OBSCURE: Versatile Software Obfuscation from a Lightweight Secure Element

Darius Mercadier[1], Viet Sang Nguyen[2], Matthieu Rivain[3] and Aleksei Udovenko[4]

[1] Google, Munich, Germany
dmercadier@google.com
[2] Université Jean Monnet, Saint-Étienne, France
viet.sang.nguyen@univ-st-etienne.fr
[3] CryptoExperts, Paris, France
matthieu.rivain@cryptoexperts.com
[4] SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
aleksei@affine.group

Abstract. Software obfuscation is a powerful tool to protect the intellectual property or secret keys inside programs. Strong software obfuscation is crucial in the context of untrusted execution environments (e.g., subject to malware infection) or to face potentially malicious users trying to reverse-engineer a sensitive program. Unfortunately, the state-of-the-art of pure software-based obfuscation (including white-box cryptography) is either insecure or infeasible in practice.

This work introduces OBSCURE, a versatile framework for practical and cryptographically strong software obfuscation relying on a simple stateless secure element (to be embedded, for example, in a protected hardware chip or a token). Based on the foundational result by Goyal *et al.* from TCC 2010, our scheme enjoys provable security guarantees, and further focuses on practical aspects, such as efficient execution of the obfuscated programs, while maintaining simplicity of the secure element. In particular, we propose a new rectangular universalization technique, which is also of independent interest. We provide an implementation of OBSCURE taking as input a program source code written in a subset of the C programming language. This ensures usability and a broad range of applications of our framework. We benchmark the obfuscation on simple software programs as well as on cryptographic primitives, hence highlighting the possible use cases of the framework as an alternative to pure software-based white-box implementations.

Keywords: Obfuscation · Secure Element · White-Box Cryptography · VBB Security

1 Introduction

In our modern world, many devices constantly run programs processing sensitive data. However, most of the time, the underlying execution environment cannot be fully trusted. Indeed, in modern systems running multiple third-party applications, a malicious software may be able to infect the environment through numerous security vulnerabilities. Another threat is the disclosure of intellectual property in a context where a legitimate user turn to be malicious and try to reverse-engineer the program.

Software obfuscation aims to protect a program by making it unintelligible to a potential adversary while maintaining its functionality. In the context of a cryptographic program, it should notably be hard to extract its internal secrets, which is further known as *white-box*
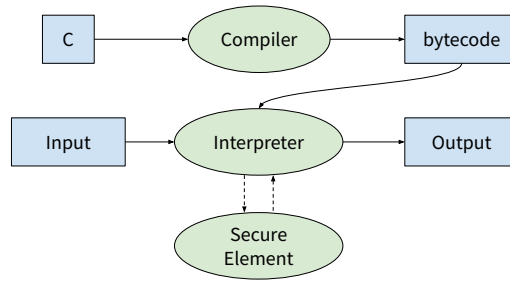
Figure 1: High-level view of our obfuscation solution.

cryptography. Proposed in seminal works of Chow *et al.* [CEJv03, CEJvO02], white-box implementations of cryptographic primitives (such as the standardized AES block cipher) aim to protect the secret keys from extraction by an adversary with full access to the implementation. Unfortunately, all initial and consequent academic white-box designs were broken by practical (and often fully automated) attacks, see *e.g.* [BGEC04, LRD+14, BHMT16, BU18, GRW20].

The motivation for this work arises from the need of practical *and* secure software obfuscation for numerous applications while state-of-the-art solutions are not fulfilling the need. On the one hand, current white-box cryptography solutions desperately lack security, illustrated by the absence of unbroken schemes in academia and, for example, by recent WhibOx competitions [PCY+17, BGK+19, KGP+21]. On the other hand, theoretical cryptographic obfuscation [BGI+01], and in particular *indistinguishability obfuscation*, is far from being feasible in practice despite recent breakthroughs proposing provably secure constructions under well-founded hardness assumptions [JLS21].

A direction this work takes is a partial delegation of trust to a tamper-proof *secure element* (SE). The underlying idea is to build secure software obfuscation relying on interaction with a secure embedded chip or a hardware token. Prominent real-world examples in this direction are the trusted execution environments, such as Intel SGX, AMD Secure Technology or ARM TrustZone. However, security of these hardware elements is uncertain due to numerous side-channel attacks [FYDX21] arising from the extremely high internal complexity of the chips. In fact, much simpler tamper-proof hardware tokens are sufficient for achieving strong obfuscation, as was shown by Goyal, Ishai, Sahai, Venkatesan and Wadia at TCC 2010 [GIS+10]. However, the focus of the latter work is purely theoretical and a straight application of this scheme would be widely inefficient.

In this work, we design and implement OBSCURE, a software obfuscation scheme with a focus on application features and simplicity of the SE functionality. We build our solution by reworking and extending the scheme from TCC 2010. OBSCURE takes a C source code as input and produces an obfuscated program (a bytecode), which can be distributed and fed into an *interpreter* evaluating the program on any given input through interaction with an SE (see Figure 1). The SE is stateless and independent of programs being obfuscated, thus being fully reusable.

Our contribution. Our contribution can be summarized as the design and implementation of practical software obfuscation based on a *lightweight* SE. By lightweight we mean that, besides basic cryptographic computation (to secure its input-output data), the SE functionality is only required to perform very simple computation such as a few RISC-like instructions involving a limited set of registers. Except for a pair of (hardcoded) cryptographic keys, the SE functionality is *stateless*: it needs no storage between any two invocations. Moreover, it only uses a small segment of RAM and does not need to deal with memory management. We explore the obfuscation of programs without data-dependent

memory accesses. While this reduces the scope of supported programs, it allows us to obtain a much simpler, efficient, and easier to protect SE functionality than in schemes supporting RAM programs and relying on *e.g.* oblivious RAM, an advanced (and rather inefficient) cryptographic primitive. Our scheme is particularly efficient in protecting programs in the scenario where only the hardcoded secrets need to be hidden, i.e., where the program structure is not sensitive (e.g., white-box cryptography, pre-trained neural networks), but can also be used for general obfuscation. Our main contributions are the following:

1. **Design and optimizations**: At its core, OBSCURE relies on the scheme from TCC 2010. We optimize it and work out all practical aspects. First, we generalize it to address modern programs made of typical CPU instructions instead of Boolean circuits. A major optimization includes batch evaluation of instructions in terms of *multi-instructions*, which are short snippets of instructions processing a fixed number of values. We develop a *clusterization* procedure that combines instructions into multi-instructions. We also simplify the original scheme by reducing the number of different SE queries down to three (instead of five in the original scheme). For practical purpose, we remove the need of a shared key between the SE and the obfuscator by relying on a public key encryption scheme, which is invoked for a single encryption per obfuscation and for a single decryption per obfuscated program evaluation.

2. **Rectangular universalization**: For programs with sensitive structure and data flow, we develop a tailored *universalization* method, allowing to serialize an original program as a hardcoded constant into a universal evaluating program. We thus reduce the goal of general obfuscation to the basic constant-protecting obfuscation (a.k.a. *white-box obfuscation*). Rectangular universalization is based on coercing the program's circuit into a rectangular (fixed-width) shape and obfuscating the wirings using *permutation-duplication networks*. We design such a network tailored to our scheme, using the well-known Beneš permutation network as a component. For the latter scheme, we show how it can be efficiently compressed by using multi-instructions (in this case, essentially, fixed-size permutation gates). We believe that this contribution is of independent interest and could be useful to other practical implementation contexts of universal circuits / programs.

3. **Implementation of a C source code obfuscator**: Our implementation of OB-SCURE includes a compiler from a program in C^- (a subset of the C language) into an obfuscated bytecode, a bytecode interpreter interacting with an SE, and a software library (in C) implementing the SE API. The source code is publicly available at

   https://github.com/CryptoExperts/OBSCURE

   Direct obfuscation of C programs highlights usability of our solution and a broad range of applications. Our implementation of OBSCURE is benchmarked on different programs from small to fairly large size which gives a good illustration of the scaling of our approach and provides some insights on the possible trade-offs.

Potential applications.   We show that our scheme achieves the strongest form of obfuscation (assuming that the underlying SE is tamper resistant), which is known as *virtual black box* (VBB) obfuscation [BGI$^+$01]. This implies that the adversary learns nothing more from the obfuscated program than from an oracle access to a "black box" computing the program. From this ground, our scheme can be applied to multiple contexts and provide the highest possible security to a software program. In the context of an encryption or decryption program, besides protecting its secrets, our obfuscator can make it *one-way*,

incompressible and/or *traceable* [DLPR14]. We notably report and benchmark the implementation of a *collusion-resistant* traceable AES decryption program using OBSCURE. Another promising application is the protection of the sensitive weights of a pre-trained neural network (or other machine learning model). This use case is particularly amenable to our constant-protecting obfuscation (obfuscation in "white-box mode"). We show-case and benchmark such an application on a multilayer perceptron (MLP) pre-trained on the MNIST dataset to recognize handwritten digits. OBSCURE thus enables embedded AI modules provably protected against the disclosure of their trained parameters. We further discuss how OBSCURE could be an interesting solution for confidential cloud computing. In this scenario, one compiles and obfuscates a program locally and delegate its computation to a public cloud without disclosing the program content or internal computation to the cloud. While we just cited a few, many more applications can benefit strong obfuscation in practice. The interested reader is referred to [BGI+01, SW14, Bar16, HB15] for further examples.

Related works. Since current pure software-based obfuscation techniques are either insecure or infeasible, we focus on related works about software obfuscation using secure hardware. These related works vary by the nature and complexity of the underlying secure hardware. Our work is heavily inspired by the work of Goyal *et al.* [GIS+10]. We improve this scheme in several ways to make it practical and provide a concrete implementation featuring a multi-stage compiler from a C source code into an obfuscated bytecode, a bytecode interpreter and a software proof-of-concept SE implementation.

Towards the other side of the spectrum are obfuscation schemes relying on SEs of medium complexity. PHANTOM [MLS+13] (CCS 2013), GhostRider [LHM+15] (ASPLOS 2015) and HOP [NFR+17] (NDSS 2017) are variants of secure processors and have a rather similar architecture. GhostRider only aims to protect the data (inputs, intermediate values and outputs), somewhat similar to our white-box mode. HOP also aims to protect the code, and, for this purpose, inserts dummy instructions to prevent timing leakage. GhostRider and HOP can obfuscate generic programs in the RAM (Random Access Machine) model. The support of random memory accesses is implemented through oblivious RAM, which adds significant complexity and performance overhead to the SE (3000 secure CPU cycles per a memory access, as reported in [NFR+17]). In addition, the GhostRider processor has more than 64 KB of secure internal memory; the HOP processor has more than 512 KB of secure internal memory. Our SE functionality, on the other side, typically relies on a few dozens of 32-bit registers (< 1.3 KB state in the largest proposed version, excluding the decryption/encryption parts). By being much simpler, our SE functionality is easier to protect against side-channel attacks and can hence lead to higher security assurance.

Let us finally mention other works based on existing generic "trusted execution environment" such as the Intel SGX (*Software Guard Extensions*): an obfuscation engine OBFUSCURO [AJX+19] (NDSS 2019), a functional encryption system IRON [FVBG17] (ACM CCS 2017). While very efficient and powerful, these schemes rely on a secure execution environment with extremely high complexity, which, in practice, unavoidably leads to security vulnerabilities [FYDX21].

Outline. Section 2 formalizes the notion of secure element (SE) -based obfuscation, describes the OBSCURE design and states its security. Section 3 describes our universalization technique which achieves better efficiency than known universal circuit constructions by taking advantage of the considered computational model. Section 4 presents the software architecture of our obfuscator. It details the syntax of input programs, the compilation pipeline and different intermediate representations of the program along the obfuscation process. Finally, Section 5 showcases applications of OBSCURE and gives some benchmarks for the obfuscation of several programs.

2 Strong Obfuscation with Secure Element

2.1 Definitions

For any two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$, we write $\mathcal{D}_1 \approx_{(t,\varepsilon)} \mathcal{D}_2$ to mean that no algorithm running in time at most t can distinguish the two distributions with probability greater than $\frac{1}{2} + \varepsilon$ (or equivalently with advantage greater than ε). Along this paper, the latter notion of computational closeness shall be used for $\mathcal{D}_1$, $\mathcal{D}_2$, t and ε being functions of a security parameter λ (which shall often be implicit in the presentation). In the following, an *adversary* (usually denoted $\mathcal{A}$) and a *simulator* (usually denoted $\mathcal{S}$) both refer to a probabilistic algorithm.

Let $\mathcal{L}$ be a formal programming language. For an algorithm $\mathcal{O}$, we shall denote $\mathcal{L}^{\mathcal{O}(\cdot)}$ the programming language obtained by augmenting $\mathcal{L}$ with oracle calls to $\mathcal{O}$. For any two programs $P_1, P_2 \in \mathcal{L}$, we shall denote $P_1 \equiv P_2$ to mean that the two programs are functionally equivalent. For any $P \in \mathcal{L}$, we shall further denote $\overline{P}$ the program obtained by setting all the constants of P to 0.[1] As a particular example, if P is a program computing a cryptographic algorithm with hardcoded secret key (embedded as instructions' constants) then $\overline{P}$ does not reveal anything about P's secret.

SE-based obfuscation.   We now formally define the concept of obfuscation based on a secure element. We consider a context where the SE is associated with a key pair $(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}})$. The public key $\mathsf{pub}_{\mathsf{SE}}$ is public information possibly authenticated by the manufacturer. The private key $\mathsf{priv}_{\mathsf{SE}}$ is stored in the SE as secret information. The SE is assumed to be tamper resistant and is hence modeled as an oracle to which the obfuscated program as well as a potential adversary only have a black-box access.

Definition 1 (SE-based Obfuscation)**.** Let $\mathcal{L}$ be a formal programming language. An SE-based obfuscator for $\mathcal{L}$ is a triplet of PPT algorithms $(\mathsf{KeyGen}, \mathsf{SE}, \mathsf{Obf})$ defined as follows:

- **Key generation:** on input 1^λ, KeyGen outputs a public-secret key pair $(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}})$,

- **Secure element:** on input a secret key $\mathsf{priv}_{\mathsf{SE}}$ and a request r, SE outputs an answer a,

- **Obfuscator:** on input a program $P \in \mathcal{L}$ and an SE public key $\mathsf{pub}_{\mathsf{SE}}$, Obf outputs an obfuscated program $\widehat{P} \in \mathcal{L}^{\mathsf{SE}(\mathsf{priv}_{\mathsf{SE}}, \cdot)}$.

This triplet of algorithms satisfies the following properties:

- **Functional correctness.** For any $P \in \mathcal{L}$, we have $\widehat{P}^{\mathsf{SE}(\mathsf{priv}_{\mathsf{SE}}, \cdot)} \equiv P$ with probability 1 for $(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}}) \leftarrow \mathsf{KeyGen}()$ and $\widehat{P} \leftarrow \mathsf{Obf}(P, \mathsf{pub}_{\mathsf{SE}})$.

- **VBB obfuscation security.** For any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ of similar running time, such that for any program $P \in \mathcal{L}$ of size $|P|$:

$$\mathcal{A}^{\mathsf{SE}(\mathsf{priv}_{\mathsf{SE}}, \cdot)}(\mathsf{pub}_{\mathsf{SE}}, \widehat{P}) \approx_{(t,\varepsilon)} \mathcal{S}^{P(\cdot)}(|P|)$$

  where $(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}}) \leftarrow \mathsf{KeyGen}()$ and $\widehat{P} \leftarrow \mathsf{Obf}(P, \mathsf{pub}_{\mathsf{SE}})$. The obfuscation is said to be (t, ε)-VBB secure.

We shall further consider a relaxed notion of security in which the program structure might be revealed to the adversary but the obfuscation is meant to protect the program data:

---

[1]More formally, $\mathcal{L}$ is assumed to include instructions which can take constant operands on some constant domain, *e.g.* $\mathbb{N} \cap [0, 256)$, and $\overline{P}$ is similar to P with all constants set to 0. If 0 is outside the constant domain, any other fixed constant can be used instead for the definition of $\overline{P}$.

- **VBB white-box security.** *For any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ of similar running time, such that for any program $P \in \mathcal{L}$:*

$$\mathcal{A}^{SE(\mathsf{priv}_{SE}, \cdot)}(\mathsf{pub}_{SE}, \widehat{P}, \overline{P}) \approx_{(t,\varepsilon)} \mathcal{S}^{P(\cdot)}(\overline{P}) \ ,$$

where $(\mathsf{pub}_{SE}, \mathsf{priv}_{SE}) \leftarrow KeyGen()$ and $\widehat{P} \leftarrow Obf(P, \mathsf{pub}_{SE})$, and where $\overline{P}$ is the zeroized-constants version of P as defined above. The obfuscation is said to be (t, ε)-white-box-VBB secure.

