![x_1 \oplus r]$, leaking the distribution (x_1, r). The second assignment is $y_0 := x_0 \oplus t$, which now leaks the distribution $(x_0, x_1 \oplus r)$. Since the register stopped the propagation of glitches the leakage of $x_1 \oplus r$ is now uniformly distributed and the attacker is unable to recover the secret x from $(x_0, x_1 \oplus r)$.

Software instruction assignment Next, MASKVERIF also supports an assignment for the transition extended probing model, which aims to model a physical default in software implementations. In this case, an intermediate variable is reused and not cleared in between assignments and leaks a joint distribution of the values it was assigned. An example of the transition assignment `var = expr` can be seen in Figure 4.2.

```
1 proc transition:
2     inputs: a[0:1]
3     outputs: b[0:1]
4
5     t = a[0];
6     b[0] = t;
7     t = a[1];
8     b[1] = t;
9 end
```

Figure 4.2: A MASKVERIF gadget with transitions.

The gadget in Figure 4.2 simply takes input shares and writes them to an intermediate variable and then to an output variable. Without the consideration of transitions, every intermediate and output variable can be perfectly simulated using a single input share.

However, considering transitions, the leakage after the second assignment of t depends on both of its assignments a_0 and a_1, therefore, revealing information about the secret. A simple way to mitigate transitions is to assure that intermediate variables are not reused or are cleared in between assignments. The same gadget can be secured by rewriting it as shown in Figure 4.3.

Additionally, the `var = expr` assignment needs to be used when calling another gadget as a function. An example of this is `b = example1(a, a)` from line 32 in Figure 4.1. Using any other assignment type will lead to a parse error and should be avoided.

Computation result assignment The fourth assignment is `var = {expr}`, which represents the notion that only a computation's result without any physical defaults

```
1 proc transition_fixed:
2     inputs: a[0:1]
3     outputs: b[0:1]
4
5     t0 = a[0];
6     b[0] = t0;
7     t1 = a[1];
8     b[1] = t1;
9 end
```

Figure 4.3: A MASKVERIF gadget where transitions no longer happen.

will leak. Going back to the example from our first assignment $y_0 = \{x_0 \oplus x_1 \oplus r\}$, now y_0 will only leak the computational result which is $x_0 \oplus x_1 \oplus r$ and is uniformly distributed.

Leakage-free assignment The last supported assignment `var <- expr`, can be used to declare that an assignment is leakage free. This can be especially useful to debug gadgets that are presumed secure but evaluated as insecure. A user can pinpoint leakage by declaring other assignments leakage-free and debugging which exact assignments are problematic.

```
1 proc leakage_free:
2     inputs: a[0:1]
3     outputs: b[0:1]
4
5     t <- a[0] + a[1];
6     t2 = t;
7     b[0] = a[0];
8     b[1] = a[1];
9 end
```

Figure 4.4: A MASKVERIF gadget with a leakage-free assignment.

In the example in Figure 4.4 the assignment $t \leftarrow a_0 \oplus a_1$ would in reality leak the secret but is considered to have no observable leakage. Using this assignment, MASKVERIF will disregard the observation $t \leftarrow a_0 \oplus a_1$ and evaluate the gadget as secure. However, when we assign $t2 = t$, MASKVERIF will substitute $a_0 \oplus a_1$ for t and flag this as insecure, since the secret is leaked. The leakage-free assignment should be used very carefully and mostly for debugging purposes or in cases where the

user has complete confidence that no leakage can be observed for the assignment in question.

Verification Parameters

Multiple verification parameters can be used in MASKVERIF gadgets, these can be seen in Table 4.4.

Table 4.4: Verification parameters in a MASKVERIF gadget.

Verification parameter	Description
`verbose` 1-4	Parameter for additional debug information
`noglitch`	Assignments never cause glitches
`Probing`	Verifies t–probing security notion
`NI`	Verifies t–NI security notion
`SNI`	Verifies t–SNI security notion

The first parameter that we discuss is the `verbose` parameter. If we omit this parameter MASKVERIF only tells us if the verification succeeded or it tells us the probe positions for which the verification failed. By adding the `verbose` parameter we can receive additional information about MASKVERIF's internal state and reasoning. The amount of information gradually increases from `verbose` 1 to 4, eventually printing out all internal states and tested probe positions. We advise testing and finding a value for this that suits the purpose, since using `verbose` 4 prints an overwhelming amount of information for any nontrivial gadget.
When verifying a gadget, a user can use assignments to model the presence or absence of glitches. If glitches are not considered, the `noglitch` parameter can be added in front of the evaluation type to disable glitch leakages. In terms of security notions, a user can choose `Probing`, `NI`, or `SNI` as parameters, and MASKVERIF will try to verify the circuit for these notions to the best of its ability. It is important to note that MASKVERIF does not offer completeness. If a gadget is evaluated as secure, it is indeed secure. However, when gadgets are evaluated as insecure this can be a false negative in MASKVERIF's verification and it is up to the user to identify if the gadget is insecure or just not verified correctly.

4.2 ISW Multiplication

In 2003, Ishai et al. [ISW03] proposed the security notion of the t–probing model and proposed an algorithm for a secure multiplication gadget. This gadget is often referred to

as **ISW AND** in the literature, named after its inventors. It is a great example to get acquainted with the different security properties and to understand what the different assignment types from the previous section are useful for.
Before looking at the gadgets, we briefly explain how the **ISW AND** algorithm works. The general idea is to split up the computation $c = a \wedge b$ into $c = \sum_{i,j} a_i b_j$, where a, b are secret inputs and c is the result. The procedure that Ishai et al. described can be seen in Algorithm 7.

Algorithm 7: **ISW AND** rewritten version of Ishai et al.'s algorithm in Section 3 [ISW03]

Input: Boolean shares $(a_0, ..., a_{n-1})$ and $(b_0, ..., b_{n-1})$ of secrets a, b.
Output: $(c_0, ..., c_{n-1})$ which fulfill $c = a \wedge b = c_0 \oplus ... \oplus c_{n-1}$.

for $i = 0$ *to* $n - 1$ **do**
  $c_i = a_i \wedge b_i$
  for $j = 0$ *to* $n - 1$ **do**
    if $i \neq j$ **then**
      $r_{i,j} \leftarrow_{\$} \{0,1\}^K$
      $r_{j,i} = r_{i,j} \oplus a_i b_j \oplus a_j b_i$
      $c_i = c_i \oplus r_{i,j}$
    end
  end
end
return $(c_0, ..., c_{n-1})$

We will now show what first-order MASKVERIF gadgets implementing the **ISW AND** algorithm look like. To convert the **ISW AND** algorithm into a gadget we first need to unroll the loops, since MASKVERIF does not support any form of loop assignment. The first version of this gadget is written in a way that disregards the occurrence of glitches since the glitch-extended probing model was not considered when this algorithm was first published. When verifying the gadget in Figure 4.5 we use the `noglitch` option to completely disregard glitches no matter what kind of assignment is used.
The verification with MASKVERIF succeeds for all security notions in the standard probing model but is evaluated as not t–NI when considering glitches. The reason is that the values in aux propagate $a_0 \wedge b_1$ and $a_1 \wedge b_0$ without the additional random mask r. To fix this, the gadget in Figure 4.6 uses register assignments for aux to stop the propagation of glitches. This gadget in Figure 4.6 achieves the t–NI property. However, it is not t–SNI because c_0 and c_1 are assigned values $a_0 \wedge b_0$ and $a_1 \wedge b_1$ respectively, and propagate these values through to the output. Therefore, the outputs are not uniformly distributed and do not fulfill the t–SNI property. We can fix this by adding register assignments for both outputs to stop glitch propagation.

```
1  proc ISW_AND:
2      (* The algorithm takes two inputs each split into two shares. *)
3      inputs: a[0:1], b[0:1]
4      (* It returns one output split in two shares. *)
5      outputs: c[0:1]
6      (* r is a fresh random local to the algorithm. *)
7      randoms: r;
8
9      c[0] := a[0] * b[0];
10     a0b1 := a[0] * b[1];
11     a1b0 := a[1] * b[0];
12     c[1] := a[1] * b[1];
13     aux  := r + a0b1;
14     aux  := aux + a1b0;
15     c[0] := c[0] + r;
16     c[1] := c[1] + aux;
17 end
18
19 noglitch Probing ISW_AND
20 noglitch NI ISW_AND
21 noglitch SNI ISW_AND
22 Probing ISW_AND
```

Figure 4.5: First-order secure **ISW_AND** gadget without glitches.

```
1  proc ISW_AND_NI_G:
2      inputs: a[0:1], b[0:1]
3      outputs: c[0:1]
4      randoms: r;
5
6      c[0] := a[0] * b[0];
7      a0b1 := a[0] * b[1];
8      a1b0 := a[1] * b[0];
9      c[1] := a[1] * b[1];
10     (* Register assignments are necessary to prevent the
11     propagation of unmasked values when probing on c[1]. *)
12     aux  = ![r + a0b1];
13     aux  = ![aux + a1b0];
14     c[0] := c[0] + r;
15     c[1] := c[1] + aux;
16 end
17
18 NI ISW_AND_NI_G
19 SNI ISW_AND_NI_G
```

Figure 4.6: First-order secure **ISW_AND_NI_G** gadget with glitches that satisfies the t–NI property.

```
1  proc ISW_AND_SNI_G:
2      inputs: a[0:1], b[0:1]
3      outputs: c[0:1]
4      randoms: r;
5
6      c[0] := a[0] * b[0];
7      a0b1 := a[0] * b[1];
8      a1b0 := a[1] * b[0];
9      c[1] := a[1] * b[1];
10     aux  = ![r + a0b1];
11     aux  = ![aux + a1b0];
12     (* Stop the propagation of glitches into the outputs
13     with register assignments, since output distributions
14     need to be independent of shares to achieve SNI. *)
15     c[0] = ![c[0] + r];
16     c[1] = ![c[1] + aux];
17 end
18
19 SNI ISW_AND_SNI_G
```

Figure 4.7: First-order secure **ISW_AND_SNI_G** gadget with glitches that satisfies the t–SNI property.