Computational model.   We shall consider a formal programming language $\mathcal{L}$ defining a set of static single assignment (SSA) straight-line programs. In such programs, each line computes a few Boolean and/or arithmetic instructions on w-bit words for some $w \in \mathbb{N}$, which we shall then call a *multi-instruction*. For some input-output length $\ell \in \mathbb{N}$, a multi-instruction computes a function from a base

$$\mathcal{F} \subseteq \left\{ f : \left(\{0,1\}^w\right)^\ell \to \left(\{0,1\}^w\right)^\ell \right\} \ .$$

This multi-instruction base can be thought of as the set of functions generated by sequences of a few base instructions. The exact definition of $\mathcal{F}$ depends on the instruction set, the maximal number of instructions, and the number of internal registers. We shall specify the definition of $\mathcal{F}$ in Section 4 although we stress that our obfuscator can work for any definition of $\mathcal{F}$ and one might choose another $\mathcal{F}$ depending on the context.

Inputs and outputs of multi-instructions are read from and written to a static single assignment memory $\mathtt{m}$ which is a two-dimensional array such that $\mathtt{m}[i][j]$ stores the jth output element of the ith multi-instruction (while $\mathtt{m}[i]$ denotes the full output). For a multi-instruction base $\mathcal{F}$ and associated parameters w, ℓ, a program $P \in \mathcal{L}$ of size s, input length n and output length m is composed of three following stages:

1. *Writing input:* the input $(x_1, \ldots, x_n) \in (\{0,1\}^w)^n$ is written at the beginning of the memory

$$\mathtt{m}[1] \leftarrow (x_1, \ldots) \ ; \ \ldots \ ; \ \mathtt{m}[L] \leftarrow (\ldots, x_n)$$

   where $L = \lceil n/\ell \rceil$.

2. *Computation:* the program executes a sequence of multi-instructions

$$\mathtt{m}[\nu] \leftarrow f_\nu(\mathtt{m}[i_{\nu,1}][j_{\nu,1}], \ldots, \mathtt{m}[i_{\nu,\ell}][j_{\nu,\ell}])$$

   for $\nu \in [L+1, L+s]$, where $f_\nu \in \mathcal{F}$, $i_{\nu,1}, \ldots, i_{\nu,\ell} \in [1, \nu)$ and $j_{\nu,1}, \ldots, j_{\nu,\ell} \in [1, \ell]$.

3. *Reading output:* the output $(y_1, \ldots, y_m) \in (\{0,1\}^w)^m$ is read from certain memory cells:

$$(y_1, \ldots) \leftarrow \mathtt{m}[\mu_1] \ ; \ \ldots \ ; \ (\ldots, y_m) \leftarrow \mathtt{m}[\mu_{L'}]$$

   where $L' = \lceil m/\ell \rceil$ and $\mu_1, \ldots, \mu_{L'} \in [1, L+s]$.

In practice, a the function $f \in \mathcal{F}$ is encoded into a *bytecode* which is a bit string uniquely identifying f. The bytecode of the νth multi-instruction shall be denoted f_ν as the underlying function by abusing notations. The tuple defining the input indices is further called the *input identity* of the multi-instruction and denoted

$$\mathrm{ID}_{\mathrm{in}}^\nu := ((i_{\nu,1}, j_{\nu,1}), \ldots, (i_{\nu,\ell}, j_{\nu,\ell})) \ . \tag{1}$$

Example 1. Figure 2 depicts an example with two multi-instructions MI_1 and MI_2, each contains 2 instructions. The input identity of MI_1 is $\mathrm{ID}_{\mathrm{in}}^1 = ((1,1),(1,2))$. This can be understood as follows: the multi-instruction MI_1 takes as input the first and the second elements of $\mathtt{m}[1]$. In other words, $\mathrm{ID}_{\mathrm{in}}^\nu$ tells us where MI_ν's input values come from. Similarly, the input identity of MI_2 is $\mathrm{ID}_{\mathrm{in}}^2 = ((2,1),(2,2))$.
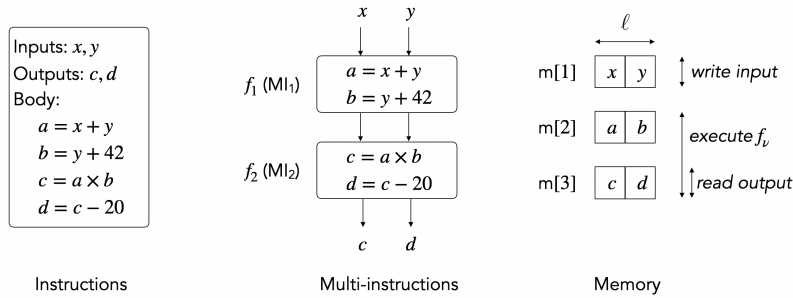
Figure 2: Example of computational model.

2.2 Obfuscator Design

OBSCURE follows the same lines as the construction of [GIS+10]. The principle is to manipulate encrypted intermediate variables (Boolean values in [GIS+10], w-bit words in our construction) and to call the SE to perform operations on these encrypted values. Additionally, the design must take into account an adversary tampering with the origin of encrypted input values (wires in [GIS+10]) and/or swapping multi-instructions (gates in [GIS+10]). The design must also be able to prevent an adversary from swapping a specific intermediate value between two different executions (corresponding to different program inputs). We use a so-called *execution identity* which is bound to the program input (in a non-forgeable way) and further to all the intermediate encrypted values. Hereafter, we present in details the design of our obfuscator.

Cryptographic primitives.   To simultaneously ensure the data confidentiality and authenticity, we use authenticated encryption with associated data (AEAD) throughout our construction. Let N be a nonce, A be associated data, K be a symmetric key, M be a plaintext, C be a ciphertext and R be the result of validating the message authentication code (MAC). R can receive the value of either "valid" or "invalid" (also denoted $\perp$). We denote the functions of encryption and decryption by $C \leftarrow \mathsf{AEnc}_K(N, A, M)$ and $(M, R) \leftarrow \mathsf{ADec}_K(N, A, C)$, respectively. The ciphertext obtained from the encryption function implicitly includes the message authentication code and the decryption function implicitly includes the verification of the MAC. The plaintext and/or associated data can be set to the empty string, which is denoted $\emptyset$. Our solution also makes use of public-key encryption (PKE) scheme. For an asymmetric key pair (pub, priv), we denote $C \leftarrow \mathsf{Enc}_{\mathsf{pub}}(M)$ the public-key encryption of a plaintext M under a public key pub, and $M \leftarrow \mathsf{Dec}_{\mathsf{priv}}(C)$ the PKE decryption of a ciphertext C using the private key priv. Finally, we shall use a hash function Hash and further denote $\mathsf{Hash}_i(\cdot) := \mathsf{Hash}(i \parallel \cdot)$ for $i \in [0, 3]$ where the prefix i is encoded on 2 bits, and where $\parallel$ denotes the concatenation operator.

We assume standard security properties for these primitives: authenticity and privacy of the AEAD scheme, ciphertext indistinguishability (under chosen plaintext attacks) for the PKE scheme, and collision resistance of the hash function (see *e.g.* [BR05, RBBK01]). Their definitions are recalled in Appendix A.

Secure element.   In OBSCURE, the key generation produces a PKE key pair (pub$_{\mathsf{SE}}$, priv$_{\mathsf{SE}}$) and an AEAD key K_{SE}. Formally, pub$_{\mathsf{SE}}$ is the public key of the SE while its secret key is composed of the pair (priv$_{\mathsf{SE}}$, K_{SE}). The latter is stored in the SE and used to answer incoming requests. A request received by the SE is composed of a label followed by a
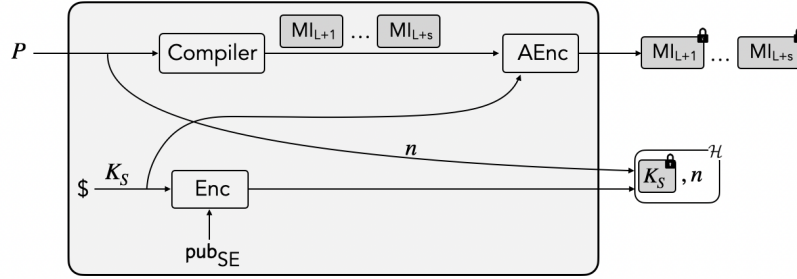
Figure 3: Obfuscation process.

sequence of arguments:

$$\mathsf{SE}(\langle\text{label}\rangle, \langle\text{argument } 1\rangle, \langle\text{argument } 2\rangle, \ldots)$$

The SE answers three type of requests with labels "Start", "Input" and "Eval" whose associated functionalities are described in Algorithm 1, Algorithm 2, Algorithm 3.

Remark 1. In practice, the key pair ($\mathsf{pub}_{\mathsf{SE}}$, $\mathsf{priv}_{\mathsf{SE}}$) is generated by the manufacturer during a key ceremony and embedded in the device together with a certificate issued by the manufacturer (or, alternatively, $\mathsf{pub}_{\mathsf{SE}}$ is distributed among developers of obfuscated programs). This setup is similar to the best practices of the secure hardware industry (*e.g.*, hardware security modules, secure smart cards). In principle, alternative key distribution schemes are possible and can be applied together with our core design.

Obfuscation process. On an input program P (following the computational model of Subsection 2.1), the obfuscator produces an authenticated and encrypted bytecode to which some header is prepended, as depicted in Figure 3. The obfuscated program consists of this data together with a fixed (public) *interpreter* program. The latter parses the encrypted bytecode into a sequence of encrypted multi-instructions and makes requests to the SE in order to sequentially execute them and finally obtain the output.

First of all, the obfuscator randomly samples a symmetric key K_{S}. This key is used to encrypt the program bytecode which shall hence be securely shared with the SE (to enable the bytecode decryption). This key is hence referred to as the *shared key*.

Secondly, the obfuscator creates a header $\mathcal{H}$ which contains the information about the number of inputs n of the program P and the ciphertext obtained by encrypting the shared key K_{S} using the public key $\mathsf{pub}_{\mathsf{SE}}$ of the SE:

$$\mathcal{H} = (n, C_{\mathcal{H}}) \quad \text{with} \quad C_{\mathcal{H}} \leftarrow \mathsf{Enc}_{\mathsf{pub}_{\mathsf{SE}}}(K_{\mathsf{S}})$$

Thirdly, the program bytecode is authenticated and encrypted. At evaluation time, the SE should only accept such an evaluation request if the given input of the current multi-instruction (MI) comes from the correct previous MIs, namely it should authenticate the MI of origin of each encrypted input value and verify that it matches the input identity of the current MI. Thus, the input identity $\mathrm{ID}_{\mathrm{in}}^{\nu}$ (*i.e.* the tuple of indices of the multi-instruction's input) must be authenticated. Besides, we use a Boolean flag b_R^{ν}, that we call a *revelation flag* and which is set whenever the current multi-instruction's output is part of the program's output and must hence be returned in plain by the SE. Formally, an authenticated and encrypted multi-instruction is defined as

$$\mathrm{MI}_{\nu} = (\nu, A_{\nu}, C_{\nu}) \text{ where } \begin{cases} A_{\nu} = (b_R^{\nu}, \mathrm{ID}_{\mathrm{in}}^{\nu}) \\ C_{\nu} = \mathsf{AEnc}_{K_{\mathsf{S}}}(\nu, A_{\nu}, f_{\nu}) \end{cases} \tag{2}$$

where we abuse notations by denoting f_ν the bytecode of the multi-instruction.

The obfuscated program is composed of the header $\mathcal{H}$, the sequence of authenticated and encrypted multi-instructions $\mathrm{MI}_{L+1}, \ldots, \mathrm{MI}_{L+s}$, and a fixed (public) program called the *interpreter* which is depicted hereafter.

Remark 2. We stress that, once the obfuscation process completed, the obfuscator should securely destroy the shared key K_S. This way, K_S is only kept in encrypted form in the obfuscated program header which can only be decrypted by the target secure element.

Obfuscated program evaluation.   In the evaluation of an obfuscated program, each w-bit intermediate value (including the program input) is stored in the memory in an encrypted form and is called an *encrypted word*. Those encryptions use the secret key K_{SE} stored in the SE. To execute a multi-instruction f_ν, its (encrypted) bytecode and the corresponding encrypted input words are appended to a request sent to the SE. The SE is then able to decrypt the provided data and execute f_ν internally. The output elements of the multi-instruction are encrypted by the SE and stored as new encrypted words in the program memory (with a special treatment for the program output which must be returned in plain).

In the design, an *execution identity* E_{ID} prevents an adversary from using an encrypted intermediate value to modify the corresponding encrypted intermediate value in another execution. The execution identity is unique for an input of an obfuscated program so that it is hard to have two different executions with the same execution identity. We give hereafter a detailed description of the evaluation of an obfuscated program. Specifically, we depict the different steps of the interpreter. Each time the interpreter calls the SE, we describe the associated computation.

The input of the interpreter is composed of the obfuscated program header $\mathcal{H}$ and authenticated and encrypted multi-instructions $\mathrm{MI}_{L+1}, \ldots, \mathrm{MI}_{L+s}$ as well as the program input $x_1, \ldots, x_n$. It shall return $(y_1, \ldots, y_m) = P(x_1, \ldots, x_n)$.

Step 1: This step is the preparation for the input writing stage of the computational model of Subsection 2.1 and for the derivation of the execution identity. The program input words $x_1, \ldots, x_n$ are grouped into L batches: $X_1 = (x_1, \ldots, x_\ell), \ldots, X_L = (x_{(L-1)\ell+1}, \ldots, x_{L\ell})$ where $L = \lceil n/\ell \rceil$. The last batch X_L is padded with 0's if n is not a multiple of ℓ. We also use the notation $X_{i,j}$ to refer to the jth element of the ith batch, as $X_{i,j} = x_{(i-1)\ell+j}$ where $1 \leq i \leq L$ and $1 \leq j \leq \ell$. The interpreter then computes the *chain of hashes* $H_1, \ldots, H_L$ where

$$H_i = \mathsf{Hash}(H_{i-1} \| X_i)$$

for $1 \leq i \leq L$ and H_0 is the all-zero string of the hash length (see Figure 4).

Step 2: Given the program header $\mathcal{H}$ and H_L, the interpreter sends to SE the request "Start" as shown in Algorithm 1. On such a request, the SE first derives an execution identity E_{ID} by hashing H_L and the header $\mathcal{H}$. All the subsequent AEAD ciphertexts are then bound to the execution identity so that it is hard to replay an encrypted word from one execution to another (corresponding to a different input of the program).

Next, the ciphertext $C_{\mathcal{H}}$ is decrypted to obtain the shared key K_S which is encrypted again using the symmetric key K_{SE} of the SE (lines 2-4). The ciphertext of the shared key is called the *execution key* E_K. It then computes the final element M_L^{in} of an *input chain MAC* (lines 6-8). This input chain MAC is presented in detail in the next step. At the end of this step, the SE returns the execution key E_K, the execution identity E_{ID} and the final input MAC M_L^{in}.

Step 3: The goal of this step is to obtained authenticated and encrypted versions of the input words $\{X_{i,j}\}$ under the SE secret key K_{SE} while using the execution identity E_{ID} and the indexes (i, j) as associated authenticated data in order to avoid a *swapping attack* (*i.e.*
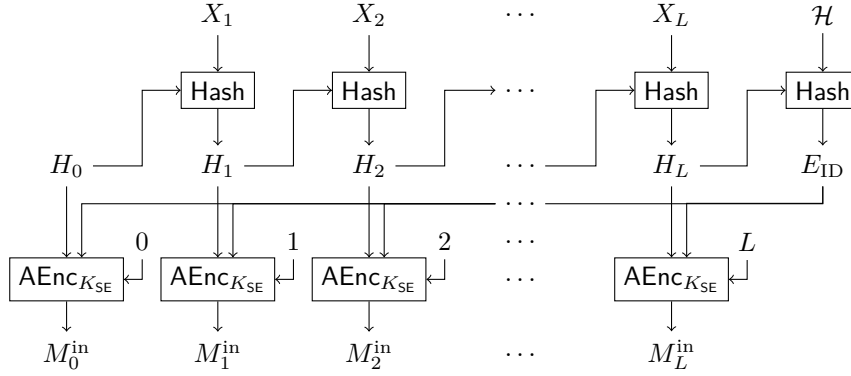
Figure 4: Hash chain and input chain MAC.

Algorithm 1 $\mathsf{SE}(\text{``Start''}, \mathcal{H}, H_L)$

1: $E_{\mathrm{ID}} \leftarrow \mathsf{Hash}_0(H_L \,\|\, \mathcal{H})$
2: $(n, C_{\mathcal{H}}) \leftarrow \mathcal{H}$
3: $K_{\mathrm{S}} \leftarrow \mathsf{Dec}_{\mathrm{priv}_{\mathsf{SE}}}(C_{\mathcal{H}})$
4: $E_{\mathrm{K}} \leftarrow \mathsf{AEnc}_{K_{\mathsf{SE}}}(\mathsf{Hash}_1(E_{\mathrm{ID}}), \emptyset, K_{\mathrm{S}})$
5: $L \leftarrow \lceil n/\ell \rceil$
6: $N_L^{\mathrm{in}} \leftarrow \mathsf{Hash}_2(E_{\mathrm{ID}} \,\|\, L)$
7: $A_L^{\mathrm{in}} \leftarrow (H_L \,\|\, L \,\|\, E_{\mathrm{ID}})$
8: $M_L^{\mathrm{in}} \leftarrow \mathsf{AEnc}_{K_{\mathsf{SE}}}(N_L^{\mathrm{in}}, A_L^{\mathrm{in}}, \emptyset)$
9: **return** $E_{\mathrm{K}}, E_{\mathrm{ID}}, M_L^{\mathrm{in}}$

---

an attack consisting in swapping intermediate variables within an execution or between different executions). To enforce the use of the associated data $A_{i,j} = (i \,\|\, j \,\|\, E_{\mathrm{ID}})$ in the encryption of $X_{i,j}$, we rely on the backward computation and verification of the *input chain MAC* $M_0^{\mathrm{in}}, \ldots, M_L^{\mathrm{in}}$ associated to the chain of hashes $H_0, \ldots, H_L$ (see Figure 4).

For i from L to 1, the interpreter sends an "Input" request to the SE with E_{ID}, i, H_{i-1}, X_i and M_i^{in} where

$$M_i^{\mathrm{in}} = \mathsf{AEnc}_{K_{\mathsf{SE}}}(N_i^{\mathrm{in}}, A_i^{\mathrm{in}}, \emptyset)$$

with $A_i^{\mathrm{in}} = (H_i \,\|\, i \,\|\, E_{\mathrm{ID}})$ and $N_L^{\mathrm{in}} = \mathsf{Hash}_2(E_{\mathrm{ID}} \,\|\, i)$. We notice that the definition of M_L^{in} in the step 2 matches the above definition of M_i^{in} when $i = L$. The computation of the SE when receiving this request is depicted in Algorithm 2. It recomputes $H_i = \mathsf{Hash}_0(H_{i-1} \,\|\, X_i)$, then verifies the MAC M_i^{in} with respect to this hash (lines 3-7) and computes M_{i-1}^{in} (lines 8-10). Let us stress that in the case a wrong X_i or H_{i-1} is inputted to the SE, the recomputation of H_i shall not match the associated data of M_i^{in} and its verification (AEAD decryption) shall fail with overwhelming probability.

Next, the SE encrypts the input words grouped in X_i as follows. For j from 1 to ℓ, the input word $X_{i,j}$ is encrypted using K_{SE} with the execution identity E_{ID} and the origin of this input word (j-th element of i-th batch) in the associated data $A_{i,j} = (i \,\|\, j \,\|\, E_{\mathrm{ID}})$ and the nonce $N_{i,j} = \mathsf{Hash}_3(E_{\mathrm{ID}} \,\|\, i \,\|\, j)$ (lines 11-14), we have:

$$C_{i,j} = \mathsf{AEnc}_{K_{\mathsf{SE}}}(N_{i,j}, A_{i,j}, X_{i,j}) \ . \tag{3}$$

The SE finally returns M_{i-1}^{in} and the encrypted words $C_{i,j}$, $1 \le j \le \ell$ to the interpreter. The encrypted words are stored in memory: $\mathtt{m}[i] \leftarrow (C_{i,1}, \ldots, C_{i,\ell})$. Note that the interpreter's memory is composed of memory cells storing ciphertexts instead of w-bit words as in the original program.

At the end of Step 3, once the "Input" requests have been made for i from L to 1, all the encrypted input words are stored at the beginning of the interpreter's memory in $\mathtt{m}[1], \ldots, \mathtt{m}[L]$. This matches the input writing step of the computational model depicted in Subsection 2.1.

Algorithm 2 $\mathsf{SE}(\text{"Input"}, E_{\mathrm{ID}}, i, H_{i-1}, X_i, M_i^{\mathrm{in}})$

1: **if** $i < 1$ **then return** $\bot$
2: **if** $i = 1$ and $H_{i-1} \neq 0$ **then return** $\bot$
3: $H_i = \mathsf{Hash}_0(H_{i-1} \parallel X_i)$
4: $A_i^{\mathrm{in}} = (H_i \parallel i \parallel E_{\mathrm{ID}})$
5: $N_i^{\mathrm{in}} = \mathsf{Hash}_2(E_{\mathrm{ID}} \parallel i)$
6: $(\emptyset, R_i^{\mathrm{in}}) \leftarrow \mathsf{ADec}_{K_{\mathsf{SE}}}(N_i^{\mathrm{in}}, A_i^{\mathrm{in}}, M_i^{\mathrm{in}})$
7: **if** $R_i^{\mathrm{in}} = \bot$ **then return** $\bot$
8: $N_{i-1}^{\mathrm{in}} \leftarrow \mathsf{Hash}_2(E_{\mathrm{ID}} \parallel i - 1)$
9: $A_{i-1}^{\mathrm{in}} \leftarrow (H_{i-1} \parallel i - 1 \parallel E_{\mathrm{ID}})$
10: $M_{i-1}^{\mathrm{in}} = \mathsf{AEnc}_{K_{\mathsf{SE}}}(N_{i-1}^{\mathrm{in}}, A_{i-1}^{\mathrm{in}}, \emptyset)$
11: **for** j from 1 to ℓ **do**
12:     $N_{i,j} \leftarrow \mathsf{Hash}_3(E_{\mathrm{ID}} \parallel i \parallel j)$
13:     $A_{i,j} \leftarrow (i \parallel j \parallel E_{\mathrm{ID}})$
14:     $C_{i,j} \leftarrow \mathsf{AEnc}_{K_{\mathsf{SE}}}(N_{i,j}, A_{i,j}, X_{i,j})$
15: **return** $M_{i-1}^{\mathrm{in}}, (C_{i,1}, \ldots, C_{i,\ell})$

Step 4: The interpreter sends "Eval" requests to the SE to sequentially execute the multi-instructions $\mathrm{MI}_{L+1}, \ldots, \mathrm{MI}_{L+s}$. For each multi-instruction $\mathrm{MI}_\nu = (\nu, A_\nu, C_\nu)$, the interpreter provides the SE with the execution identity E_{ID}, the execution key E_{K} and the encrypted words of input indices extracted from the input identity $\mathrm{ID}_{\mathrm{in}}^\nu$ (see Equation 1). We note that these encrypted words are looked up from the interpreter's memory as follows:

$$(C_1^*, \ldots, C_\ell^*) \leftarrow (\mathtt{m}[i_{\nu,1}][j_{\nu,1}], \ldots, \mathtt{m}[i_{\nu,\ell}][j_{\nu,\ell}])$$

(where C_k^* is a lighter notation for $C_{i_{\nu,k}, j_{\nu,k}}$).

Upon such a request, the SE performs the computation depicted in Algorithm 3. In this algorithm, it first decrypts the shared key K_{S} from E_{K} (lines 1-2). From the shared key, the SE decrypts the bytecode f_ν and further verifies the authenticity of the multi-instruction data which is b_R^ν, $\mathrm{ID}_{\mathrm{in}}^\nu$ and f_ν (lines 3-6). It then decrypts the encrypted input words by deriving the nonce and associated data from the authenticated input identity $\mathrm{ID}_{\mathrm{in}}^\nu$ (lines 7-12). If any of these decryptions fail, it stops and returns a failure. Otherwise, the function f_ν is evaluated on the obtained plain values $(u_1, \ldots, u_\ell)$, which gives the words $(v_1, \ldots, v_\ell)$ corresponding to the ν-th multi-instruction plain output (line 13). If the revelation flag b_R^ν is set, these output words are returned in plain (lines 14-15). Otherwise, they are encrypted into $C_{\nu,1}, \ldots, C_{\nu,\ell}$ (following Equation 3) before being returned to the interpreter (lines 17-20). The output (encrypted) words are stored in $\mathtt{m}[\nu]$ by the interpreter.

Proceeding this way from MI_{L+1} to MI_{L+s}, the interpreter evaluates the program with encrypted words in memory, except for the program output words which are returned in clear by the SE. Once the last multi-instruction MI_{L+s} has been processed, the interpreter looks up the plain output in memory and returns it.

Example 2. Figure 5 depicts the evaluation of the program given in Example 1. As we can see, the intermediate values, the shared key K_{S} and the bytecodes of multi-instructions are all encrypted outside the SE (Figure 5a). They are decrypted for the executions of multi-instructions only inside the SE (Figure 5b). We note that we return the output of
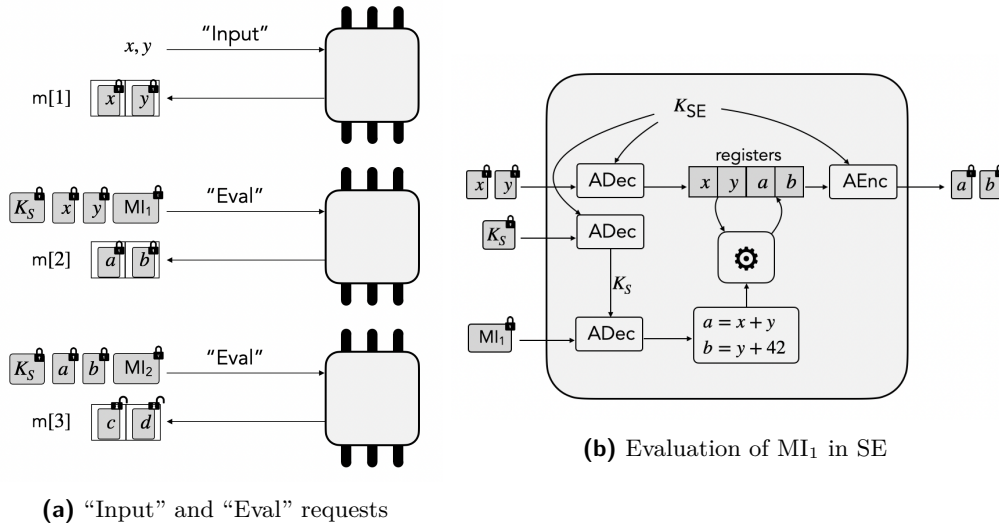
(a) "Input" and "Eval" requests

(b) Evaluation of MI_1 in SE

Figure 5: Example of obfuscated program evaluation.

MI_2 (c, d) without encryptions because this is the output of the program. In this case, the revelation flag b_R^2 in MI_2 is set (see Equation 2).

2.3 Obfuscation Security

OBSCURE comes with two different security modes:

White-box mode.   In this mode, the program's instructions and internal data are obfuscated while the program's control flow is (partly) revealed. Although the control flow internal to the multi-instructions is obfuscated (by encryption of their bytecode), the data dependency between the different multi-instructions is leaked through the input identities $\text{ID}_{\text{in}}^{\nu}$. The latter must be known to the interpreter to select the encrypted words in input of the "Eval" requests and hence they cannot be encrypted along with the bytecode. A typical use case for the white-box mode is the white-box protection of a cryptographic algorithm embedding hardcoded secrets (a.k.a. white-box cryptography).

Full obfuscation mode.   In this mode, one aims at additionally obfuscating the (full) program control flow. To achieve this, a preliminary step is added to the obfuscation process which is called *universalization*. This step refactors the input program P as a program P' which for given *width* and *depth* parameters has constant input identities $\text{ID}_{\text{in}}^{\nu}$ independent of the original program P (the width and depth parameters must be chosen in such a way that the program fits in). Our universalization technique is described in Section 3. For the purpose of the current section, we simply assume that in full obfuscation mode, the program P' in input of the obfuscation process indeed fulfills the requirement of constant input identities.

The following theorem states the security of our scheme:

Theorem 1. *Let OBSCURE be the SE-based obfuscator depicted in Subsection 2.2. Assume that*

- *the used AEAD scheme satisfies (t, ε_{pr})-privacy,*
- *the used AEAD scheme satisfies (t, ε_{au})-authenticity,*

Algorithm 3 $\mathsf{SE}(\text{``Eval''}, E_{\text{ID}}, E_{\text{K}}, \text{MI}_\nu, C_1^*, \ldots, C_\ell^*)$

1: $(K_{\text{S}}, R_{\text{S}}) \leftarrow \mathsf{ADec}_{K_{\text{SE}}}(\mathsf{Hash}_1(E_{\text{ID}}), \emptyset, E_{\text{K}})$
2: **if** $R_{\text{S}} = \bot$ **then return** $\bot$
3: $(\nu, A_\nu, C_\nu) \leftarrow \text{MI}_\nu$
4: $(b_R^\nu \parallel \text{ID}_{\text{in}}^\nu) \leftarrow A_\nu$
5: $(f_\nu, R_\nu) \leftarrow \mathsf{ADec}_{K_{\text{S}}}(\nu, A_\nu, C_\nu)$
6: **if** $R_\nu = \bot$ **then return** $\bot$
7: **for** k from 1 to ℓ **do**
8:     $(i', j') \leftarrow \text{ID}_{\text{in}}^\nu[k]$
9:     $N' \leftarrow \mathsf{Hash}_3(E_{\text{ID}} \parallel i' \parallel j')$
10:    $A' \leftarrow (i' \parallel j' \parallel E_{\text{ID}})$
11:    $(u_k, R') \leftarrow \mathsf{ADec}_{K_{\text{SE}}}(N', A', C_k^*)$
12:    **if** $R' = \bot$ **then return** $\bot$
13: $(v_1, \ldots, v_\ell) \leftarrow f_\nu(u_1, \ldots, u_\ell)$
14: **if** b_R^ν is true **then**
15:    **return** $(v_1, \ldots, v_\ell)$
16: **else**
17:    **for** j from 1 to ℓ **do**
18:        $N_{\nu,j} \leftarrow \mathsf{Hash}_3(E_{\text{ID}} \parallel \nu \parallel j)$
19:        $A_{\nu,j} \leftarrow (\nu \parallel j \parallel E_{\text{ID}})$
20:        $C_{\nu,j} \leftarrow \mathsf{AEnc}_{K_{\text{SE}}}(N_{\nu,j}, A_{\nu,j}, v_j)$
21:    **return** $(C_{\nu,1}, \ldots, C_{\nu,\ell})$

- *the used PKE scheme satisfies (t, ε_{ind})-IND-CPA,*
- *the used hash function satisfies (t, ε_{cr})-collision resistance.*

OBSCURE achieves

- *VBB (t, ε)-white-box security in white-box mode,*
- *VBB (t, ε)-obfuscation security in full obfuscation mode,*

with $\varepsilon \leq 2\varepsilon_{pr} + \varepsilon_{au} + \varepsilon_{ind} + \varepsilon_{cr}$.

The proof is provided in Appendix A where we further recall the formal definitions of the necessary security notions.

Remark 3. In full obfuscation mode, the width and depth parameters are revealed by the obfuscated program, so that formally, the achieved VBB obfuscation notion is for a simulator which is given the width and depth of P as the "size" $|P|$.

3   Universalization

A *universal circuit* is a commonly used tool for obfuscation. It takes as input a description of a circuit and an input to it, and evaluates the circuit on that input, yielding the circuit's output. Essentially, it allows to hide the *structure* of a program being obfuscated and to reduce the obfuscation goal to hiding only the inputs and the intermediate *values* of the universal circuit.

Valiant [Val76] proved a fundamental asymptotic lower bound of $\Omega(N \log N)$ gates for a universal circuit computing arbitrary Boolean circuits with N gates, and provided a construction with $19N \log N$ gates (with $4.75N \log N$ AND gates). This construction was recently improved in a sequence of works [KS16, ZYZL19, AGKS20, LYZ+21] to finally achieve $12N \log N$ gates (with $3N \log N$ AND gates) in [LYZ+21].