The resulting gadget in Figure 4.7 now fulfills all of MASKVERIF's security notions, even in the glitch-extended model.

4.3 t–SNI Secure Addition

Now that we have presented the **ISW AND** algorithm, we will examine a t–SNI secure addition algorithm by Coron et al. [CGV14] that uses the **ISW AND** algorithm as a subroutine. The motivation for this algorithm is to avoid conversion and perform an addition directly on Boolean shares instead. Coron proposed two variants of secure addition variants, of which we chose the first one. A rewritten version of Coron et al.'s algorithm can be seen in Algorithm 8.

This algorithm makes use of the property that an addition $d = a + b \bmod 2^k$, can also be computed as $d = a \oplus b \oplus c$, where c is the carry of the addition. Conveniently, we already have an algorithm to securely compute the carry of a and b. Coron classifies his algorithm as secure in the ISW model, which refers to the security notion that we call t–probing security. However, we will show that Coron's algorithm achieves the much stronger notion of t–SNI. Algorithm 7 allows us to compute a bitwise $\wedge$ while maintaining the t–SNI property. Since t–SNI offers compositional guarantees, meaning that all outputs of **ISW AND** are uniformly distributed, the values c_i are all uniformly distributed. Each

Algorithm 8: SEC ADD rewritten version of Coron et al.'s Algorithm 2 [CGV14].

Input: Boolean shares $(a_0, ..., a_{n-1})$ and $(b_0, ..., b_{n-1})$ of secrets a, b of size k.
Output: $(d_0, ..., d_{n-1})$ which fulfill $d = a + b = d_0 \oplus ... \oplus d_{n-1}$.

$(c_i^0)_{i \in n} \leftarrow 0$
for $j = 0$ *to* $k - 2$ **do**
    $(ab_i^j)_{i \in n} \leftarrow$ **ISW AND**$((a_i^j)_{i \in n}, (b_i^j)_{i \in n})$
    $(ac_i^j)_{i \in n} \leftarrow$ **ISW AND**$((a_i^j)_{i \in n}, (c_i^j)_{i \in n})$
    $(bc_i^j)_{i \in n} \leftarrow$ **ISW AND**$((b_i^j)_{i \in n}, (c_i^j)_{i \in n})$
    $(c_i^{j+1})_{i \in n} \leftarrow (ab_i^j)_{i \in n} \oplus (ac_i^j)_{i \in n} \oplus (bc_i^j)_{i \in n}$
end
$(d_i)_{i \in n} \leftarrow (a_i)_{i \in n} \oplus (b_i)_{i \in n} \oplus (c_i)_{i \in n}$
return $(d_0, ..., d_{n-1})$

share of d is then computed through an xor of the corresponding shares of a and b, and the xor with each of the uniformly distributed carries c_i masks the result. Hence, all shares of d are simulatable without any input shares. Therefore, the **SEC ADD** algorithm achieves t–SNI security. One exception to this is the case of $k = 1$ where the **ISW AND** algorithm is not applied. Every output share is then computed as $d_i = a_i \oplus b_i$. To be able to simulate one of the outputs d_i, a simulator now requires the respective input shares a_i and b_i, satisfying the t–NI property. However, since the output d_i still requires input shares for simulation, the t–SNI property is not satisfied for this specific case.

We chose to implement this algorithm into MASKVERIF gadgets because it is a relevant use case of the **ISW AND** algorithm. Another reason was that the secure addition on a Boolean sharing offers an alternative to B2A conversion. Since we knew that B2A conversion is not verified correctly with MASKVERIF, it was interesting to see if the **SEC ADD** algorithm can be verified correctly.

The MASKVERIF gadget in Figure 4.8 shows a first-order implementation of the **SEC ADD** algorithm for 8-bit values. Implementing this algorithm revealed some problems that can occur when implementing gadgets. First, MASKVERIF always leaks an entire variable, even when only certain bits are extracted and used in a calculation. This gives us two choices for the **SEC ADD** algorithm.

The first option is to implement a gadget that is equivalent in instruction to the algorithm but over-approximates leakage. The second option is to write a gadget that is equivalent in leakage but does not implement instructions according to the algorithm. We chose the first option, requiring us to generate 1-bit masks to give us the ability to select bits from both inputs and calculate bitwise operations correctly. What adds to this problem, is that constants only exist as one-bit values, which are incompatible with the 8-bit values used in the algorithm. Therefore, a helper function is necessary to expand a 1-bit value into

an 8-bit value to avoid incompatibilities between the data types. From that point on, the algorithm can be executed normally, the only other change is that we need to add up the individual carry bits into an 8-bit mask in lines 91-93 of Figure 4.8.
We implemented **SEC ADD** gadgets for up to fifth-order with input sizes 1, 8, 16, 32, and 64-bit values. These gadgets are verified as t–SNI using MASKVERIF, for up to fifth-order with input sizes up to 32-bit. The verification was performed without parallelization enabled and on a regular desktop computer. We stopped verification at the fifth-order gadget for 32-bit values when the verification time exceeded two hours. Should a more powerful computer perform the verification with parallelization enabled, it stands to reason that slightly higher orders can be verified.

```
1  proc ISW_AND:
2     inputs: w8 a[0:1], w8 b[0:1]
3     outputs: w8 c[0:1]
4     shares: w8 a0b1[0], w8 a1b0[0], w8 aux[0]
5     randoms: w8 r;
6
7     c[0] := a[0] &w8 b[0];
8     a0b1[0] := a[0] &w8 b[1];
9     a1b0[0] := a[1] &w8 b[0];
10    c[1] := a[1] &w8 b[1];
11    aux[0]  := r ^w8 a0b1[0];
12    aux[0]  := aux[0] ^w8 a1b0[0];
13    c[0] := c[0] ^w8 r;
14    c[1] := c[1] ^w8 aux[0];
15 end
16
17 (* This is a helper function to convert a 1 bit into an 8 bit value. *)
18 op #expand : w1 -> w8
19 (* This is a helper function to perform a 1 bit left shift. *)
20 op #lsl1 : w8 -> w8
21
22 (* SNI secure ADD *)
23 proc SEC_ADD:
24    inputs: w8 a[0:1], w8 b[0:1]
25    outputs: w8 d[0:1]
26    shares: w8 maska[0:1], w8 maskb[0:1], w8 carry[0:1],
27    w8 auxa0[0:1], w8 auxa1[0:1], w8 auxa2[0:1], w8 auxa3[0:1],
28    w8 auxa4[0:1], w8 auxa5[0:1], w8 auxa6[0:1], w8 auxb0[0:1],
29    w8 auxb1[0:1], w8 auxb2[0:1], w8 auxb3[0:1], w8 auxb4[0:1],
30    w8 auxb5[0:1], w8 auxb6[0:1], w8 auxb7[0:1], w8 ab0[0:1],
31    w8 ab1[0:1], w8 ab2[0:1], w8 ab3[0:1], w8 ab4[0:1], w8 ab5[0:1],
32    w8 ab6[0:1], w8 ab7[0:1], w8 ac0[0:1], w8 ac1[0:1], w8 ac2[0:1],
33    w8 ac3[0:1], w8 ac4[0:1], w8 ac5[0:1], w8 ac6[0:1], w8 ac7[0:1],
34    w8 bc0[0:1], w8 bc1[0:1], w8 bc2[0:1], w8 bc3[0:1], w8 bc4[0:1],
35    w8 bc5[0:1], w8 bc6[0:1], w8 bc7[0:1], w8 c0[0:1], w8 c1[0:1],
36    w8 c2[0:1], w8 c3[0:1], w8 c4[0:1], w8 c5[0:1], w8 c6[0:1], w8 c7[0:1];
37
38    (* This inconvenient part is necessary to simulate the computation of
39    single bit operations while using the 8-bit data type. *)
40    c0[0:1] = #expand((0d0 : w1));
41    maska[0:1] = #expand((0d1 : w1));
42    maskb[0:1] = #expand((0d1 : w1));
43    auxa0[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
44    auxa1[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
45    auxa2[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
46    auxa3[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
47    auxa4[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
48    auxa5[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
49    auxa6[0:1] = a[0:1] &w8 #lsl1(maska[0:1]);
50    auxb0[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
51    auxb1[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
52    auxb2[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
53    auxb3[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
54    auxb4[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
55    auxb5[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
56    auxb6[0:1] = b[0:1] &w8 #lsl1(maskb[0:1]);
```

```
57      (* This is where the computation of individual carry bits happens. *)
58
59      ab0[0:1] = ISW_AND(auxa0[0:1], auxb0[0:1]);
60      ac0[0:1] = ISW_AND(auxa0[0:1], c0[0:1]);
61      bc0[0:1] = ISW_AND(auxb0[0:1], c0[0:1]);
62      c1[0:1] := ab0[0:1] ^w8 ac0[0:1] ^w8 bc0[0:1];
63      ab1[0:1] = ISW_AND(auxa1[0:1], auxb1[0:1]);
64      ac1[0:1] = ISW_AND(auxa1[0:1], c1[0:1]);
65      bc1[0:1] = ISW_AND(auxb1[0:1], c1[0:1]);
66      c2[0:1] := ab1[0:1] ^w8 ac1[0:1] ^w8 bc1[0:1];
67      ab2[0:1] = ISW_AND(auxa2[0:1], auxb2[0:1]);
68      ac2[0:1] = ISW_AND(auxa2[0:1], c2[0:1]);
69      bc2[0:1] = ISW_AND(auxb2[0:1], c2[0:1]);
70      c3[0:1] := ab2[0:1] ^w8 ac2[0:1] ^w8 bc2[0:1];
71      ab3[0:1] = ISW_AND(auxa3[0:1], auxb3[0:1]);
72      ac3[0:1] = ISW_AND(auxa3[0:1], c3[0:1]);
73      bc3[0:1] = ISW_AND(auxb3[0:1], c3[0:1]);
74      c4[0:1] := ab3[0:1] ^w8 ac3[0:1] ^w8 bc3[0:1];
75      ab4[0:1] = ISW_AND(auxa4[0:1], auxb4[0:1]);
76      ac4[0:1] = ISW_AND(auxa4[0:1], c4[0:1]);
77      bc4[0:1] = ISW_AND(auxb4[0:1], c4[0:1]);
78      c5[0:1] := ab4[0:1] ^w8 ac4[0:1] ^w8 bc4[0:1];
79      ab5[0:1] = ISW_AND(auxa5[0:1], auxb5[0:1]);
80      ac5[0:1] = ISW_AND(auxa5[0:1], c5[0:1]);
81      bc5[0:1] = ISW_AND(auxb5[0:1], c5[0:1]);
82      c6[0:1] := ab5[0:1] ^w8 ac5[0:1] ^w8 bc5[0:1];
83      ab6[0:1] = ISW_AND(auxa6[0:1], auxb6[0:1]);
84      ac6[0:1] = ISW_AND(auxa6[0:1], c6[0:1]);
85      bc6[0:1] = ISW_AND(auxb6[0:1], c6[0:1]);
86      c7[0:1] := ab6[0:1] ^w8 ac6[0:1] ^w8 bc6[0:1];
87
88      (* Assemble the individual carry bits into
89      an 8 bit value and perform the instruction
90      from line 8 of the secure addition algorithm.*)
91
92      carry[0:1] = #expand((0d0 : w1));
93      carry[0:1] := carry[0:1] ^w8 c0[0:1] ^w8 c1[0:1]
94                ^w8 c2[0:1] ^w8 c3[0:1] ^w8 c4[0:1]
95                ^w8 c5[0:1] ^w8 c6[0:1] ^w8 c7[0:1];
96      d[0:1] := a[0:1] ^w8 b[0:1] ^w8 carry[0:1];
97  end
98
99  noglitch SNI SEC_ADD
```