3.1   Rectangular Universal Circuits

In this work, with the goal of lightweight obfuscation in mind, we consider a less generic but lighter technique of universalization. The idea is to fix (or upper-bound) the maximum *width* of a circuit. Then, each computational circuit, up to addition of dummy gates, can be represented in a *rectangular* shape: a sequence of layers of computational nodes (in our framework, multi-instructions), connected by layers of wirings (see Figure 6). This shall typically be the case for cryptographic primitives (*e.g.*, block ciphers such as the AES [AES01]), which are typical targets for obfuscation methods. We believe that rectangular universal circuits may find use beyond our obfuscation framework, *i.e.*, be a lighter practice-oriented replacement for general universal circuits.
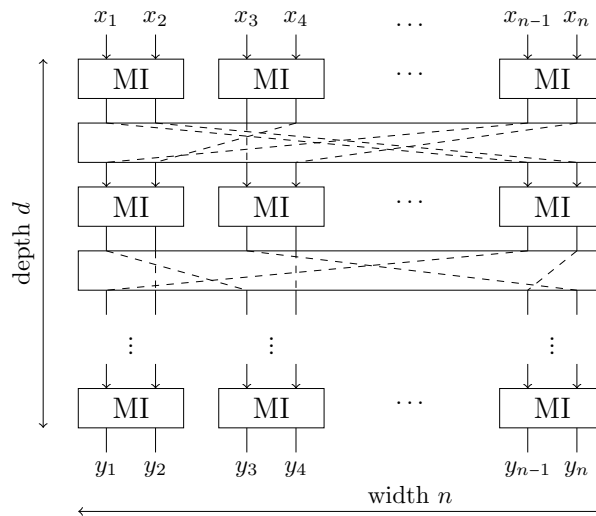

Figure 6: Rectangular-shaped circuit (simplified). "MI" stands for multi-instruction, or equivalently, an arbitrary programmable gate.

Universalization of rectangular circuits requires to universalize operation(s) performed in the computational nodes and the wirings, *i.e.*, serialize operations to bitstrings and describe a universal evaluator circuit taking these descriptions as an extra input. In OBSCURE, the computational nodes are protected by construction, through the encryption of the instructions and secure evaluation in the SE. We are thus left with the problem of protecting the wirings, which we solve by using *permutation networks*. Rectangular universalization is slightly weaker than general universalization, because it leaks (upper bounds on) the width and the depth of supported circuits (the two dimensions of the rectangle). Note that general universal circuits also leak (an upper bound on) the size of supported circuits.

Our construction of universal rectangular circuits requires $\sim dn \log n$ gates, where d is the depth and n is the width of the circuit. For a *dense rectangular circuit*, *i.e.*, a circuit of $N = \Omega(dn)$ gates, we thus obtain universal circuits of $\mathcal{O}(N \log n)$ gates, *i.e.*, with an extra logarithmic factor only in the circuit width. This is asymptotically optimal for circuits with constant width (which are dense rectangular circuits by definition) for which our construction achieves a linear scaling. For circuits with sublinear width, this is still asymptotically better than the size $\mathcal{O}(N \log N)$ of standard universal circuits.

Furthermore, we show how to practically improve our constructions by using gates with 2^l inputs and outputs. Such a *clusterized* version allows to reduce the cost by a factor of $l2^l/2$, yielding the final gate complexity $\sim \frac{2dn \log n}{l2^l}$ as l is constant and n tends to ∞.

3.2   Permutation Networks

In the following, S_n denotes the set of all permutations of the n-element tuple $(1, 2, \ldots, n)$.

Definition 2. A *permutation network* is a computational circuit P that takes as input n arbitrary elements and a description σ_π of a permutation $\pi \in S_n$, and outputs the n elements in the order given by π. Symbolically,

$$P : ((x_1, \ldots, x_n), \sigma_\pi) \mapsto (x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}).$$

Common permutation networks consist of *controlled swap* gates and static wires interconnecting these gates. The description σ_π of a permutation π consists of control bits for the swap gates. Each such bit determines whether the associated controlled swap gate should perform the swap or not.

Beneš network [Ben64] is a well-known permutation network based on controlled swaps. It permutes $n = 2^m$ elements using $2m - 1$ layers of $n/2$ controlled swap gates. In the i-th layer, $1 \leq i \leq m$, the 2^m entries in the list are grouped by their 0-based indexes in pairs $(j, j \oplus 2^i)$, *i.e.*, pairs of indexes differing only in the i-th least significant bit. A controlled swap gate is applied "in-place" to each such pair. For example, the first layer applies the gates to index pairs $(0, 1), (2, 3), \ldots, (2^m - 2, 2^m - 1)$; the second layer applies the gates to index pairs $(0, 2), (1, 3), (4, 6), (5, 7), \ldots, (2^m - 3, 2^m - 1)$, etc. The other $m - 1$ layers are the same as the first $m - 1$ ones applied in the reverse order. More precisely, the i-th layer with $i > m$ has the same structure as the $(2m - i)$-th layer (but the control bits computed for a concrete permutation may differ). The Beneš network's data flow is illustrated in Figure 7.
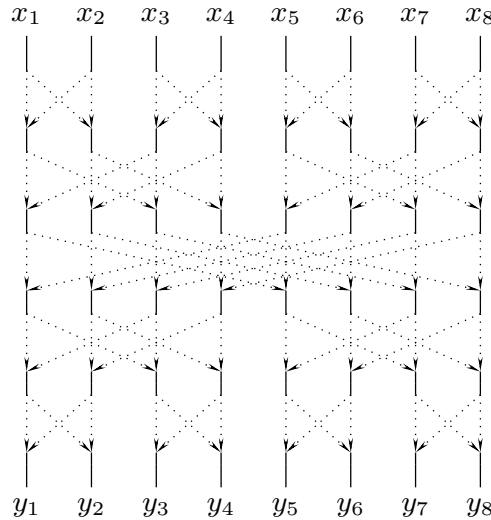

Figure 7: Beneš network data flow for $n = 8$.

Given a permutation $\pi \in S_n$, the respective control bits for the n-sized Beneš network can be computed efficiently. The variety of algorithms for solving this problem is out of scope of this paper. In our implementation, we rely on the recent verified code by Bernstein [Ber20], who also provides an extensive literature review on the topic.

3.3   Copy-Permutation Networks

Pure permutation networks are not natural for typical programs in which a single variable may be input of many subsequent operations, which means gate fan-out greater than 1

or 2 for the underlying circuit. Moreover, a limited program memory size translates to a limited circuit width provided that a gate (or multi-instruction) in a layer can connect to any number of gates in the next layer. Therefore, we also consider the problem of construction copy-permutation networks, *i.e.*, permutation networks with copying. The difference from simple permutation networks is that the ordering π is allowed to be taken from the set $\{1, \ldots, n\}^n$ rather than only from the set S_n of all n-element permutations. Constructions of such networks require extending conditional swap gates by allowing them to copy one of their inputs into both outputs (in the case of 2×2 swap/copy gates). Such a gate requires 2 control bits to encode all 4 possible actions: passing through, swapping the values, copying the first value, copying the second value. In our obfuscation framework, this is naturally achieved by instructions moving values in a required order.

Unfortunately, it can be shown that Beneš networks with extended 2×2 swap/copy gates can not compute arbitrary orderings $\pi \in \{1, \ldots, n\}^n$ (see Appendix B). Instead, we propose a new simple construction based on two permutation networks and a *sequential copy network*, which may copy an element at index i only to a contiguous segment $i, i+1, \ldots, i+t$ of output indexes, for some integer $t \geq -1$ ($t = -1$ meaning an empty segment, occurring when the position i is overwritten with a copy of an element from a smaller index). Our construction is close to an optimal one in the number of gates (larger by at most a factor of $2 + \mathcal{O}(1/\log n)$), since a copy-permutation network can not be smaller than a pure permutation network, which in turn also has the information-theoretic lower bound $\Omega(n \log n)$.

High-level construction. Let $\pi \in \{1, \ldots, n\}^n$ be the target ordering. Define $s \in S_n$ be any permutation such that $s \circ \pi$ is sorted in a non-decreasing order. We will first compute $s \circ \pi$ and then apply s^{-1} in a standard permutation network. Let $I \subseteq \{1, \ldots, n\}$ denote the set of positions i such that $i = 1$ or $(s \circ \pi)_i > (s \circ \pi)_{i-1}$. Define $\pi' \in S_n$ to be any permutation agreeing with $s \circ \pi$ on all positions I (this is possible since the values of $s \circ \pi$ on this restriction are strictly increasing). Observe that for the remaining positions $i \in \{1, \ldots, n\} \setminus I$ it is $(s \circ \pi)_i = (s \circ \pi)_{i-1}$. Therefore, the ordering $s \circ \pi$ can be obtained from π' by *sequential copying*. Let c denote the respective mapping, equal to $(s \circ \pi) \circ \pi'^{-1}$. Then, the target ordering π can be computed as $s^{-1} \circ c \circ \pi'$, which can be implemented by programming a composition of a permutation network, a sequential copy network and another permutation network (see Figure 8).

Sequential copy network. A sequential copy network can be implemented by a chain of $n-1$ *multiplexer* gates, which can be viewed as restricted copy gates. A multiplexer 2×1 gate mux simply selects one of its inputs depending on the value of the control bit. Formally, let $x = (x_1, \ldots, x_n)$ be the input of the network and let $y = (y_1, \ldots, y_n)$ be the output of the network. Observe that $y_1 = x_1$. Let the i-th mux gate, $1 \leq i \leq n-1$, take as input y_i and x_{i+1}, and return y_{i+1}. This is possible since either $y_{i+1} = y_i$ (copy) or $y_{i+1} = x_{i+1}$ (passthrough). This solution has depth $n-1$ since each gate in the sequence takes the output of the previous gate as one of the inputs. In OBSCURE, the depth of the circuit has no effect on the final performance, since full program is evaluated on a single SE, one multi-instruction at a time.

3.4 Native Clusterization of our CP Network

OBSCURE uses *multi-instructions*, which can be viewed as gates with a larger number of inputs and outputs. For simplicity, we assume $2^l \times 2^l$ gates, for an integer $l \geq 2$. We now show how to group the 2×2 gates from our copy-permutation network into a smaller amount of larger $2^l \times 2^l$ gates. We refer to this process as *native clusterization* (see also Section 4 for the case of arbitrary circuits). In the following, we assume that the width n
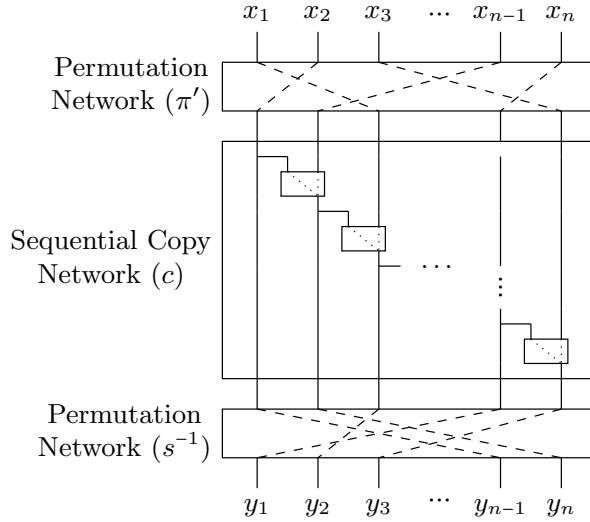
**Figure 8:** Copy-permutation network, based on a sequential copy network.

is a power of two and denote $m = \log n$. Example illustrations of native clustering in the Beneš networks and in the sequential copy networks are provided in Figure 9.

Beneš network. Beneš network has high clusterization potential. All the gates in the middle $2l - 1$ layers can be perfectly grouped into parallel $n/2^l = 2^{m-l}$ clusters (assuming $l \leq m$). Indeed, the values with 0-based indexes having $m - l$ least significant bits in common are permuted independently of other values in these layers, by the construction of the Beneš network. More precisely, for each integer i, $0 \leq i \leq 2^{m-l}$, the 2^l values with 0-based indexes $\{i + 2^{m-l}j\}_{0 \leq j < 2^l}$ are permuted by the middle $2l - 1$ layers independently of other values. Each such i-based group can be permuted using a single $2^l \times 2^l$ gate, leading to 2^{m-l} such gates in total for the middle $2l - 1$ layers.

The remaining $2m - 1 - (2l - 1) = 2(m - l)$ layers split into two symmetric cases of $m - l$ layers (before and after the middle layers). Similarly to the middle case, each l consequent layers can be merged into one layer of parallel full-sized clusters, with the only difference that the fixed bits in 0-based indexes are not the least significant bits. If $m - l$ is not divisible by l, then l' consequent layers will remain (on each of the two sides), with $0 < l' < l$. These layers can also be grouped into one layer of parallel full-sized clusters. Indeed, while the layers split into $2^{m-l'} > 2^{m-l}$ independent groups, these smaller independent groups may still be combined into full-sized clusters. Formally, the indexes are grouped into 2^{m-l} clusters by $m - l$ most significant bits. It is easy to verify that each such cluster permutes values "in-place", *i.e.*, independently from other clusters. This leads to an implementation of Beneš network of size $n = 2^m$ in $1 + 2 \left\lceil \frac{m-l}{l} \right\rceil = 2 \left\lceil \frac{m}{l} \right\rceil - 1$ layers of 2^{m-l} clusters each (i.e., $2^l \times 2^l$ gates).

Sequential copy network. In our construction based on a chain of $n - 1$ multiplexer gates, clusterization is trivial by grouping consequent multiplexer gates into sub-chains. Each sub-chain of $2^l - 1$ gates takes 2^l inputs and returns $2^l - 1$ outputs. Therefore, the $n - 1$ gates can be split into chained groups of size at most $2^l - 1$, leading to $\left\lceil \frac{n-1}{2^l-1} \right\rceil$ clusters. Formally, the i-th cluster takes as inputs values with indexes

$$\left(1 + (i-1)(2^l - 1), \ \ldots, \ \min(2^l + (i-1)(2^l - 1), n)\right)$$
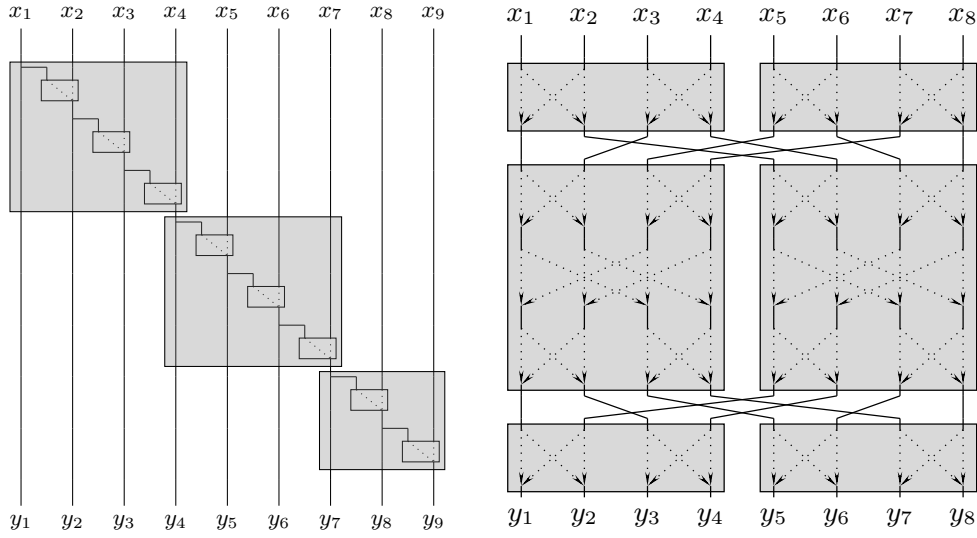
(a) Native clustering of the sequential copy network on $n = 9$ elements with 3×3 clusters/gates (each cluster shaded in gray).

(b) Native clustering of the Beneš network with $n = 8$ elements using 3 layers of 4×4 clusters/-gates (each cluster shaded in gray).

Figure 9: Examples of native clustering.

and outputs values with output indexes

$$\left(2 + (i-1)(2^l - 1),\ \ldots,\ \min(2^l + (i-1)(2^l - 1), n)\right).$$

Since the first input of each cluster is equal to the respective output of the network, it may be passed through the cluster (unchanged) to make the cluster have precisely 2^l outputs (instead of actually computed $2^l - 1$ outputs).

Full complexity. The total complexity of our implementation of a copy-permutation network of $n = 2^m$ elements is equal to

$$2^{m-l}\left(2\left\lceil\frac{m}{l}\right\rceil - 1\right) + \left\lceil\frac{n-1}{2^l - 1}\right\rceil$$

gates of type $2^l \times 2^l$, if $l < m$ (the case of $l \geq m$ can be trivially computed using one gate). Asymptotically, this is equal to $\sim 2^{m-l+1}\frac{m}{l} = 2\frac{n \log n}{l2^l}$ when l is constant and $n \to \infty$.

Full rectangular universalization requires $(d+1)$ permutation layers and $dn/2^l$ main computational gates, totaling to

$$d\left(\frac{2n \log n}{l2^l} + \frac{n-1}{2^l - 1}\right),$$

ignoring rounding errors. If l is constant and n is increasing, only the first term of the sum contributes asymptotically. However, in the scenario where the width n is constant too, the second term contributes to the constant in the $\mathcal{O}(d)$ complexity of our scheme.

4 Compiler Software Architecture

Figure 10 presents the overall architecture of our compiler which consists of 6 passes. Initially, the input C program is transformed into an *abstract syntax tree* (AST) by a

parser. We next remove high-level constructions such as loops and functions from this AST, and convert them into a *straight-line program* (SLP, *i.e.*, a sequence of basic operations without any branches, loops, conditional statements, or comparisons) in our *high-level intermediate representation* (HLIR). We then merge neighboring instructions together and produce a *data-flow graph* (DFG) whose nodes are in our *mid-level intermediate representation* (MLIR). Then, universalization produces a universal DFG for this program. The final lowering pass then generates from this DFG an SLP in our *low-level intermediate representation* (LLIR), and performs register allocation for each node of the DFG. Finally, this SLP is serialized into a bytecode, which can then be executed by the interpreter and the SE.
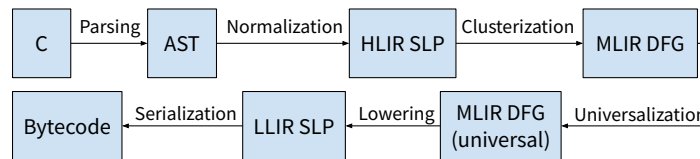

Figure 10: Software architecture of our compiler

Input C program and parsing. Our compiler accepts as input a program written in a subset of C, which we call C⁻. The main restrictions of C⁻ programs include: (1) the control flow must be statically computable, (2) pointers must alias to the same memory region during their whole lifetime, (3) all variables must be of type either `unsigned int` or `unsigned int*` and (4) we support ternary expressions (`?:`), comparisons (`<`, `==`), `for` loops and do not support `if`, `switch` and `while` loops. A program in C⁻ is a list of functions. The last function is considered as the main, by convention. Functions can call each other, but we forbid recursion, in order to be able to inline all function calls. The body of a function is a list of statements, which can be variable declarations, assignments, function calls, or *for* loops. OBSCURE supports the following operations:

- *Unary*: $-$, $\sim$

- *Binary*: $+$, $-$, $*$, $/$, $\%$, $\&$, $|$, $\hat{\ }$, $\ll$, $\gg$, $<$, $==$

- *Ternary*: $?:$

Assignments can use either a simple equation ($=$) or compound assignment operators (*e.g.*, $+=$, $\gg=$). The stopping condition of a `for` loops must be a comparison ($==$, $>=$, $<=$, $>$, $<$) between a variable and an expression, and the increment must be an expression (*e.g.*, a `var++` or `var-`) or an assignment (*e.g.*, `var+=2`). Both the stopping condition and the increment must be computable statically (in order to enable us to unroll all loops).

We use `pycparser` [Ben22], a standard C parser, to convert the input program into an abstract syntax tree (AST). This data structure is simply a tree that represents the syntax of the program, and is used in most front-ends of C compilers, such as GCC and Clang.

Normalization. The goal of the *normalization* is to remove high-level constructions of C, in order to eventually convert the AST into an assembly-like straight-line program, which we call *high-level intermediate representation* (HLIR). A HLIR program under this representation is a list of inputs, a list of outputs, and a list of *high-level instructions* (HLI). A high-level instruction has the following format:

```
OP dst, src1, src2, src3
```

where `OP` can be one of {`MOV, XOR, OR, AND, LSL, LSR, ADD, SUB, MUL, DIV, MOD, CMOV`}, `dst` is an integer representing a memory address, `src1`, `src2` and `src3` can be either memory addresses, or immediates. In case of unary (or binary) operations, `src2` and `src3` (or only `src3`) are empty. For instance, the following IR program takes two inputs and returns their sum multiplied by 42 (`m[x]` is the memory location `x` and `#42` is the immediate 42):

```
inputs: m[0], m[1]
outputs: m[3]
instructions:
  add m[2], m[0], m[1]
  mul m[3], m[2], #42
```

To transform the AST into a HLIR program, the normalization starts by inlining function calls and unrolling loops. Then, it removes arrays and pointers by replacing them with scalar variables: an array of size n is replaced by n variables. Additionally, we also remove nested expressions during this pass. For instance, `a = b + c + d` would be simplified into `tmp = b + c; a = tmp + d`. Thus, after the normalization, all instructions are assignments of unary or binary operations between variables/constants.

Clusterization. Recall that a multi-instruction is composed of several instructions. This pass merges high-level instructions (HLIs) together to produce multi-instructions. The motivation of performing clusterization is twofold: (1) Some intermediate variables will be alive only inside a multi-instructions, and will not need to be returned by the SE. Since inputs and outputs of the SE are encrypted, this directly translates into a reduced execution time. (2) The width and depth of the program will be reduced, thus reducing the size of the permutation networks and the number of permutation networks required (thanks to the smaller width and depth, respectively).

To perform clusterization, we use yet another intermediate representation: the *mid-level intermediate representation* (MLIR), which introduces multi-instructions. A *mid-level multi-instruction* (MLMI) is represented by a list of inputs, a list of outputs and a list of HLIs. In turn, an MLIR program consists of a list of inputs, a list of outputs, and a list of MLMIs. We start clusterization by trivially transforming the HLIR program into an MLIR program where each MLMI contains a single HLI. This simplifies the implementation of the clusterizer, which simply merges MLMIs together, without having to perform a change of representation.

In order to actually perform the clusterization (and the universalization), it is useful to represent the program as a directed (acyclic) graph, called *data-flow graph* (DFG), whose nodes are MLMIs, and whose edges represent assign-use relations: there is a edge from a node `a` to a node `b` if and only if `b` uses one of the variables that was assigned in `a`. Our clusterizer proceeds in three passes: (1) nodes with a single exit edge are merged with the destination of this edge, (2) nodes with a common parent are merged together and (3) nodes linked by an edge are merged together. These passes also help to reduce the DFG size, its width and its depth. Consequently, the number of decryptions/encryptions for inputs/outputs is reduced. Note that we do clusterization with respect to the maximal number of HLIs s, the maximal number of inputs/outputs ℓ and the number of registers r required to execute for each MLMI (parameters s, ℓ and r formally define an SE in Subsection 5.3).

The first and third passes have time a linear complexity in the size of the graph. The second one has a worst-case quadratic complexity. However, this complexity is only reached for a flat graph (in case the program simply outputs its inputs). On most (if not all) non-trivial programs, this second pass will actually have a sub-quadratic time complexity.

Universalization. As explained in Section 3, in order to protect the data-flow whenever full obfuscation mode is activated, we apply a universalization pass. The main idea is

to represent the program as a list of layers (multi-instructions that can be executed in parallel), and insert a permutation network between each layer, in order to mask data dependencies. We call *layerization* the pass that extracts layers from the DFG. All of the inputs of a given layer must come from the previous layer, and all of its outputs must flow into the very next layer. In case some edges go through a given layer, we insert copies of the source node to that layer to ensure this property. The outputs of the program are ensured to present at the last layer by the same way. Then, we add a pair of external layers to hide where inputs are used and where outputs come from. Let us take the following code as an example:

```
x = 42;
y = x + a;
z = x + y;
```

The DFG representing this program would have three nodes (of one instruction each, assuming that $s = 1$), all of them in a different layer, but the first one would have an edge from both the second and the third one (since x is an input of both these nodes):
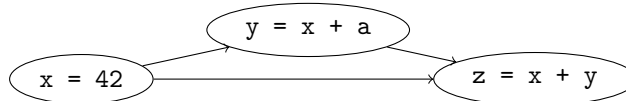
When such a situation arise, we break the edges that go through some layers by inserting copies. The above snippet would thus be converted into:

```
x = 42;
y = x + a;
x2 = x;
z = x2 + y;
```

This new snippet corresponds to the following DFG, which does not have any edges that go through a layer:
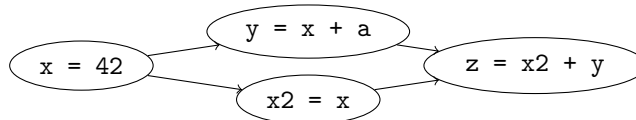
As in Section 3, the program should be represented in a rectangular shape with its width and its depth. We insert dummy instructions so that all layers have the same width. We can specify the width and the depth of a program (through the flags `-width` and `-depth` of the compiler) to strengthen the obfuscation. If not specified, they are the smallest possible width and depth of the program. We ensure that every MLMI has ℓ inputs/outputs and s HLIs by adding dummy inputs/outputs and dummy instructions (with respect to the number of registers r). Finally, we insert a permutation layer in between each layer following the process presented in Section 3.

Lowering.   Our SE does not have access to the global memory of the interpreter, hence cannot execute the HLIs contained in the MLMIs, since their operands refer to this global memory. Instead, the SE has its own internal memory (called *registers*), to which it copies the inputs of the MLMIs before executing the HLIs. The HLIs must the be "lowered" into low-level instructions (LLIs), which manipulate the SE's registers.

Our final intermediate representation is the *low-level intermediate representation* (LLIR). Similarly to MLIR, it contains inputs, outputs and low-level multi-instructions (LLMI). LLMIs are similar to MLMIs, except that they contain low-level instructions (LLI) instead of HLIs. An LLI is, in turn, similar to a HLI, except that the operands of LLIs are either immediates or registers (and not memory addresses as HLIs). The lowering pass converts the

MLMIs to LLMIs by performing register allocation. In the general case, register allocation is NP-complete, if there are more variables alive than registers available [CAC+81]: in that case, some variables must be stored in memory ("spilled"), and reloaded into the registers right before being used. However, in our case, the SE only has registers, but no memory, which is why clusterization and universalization make sure that no spilling will be necessary. This, in turn, makes register allocation fairly trivial using a simplified version of the Linear Scan algorithm [PS99] (simplified, because the original linear scan algorithm takes spilling into account). This algorithm computes the live ranges of each variables, and then iterates all instructions, allocating registers to variables that are born, and freeing the registers of the variables that die.

Serialization. The serialization converts the LLIR program into bytecode. It iterates linearly on all LLMIs, encodes their inputs and outputs (represented by memory cells), and encrypts (with authentication) their LLIs. In other word, the ν-th LLMI corresponds to the ν-th multi-instruction MI_ν, which is formally defined in **??**. Its input identities are represented by $\mathrm{ID}_{\mathrm{in}}^\nu$ and the set of LLIs is represented by f_ν in the multi-instruction. We refer to the formal definitions in **??** for the details of how we encode and encrypt in this pass. While any AEAD cipher and hash function can be used, we selected SPARKLE suite [BBdS+21] for its lightweightness. Still, changing the AEAD cipher and hash function in our implementation would come at very little effort.

Remark 4. As described above, OBSCURE only supports a subset of C defining straight-line programs. In particular, OBSCURE does not support programs containing conditional branches or loops with dynamic (data-dependent) number of iterations. Adding those features to the current obfuscation framework of OBSCURE can be done in a "simple way" which consists in unrolling and executing all the branches, using the condition evaluation as a selector for the branch to keep. While such an approach is conceptually simple, it significantly complicates the compilation and might also lead to inefficient unrolled obfuscated programs. For those reasons, we chose to restrict OBSCURE to straight-line programs. Although less flexible, this choice further increases the required developer awareness of the obfuscation process, which is better for the sake of obfuscated programs' efficiency.

5 Applications & Benchmarks

This section presents applications and benchmarks of OBSCURE for different programs.

To demonstrate the practicability of applying our obfuscator to cryptographic primitives and thus achieve secure white-box cryptography (with the help of an SE), we first report the application of OBSCURE to different ciphers. Specifically, we apply OBSCURE to Ascon [DEMS21], Photon [BCD+19], TinyJAMBU [WH21] - three ciphers submitted to the NIST lightweight cryptography competition,[2] as well as to the AES [AES01]. We use bitsliced implementations of Photon and AES in order to be able to compute the S-boxes without lookup tables. We further evaluate OBSCURE on simple array-processing functions, namely `findmax` and `sum`. We explain how the implementation of these functions impacts our universalization technique and hence the performance of their obfuscation (in full obfuscation mode).

We consider two further applications: (1) a traceable AES for the digital rights management use case, (2) the protection of sensitive intellectual property in neural networks. These two applications are described hereafter (Subsection 5.1 and Subsection 5.2), before providing benchmarks and discussions in Subsection 5.3.

---

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

5.1 Traceable AES

To highlight the potential cryptographic applications of OBSCURE beyond protecting white-boxed ciphers against key extraction, we evaluate an implementation of traceable AES inspired from [DLPR14] for a digital rights management (DRM) use case. The goal is to enforce a traceability feature to an AES decryption program by the use of obfuscation: one want to be able to trace malicious user(s) that would *e.g.* monetize access to their decryption programs. As proposed in [DLPR14], a simple transformation can make a decryption program traceable in the former sense by introducing some hidden perturbations. By applying a secure VBB obfuscator to such a program, one ensures that these hidden permutations cannot be recovered and removed from the program.

We defined the traceable AES decryption as depicted in Algorithm 4. In a nutshell, this traceable AES first decrypts the input ciphertext with the tracing key t. If the obtained ciphertext, interpreted as a 128-bit integer, is smaller that the index of the user $\mathtt{ind} \in \{1, \ldots, u\}$ (where u denotes the total number of users), then a perturbation $\mathtt{test} = 1$ will be XOR-ed to the right decryption of c under the key k. This algorithm hence implements an AES decryption under k, with a set of *dysfunctional ciphertexts* under the definition of [DLPR14] which is defined as

$$\langle c \rangle = \{c \in \{0,1\}^{128} \mid \mathsf{AES\text{-}Dec}_t(c) \leq \mathtt{ind}\}$$

for the user of index $\mathtt{ind} \in \{1, \ldots, u\}$. Defined this way, and applying our (SE-based) VBB obfuscation on top of it, this algorithm verifies the *Perturbation-Value Hiding* (PVH) and *Perturbation-Index Hiding* (PIH) from [DLPR14] which makes it traceable in a collusion-resistant way. The interested reader is referred to [DLPR14] for further details and the proof of traceability.

Algorithm 4 Traceable AES decryption. $k \in \{0,1\}^{128}$ is the hardcoded decryption key. $t \in \{0,1\}^{128}$ is the hardcoded tracing key. $\mathtt{ind} \in \{1, \ldots, u\}$ is the hardcoded user index.

Require: Input ciphertext $c \in \{0,1\}^{128}$
Ensure: Output plaintext $p \in \{0,1\}^{128}$
  1: $z \leftarrow \mathsf{AES\text{-}Dec}_t(c)$
  2: $\mathtt{test} \leftarrow (z < ind)$ ▷ 1 if ($z < \mathtt{ind}$), 0 otherwise
  3: $p \leftarrow \mathsf{AES\text{-}Dec}_k(c)$
  4: **Return** $p \oplus \mathtt{test}$

5.2 Neural Network

To further illustrate the potential applications of OBSCURE in non-cryptographic contexts, we consider the obfuscation of sensitive AI models, and specifically neural networks. In such a context, one wants to protect the sensitive intellectual property residing in the weights and/or the architecture of a pre-trained neural network. To demonstrate the potential of OBSCURE, we apply it to a neural network trained on the MNIST dataset [Den12]. This network takes as input a 28×28 image representing a handwritten digit and produces the recognized digit in $\{0, 1, \ldots, 9\}$ as the output. The network is a multilayer perceptron (MLP) composed of r layers including an input layer, $r - 2$ hidden layers, and an output layer. Denote N_i the number of neurons in the i-th layer ($1 \leq i \leq r$). For MNIST, in particular, $N_1 = 784$ and $N_r = 10$ since the network takes 28×28 images as the input and produces a predicted digit in $\{0, 1, \ldots, 9\}$ as the output. In our application, we choose $r = 10$ and $N_i = 100$ for $2 \leq i \leq 9$.