Figure 4.8: First-order secure **SEC ADD** MASKVERIF gadget for 8-bit values.

4.4 MASKVERIF's Verification Algorithms

After taking a look at some MASKVERIF gadgets and their notation, we will now discuss some high-level functions and how MASKVERIF uses these to reason about security. The algorithms can be divided into a verification algorithm, called **Check**, that verifies the independence of a set of observations from secrets and an exploration algorithm, called **CheckAll** that calls the verification algorithm on all possible sets of observations. A gadget is successfully verified if all sets of observations were determined to be free of leakage according to the chosen verification parameters. The key idea behind MASKVERIF's verification is to take a set of observations O and transform it into a set O' with the same distribution, which is independent of secrets. This is done by successively applying transformations following the optimistic- and general optimistic sampling rule on the set O. If further transformations are no longer possible while independence from secrets has not been reached the verification fails.
We will now present the verification algorithms as they are defined by Barthe et al. [BBC+19] and offer a high-level explanation of how they work.

4.4.1 Single Set Verification

The algorithms used by MASKVERIF can be split into two categories. The first set of algorithms is used to verify a single set of observations and the second set of algorithms uses the first algorithms on all possible observation sets. The algorithms used for single-set verification can be seen in Figure 4.9.
The main verification algorithm is called **Check**, which additionally uses the functions **Test**, **Select**, **Simplify**. First, we will examine the **Test** function. This function checks if a set of observations O' is independent of secret values, depending on the security notion.

Verification algorithm

proc Select(R, O) =
  if $\exists r, e, C \mid O = C[e + r] \wedge r \notin e \cup C$ **then**
    return $(R, (e, r), C[r])$;
  if $\exists r, e, C \mid O = C[e + r] \wedge r \notin e \cup R$ **then**
    $O' =$ Simplify$(O\{r \leftarrow e + r\})$;
    return $(R \cup \{r\}, (e, r), O')$;
  else fail ;

proc Check(R, B, O) =
  if Test(O) **then return** B
  $(R', b, O') =$ Select(R, O);
  Check$(R', B :: b, O')$;

Figure 4.9: These are the single set verification algorithms as described by Barthe et al. in Figure 5 [BBC+19].

- For t–probing security it is checked if O', with t probe positions, does not contain the secret inputs.
- For t–NI check if O' contains at most t shares of each input.
- For t–SNI check if O' contains at most t_{in} shares of each input, where t_{in} is the number of internal observations in O'.

If **Test** on O evaluates to true, **Check** returns the set of bijections B that were used to transform the original observation set into an independent one. In the case that **Test** does not succeed, **Check** calls **Select** with the current set O and a set of randoms R that was already used for transformation.
Select now tries to apply two possible rules on O. The first one is the **optimistic sampling rule** which the first if statement in **Select** covers and the second is the **general optimistic sampling rule**, which the second if statement covers. If neither of these rules can be applied then the verification fails, since a set O' that is independent of secrets could not be found.
The **optimistic sampling rule** is MASKVERIF's strongest simplification rule to transform a set of observations O into a smaller set O' with the same distribution. Simply put, for an expression e in O and a random variable r, if $r \notin e$ then $e + r$ can be replaced by r. In the original paper, this is described over a context $C[\cdot]$ which is not formally defined. However, our knowledge of the tool indicates that $O = C[e + r]$ describes that the observation set O contains $e + r$, as well as some other expressions. For this first version of the optimistic sampling rule, it is important that r does not occur in e or the context C, which are all other expressions. Another way of thinking about this is that r can only be used once in the current set O that is being checked. This rule is applicable for any bijective operator, which is why custom operators should be declared as such. Otherwise, MASKVERIF can not apply this specific rule to certain observations. Therefore, $e \oplus r$ can be replaced by r and leakage no longer depends on e.
The **general optimistic sampling rule** is a variation of the previous rule which is applied if the first rule can not be used. The constraint of r only occurring once is removed, as long as r does not occur in e for $O = C[e + r]$ and r is not contained in the set of already used randoms R, this rule can be applied. In this case r is now substituted by $e + r$ in $O' = O\{r \longleftarrow e + r\}$, meaning all occurrences of r are replaced. Since the size of O' does not necessarily decrease, r needs to be added to the set R to ensure termination.

4.4.2 Extending to Combinations of Observation Sets

The previous algorithms work to verify a single set of observations, however, since all combinations of t observations can be probed MASKVERIF has to efficiently check increas-

Exploration algorithm

```
proc Replay(B, O) =
  if B = [] then return Test(Simplify(O))
  if B = (e, r) :: B' then
    Replay(B', O{r ← e + r})

proc Extend(B, X) =
  {O ∈ X |
    Replay(B, O)}

proc OptSampling(X) =
  if ∃r, e, C_X | X = C_X[e + r] ∧ r ∉ e ∪ C_X then
    OptSampling(C_X[r]);
  else return X;

proc CheckAll(X) =
  if X = ∅ return true;
  X = OptSampling(X);
  O = Choose(X);
  B = Check(∅, [], O);
  X_0 = Extend(B, X);
  CheckAll(X \ X_0);
```

Figure 4.10: Algorithms to verify all possible observation sets as described by Barthe et al. in Figure 5 [BBC+19].

ingly larger sets of observations for independence. The algorithms to verify all possible combinations of observations, that Barthe et al. [BBC+19] defined, can be seen in Figure 4.10.

Instead of checking all possible combinations of observations independently, MASKVERIF uses the property that if set O is independent of the secret, then all sub-tuples of observations are also independent of the secret. This is used by determining sets independent of secrets through bijections and then extending these bijections to other observations and determining if the bijections still hold. If this is the case then a larger set independent of secrets has been found and the process can be repeated until all possible combinations of sets have been deemed secure.

More specifically **CheckAll** receives a set of all tuples X that need to be verified. The first if statement checks the trivial condition of X being empty, and therefore independent of secrets. Afterward, the optimistic sampling rule is applied globally wherever the condition is satisfied. This is an important step because it allows us to share the transformation across all tuples instead of trying it on sub-tuples and extending it later.

Next, an observation O is chosen from X with **Choose**. Then the verification algorithm **Check**, which was discussed previously, is applied. If the algorithm succeeds then B contains a list of transformations that were performed to prove that O is independent of secrets. Then **Extend** attempts to increase the set of observations while making sure that the bijections in B still apply. This is done by calling **Replay** on B with all observations in X. **Replay** then recursively checks if the transformations in B can be performed on those observations.

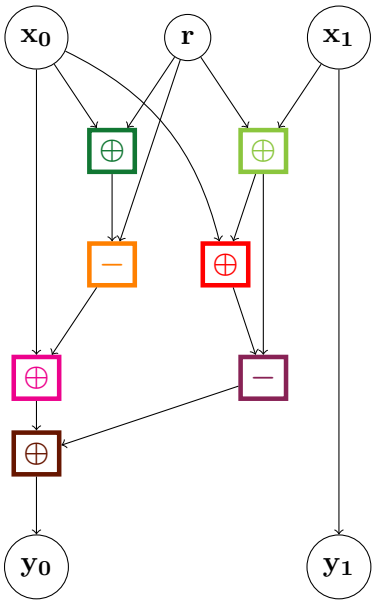
After this process, X_0 contains the observations to which the bijections in B could be extended. Finally, **CheckAll** is called again with all observations which could not be verified yet. This way the set of observations that still need to be checked gets smaller until it is empty and the if statement $X = \emptyset$ is true and verification succeeds, or no further simplifications are possible and the verification fails. Now that we have discussed the algorithms that MASKVERIF uses to reason about a gadget's security, we can discuss the implementation of Boolean-to-arithmetic conversion in the next chapter.