The computation in a neuron of a hidden layer includes a weighted sum and an activation function. Let $x_{i,k}$ be the output value of the k-th neuron in the i-th layer, $w_{j,k}^{(i)}$

be the weight associated to the connection from the k-th neuron in layer i to the the j-th neuron in layer $i+1$, and let $b_{i,j}$ be the bias of the j-th neuron in the layer i. The j-th neuron of layer i first computes the following weighted sum:

$$y_{i,j} = \sum_{k=1}^{N_{i-1}} w_{j,k}^{(i-1)} x_{i-1,k} + b_{i,j}. \tag{4}$$

Then it applies the activation function to $y_{i,j}$ to get the neuron output $x_{i,j}$. For our network, we use the ReLU function [Aga18], denoted f hereafter. We have:

$$x_{i,j} = f(y_{i,j}) = \begin{cases} 0 & \text{if } y_{i,j} \le 0, \\ y_{i,j} & \text{otherwise.} \end{cases} \tag{5}$$

Unlike in the hidden layers, a softmax function is used in the neurons of the last layer instead of the activation function. This softmax function returns an array of probability scores corresponding to the array of possible predicted digits. For the sake of simplicity, we replace the softmax function with argmax function which does not change the outcome of the predictions. Denote z the digit predicted by the neural network, we have $z = \text{argmax}_j(y_{r,j})$.

After training on the MNIST dataset, we obtain a neural network which predicts the right digit with a 97.22% accuracy. However, in this trained neural network, the computation of the weighted sum (Equation 4) and the activation function (Equation 5) takes place on real floating-point numbers while OBSCURE only supports operations on unsigned 32-bit integers. Therefore, as a second step, we *discretize* the obtained network. Namely, we transform the two above functions into new functions which work on unsigned 32-bit integers while still maintain the functionality of the neural network. This transformation is composed of 3 passes which are described in Appendix C.

5.3 Benchmarks

Parameter setting. An SE is defined by three parameters: the number of instructions s of a multi-instruction, the number of inputs/outputs ℓ and the number of *registers* r (size of internal memory) in the SE. Table 1 contains the parameters of four different instances used for our benchmarks. The number of registers r was fixed to 5ℓ: ℓ registers to store the inputs, ℓ registers to store the outputs, and 3ℓ registers for intermediate variables.

Table 1: SE parameters for our evaluation.

SE name	ℓ	s	Estimated performance on ARM Cortex-M3 (120 MHz)
small	8	32	600 LLMIs/sec
medium	16	64	300 LLMIs/sec
large	32	128	150 LLMIs/sec
extra-large	64	256	75 LLMIs/sec

We also include the estimated performances of the different SE instances on an ARM Cortex-M3 microcontroller (clocked at 120 MHz) in Table 1. The estimations are based on the cost of the input decryptions, the output encryptions and the hash computations in Algorithm 3, which dominate the overall complexity. The computation speed is taken from the reported benchmarks of the SPARKLE suite [BBdS+21].

White-box mode. We consider a scenario where the data-flow graph does not need to be protected, and we thus disable universalization. We report in Table 2 the number of

instructions in the initial program (column `#HLIs`), and the number of multi-instructions at the end of the compilation (column `#LLMIs (final)`).

Estimated execution times of the different ciphers range between 0.4 and 1.1 seconds, except for Photon (which processes 32 blocks at once). In comparison a fast implementation of AES takes about 13 microseconds on Cortex-M3 clocked at 120MHz [SS17], which is of course much smaller, but does not provide VBB obfuscation for *any* cipher or program of similar size (and advanced features such as traceability). We observe that the clusterizer is able to efficiently make use of the larger SEs. While doubling ℓ and s approximately reduces the number of LLMIs by a factor 2 for the "small" ciphers (which does not change the estimated execution time), it has a higher impact on larger programs as illustrated with the fully-bitsliced implementation of Photon for which execution time is halved when stepping the SE instance. We note that despite of requiring less hardware resources than AES as their primary design's goal, the lightweight ciphers (Ascon and TinyJAMBU) have more instructions and thus worse performance than AES in our evaluation since their internal round numbers are higher.

The obfuscated neural network results in larger compilation and execution times which comes from its high number of HLIs, 230k, between one and two order of magnitude higher than for the ciphers.[3] The estimated execution time, 36.7 sec, is pretty high. For practical application, this should be improved *e.g.* by using a faster SE and/or a lighter neural network architecture (*e.g.* avoiding too many dense layers). We observe that the execution time does not scale down while increasing the SE instance. This can be explained by the use of fully connected layers in the neural network which gives a hard time to the clusterizer. In other words, increasing the number of instructions processed by one request to the SE (number of instructions in one LLMI) does not enable the clusterization to save communication with the SE (input-output of LLMIs) because of the high connectivity in the data-flow graph of the network. This suggests that, for such neural networks with fully connected layers, a small instance of the SE performs just as well as a larger instance.

Let us stress that OBSCURE makes it possible to amortize the obfuscation time between several users. Indeed, a program can be obfuscated once with some shared key K_S. Then only the header (*i.e.*, the encryption of K_S under the SE public key) shall change from one user to another.

Full obfuscation mode. In this mode, the data-flow graph is protected by enabling universalization. We evaluate this mode on the considered ciphers and on two simple algorithms, namely `sum` and `findmax` (for an array of size 1000). We report the results in Table 3. The reported depth and width are the smallest possible for each program with each SE configuration. Rectangularization increases the number of MLMIs by a factor between 4 and 10 depending on the ciphers and the SEs. This is often due to a combination of two factors: the layers of the initial programs are of unequal sizes, and a lot of values are used in several layers after their computation, and thus need to be copied in the layers in between. Universalization increases the number of MLMIs even more significantly. This is expected, since, as shown in Section 3, universalization requires $2dn \log n$ instructions, where n is the maximum width of a layer and d is the number of layers.

In the full obfuscation mode, the way that one implements a program can have a significant impact to the size of its rectangular representation, and thus to the final number of LLMIs. We illustrate this impact by considering two approaches for the `sum` and `findmax` algorithms (the "naive approach" and the "tree approach"). In the naive approach, a variable accumulates the values of each array element which means that the instructions cannot be executed in parallel. This leads to a high program depth and thus a huge impact

---

[3]As detailed in Appendix C, the network is made of 8 dense hidden layers each featuring 100 neurons, making 100×100 connections per layer, as well as the initial layer featuring 28^2 neurons and hence $28^2 \times 100$ connections with the second layer.

Table 2: General evaluation on several ciphers in white-box mode (without universalization). Estimated execution times are for a Cortex-M3 microcontroller clocked at 120 MHz and using the SPARKLE suite [BBdS+21] for the authenticated encryption and hash primitives.

[*] The implementation of Photon is fully bitsliced, *i.e.* computing 32 blocks at once, and it thus has much more instructions than the others.

Cipher	Secure Element	#HLIs	#LLMIs (final)	Compil. time	Exec. time (estimated)
AES	small	5.3k	290	3.2 sec	0.5 sec
	medium		120	3.1 sec	0.4 sec
	large		59	3.1 sec	0.4 sec
	xlarge		29	3.2 sec	0.4 sec
Ascon	small	14k	680	6.2 sec	1.1 sec
	medium		300	6.4 sec	1.0 sec
	large		170	6.5 sec	1.1 sec
	xlarge		66	7.0 sec	0.9 sec
Tiny JAMBU	small	6.3k	350	1.7 sec	0.6 sec
	medium		170	1.9 sec	0.6 sec
	large		85	1.9 sec	0.6 sec
	xlarge		44	2.0 sec	0.6 sec
Photon*	small	47k	6.7k	180 sec	11.2 sec
	medium		3.3k	130 sec	11.0 sec
	large		900	130 sec	6.0 sec
	xlarge		270	140 sec	3.6 sec
Traceable AES	small	11k	580	4.8 sec	1.0 sec
	medium		240	4.4 sec	0.8 sec
	large		120	4.8 sec	0.8 sec
	xlarge		59	4.7 sec	0.8 sec
Neural Net	small	230k	22k	220 min	36.7 sec
	medium		11k	58 min	36.7 sec
	large		5.5k	21 min	36.7 sec
	xlarge		2.6k	520 sec	36.7 sec

of rectangularization. In contrary, implementing the algorithm with the tree approach yields a much lower program depth: 6 compared to 250 for the `sum` program on a `small` SE (Table 3) since the instructions can be parallelized and clusterized to multi-instructions in the same layer. As a result, the number of final LLMIs is significantly reduced (28k compared to 520k). We observe a similar behavior for the `findmax` algorithm.

Clusterizer evaluation. Table 2 allows to clearly see the clusterizer in action, and how it is able to reduce the number of instructions of the program (and therefore, of requests to the SE). To analyze the quality of our clusterizer, we can define the *ideal clusterization* as the number of LLMIs that would be generated if each LLMI was actually computing s instructions, and compute the ratio between the actual number of LLMIs generated and this ideal number. A ratio of 1 means that our clusterizer was optimal, whereas a ratio close to 0 means that we generated much more LLMIs than needed. We report this metric in Table 4 for the `xlarge` SE. The "Actual" column of this table is taken from the "#LLMIs" column of Table 2.

Overall, we observe actual-ideal ratios between 0.57 and 0.83. To understand those numbers, we recall that our algorithm locally aggregates MLMIs into larger MLMIs. In the worst case, all LLMIs would contain $s/2 + 1$ instructions after clusterization, which would

Table 3: General evaluation on several programs in full obfuscation mode (with universalization).

** The implementation of Photon is fully bitsliced, *i.e.* computing 32 blocks at once, and it thus has much more instructions than the others.*

Cipher	Secure Element	#HLIs	Depth	Width	#MLMIs (clusterized)	#MLMIs (rectangular)	#LLMIs (final)	Compilation time
AES	small		190	7	290	1.3k	12k	60 sec
	medium	5.3k	110	3	120	320	3.4k	4.7 sec
	large		58	2	59	120	1000	3.9 sec
	xlarge		29	1	29	31	120	3.6 sec
Ascon	small		330	21	680	6.9k	120k	22 sec
	medium	14k	180	14	300	2.5k	23k	17 sec
	large		170	2	170	330	2.8k	9.1 sec
	xlarge		64	2	66	130	1.1k	7.8 sec
TinyJAMBU	small		350	3	350	1.1k	11k	3.9 sec
	medium	6.3k	170	2	170	350	3k	3.1 sec
	large		84	2	85	170	1.4k	3.2 sec
	xlarge		43	2	44	88	750	3.5 sec
Photon*	small		540	60	6.7k	32k	410k	250 sec
	medium	47k	340	40	3.3k	13k	250k	230 sec
	large		110	16	900	1.8k	15k	150 sec
	xlarge		55	8	270	450	3.6k	150 sec
traceable AES	small		190	13	580	2.5k	37k	10 sec
	medium	11k	110	5	240	550	6.9k	7.4 sec
	large		62	3	120	190	2k	6.7 sec
	xlarge		32	2	59	66	560	5.8 sec
sum(naive)	small		250	130	250	31k	520k	83 sec
	medium	1000	120	63	120	7.9k	97k	41 sec
	large		61	32	61	2k	16k	28 sec
	xlarge		30	16	30	500	4k	15 sec
sum(tree)	small		6	190	270	1.2k	28k	6.6 sec
	medium	1000	3	63	67	250	3.1k	2.5 sec
	large		4	56	62	260	3.8k	6.2 sec
	xlarge		3	46	51	160	2k	18 sec
findmax(naive)	small		250	130	250	31k	520k	84 sec
	medium	2k	120	63	120	7.9k	97k	42 sec
	large		62	32	62	2k	16k	29 sec
	xlarge		31	16	31	510	4.1k	16 sec
findmax(tree)	small		5	190	260	1.1k	24k	6.3 sec
	medium	2k	3	63	67	250	3.1k	30 sec
	large		3	57	62	200	3k	5.6 sec
	xlarge		3	47	52	160	2k	18 sec

prevent further merges, and would result in a ratio of 0.5 (meaning that we generated twice more LLMIs that the ideal number). On top of that, we must account for the constraint on the number of inputs and outputs of each MLMI ($\leq \ell$), which could also prevent some merges and thus lower the actual-ideal ratio than even an optimal clusterizer would reach (or, alternatively, lower the ratio of our clusterizer below 0.5). Therefore, our clusterizer performs significantly better than the worst-case scenario, keeping in mind that optimal clusterization is NP-hard, and that our clusterization algorithm is sub-quadratic.

Impact of SE's parameters.   To give more insights into the impact of the secure element's parameters, we benchmarked TinyJAMBU with various combinations of ℓ and s. We forbade s to be less than ℓ, in order to be able to generate nodes that simply copy their inputs to their outputs. We fix $r = 5\ell$ as before. Table 5 provides the final numbers of LLMIs generated. It shows that increasing either ℓ or s separately does not yield much improvement (the number of LLMIs is not reduced significantly). For instance, we consider $\ell = 4$, for $s \in \{16, 32, 64, 128, 256\}$ it generates 66-69k LLMIs. Similarly, we consider $s = 128$, for $\ell \in \{16, 32, 64, 128\}$ it generates the same number of LLMIs (1.4k). Therefore, to reduce the number of final LLMIs, we should increase both ℓ and s.

Table 4: Analysis of the quality of the clusterizer.

Cipher	HLIs	LLMIs		Ratio
		Ideal	**Actual**	
AES	5.3k	20	29	0.69
Ascon	14k	55	66	0.83
Photon	47k	184	270	0.68
TinyJAMBU	6.3k	25	44	0.57

Table 5: Impact of the SE parameters on the number of LLMIs produced by the compilation for TinyJAMBU.

s \ ℓ	**4**	**8**	**16**	**32**	**64**	**128**	**256**
4	200k	-	-	-	-	-	-
8	100k	35k	-	-	-	-	-
16	68k	22k	12k	-	-	-	-
32	69k	11k	6k	6k	-	-	-
64	68k	5.6k	3k	3k	3k	-	-
128	67k	2.9k	1.4k	1.4k	1.4k	1.4k	-
256	66k	1.6k	750	750	750	750	750

6   Further Discussion

This section provides further discussion on the security of OBSCURE compared against trusted execution environments (TEE) as well as its use in a cloud computing use case.

6.1   Security of OBSCURE versus TEE

A common way to protect sensitive computation in untrusted environments (such as multi-applicative smart devices) is to rely on trusted execution environments (TEE) such as Intel SGX, AMD Secure Technology or ARM TrustZone. While TEE certainly improve the security of the underlying computation, they still suffer numerous side-channel attacks arising from their high internal complexity [FYDX21]. In comparison, relying on a simple secure element (*e.g.* implementing the OBSCURE functionality) provides more assurance in practice thanks to its physical isolation and limited functionality.

Physical isolation. A secure element is a separate, dedicated microprocessor or a digital circuit, which provides physical isolation from the main processor, which is not the case of TEE. This physical isolation makes it more resistant to attacks that exploit vulnerabilities in the main processor or its software. In contrast, TEE, although isolated, still share resources with the main OS, which could potentially lead to vulnerabilities.

Limited functionality. Secure elements are designed for specific tasks like storing cryptographic keys or performing encryption, with a very limited and defined scope. This simplicity reduces the attack surface compared to a TEE, which is designed to handle a broader range of more complex tasks. In our context, the proposed SE functionality is much simpler than what is typically offered by a TEE.

In addition, OBSCURE only requires a stateless secure element, it is thus not vulnerable to attacks exploiting stateful functional units. An OBSCURE 's obfuscated program is deterministic (no branching), so the constant-time property is preserved in terms of software (hardware design is out of scope of this work).

6.2   OBSCURE for Confidential Could Computing

An potential application of OBSCURE is confidential computing in a public cloud. In this context, an untrusted cloud service is involved to securely run programs with the help of trusted secure elements implementing the OBSCURE functionality. Users can then compile their programs locally and have the cloud execute them without disclosing any information on the internal computation of the programs (nor on the program structure in full obfuscation mode). In such a scenario, the public cloud typically relies on many secure elements. Hence a possible issue is that the local compiler might not know ahead of time to which SE the obfuscated program will be distributed.