5 Implementing Boolean to Arithmetic Conversion

In this section, we explain our initial approach to identify why the verification of B2A algorithms leads to false negatives. We then present a security proof for Goubin's B2A algorithm showing that it is t–probing secure, but not t–NI. By carefully examining which semantically equivalent expressions are correctly verified by MASKVERIF, we can define simplification and substitution rules that lead to the correct verification of first-order B2A gadgets. Lastly, we discuss MASKVERIF's data structures and how our rules were implemented in MASKVERIF's codebase.
It is important to note that the rules we have chosen are very similar to the rules that Coron used to incorporate B2A conversion into his tool CHECKMASKS [Cor18]. We encountered this tool while finishing the write-up of the thesis and found that we had independently reached a similar approach to the one that Coron presented. His tool is based on the first and second versions of MASKVERIF [BBD+15, BBD+16] and uses similar verification methods. The rules that Coron chose also aim to find expressions violating the optimum sampling rule and replace them. CHECKMASKS operates on Common Lisp files and functions for verification and does not support Verilog or MASKVERIF gadgets. In contrast to MASKVERIF version three, CHECKMASKS can only verify t–probing, t–NI, and t–SNI notions without the consideration of glitches and transitions. This is also one of the reasons that MASKVERIF is used a lot, since the consideration of physical defaults is important for practical analysis.

5.1 Goubin's B2A Algorithm

Initially, we knew that B2A and A2B conversion were not verified correctly in MASKVERIF, however, the precise reason was unknown. Therefore, the first step was understanding the algorithms and writing verification gadgets for them. We started with Goubin's B2A algorithm [Gou01] since it is a classic work in the field of B2A and A2B conversion. The corresponding gadget and a circuit representation can be seen in Figure 5.1.
In the early stages of testing, we used the custom operator `bij op #sub: w1,w1->w1` since MASKVERIF does not provide arithmetic operations natively. Later on, we deemed it necessary to implement a native subtraction operator in MASKVERIF, which is used in the gadget displayed in Figure 5.1.
At first glance, we believed that this gadget should fulfill the t–NI property, for $t = 1$, and




```
1  proc B2A_Goub:
2      inputs: x[0:1]
3      outputs: y[0:1]
4      randoms: r;
5      t1 := x[0] + r;
6      t2 := t1 - r;
7      t3 := t2 + x[0];
8      t4 = ![r + x[1]];
9      t5 := x[0] + t4;
10     t6 := t5 - t4;
11     t7 = t6 + t3;
12
13     y[0] = t7;
14     y[1] = x[1];
15
16 end
17
18 Probing B2A_Goub
19 NI B2A_Goub
```

Figure 5.1: Circuit and MASKVERIF gadget representation of Algorithm 3.

tested a combination of normal assignments and leakage-free assignments to determine where the verification fails. As it turns out, operation t_7 does not allow for the application of the optimal sampling rule, and MASKVERIF is unable to verify this exact observation. The two observations t_6 and t_3, which occur in t_7, can be verified individually. This is possible because MASKVERIF can find a bijection for t_6 determining that t_6 depends on t_5 and t_4, where $t_5 = t_4 \oplus x_0$, meaning that t_4 occurs twice, but the randomness r only occurs once in $t_4 = x_1 \oplus r$. Consequently, MASKVERIF can verify that this statement does not violate t–NI and does not depend on x_1. For t_3, MASKVERIF can do the same thing. The problem arises when both intermediate values are combined and MASKVERIF has two different expressions, namely $t_2 = (x_0 \oplus r) - r$ and $t_6 = (x_0 \oplus x_1 \oplus r) - (x_1 \oplus r)$, for which it cannot find a bijection such that a chain of dependencies is built where only one sub-expression depends on r and every following expression only has exactly one expression in which it is used.
After finding the problematic instruction we decided to go through a manual proof to determine which security notion Goubin's algorithm satisfies exactly with regard to modern terminology.

5.1.1 Goubin B2A Security

Goubin talks about the security of his algorithm in Section 3.3 [Gou01], where he analyzes security against DPA attacks. However, since Goubin's security proof does not include

any of the formal security notions like t–probing, t–NI, or t–SNI, which did not exist at the time, the question of what correct verification should result in, arises. Therefore, it was necessary to analyze Goubin's B2A algorithm and offer a formal proof that allows us to modify MASKVERIF's verification.

Lemma 5.1. *The gadget in Figure 5.1 is t–probing secure and is not t–NI, for $t = 1$.*

In this subsection, we show which parts of Goubin's algorithm are in accordance with the t–NI property and outline where the algorithm violates t–NI.
For this proof, we use the following from Goubin's work [Gou01]:

Corollary 1.2 [Gou01] *For any value r, if we denote $A = (x_0 \oplus x_1) - x_1$, we also have*

$$A = [(x_0 \oplus r) - r] \oplus x_0 \oplus [(x_0 \oplus (x_1 \oplus r)) - (x_1 \oplus r)].$$

Corollary 1.2 shows that an arithmetic share can still be computed correctly even when a random mask r is added during specific points of the calculation. This corollary is vital to the secure and correct calculation of the arithmetic share A.
Next, Theorem 1 states an alternative representation of calculating the arithmetic share A:

Theorem 1 [Gou01]

$$B2A(x_0, x_1) = x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i x_1)]$$

Lastly, we need **Lemma 2** in **Annex 1** of [Gou01], which states that for any integers u, v, the following holds:

$$u - v \equiv (u \oplus v) - 2(\bar{u} \wedge v) \bmod 2^K.$$

Proof (Proof of Lemma 5.1). We will show that every step of Goubin's algorithm, except for the output value t_7, can be simulated with at most one input share. The output t_7 depends on both shares, however, the result of the computation is uniformly distributed, if x_1 is uniformly distributed.
As r is randomly chosen and uniformly distributed on $\{0,1\}^K$, where K is the register size in the computation, one can see that r, t_1, t_4, and t_5 are uniformly distributed on $\{0,1\}^K$. This means that a simulator can produce values with the same output distribution as these intermediate values by sampling a random value. The terms t_2 and t_3 only contain the share x_0 and are independent of x_1. The output y_1 depends only on the input share x_1 and can thus be simulated with one share.

Next, we will show that the distribution of t_6 depends only on x_0 and not on x_1. Therefore, t_6 can be simulated with the share x_0 and a uniformly random sampled value.

We will use Lemma 2 to simplify the terms containing subtractions and show that they only depend on x_0.

Proof that t_6 is in accordance with the t–NI property:
First, we apply Goubin's previously mentioned Lemma 2 to $t_6 = (x_0 \oplus x_1 \oplus r) - (x_1 \oplus r)$ and obtain:

$$
\begin{aligned}
t_6 &= (x_0 \oplus x_1 \oplus r) - (x_1 \oplus r) \\
&= (x_0 \oplus x_1 \oplus r \oplus x_1 \oplus r) - 2(\overline{(x_0 \oplus x_1 \oplus r)} \wedge (x_1 \oplus r)) \\
&= x_0 - 2(\overline{(x_0 \oplus x_1 \oplus r)} \wedge (x_1 \oplus r)) \\
&= x_0 - 2((\overline{x_0} \oplus x_1 \oplus r) \wedge (x_1 \oplus r)) \\
&= x_0 - 2(((x_1 \oplus r) \wedge \overline{x_0}) \oplus ((x_1 \oplus r) \wedge x_1) \oplus ((x_1 \oplus r) \wedge r)) \\
&= x_0 - 2((\overline{x_0} \wedge r) \oplus (\overline{x_0} \wedge x_1) \oplus (x_1 \wedge x_1) \oplus (x_1 \wedge r) \oplus (x_1 \wedge r) \oplus (r \wedge r)) \\
&= x_0 - 2((\overline{x_0} \wedge r) \oplus (\overline{x_0} \wedge x_1) \oplus x_1 \oplus (x_1 \wedge r) \oplus (x_1 \wedge r) \oplus r) \\
&= x_0 - 2((\overline{x_0} \wedge r) \oplus (\overline{x_0} \wedge x_1) \oplus x_1 \oplus r) \\
&= x_0 - 2((\overline{x_0} \wedge (x_1 \oplus r)) \oplus x_1 \oplus r).
\end{aligned}
$$

At this point we substitute $z = x_1 \oplus r$ to improve readability and obtain

$$
\begin{aligned}
&= x_0 - 2((\overline{x_0} \wedge z) \oplus z) \\
&= x_0 - 2((\overline{(\overline{x_0} \wedge z)} \wedge z) \vee (\overline{x_0} \wedge z \wedge \overline{z})) \\
&= x_0 - 2((x_0 \vee \overline{z}) \wedge z) \\
&= x_0 - 2((x_0 \wedge z) \vee (z \wedge \overline{z})) \\
&= x_0 - 2(x_0 \wedge z) \\
&= x_0 - 2(x_0 \wedge (x_1 \oplus r)).
\end{aligned}
$$

If given x_0, a simulator can generate an output with the same distribution as t_6, since it can sample a uniformly distributed random value that has the same distribution as $x_1 \oplus r$. Consequently, x_1 has no impact on the distribution of t_6 and the t–NI property holds.

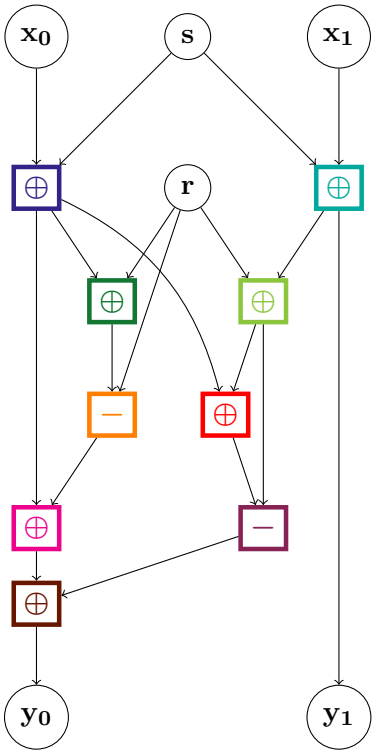
Proof that t_7 violates the t–NI property:
The distribution of t_7 depends on both input shares x_0 and x_1, meaning that the t–NI property is violated. To see why the random value r does not influence the distribution of t_7, we will apply the previously mentioned Theorem 1 from Goubin's work to t_7:

$$
\begin{aligned}
t_7 &= x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i r)] \oplus x_0 \oplus \\
&\quad x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i (r \oplus x_1))] \\
&= x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i r)] \\
&\quad \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i (r \oplus x_1))] \\
&= x_0 \oplus \bigoplus_{i=1}^{K-1} [((\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i r)) \oplus ((\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i (r \oplus x_1)))] \\
&= x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (((2^i x_0) \wedge (2^i r)) \oplus ((2^i x_0) \wedge (2^i (r \oplus x_1))))] \\
&= x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge ((2^i r) \oplus (2^i (r \oplus x_1)))] \\
&= x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge ((2^i r) \oplus (2^i r) \oplus (2^i x_1))] \\
&= x_0 \oplus \bigoplus_{i=1}^{K-1} [(\bigwedge_{j=1}^{i-1} (2^j \overline{x_0})) \wedge (2^i x_0) \wedge (2^i x_1)] \\
&= B2A(x_0, x_1) = x - x_1.
\end{aligned}
$$

The result $x - x_1$ is only uniformly distributed if x_1 is uniformly distributed. Since the notion of t–NI does not require shares to be uniformly distributed, t_7 can not be simulated without both shares x_0, x_1. However, the notion of t–probing security assumes shares to be uniformly distributed, in which case t_7 is also uniformly distributed. Therefore, Goubin's algorithm achieves t–probing security, but not t–NI. □

5.1.2 Coron B2A Security

Coron extends Goubin's algorithm in [Cor17] to achieve the t–SNI property. This is done by applying additional randomness $s \leftarrow\!\!\$\ \{0,1\}^K$ to both shares, such that $a_0 = x_0 \oplus s$ and $a_1 = x_1 \oplus s$. Then all of the intermediate computations of Goubin's algorithm are performed with a_0, a_1 instead of the original shares x_0, x_1. The circuit and MASKVERIF gadget for Coron's first-order t–SNI algorithm can be seen in Figure 5.2.
Since all the intermediate steps already fulfilled the t–NI property and additional fresh randomness is added for remasking, all of these values stay secure. Remember that the t–SNI property requires that the distribution of outputs can be simulated without input shares.




```
1  proc B2A_SNI:
2      inputs: x[0:1]
3      outputs: y[0:1]
4      shares: a[0:1]
5      randoms: r, s;
6      a[0] := x[0] + s;
7      a[1] := x[1] + s;
8      t1 := a[0] + r;
9      t2 := t1 - r;
10     t3 := t2 + a[0];
11     t4 = ![r + a[1]];
12     t5 := a[0] + t4;
13     t6 := t5 - t4;
14     t7 = t6 + t3;
15
16     y[0] = t7;
17     y[1] = a[1];
18
19 end
20
21 Probing B2A_SNI
22 NI B2A_SNI
```

Figure 5.2: Circuit and MASKVERIF gadget representation of Algorithm 4.

The first output $y_0 = t_7 = B2A(a_0, a_1) = (a_0 \oplus a_1) - a_1 = (x_0 \oplus x_1 \oplus s \oplus s) - (x_1 \oplus s) = x - (x_1 \oplus s)$ is always uniformly distributed, since $x_1 \oplus s$ is uniformly distributed and subtracting a uniformly distributed value from the secret x results in a uniformly distributed value. Consequently, y_0 can be simulated without any input shares.
The second output $y_1 = a_1 = x_1 \oplus s$ is also uniformly distributed, therefore, Coron's algorithm satisfies the t–NI property.

5.1.3 Determining Verifiable Expressions

After examining the security notions that each algorithm satisfies, the next task was to determine a suitable method to allow MASKVERIF to verify these algorithms correctly.
The original term $t_7 = ((x_0 \oplus r) - r) \oplus x_0 \oplus ((x_0 \oplus x_1 \oplus r) - (x_1 \oplus r))$ can not be simplified with the rules that MASKVERIF has at its disposal. Our next step was to test which expressions MASKVERIF could verify correctly and found that the term $x - x_1$ is classified correctly as t–probing secure. We determined this by manually verifying different semantically equivalent expressions and evaluating how MASKVERIF reasons about their security.
To avoid the problems with the optimal sampling rule we implemented our first rule, which we call **Goubin substitution**. This rule checks expressions to look for the pattern

$((x_0 \oplus r) - r) \oplus ((x_0 \oplus x_1 \oplus r) - (x_1 \oplus r))$ and simplifies the expression into $((x_0 \oplus x_1) - x_1) \oplus x_0$. When applied to the term t_7, this results in $((x_0 \oplus x_1) - x_1) \oplus x_0 \oplus x_0$. To further simplify this, we first need to understand how t–probing security is verified in MASKVERIF.

For this security notion, MASKVERIF internally defines t–probing security as t–NI under the restriction of uniformly sampled shares, for all but one share. This means that all but one share are eligible for the optimal sampling rule and the only non-uniformly distributed share is now defined as $x_0 = x \oplus x_1 \oplus ... \oplus x_{n-1}$ for a secret x split into n shares.

This changes the internal representation of the result $t_7 = ((x_0 \oplus x_1) - x_1) \oplus x_0 \oplus x_0$ to $((x \oplus x_1 \oplus x_1) - x_1) \oplus (x \oplus x_1) \oplus (x \oplus x_1)$. Since x_1 still occurs in multiple expressions the verification fails, however, all the problems with this expression can be remedied by implementing another rule we call **xor simplification**.

This rule aims to eliminate any even number of occurrences where one variable is xored with itself. Meaning the term $x_1 \oplus (x_1 \oplus x_1)$ is simplified into the term x_1 and $x_1 \oplus x_1$ is removed. When applied to the previous expressions we obtain $x - x_1$, which can be verified correctly.

Additionally, we implemented an **and simplification** rule which substitutes any sequence of a variable that has the $\wedge$ operation performed with itself for a single occurrence of a variable, meaning $x_1 \wedge x_1 \wedge ... \wedge x_1$ is substituted by x_1. This specific rule has no bearing on our example, it should, however, allow for simplification of expressions generally.

Our rules are applied as a preprocessing on the observations that MASKVERIF builds before any verification function is called. The **Goubin substitution** rule will only detect and substitute terms with the specific structure we mentioned and the correctness of the substitution has been shown with the previous proofs and Goubin's Corollary 1.2. The **and simplification** and **xor simplification** rules are checked for every expression in the full set of observations and allow MASKVERIF to apply its verification function on simpler, yet semantically equivalent expressions, that can be verified more easily.

For Coron's algorithm the expression $t_7 = ((a_0 \oplus r) - r) \oplus a_0 \oplus ((a_0 \oplus a_1 \oplus r) - (a_1 \oplus r))$ can be reduced to $(a_0 \oplus a_1) - a_1 = (x_0 \oplus x_1 \oplus s \oplus s) - (x_1 \oplus s) = (x_0 - x_1) - (x_1 \oplus s)$ using the same rules described before. Since t–SNI does not assume shares to necessarily be randomly sampled the only randomness that MASKVERIF can use for its optimal sampling rule is s. Fortunately, s only occurs once after our simplification rules are applied and the term t_7 is correctly evaluated as t–SNI, for $t = 1$.

After briefly describing the rules we used to fix the verification of first-order B2A conversion algorithms, we will highlight the implementation details in the next section.

5.2 Implementation Details

In this section, we will explain how the three rules we previously described are implemented into MASKVERIF's codebase. Since MASKVERIF is written in the functional programming language OCaml, most of the following pseudocode is written in an abstract functional manner and substantially simplified. Before describing the functions we wrote we first need to look at the data structure that MASKVERIF uses to represent expressions.

```
type expr = {
    eid : int;
    enode : exprnode}
```

Figure 5.3: Expression data type in MASKVERIF.

Expressions are a tuple of an expression ID and an expression node and the definition of the data type can be seen in Figure 5.3. When an expression is constructed, an expression node receives an ID and is added to a hash table. If the node is already present in the hash table the same ID is assigned to the node.

```
type exprnode =
    | Etop
    | Ernd   of rnd
    | Eshare of param * int * var
    | Epub   of var
    | Epriv  of var
    | Eop1   of operator * expr
    | Eop2   of operator * expr * expr
    | Eop    of bool * operator * expr array
    | Econst of constant
```

Figure 5.4: Expression node data type in MASKVERIF.