A possible setting could be that the same public-private key pair ($\mathsf{priv}_{\mathsf{SE}}, \mathsf{pub}_{\mathsf{SE}}$) is shared (securely transported) by different SEs of the cloud service. But this is not ideal in terms of security and administration. In a setting where all the SEs have their own key pairs (which are generated inside the SE and never go out), the trivial solution would be to make the obfuscated program compatible with all (or a significant part of) the SEs. This might be an acceptable solution as the extent of the encrypted material to a target SE is small (a single symmetric key K_{S}). But this is far from being ideal with respect to performances, communication and flexibility (*e.g.* the obfuscator is required to keep a list of SE public keys which might frequently evolve).

Another solution to avoid sharing the same key among different SEs would be to use the principle of proxy re-encryption (see, *e.g.*, [AFGH05]). In such a scenario, the shared key K_{S} would be encrypted with a common public key for which a central proxy has several re-encryption keys (one per SE). This solution involves an additional trusted party which owns the private key corresponding to the proxy public key and which has the ability to produce the re-encryption keys for the SEs.

7   Conclusion

In this work, we present OBSCURE, a versatile framework for practical and cryptographically strong software obfuscation relying on a simple stateless secure element. By a formal proof, we show that OBSCURE achieves virtual black-box security, the strongest form of obfuscation, provided that the secure element is tamper-resistant.

One of the main features of OBSCURE is the simplicity of the functionality that must be embedded in a secure hardware. It works as a simple and small arithmetic unit with small internal memory. This way it can be easily included as a small software library in a secure element or implemented in hardware as a simple circuit. This should be smaller by order of magnitudes than solutions employing oblivious RAM or solutions relying on SGX.

We describe and benchmark the programs of some potential applications to show that OBSCURE can be an alternative to pure software-based white-box implementations. Additionally, we propose a new rectangular universalization technique, which is also of independent interest. Last but not least, by open-sourcing the implementations of OBSCURE, we will ease follow-up works and future comparisons.

Acknowledgement

References

[AES01]   Advanced Encryption Standard (AES). National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, November 2001.

[AFGH05]   Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *NDSS 2005*. The Internet Society, February 2005.

[Aga18]   Abien Fred Agarap. Deep learning using rectified linear units (ReLU). *CoRR*, abs/1803.08375, 2018.

[AGKS20]   Masaud Y. Alhassan, Daniel Günther, Ágnes Kiss, and Thomas Schneider. Efficient and scalable universal circuits. *Journal of Cryptology*, 33(3):1216–1271, July 2020.

[AJX+19]   Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. OBFUSCURO: A commodity obfuscation engine on intel SGX. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, 2019.

[Bar16]   Boaz Barak. Hopes, fears, and software obfuscation. *Commun. ACM*, 59(3):88–96, feb 2016.

[BBdS+21] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Qingju Wang, and Alex Biryukov. Schwaemm and Esch: lightweight authenticated encryption and hashing using the Sparkle permutation family. version v1.2. *NIST Lightweight Cryptography Finalists*, 2021.

[BCD+19]   Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. PHOTON-beetle authenticated encryption and hash family. *NIST Lightweight Cryptography Finalists*, 2019.

[Ben64]   V. E. Beneš. Permutation groups, complexes, and rearrangeable connecting networks. *The Bell System Technical Journal*, 43(4):1619–1640, 1964.

[Ben22]   Eli Bendersky. pycparser v2.21, 2022.

[Ber20]   Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks. Cryptology ePrint Archive, Report 2020/1493, 2020. https://eprint.iacr.org/2020/1493.

[BGEC04]   Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240. Springer, Heidelberg, August 2004.

[BGI+01]   Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

[BGK+19]   Andrey Bogdanov, Louis Goubin, Stefan Kölbl, Pascal Paillier, Matthieu Rivain, Elmar Tischhauser, and Junwei Wang. CHES 2019 Capture The Flag Challenge. The WhibOx Contest, 2nd Edition, 2019. https://whibox.io/contests/2019/.

[BHMT16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 215–236. Springer, Heidelberg, August 2016.

[BR05]   Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography, 2005. Course Notes.

[BU18]   Alex Biryukov and Aleksei Udovenko. Attacks and countermeasures for white-box designs. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 373–402. Springer, Heidelberg, December 2018.

[CAC⁺81] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[CEJv03] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, August 2003.

[CEJvO02] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *J. Cryptol.*, 34(3):33, 2021.

[Den12]   Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[DLPR14] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264. Springer, Heidelberg, August 2014.

[FVBG17] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 765–782, New York, NY, USA, 2017. Association for Computing Machinery.

[FYDX21] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Comput. Surv.*, 54(6), jul 2021.

[GIS⁺10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 308–326. Springer, Heidelberg, February 2010.

[GRW20]   Louis Goubin, Matthieu Rivain, and Junwei Wang. Defeating state-of-the-art white-box countermeasures. *IACR TCHES*, 2020(3):454–482, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8597.

[HB15]   Máté Horváth and Levente Buttyán. The birth of cryptographic obfuscation –
            a survey. Cryptology ePrint Archive, Paper 2015/412, 2015. https://eprint.
            iacr.org/2015/412.

[JLS21]   Aayush Jain, Huijia Lin, and Amit Sahai. *Indistinguishability Obfuscation
            from Well-Founded Assumptions*, page 60–73. Association for Computing
            Machinery, New York, NY, USA, 2021.

[KGP+21]   Stefan Kölbl, Louis Goubin, Pascal Paillier, Matthieu Rivain, Aleksei Udovenko,
            and Junwei Wang. CHES 2021 Capture The Flag Challenge. The WhibOx
            Contest, 3nd Edition, 2021. https://whibox.io/contests/2021/.

[KS16]   Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In
            Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*,
            volume 9665 of *LNCS*, pages 699–728. Springer, Heidelberg, May 2016.

[LHM+15]   Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and
            Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivi-
            ous computation. In *Proceedings of the Twentieth International Conference on
            Architectural Support for Programming Languages and Operating Systems*, AS-
            PLOS '15, page 87–101, New York, NY, USA, 2015. Association for Computing
            Machinery.

[LRD+14]   Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart
            Preneel. Two attacks on a white-box AES implementation. In Tanja Lange,
            Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*,
            pages 265–285. Springer, Heidelberg, August 2014.

[LYZ+21]   Hanlin Liu, Yu Yu, Shuoyao Zhao, Jiang Zhang, Wenling Liu, and Zhenkai Hu.
            Pushing the limits of valiant's universal circuits: Simpler, tighter and more
            compact. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*,
            volume 12826 of *LNCS*, pages 365–394, Virtual Event, August 2021. Springer,
            Heidelberg.

[MLS+13]   Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste
            Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious
            computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC
            Conference on Computer & Communications Security*, CCS '13, page 311–324,
            New York, NY, USA, 2013. Association for Computing Machinery.

[NFR+17]   Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran,
            Satya V. Lokam, Elaine Shi, and Vipul Goyal. HOP: hardware makes obfus-
            cation practical. In *24th Annual Network and Distributed System Security
            Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1,
            2017*. The Internet Society, 2017.

[PCY+17]   Emmanuel Prouff, Chen-Mou Cheng, Bo-Yin Yang, Thomas Baignères,
            Matthieu Finiasz, Pascal Paillier, and Matthieu Rivain. CHES 2017 Cap-
            ture The Flag Challenge. The WhibOx Contest, 2017. https://whibox.io/
            contests/2017/.

[PS99]   Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM
            Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–
            913, 1999.

[RBBK01]   Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 196–205. ACM Press, November 2001.

[SS17]   Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 180–194, Cham, 2017. Springer International Publishing.

[SW14]   Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.

[Val76]   Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, page 196–203, New York, NY, USA, 1976. Association for Computing Machinery.

[WH21]   Hongjun Wu and Tao Huang. TinyJAMBU: A family of lightweight authenticated encryption algorithms (version 2). *NIST Lightweight Cryptography Finalists*, 2021.

[ZYZL19]   Shuoyao Zhao, Yu Yu, Jiang Zhang, and Hanlin Liu. Valiant's universal circuits revisited: An overall improvement and a lower bound. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 401–425. Springer, Heidelberg, December 2019.

A   Security Proof

We assume that the used AEAD scheme satisfies the security notions of *privacy* and *authenticity* as defined in [RBBK01], which we recall hereafter.

An adversary $\mathcal{A}$ is *nonce-respecting* if it never repeats a nonce: if $\mathcal{A}$ asks its oracle an encryption query with nonce N, it will never subsequently ask its oracle a query with same nonce N, regardless of its randomness (if any) and regardless of oracle responses. In the above security notions, the adversary is assumed to be nonce-respecting. This means that for these security notions to hold, the usage of the authenticated encryption scheme must ensure that nonces are not repeated (which is the case in our context as we shall see in the proof of Theorem 1). In the following $\mathcal{K}$ denotes the key space of the AEAD scheme.

- Privacy: Consider an adversary $\mathcal{A}$ that has one of two types of oracles: a "real" encryption oracle or a "fake" encryption oracle. A real encryption oracle, $\mathsf{AEnc}_K(\cdot, \cdot, \cdot)$, takes as input N, A, M and returns $C \leftarrow \mathsf{AEnc}_K(N, A, M)$. A fake encryption oracle, $\$(\cdot, \cdot, \cdot)$, takes as input N, A, M and returns $C \leftarrow \{0,1\}^{\gamma(A,M)}$ where $\gamma(A, M)$ is the bit-length of an encryption of a message M with associated data A. The AEAD scheme achieves $(t, \varepsilon_{\mathrm{pr}})$-*privacy* if for every adversary $\mathcal{A}$ running in time t:

$$\big| \Pr[K \leftarrow \mathcal{K} \; : \; \mathcal{A}^{\mathsf{AEnc}_K(\cdot,\cdot,\cdot)} = 1]$$
$$- \Pr[K \leftarrow \mathcal{K} \; : \; \mathcal{A}^{\$(\cdot,\cdot,\cdot)} = 1] \big| \; \leq \; \varepsilon_{\mathrm{pr}}$$

- Authenticity: Consider an adversary $\mathcal{A}$ with an encryption oracle $\mathsf{AEnc}_K(\cdot, \cdot, \cdot)$ for some key K. Adversary $\mathcal{A}$ *forges* if $\mathcal{A}$ outputs (N, A, C) such that $(M, R) \leftarrow \mathsf{ADec}_K(N, A, C)$ with $R =$ "valid" and $\mathcal{A}$ made no earlier query (N, A, M) to the oracle which resulted in a response C. The AEAD scheme achieves $(t, \varepsilon_{\mathrm{au}})$-*authenticity* if for every adversary $\mathcal{A}$ running in time t:

$$\Pr[K \leftarrow \mathcal{K} \; : \; \mathcal{A}^{\mathsf{AEnc}_K(\cdot,\cdot,\cdot)} \text{ forges}] \leq \varepsilon_{\mathrm{au}}$$

We further assume the used public-key encryption scheme achieves indistinguishability under chosen plaintext attacks (IND-CPA) and the used hash functions is sample among a family of collision resistant hash functions (see *e.g.* [BR05]):

- Indistinguishability under CPA: Consider an encryption oracle $\mathcal{E}$, which on input (b, M_0, M_1) where $b \in \{0,1\}$ returns an encryption $\mathsf{Enc}_{\mathsf{pub}}(M_b)$. The PKE scheme achieves $(t, \varepsilon_{\mathrm{ind}})$-*IND-CPA* if for every adversary $\mathcal{A}$ running in time t:

$$\big| \Pr[\mathcal{A}^{\mathcal{E}(0,\cdot,\cdot)}(\mathsf{pub}) = 1] - \Pr[\mathcal{A}^{\mathcal{E}(1,\cdot,\cdot)}(\mathsf{pub}) = 1] \big| \leq \varepsilon_{\mathrm{ind}}$$

  where the above probabilities are over a random sampling of $(\mathsf{pub}, \mathsf{priv})$ on the PKE key space, the randomness of $\mathcal{A}$ and the randomness of $\mathcal{E}$.

- Collision resistance: Consider a hash function randomly sampled from a family $\mathcal{H} := \{\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^{2\lambda}\}$. This hash function (family) is $(t, \varepsilon_{\mathrm{cr}})$-*collision resistant* if for every adversary $\mathcal{A}$ running in time t:

$$\Pr\left[\begin{array}{c} x \neq x' \\ \cap\ \mathsf{Hash}(x) = \mathsf{Hash}(x') \end{array} \ \middle| \ \begin{array}{l} \mathsf{Hash} \leftarrow \mathcal{H}; \\ (x, x') \leftarrow \mathcal{A}(\mathsf{Hash}) \end{array} \right] \leq \varepsilon_{\mathrm{cr}}$$

Proof of Theorem 1. We demonstrate the VBB white-box security of our obfuscator in the white-box mode. We note that the VBB obfuscation security of our obfuscator (in full obfuscation mode) directly holds from its VBB white-box security and using an universalized program.

We assume that the considered program is such that each output batch (*i.e.* a block of ℓ output words revealed in plain by the SE at the end of the program) depends on all the input batches $X_1, \ldots, X_L$ in the data flow of the program (*i.e.* looking at the data flow graph, each output batch has all the input batches as ancestor nodes). Note that this is always the case for a universalized program (due to the permutation networks fully connecting nodes between layers) but might not be the case for a non-universalized program. This assumption which we refer to as the *input-output dependency assumption* is only made for the sake of simplicity and we explain how it can be relaxed at the end of the proof.

Our proof consists of a sequence of experiments where the first experiment corresponds to the white-box VBB security experiment while an efficient simulator can be extracted from the last experiment. At each step, we upper bound the computational closeness of two successive experiments' outputs.

Experiment 1. This experiment takes as input a program P and an adversary $\mathcal{A}$. It first runs the key generation KeyGen with input 1^λ to get a key pair $(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}})$. Then run the obfuscator on the program P with SE public key $\mathsf{pub}_{\mathsf{SE}}$. Finally it runs the adversary $\mathcal{A}$ on $(\mathsf{pub}_{\mathsf{SE}}, \widehat{P}, \overline{P})$ with oracle $\mathsf{SE}(\mathsf{priv}_{\mathsf{SE}}, \cdot)$ and returns the adversary's output out_1.

Experiment 1 (on input P and $\mathcal{A}$):

$(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$\widehat{P} \leftarrow \mathsf{Obf}(P, \mathsf{pub}_{\mathsf{SE}})$
$out_1 \leftarrow \mathcal{A}^{\mathsf{SE}(\mathsf{priv}_{\mathsf{SE}}, \cdot)}(\mathsf{pub}_{\mathsf{SE}}, \widehat{P}, \overline{P})$
Return out_1

We shall denote t the total running time of Experiment 1, which we assume dominated by the adversary computation time $\mathcal{A}$. Without loss of generality, we assume that $\mathcal{A}$ never makes twice the same request to SE.

Through a few experiment transitions, we will show how we can define another experiment working with a different definition of SE and solely taking $\overline{P}$ as input and which produces an output computationally close to out_1.

Experiment 2 (no hash collisions). This experiment is similar to the previous one, with the following difference. It keeps a list $\mathcal{Q} = \{(x,y)\}$ of all the hash computation performed by the secure element. More precisely, $\mathcal{Q}$ is initialized to $\emptyset$ at the beginning of the experiment. Whenever a request is sent to the secure element and for each hash computation in this request, one computes $y = \mathsf{Hash}(x)$ where x is the hash function input and checks whether there exists $(x',y) \in \mathcal{Q}$ with $x' \neq x$. If so, Experiment 2 fails, namely it stops and outputs $\bot$. Otherwise Experiment 2 adds (x,y) to $\mathcal{Q}$ and continues normally with y as output of the hash function.

Let out_2 denotes the output of Experiment 2. By collision resistance of the hash function, we have

$$out_2 \approx_{(t,\varepsilon_{\mathrm{cr}})} out_1 \ .$$

Experiment 3 (no forgeries). This experiment is similar to the previous one, with the following difference. It keeps a list $\mathcal{C} = \{(C,K)\}$ of all the AEAD ciphertexts C computed by the secure element for a key K. More precisely, $\mathcal{C}$ is initialized to $\emptyset$ at the beginning of the experiment. Whenever a request is sent to the secure element and for each AEAD encryption $C \leftarrow \mathsf{AEnc}_K(N,A,M)$ in this request, the pair (C,K) is added to $\mathcal{C}$. Moreover, whenever a request is sent to the secure element and for each AEAD decryption $(M,R) \leftarrow \mathsf{ADec}_K(N,A,C)$ in this request for which $R \neq \bot$, the experiment checks whether (C,K) is in $\mathcal{C}$ or $(K,N,A,C) = (K_\mathrm{S},\nu,A_\nu,C_\nu)$ for some $\mathrm{MI}_\nu = (\nu,A_\nu,C_\nu)$ part of the obfuscated program $\widehat{P}$. If none of these two cases occur, which means that the ciphertext C is a forgery input by the adversary to the current SE request, then Experiment 3 fails, namely it stops and outputs $\bot$.