Expressions and expression nodes are recursively defined over one another, where every expression has an ID and a node that is either a leaf in a tree or another expression. The data type expression node, as it is defined in MASKVERIF, can be seen in Figure 5.4.
The leaves of our expression nodes are the empty node as `Etop`, a random variable `rnd`, a share `Eshare`, a public value `Epub`, a private value `Epriv`, or a constant value `Econst`. For the leaves, this high-level description suffices to understand our later approach. The nodes in the tree are constructed with operators that have expressions as arguments, namely operators with a single argument `Eop1`, operators with two arguments `Eop2`, and

operators with more than two arguments `Eop`.
The implementation of our rules required us to detect expressions that are semantically equivalent but syntactically different due to having an associative operator. When looking at the two terms $x_0 \oplus x_1$ and $x_1 \oplus x_0$ it is obvious that these are semantically equivalent. However, MASKVERIF will assign two different IDs to their expressions and structural comparisons on these expression nodes will evaluate to false. This behavior makes pattern matching and equivalence checks more troublesome and can be avoided by first sorting the leaves and subtrees of expressions by IDs. We implemented the function `orderexpr` which uses a recursive helper function `ordernodes` and can be seen in Figure 5.5.

```
1  function orderexp(exprlist):
2      function ordernodes(exprnode):
3          switch exprnode:
4              case Eop1(op, e1):
5                  return Eop1(op, ordernodes(e1.enode))
6              case Eop2(op, e1, e2):
7                  return Eop2(op,
8                      ordernodes(e1.enode),
9                      ordernodes(e2.enode))
10             case Eop(op, exprarray):
11                 return Eop(op,
12                     foreach exp in exprarray: ordernodes(exp.enode))
13             otherwise:
14                 return exprnode
15
16     return foreach exp in exprlist: ordernodes(exp.enode)
```

Figure 5.5: A function to order expressions by their IDs.

This pseudocode shows that we recursively order expressions along their operators. Some specific checks and conditions, regarding how the ordering is done, are left out for simplicity's sake and can be found in the source code.

5.2.1 Operator Simplification

Now we have ordered expressions to perform our substitution and simplification rules. Next, we will describe the functions that implement the **and simplification** and **xor simplification** rules. Both **and simplification** and **xor simplification** are implemented through a function called `operator_simplification`. This function receives an operator and a list of expressions and tries to simplify all expressions for the specified operator.
To find potential sub-expressions for simplification within the global expression two re-

cursive functions calling each other are used.

The first function `build_subtree` builds a subexpression for all connected nodes with the same operator and returns a tuple containing the subexpression and a list of simplification candidates.

The second function `find_op` finds expression nodes with the operator in question and then calls `build_subtree` and `reduce_expression` on them.

Additionally, another helper function called `reduce_expression`, is used. It receives an operator and a tuple containing a list of expressions that should be reduced and the expression node for which the reduction should be performed.

Figure 5.6 shows an abstract version of the code that implements the **and simplification** and **xor simplification** rules, that could be used for other potential operators as well. This means that we could also use the `operator_simplification` function for the subtraction operator. We currently do not perform this though, since the only MASKVERIF gadgets using the newly implemented subtraction operator are our gadgets, in which expressions like $e - e$ never occur. Consequently, calling the function for the subtraction operator is useless. The simplification rule can be extended for other operators with a single line of code, should the number of gadgets using our subtraction operator increase.

Next, we will briefly explain how the pseudocode shown in Figure 5.6 works. First, `find_op` explores the expression tree and compares operators it finds with the operator that was passed to `operator_simplification`. When the correct operator is found `build_subtree` is called, which identifies all sub-expressions that have the specified operator between them. This chain needs to be uninterrupted, otherwise, the order of operations and associativity do not necessarily allow for trivial simplification. Therefore, finding any other operator requires another call of `find_op` to check if we can still simplify further in that sub-expression. All uninterrupted chains of expressions are then simplified with the `reduce` function according to the **and simplification** or **xor simplification** rules.

5.2.2 Goubin Substitution

This leads us to the **Goubin substitution** rule which is implemented through two core functions. The first one is `goubin_substitution`, which uses a structure similar to the `operator_simplification` function to find potential candidates in sub-expressions that are connected by the correct operators. The second one is `goubin_solve`, which receives a list of expression nodes with a subtraction as their operator, making these expressions potential candidates for substitution, and the current sub-expression which was checked. Additionally, several helper functions are used:

- `is_candidate` checks if an expression with a subtraction operator fulfills the conditions to be a candidate for substitution,
- `flatten` takes an expression and turns it into a list of nodes along the xor operator,
- `find_matches` goes over candidates and finds the best matches, such that the substitution structure can create the smallest terms possible,
- `simplify` performs the substitution and returns simplified expressions, and
- `reconstruct` takes the flattened expressions we have been working on and returns them to their regular structure.

Before we take a look at the pseudocode, we will briefly explain the requirements that have to be met to substitute expressions. Two expressions presenting the pattern $e_0 = (x \oplus (y_0 \oplus ... \oplus y_{n-1})) - (y_0 \oplus ... \oplus y_{n-1})$ and $e_1 = ((x \oplus z) \oplus (y_0 \oplus ... \oplus y_{n-1})) - (z \oplus (y_0 \oplus ... \oplus y_{n-1}))$ need to be xored with each other. This means that both expressions have the structure $x \oplus y - y$, while one expression contains additional variables.
According to Goubin's Corollary 1.2, we can eliminate all variables which occur in both expressions. Initially, we performed a three-argument matching for Corollary 1.2, meaning that we checked if $(x \oplus y) - y$, $(x \oplus y \oplus z) - (y \oplus z)$, and x were present and substituted all three expressions for $x \oplus z - z$. We later changed our approach to a two-argument matching only checking for $(x \oplus y) - y$ and $(x \oplus y \oplus z) - (y \oplus z)$ since this reduced the effort to find the necessary structure. In turn, we now need to substitute $((x \oplus z) - z) \oplus x$ for these two expressions. The extra variable x needs to be added to fulfill the equation and ensure that our substitution is correct. After substitution, we obtain $((x \oplus z) - z) \oplus x \oplus x = (x \oplus z) - z$, which is an expression that MASKVERIF can correctly verify.
Next, we explain the pseudocode of the `goubin_solve` function shown in Figure 5.7. It receives a list of candidate expressions for which the substitution will be attempted. Then, the `flatten` function is called on all expressions, turning the recursive data structure into a list, allowing us to perform easier comparisons and checks between expressions. Now, `find_matches` finds expressions that fulfill the substitution condition mentioned above and matches them according to the largest set of variables that can be eliminated. If no matches are found, then no simplification is possible and the original expression is returned. Afterward, `simplify` performs the substitution and returns the simplified expression. The last step is to call `reconstruct` and turn the simplified expressions back into the original data structure and insert them back into the global expression.
Finally, we will explain the `goubin_substitution` function visible in Figure 5.8. This function receives a list of expressions and performs the **Goubin substitution** rule wherever possible.

The overall structure of this function is similar to the `operator_simplification` function, where the `find_op` function looks for an operator, in this case, the $\oplus$ operator, and then calls the `build_subtree` function to generate a list of potential candidates for the substitution. A slight change to `build_subtree`, however, is that now a chain of xored expressions is checked for expressions containing a subtraction as their operator. Should a subtraction be found, a check is performed by calling `is_candidate`, to determine if this expression fits the structure of $(x \oplus y) - y$, where y can be any arguments. If this is the case, we found a potential candidate for substitution and add it to the list of candidates. After checking an expression connected by xors and finding all candidates, `goubin_solve` is called and potentially returns a substituted expression. This new expression is then inserted into the tree in the position of the old expression. By performing these checks on all expressions we can substitute all relevant terms.

It is important to note that the order in which our rules are applied matters. The **Goubin substitution** rule needs to be applied before the **xor simplification** rule, to guarantee that the substituted terms are simplified correctly. If the simplification is not performed after the **Goubin substitution**, the resulting expression can not be verified correctly as first-order probing secure by MASKVERIF.
An example of this is the expression $t_7 = ((x_0 \oplus r) - r) \oplus x_0 \oplus ((x_0 \oplus x_1 \oplus r) - (x_1 \oplus r))$ that we previously looked at. If the **xor simplification** rule is applied first, it has no impact on the expression.
The following application of the **Goubin substitution** leads to $t_7 = ((x_0 \oplus x_1) - x_1) \oplus x_0 \oplus x_0$. This term is evaluated as not t–probing secure by MASKVERIF. If the rules are applied in reverse order, the resulting expression is $t_7 = ((x_0 \oplus x_1) - x_1)$, which is correctly verified as t–probing secure. This means that the rules need to be applied in a specific order, or applied multiple times to guarantee an exhaustive simplification of expressions.

```
function operator_simplification(operator, exprlist):
    function build_subtree(expr):
        switch expr.enode:
            case Eop1(op, e1):
                return [Eop1(op, e1)], Eop1(op, find_op(e1))
            case Eop(b, op, exprarray):
                return [Eop(b, op, exprarray)],
                    Eop(b, op, foreach exp in exprarray: find_op(exp))
            case Eop2(op, e1, e2):
                if op == operator:
                    l1, node1 = build_subtree(e1)
                    l2, node2 = build_subtree(e2)
                    return List.append(l1, l2),
                        find_op(Eop2(op, node1, node2))
                else:
                    return [Eop2(op, e1, e2)], find_op(Eop2(op, e1, e2))
            otherwise:
                return [expr.enode], expr.enode

    function find_op(expr):
        switch expr.enode:
            case Eop1(op, e1):
                return Eop1(op, find_op(e1))
            case Eop(b, op, exprarray):
                return Eop(b, op, foreach exp in exprarray: find_op(exp))
            case Eop2(op, e1, e2):
                if op == operator:
                    return reduce_expression(op,
                        build_subtree(Eop2(op, e1, e2)))
                else:
                    return Eop2(op, find_op(e1), find_op(e2))
            otherwise:
                return expr.enode

    return foreach exp in exprlist: find_op(exp)
```

Figure 5.6: Pseudocode for the `operator_simplification` function.

```
function goubin_solve(candidates, expr):
    foreach c in candidates:
        flatten(c)
    matches = find_matches(candidates)
    if matches == []:
        return expr
    simplify(matches)
    return reconstruct(matches)
```

Figure 5.7: Pseudocode for the `goubin_solve` function.

```
function goubin_substitution(exprlist):
    function build_subtree(expr):
        switch expr.enode:
            case Eop1(op, e1):
                return [], Eop1(op, find_op(e1))
            case Eop(b, op, exprarray):
                return [],
                    Eop(b, op, foreach exp in exprarray: find_op(exp))
            case Eop2(op, e1, e2):
                if op == Xor:
                    l1, node1 = build_subtree(e1)
                    l2, node2 = build_subtree(e2)
                    return List.append(l1, l2),
                        find_op(Eop2(op, node1, node2))
                else if op == Sub:
                    if is_candidate(e1, e2) == true:
                        return [(e1, e2)], Eop2(op, e1, e2)
                else:
                    return [], find_op(Eop2(op, e1, e2))
            otherwise:
                return [], expr.enode

    function find_op(expr):
        switch expr.enode:
            case Eop1(op, e1):
                return Eop1(op, find_op(e1))
            case Eop(b, op, exprarray):
                return Eop(b, op, foreach exp in exprarray: find_op(exp))
            case Eop2(op, e1, e2):
                if op == Xor:
                    return goubin_solve(build_subtree(Eop2(op, e1, e2)))
                else:
                    return Eop2(op, find_op(e1), find_op(e2))
            otherwise:
                return expr.enode

    return foreach exp in exprlist: find_op(exp)
```

Figure 5.8: Pseudocode for the `goubin_substitution` function.

6 Documentation of MASKVERIF

Over the course of this work, it was necessary to get a deeper understanding of MASKVERIF's code and how it processes gadgets. In this section, we document some files and functions which we believe to be important for understanding and extending MASKVERIF. Understanding and working with MASKVERIF was the most time-consuming part of this thesis since the tool's code is completely undocumented and is far from self-explanatory. The codebase of MASKVERIF contains approximately 5000 lines of code contained in 22 files, which makes an overview of its core files and functions a necessity. Therefore, we offer documentation that allows others that want to get a deeper understanding of MASKVERIF's code or extend the tool to get a head start by identifying important functions and having an overview of the codebase. The structure of this chapter is as follows, a section represents a `.ml` file, while subsections represent functions inside that file. Over the course of this thesis, we have written close to 1000 lines, of which approximately half is code and the other half is documentation. Providing documentation for ourselves and others became a necessity because a large chunk of MASKVERIF's program flow, function names, and state are hard to identify when comparing the actual codebase and the MASKVERIF version three paper [BBC+19]. Most of the codebase documentation that we provide is focused on establishing a connection between the pseudocode in the paper and the existing codebase.

6.1 File preprocess.ml

All the additions we have made to enable our **Goubin substitution**, **and simplification** and **xor simplification** rules can be found in `preprocess.ml`. These rules are applied to the observations that are built by `Prog.build_obs_func` and therefore serve, as the file name indicates, as a preprocessing that is applied before MASKVERIF's core verification routine. This also makes it fairly simple to extend these rules further, since understanding the internal verification process is helpful, but not strictly necessary when working on the observations instead of the graph state that is later initialized for the verification routine.

6.2 File main.ml

The `main.ml` file has a large part of its code devoted to parsing, the most interesting parts, however, are the functions `check_threshold`, `check_ni`, `check_sni` and `check_spini`. These are the different possible security notions that MASKVERIF can check and the function that is called depends on the specified verification parameter. In all of these functions, a set of observations is built with the `Prog.build_obs_func` call from the `prog.ml` file. This function builds the set of observations that has to be checked depending on multiple parameters specified in the gadget, e.g., glitches, assignment types, and security notions. Afterward `check_threshold`, `check_ni`, `check_sni` and `check_spini` call their respective counterparts in the `checker.ml` file and pass the observations and verification parameters along.

6.3 File checker.ml

For this file we have the main verification functions `check_threshold`, `check_ni`, `check_sni` and `check_spini` which mostly differ in their parameters and restrictions. The underlying verification process is similar and that is what we will focus on. All the previously named functions start by first building a state of expressions from the observations that were passed in `main.ml`. Additionally, the number of maximum shares is initialized and used as a checking condition to verify independence from t probes under the respective security notions. Then `check_all_opt` is called, which decides if the verification is parallelized or not, depending on if the parameter was specified in the gadget or not. Over the course of this work, we have almost exclusively worked with non-parallel verification, since it allows for significantly easier program flow analysis and debugging.

6.3.1 Function check_all

This is the first notable function in the verification process and is named similarly to the respective function in the algorithmic description of MASKVERIF version 3 [BBC+19]. However, the algorithmic description that the authors presented only loosely relates to the actual function names and program flow that happens within the tool. The first key difference is that instead of passing sets of observations and bijections to further functions almost all the following functions operate on a graph-based state which contains all variables currently being checked. This data structure is used to nicely illustrate dependencies between sub-expressions and the bijections that MASKVERIF identifies are realized by changing parent and child nodes within the graph.
After setting up the state, `check_all` tests if the current observations are independent of

secrets. Then `simplify_ldfs` is called, attempting to apply the **OptSampling(**X**)** call from Figure 4.10 on all oberservations. Consequently, we now obtain a potentially simplified set of observations after this function call. The check `continue1` is performed to test if verification needs to continue or if independence is reached. Should further verification be necessary, `find_bij` is called, which in part fulfills the functionality of the single-set **Check** algorithm from Figure 4.4, as well as the **Choose**, **Extend** and **Replay** algorithms from Figure 4.10. An updated state of observations is returned for which applications of the optimal sampling rule were found and all the observations that these bijections hold for are no longer considered in the rest of the verification.
At this point, the set of observations is divided and combinations of probes into the separated parts are checked by calling `check_all` on them. Here, the same verification process begins anew.

6.3.2 Function simplify_ldfs

This function loosely corresponds to the $X =$ **OptSampling(**X**)** call from the exploration algorithm we described in Chapter 4 Figure 4.10. The state is initialized, by clearing it and adding all available observations. Afterward `simplify_until`, from `state.ml`, is called, which consists of a series of function calls that essentially correspond to the optimal sampling rule application. Now the state is either independent of secrets, in which case we terminate, or further verification effort is attempted. This is done by calling `simplified_expr` with the state to globally rewrite expressions with a simplified form. This essentially equates to the **OptSampling(**X**)** call, but instead of obtaining bijections that hold for all observations, MASKVERIF now chooses to substitute the expressions for which the optimal sampling rule could be applied globally and continue with these. Lastly, a new set of observations is derived and we return into `check_all`.

6.3.3 Function find_bij

The name of this function alludes to its task to find bijections for observations through applications of the optimal sampling rule. It operates by first choosing a subset of all observations based on a heuristic and adding these to the state, which corresponds to the **Choose** algorithm from Figure 4.10. A verification attempt is then made for this set of observations by calling `simplify_until`, from `state.ml`, on the state to obtain expressions that have been simplified. Should this simplification fail, MASKVERIF attempts to apply strategies to mitigate the number of false negatives and, should these fail, asserts that could not verify the gadget for the given security notion at that currently tested probe position. If simplifications were possible, `get_bij` is used to obtain the bijections that

were applied, and more expressions are added to the state.

Now, `replay_bij` and `simplify_until_with_clear` are used to try and find expressions to which the bijections also apply, which corresponds to the **Extend** and **Replay** of the original algorithms. Note that these are still performed in `find_bij` and not in `check_all` as the pseudocode in Figure 4.10 describes it. At this point, we return to `check_all` with the updated state of observations.

6.4 File state.ml

The `state` manages, as the name indicates, the state that MASKVERIF uses during its verification process. In addition to functions that initialize, update, and manage the state, some other functions which are part of the actual verification logic are also present in this file. Some notable ones include `simplify_until`, `simplify`, `simplify1`, `apply_bij` and `is_rnd_for_bij`.

6.4.1 Function simplify_until

This function is called by multiple functions from `checker.ml` and initiates the application of the optimal sampling rule. As long as the predicate `continue` is true, meaning the observations are still dependent on secret values, `simplify` is called. If `simplify` has no more random values eligible for optimal sampling rule simplification, but simplifications were performed, it will return true, and `simplify_until` will recursively call itself and attempt further simplification. If `continue` is true, but no simplification through `simplify` is possible `simplify_until` returns false, indicating that simplification is necessary but can not be performed.

6.4.2 Function simplify

The main purpose of this function is to call `simplify1`, which in turn attempts to apply the optimal sampling rule to all random variables which are eligible. If `simplify1` returns true, meaning that at least one application of the optimal sampling rule could be applied, it recursively applies `simplify1` to the new state. If `simplify1` was called once and can not be applied anymore, `simplify` returns true. However, when `simplify1` can not be applied at all, meaning no randoms could be considered for the optimal sampling rule, it returns false.

6.4.3 Function simplify1

The application of the optimal sampling rule happens here, `state.s_todo` contains all randoms that could potentially be used for simplification.
Then the function `is_rnd_for_bij` is called, which returns true if the optimal sampling rule can be applied for the random in question. If this returned true, `apply_bij` is called for the random variable, which substitutes a random value r for an expression c by changing its parent and descriptor to r. Additionally, the bijection used is saved and `is_rnd_for_bij` on c to check if c could also be used for further simplification with the optimal sampling rule. After all available randoms in `state.s_todo` were checked, `simplify1` returns true. If `state.s_todo` was initially empty and no simplifications could be performed, false is returned.

6.4.4 Function is_rnd_for_bij

This is the check to establish if a random variable r can be used for the optimal sampling rule. First, two conditions need to be fulfilled, the variable r needs to be a random variable and it needs to have exactly one child, meaning that only one expression in the current state is dependent on r. If these conditions are fulfilled, the function checks if the operator, which r is used in, is bijective. This is a mandatory requirement since the leakage of an expression e can otherwise not be substituted by r.

6.4.5 Function apply_bij

The function receives the state and a random variable r. A bijection chain is built for a child c of r, where c's descriptor is replaced by r's, essentially substituting r for c, but maintaining the expression c and its children. Then c is classified as a random variable and the bijection is added to the state. In the case that c now also satisfies `is_rnd_for_bij`, the potential candidate c is added to the stack to be used for another bijection.

6.5 File expr.ml

The `expr.ml` file contains all functions and data structures necessary to initialize and operate on expressions and to define operators. This is where we added our subtraction operator (with some small changes to the parser and lexer as well) and where a lot of potential for the extension of operators into MASKVERIF still exists. Of special interest are operators and expressions as data types.
We have already shown the expression and expression data structure in Section 5. To summarize, expressions `expr` and expression nodes `exprnode` are recursively defined over

one another to construct a tree. The leaves of this tree are either randoms `rnd` or shares constructed over a parameter `param` indicating input, output, or intermediate share, `int` to indicate the number of shares, and a variable `var`. Additionally, leaves can be public or private variables of type `var` or constants of type `constant`.
The nodes of an expression are operators, either with one, two, or more arguments. To understand these further we will outline the operator data type in Figure 6.1.

```
type operator = {
    op_id    : int;
    op_name  : string;
    op_ty    : (ty list * ty) option;
    op_bij   : bij_kind;
    op_kind  : op_kind
}
```

Figure 6.1: Operator data type in MASKVERIF.

An operator has an ID `op_id` which uniquely identifies it and is assigned when it is constructed. The name of an operator `op_name` is defined by what kind of operator it is and the size of its arguments in bits. An example of this is that a $\wedge$ operator for two 32-bit variables has the name &w32. Additionally, the argument sizes in a bit are specified as `op_ty` and can be 1, 8, 16, 32, or 64 bit. Every operator is either bijective or not, specified by `op_bij`. The value `op_kind` indicates the operation that is performed, currently, these can be `Add` (which is a bitwise $\oplus$), `Mul` (bitwise $\wedge$), `Neg` (bitwise complement), `Sub` (the subtraction mod 2^k we added) and `Other`, which is used for the custom defined operators in gadgets.
The `expr.ml` file contains more functions that handle these data types, but the knowledge we presented is sufficient to understand operators and expressions on a deep enough level to work with MASKVERIF. Expressions are the predominant data type used to represent the symbolic values that MASKVERIF analyses for verification and understanding them will allow both users and programmers to easily use and extend the tool.

7 Conclusions

We will now briefly summarize the contents and contributions of this work and outline some open problems regarding the topic.

7.1 Summary

In this thesis, we extended MASKVERIF, one of the leading formal verification tools for masked implementations, by implementing rules that simplify the observations that MASKVERIF operates on. These rules allowed us to verify two important Boolean-to-arithmetic conversion algorithms with MASKVERIF without the occurrence of false negatives for the first time.

The first algorithm is from Goubin's seminal work in 2001 [Gou01], which started opening up the area of research for Boolean-to-arithmetic masking conversion and vice versa. Since this algorithm was published over 20 years ago, the current security notions of t–probing security, t–non–interference, and t–strong non–interference were not formally verified in Goubin's security proof. We offered a proof of t–probing security for Goubin's B2A algorithm and showed at which point the t–non–interference property is violated. This algorithm is now correctly verified as t–probing secure by MASKVERIF using our simplification rules added to the code.

The second algorithm we focused on is Coron's t–SNI variant of Goubin's B2A algorithm, which was incorrectly assessed as not even t–probing secure by MASKVERIF. Our rules also lead to the correct verification of this algorithm as t–SNI.

Additionally, we offered extensive documentation of MASKVERIF's codebase and showed how it is connected to the pseudocode that Barthe et al. [BBC+19] provide for MASKVERIF version three. This documentation hopefully provides researchers with a helpful tool for an easier understanding of MASKVERIF's verification functions which in turn makes its use or extension more straightforward.

Lastly, we provided several MASKVERIF gadgets for different use cases like first and second-order Boolean-to-arithmetic conversion, first-order arithmetic-to-Boolean conversion, mask refreshing up to tenth-order, and t–SNI secure addition up to fifth-order for register sizes of 1, 8, 16, 32, or 64 bits.

7.2 Discussion and Open Problems

Over the course of this work, many interesting open problems revealed themselves.

- The first open problem is the extension of our first-order rules to correctly verify higher-order Boolean-to-arithmetic conversion algorithms without false negatives. Currently, the higher-order version of Coron's algorithm in [Cor17] still leads to false negatives. Coron was able to correctly verify this algorithm in his tool with rules similar to ours, indicating that while the approach may be correct, MASKVERIF is currently unable to handle the expressions that the algorithm produces. As it currently stands we believe that simply incorporating Coron's rules into MASKVERIF will not fix the false negatives in higher-order B2A and further research on our part is necessary to determine the root cause and potential solutions.

- Secondly, MASKVERIF is currently not able to verify arithmetic-to-Boolean algorithms of any order. A first step here would be to extend MASKVERIF in a way that leads to the correct verification of Goubin's first-order arithmetic-to-Boolean conversion algorithm [Gou01]. Later on, the correct verification of higher-order arithmetic-to-Boolean conversion algorithms can also be pursued.

- Another interesting topic is researching how to minimize the number of false negatives that MASKVERIF's verification produces. It is expected that MASKVERIF's heuristics and simplifications lead to some loss in completeness, however, minimizing the gap in completeness between MASKVERIF and its competitors, while keeping a lead in performance, would drastically improve the quality of the tool further for many use cases.

- Last but not least, many other state-of-the-art tools are now focusing on combined security (passive and active attackers) and it would be interesting to see if MASKVERIF can be extended to incorporate active attackers inducing faults as well. Most other tools again focus on completeness, causing performance to often suffer heavily for combined security. This happens because many tools look at all possible fault positions and then verify the different notions of probing security for all positions. If MASKVERIF could be extended to fulfill the role of a tool for formal verification of combined security that does not offer completeness but also does not incur such a performance loss for combined security, it could be a very useful tool for the verification of larger implementations.

References

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and Francois-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, *Computer Security – ESORICS 2019*, pages 300–318, Cham, 2019. Springer International Publishing.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 457–485. Springer, 2015.

[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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129, 2016.

[BBE+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 *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 354–384. Springer, 2018.

[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 *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part I*, pages 339–368. Springer, 2020.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 22–45, 2018.

[BGG+15] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications*, pages 64–81, Cham, 2015. Springer International Publishing.

[BGR+21] Joppe W Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First-and higher-order implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 173–214, 2021.

[BMRT22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. Ironmask: Versatile verification of masking security. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 142–160. IEEE, 2022.

[BNN+15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia Tokareva, and Valeriya Vitkup. Threshold implementations of small s-boxes. *Cryptography and Communications*, 7:3–33, 2015.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 513–525, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[CG00] Jean-Sébastien Coron and Louis Goubin. On boolean and arithmetic masking against differential power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 231–237. Springer, 2000.

[CGTZ23] Jean-Sebastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of ntru. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 180–211, 2023.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 188–205. Springer, 2014.

[CJRR99a] Suresh Chari, Charanjit Jutla, Josyula R Rao, and Pankaj Rohatgi. A cautionary note regarding evaluation of aes candidates on smart-cards. In *Second Advanced Encryption Standard Candidate Conference*, pages 133–147. Citeseer, 1999.

[CJRR99b] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International workshop on cryptographic hardware and embedded systems*, pages 292–302. Springer, 1999.

[Cor17] Jean-Sébastien Coron. High-order conversion from boolean to arithmetic masking. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 93–114, Cham, 2017. Springer International Publishing.

[Cor18] Jean-Sébastien Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 65–82, Cham, 2018. Springer International Publishing.

[CRZ17] Jean-Sebastien Coron, Franck Rondepierre, and Rina Zeitoun. High order masking of look-up tables with common shares. Cryptology ePrint Archive, Paper 2017/271, 2017. `https://eprint.iacr.org/2017/271`.

[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*, 15:2542–2555, 2020.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: from probing attacks to noisy leakage. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 423–440. Springer, 2014.

[DR99] Joan Daemen and Vincent Rijmen. Resistance against implementation attacks: A comparative study of the aes proposals. In *Proceedings of the Second Advanced Encryption Standard (AES) Candidate Conference*, volume 82, 1999.

[FGMDP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the

presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, Aug. 2018.

[FRBSG22] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. Cini minis: Domain isolation for fault and combined security. Cryptology ePrint Archive, Paper 2022/1131, 2022. `https://eprint.iacr.org/2022/1131`.

[Gou01] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 3–15, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 444–461, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 463–481. Springer, 2003.

[KDVB+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked saber. In *Security and Cryptography for Networks: 13th International Conference, SCN 2022, Amalfi (SA), Italy, September 12–14, 2022, Proceedings*, pages 93–116. Springer, 2022.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KLRBG22] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. In *Post-Quantum Cryptography: 13th International Workshop, PQCrypto 2022, Virtual Event, September 28–30, 2022, Proceedings*, pages 309–326. Springer, 2022.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology —*

CRYPTO '96, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. Silver–statistical independence and leakage verification. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 787–816. Springer, 2020.

[MDS99a] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Investigations of power analysis attacks on smartcards. *Smartcard*, 99:151–161, 1999.

[MDS99b] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 144–157, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium: efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, pages 344–362. Springer, 2019.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M Gammel. Side-channel leakage of masked cmos gates. In *Cryptographers' Track at the RSA Conference*, pages 351–365. Springer, 2005.

[RBFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. Verica - verification of combined attacks automated formal verification of security against simultaneous information leakage and tampering. *IACR Transactions Cryptographic Hardware Embedded Systems (TCHES)*, 2022(4):255–284, 2022.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I 35*, pages 764–783. Springer, 2015.

[RBSS+21] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. Fiver–robust verification of countermeasures against fault injections. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 447–473, 2021.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*, pages 413–427. Springer, 2010.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *IACR International Workshop on Public Key Cryptography*, pages 534–564. Springer, 2019.