Now let us recall that the secure element makes four types of AEAD encryptions:

1. an execution key $E_\mathrm{K} = \mathsf{AEnc}_{K_\mathsf{SE}}(\mathsf{Hash}_1(E_\mathrm{ID}),\emptyset,K_\mathrm{S})$ with $\mathsf{Hash}_1(E_\mathrm{ID})$ as nonce, in a "Start" request,

2. an input chain MAC $M_{i-1}^{\mathrm{in}} = \mathsf{AEnc}_{K_\mathsf{SE}}(N_{i-1}^{\mathrm{in}},A_{i-1}^{\mathrm{in}},\emptyset)$ with $N_{i-1}^{\mathrm{in}} = \mathsf{Hash}_2(E_\mathrm{ID} \,\|\, i-1)$ as nonce, in an "Input" request,

3. an encrypted word $C_{i,j} = \mathsf{AEnc}_{K_\mathsf{SE}}(N_{i,j},A_{i,j},X_{i,j})$ with $N_{i,j} = \mathsf{Hash}_3(E_\mathrm{ID} \,\|\, i \,\|\, j)$ as nonce, in an "Input" request,

4. an encrypted word $C_{\nu,j} = \mathsf{AEnc}_{K_\mathsf{SE}}(N_{\nu,j},A_{\nu,j},v_j)$ with $N_{\nu,j} = \mathsf{Hash}_3(E_\mathrm{ID} \,\|\, \nu \,\|\, j)$ as nonce, in an "Eval" request.

The absence of hash collision readily implies the absence of nonce collision between these different encryptions (either two different item or same item with different indexes ν, i and/or j). Indeed all the nonces are hash outputs with strictly distinct inputs while E_ID cannot collides either for two different executions due to the absence of hash collisions.

Let us now show that the same nonce cannot be used twice with different plaintexts and/or associated data without a previous forgery.

- For the first type of encryption (execution key), we cannot have twice the same nonce for different K_S because this would imply a hash collision (since for a different K_S we have a different $\mathcal{H}$ and hence a different E_ID).

- For the second type of encryption (which is actually a MAC), obtaining two MACs M_{i-1}^{in} and $M_{i-1}^{\mathrm{in}\,\prime}$ for same nonce N_{i-1}^{in} and different associated data $A_{i-1}^{\mathrm{in}} = (H_{i-1} \,\|\, i-1 \,\|\, E_\mathrm{ID})$ and $A_i^{\mathrm{in}\,\prime}$ can only be achieved through two requests

$$\mathsf{SE}(\text{"Input"}, E_\mathrm{ID}, i, H_{i-1}, X_i, M_i^{\mathrm{in}})$$

with different H_{i-1} and H'_{i-1} (the execution identity E_{ID} and index i must be the same to get the same N^{in}_{i-1}). For these two requests to proceed and output the two MACs M^{in}_i and $M^{\text{in}'}_i$, they must be respectively given in input X_i, M^{in}_i and $X'_i, M^{\text{in}'}_i$ where M^{in}_i (resp $M^{\text{in}'}_i$) is a valid MAC of $A^{\text{in}}_i = (H_i \parallel i \parallel E_{\text{ID}})$ with $H_i = \mathsf{Hash}_0(H_{i-1} \parallel X_i)$ (resp. $A^{\text{in}'}_i$ defined in the same way). Since, by assumption, no previous forgery occurred, these two valid MACs have been obtained from two previous call to $\mathsf{SE}(\text{"Input"}, E_{\text{ID}}, \ldots)$. Rewinding the chain of calls to the secure element, we thus get two initial calls to $\mathsf{SE}(\text{"Start"}, \ldots)$ which gives rise to the same E_{ID} with two different chains of hashes (since $H_{i-1} \neq H'_{i-1}$). However this cannot occur since we have from Experiment 2 that no hash collisions occur.

- For the third type of encryption, the argument is similar to the above. The adversary should be able to make two "Input" requests with different for the same execution identity E_{ID} and same index i with different input word $X_{i,j}$ (the associated data $A_{i,j} = (i \parallel j \parallel E_{\text{ID}})$ is necessarily the same to get the same nonce $N_{i,j} = \mathsf{Hash}_3(E_{\text{ID}} \parallel i \parallel j)$). For such two requests cannot occur in the absence of hash collision and previous forgery.

- For the fourth type of encryption, the argument is similar. In order to obtain two valid ciphertexts $C_{\nu,j}$ and $C'_{\nu,j}$ with same nonce $N_{\nu,j} = \mathsf{Hash}_3(E_{\text{ID}} \parallel \nu \parallel j)$ and different plaintext v_j, the adversary should be able to make two non-failing calls

$$\mathsf{SE}(\text{"Eval"}, E_{\text{ID}}, E_{\text{K}}, \text{MI}_\nu, C^*_1, \ldots, C^*_\ell)$$

with same E_{ID} and same ν but leading to a different computation of $(v_1, \ldots, v_\ell) \leftarrow f_\nu(u_1, \ldots, u_\ell)$. If either E_{K} or MI_ν (or both) is different in these two calls, then a previous forgery was achieved. If one or several of the C^*_k's are different then we can loop the argument and we also get that a previous forgery must have been achieved.

In the absence of hash collision and before a forgery was achieved by the adversary, we hence have that all the AEAD encryptions computed and returned by the secure element are nonce-respecting. In this context, Experiment 3 fails due to a forgery if and only if the adversary breaks the authenticity of the AEAD scheme.

Let out_3 denotes the output of Experiment 3. By the authenticity of the AEAD scheme, we have

$$out_3 \approx_{(t, \varepsilon_{\text{au}})} out_2 \ .$$

Experiment 4 (associative array). This experiment is similar to the previous one, with the following difference. It builds an associative array denoted T with the following fields:

- $T[E_{\text{ID}}][\text{"}L\text{"}]$ keeps the length L corresponding to an execution identity E_{ID},

- $T[E_{\text{ID}}][\text{"}E_{\text{K}}\text{"}]$ keeps the execution key E_{K} corresponding to an execution identity E_{ID},

- $T[E_{\text{ID}}][\text{"}X_i\text{"}]$ keeps the i-th input batch X_i corresponding to an execution identity E_{ID}, for $i \in [1, L]$.

During a call to $\mathsf{SE}(\text{"Start"}, \ldots)$, a new entry to the associative array is created for the corresponding execution identity E_{ID}. The fields $T[E_{\text{ID}}][\text{"}L\text{"}]$ and $T[E_{\text{ID}}][\text{"}E_{\text{K}}\text{"}]$ are set to the computed length L and execution key E_{K}. During a call $\mathsf{SE}(\text{"Input"}, \ldots)$, the field $T[E_{\text{ID}}][\text{"}X_i\text{"}]$ is set to the input X_i before returning the output, *i.e.* if and only if the different checks have passed. Note that, unless the experiment fails, each call to $\mathsf{SE}(\text{"Start"}, \mathcal{H}, H_L)$ gives rise to a different $E_{\text{ID}} = \mathsf{Hash}_0(H_L \parallel \mathcal{H})$ since the adversary does not make the same call twice and since no hash collision occurs.

The experiment is further modified as follows: for any request of the form $\mathsf{SE}(\text{“Input”}, E_{\mathrm{ID}}, \ldots)$ or $\mathsf{SE}(\text{“Eval”}, E_{\mathrm{ID}}, \ldots)$, the experiment first checks whether an entry in T exists for E_{ID}. If no entry exists for E_{ID} (the adversary is *bluffing* with the input execution ID), the request is answered by $\perp$, otherwise it runs as usual.

Let us remark that any request of the form $\mathsf{SE}(\text{“Input”}, E_{\mathrm{ID}}, \ldots)$ or $\mathsf{SE}(\text{“Eval”}, E_{\mathrm{ID}}, \ldots)$ includes one or several AEAD decryption(s)

- $(\emptyset, R_i^{\mathrm{in}}) \leftarrow \mathsf{ADec}_{K_{\mathsf{SE}}}(N_i^{\mathrm{in}}, A_i^{\mathrm{in}}, M_i^{\mathrm{in}})$, or

- $(u_k, R') \leftarrow \mathsf{ADec}_{K_{\mathsf{SE}}}(N', A', C_k^*)$

and returns $\perp$ whenever the decryption fails. Therefore in such a request, either the input MAC M_i^{in} is valid (resp. the input ciphertexts C_k^* are valid) or the request returns $\perp$. In the latter case, Experiment 4 behaves like Experiment 2. Now assume the input MAC (or ciphertexts) is valid. Since we have from Experiment 3 that no forgeries occur, an entry for E_{ID} necessarily exists in the table. Indeed, for each request with valid input MAC (resp. ciphertexts), there exists a previous request producing this valid MAC (resp. ciphertexts) which itself takes as input a valid MAC (resp. ciphertexts), until the original $\mathsf{SE}(\text{“Start”}, \ldots)$ request which creates the entry for E_{ID}. Therefore Experiment 4 behaves like Experiment 2.

Let out_4 denotes the output of Experiment 4. We thus have

$$out_4 \approx_{(t,0)} out_3 .$$

Experiment 5 (output revealing requests). This experiment is similar to the previous one, with the following difference. It adds a field $T[E_{\mathrm{ID}}][\text{“output”}]$ to the associative array. Whenever a request $\mathsf{SE}(\text{“Input”}, E_{\mathrm{ID}}, i, \ldots)$ with $i = 1$ is made by the adversary, the experiment reconstructs the program input $x_1, \ldots, x_n$ from the batches $X_1, \ldots, X_L$ stored in $T[E_{\mathrm{ID}}][\text{“}X_1\text{”}], \ldots, T[E_{\mathrm{ID}}][\text{“}X_L\text{”}]$. If one of these entries is missing from T then the experiment fails, namely it stops and returns $\perp$. Otherwise, the experiment evaluates the program on the reconstructed input to get the output $(y_1, \ldots, y_{n'}) = P(x_1, \ldots, x_n)$ and stores it in $T[E_{\mathrm{ID}}][\text{“output”}]$.

Now, whenever a request $\mathsf{SE}(\text{“Eval”}, E_{\mathrm{ID}}, \ldots)$ is made by the adversary with a multi-instruction MI_ν for which the revelation flag b_R^ν is set to true, the experiment checks whether the associative array has an entry $T[E_{\mathrm{ID}}][\text{“output”}]$. If so, it defines the plain output $(v_1, \ldots, v_\ell)$ of the SE request as the corresponding subpart of $T[E_{\mathrm{ID}}][\text{“output”}]$. Otherwise, the experiment fails, namely it stops and outputs $\perp$.

If the experiment does not fail, then we know by definition of Experiment 3 that no forgeries occur. It means that the requests $\mathsf{SE}(\text{“Eval”}, E_{\mathrm{ID}}, \ldots)$ are only called with genuine multi-instructions MI_ν (and hence genuine revelation flags b_R^ν) from the obfuscated program $\widehat{P}$, and with genuine encrypted words $(C_1^*, \ldots, C_\ell^*)$ from previous SE requests. Moreover, according to the input-output dependency assumption (stated at the beginning of the proof), we know that a genuine request $\mathsf{SE}(\text{“Eval”}, E_{\mathrm{ID}}, \ldots)$ with revelation flag $b_R^\nu = \text{true}$ can only occur if all the $\mathsf{SE}(\text{“Input”}, E_{\mathrm{ID}}, \ldots)$ requests have been previously executed and hence the $T[E_{\mathrm{ID}}][\text{“output”}]$ entry well exists (otherwise a forgery would have been achieved by the adversary). The plain answers $(v_1, \ldots, v_\ell)$ to the requests $\mathsf{SE}(\text{“Eval”}, E_{\mathrm{ID}}, \ldots)$ with $b_R^\nu = \text{true}$ then always match between Experiment 5 and Experiment 4 by functional correctness of the obfuscator.

Let out_5 denotes the output of Experiment 5. We thus have

$$out_5 \approx_{(t,0)} out_4 .$$

Experiment 6 (random intermediate variables). This experiment is similar to the previous one, with the following difference. In a call $\mathsf{SE}(\text{“Eval”}, \ldots)$ for which the

revelation flag b_R^ν is set to false, instead of computing the multi-instruction output as $(v_1, \ldots, v_\ell) \leftarrow f_\nu(u_1, \ldots, u_\ell)$, the Experiment 6 samples it uniformly at random:

$$(v_1, \ldots, v_\ell) \leftarrow \left(\{0,1\}^w\right)^\ell .$$

By the privacy of the AEAD scheme, the adversary cannot efficiently distinguish Experiment 5 and Experiment 6 and essentially behaves similarly. Formally, we have

$$out_6 \approx_{(t,\varepsilon_{\mathrm{pr}})} out_5$$

where out_6 denotes the output of Experiment 6.

Experiment 7 (no bytecode decryption). This experiment is similar to the previous one, with the following difference. In a call $\mathsf{SE}(\text{"Eval"}, \ldots)$, we replace the decryption and authentication of the multi-instruction $\mathrm{MI}_\nu = (\nu, A_\nu, C_\nu)$, namely the steps

   $(f_\nu, R_\nu) \leftarrow \mathsf{ADec}_{K_\mathrm{S}}(\nu, A_\nu, C_\nu)$
   if $R_\nu = \bot$ **then return** $\bot$

by a check that MI_ν is indeed part of the obfuscated program $\widehat{P}$. The experiment fails whenever the check does not pass.

Since f_ν is no more used from Experiment 6 and since no request is made with a forged C_ν, Experiment 7 behaves as Experiment 6. Formally, we have

$$out_7 \approx_{(t,0)} out_6$$

where out_7 denotes the output of Experiment 7.

Experiment 8 (no shared key decryption). This experiment is similar to the previous one, with the following difference. In a call $\mathsf{SE}(\text{"Eval"}, E_\mathrm{ID}, E_\mathrm{K}, \ldots)$, we replace the decryption and authentication of the execution key (resulting in the shared key):

   $(K_\mathrm{S}, R_\mathrm{S}) \leftarrow \mathsf{ADec}_{K_\mathrm{SE}}(\mathsf{Hash}_1(E_\mathrm{ID}), \emptyset, E_\mathrm{K})$
   if $R_\mathrm{S} = \bot$ **then return** $\bot$

by a check that the input execution key E_K matches the record $T[E_\mathrm{ID}][\text{"}E_\mathrm{K}\text{"}]$ (for the input execution identity E_ID). The experiment fails whenever the check does not pass.

Since K_S is no more used from Experiment 7 and since no request is made with a forged E_K, Experiment 8 behaves as Experiment 7. Formally, we have

$$out_8 \approx_{(t,0)} out_7$$

where out_8 denotes the output of Experiment 8.

Experiment 9 (random shared key). This experiment is similar to the previous one, with the following difference. In the definition of the obfuscator, two different shared key K_S and K_S' are generated. While the first one, K_S, is used to encrypt and authenticate the multi-instructions as in the previous experiment, the second one is encrypted

$$C_\mathcal{H} \leftarrow \mathsf{Enc}_{\mathsf{pub}_{\mathsf{SE}}}(K_\mathrm{S}')$$

and embedded in the header $\mathcal{H}$.

The indistinguishability of the PKE scheme under chosen plaintext attacks prevents the adversary from distinguishing between the previous experiment in which $C_\mathcal{H}$ is the encryption of K_S (which is used to encrypt the multi-instructions) and the current experiment in which $C_\mathcal{H}$ is the encryption of a random and independent K_S'. Let out_9 denotes the output of Experiment 9, we thus have

$$out_9 \approx_{(t,\varepsilon_{\mathrm{ind}})} out_8 .$$

Experiment 10 (program switching). This experiment is similar to the previous one, with the following difference. The obfuscator is given $\overline{P}$, the zeroized-constants version of P, as input in place of P. In other words, the original multi-instructions $\mathrm{MI} = (\nu, A_\nu, C_\nu)$ with $A_\nu = (b_R^\nu, \mathrm{ID}_{\mathrm{in}}^\nu)$ and $C_\nu = \mathsf{AEnc}_{K_{\mathrm{S}}}(\nu, A_\nu, f_\nu)$ are replaced by

$$\overline{\mathrm{MI}} = (\nu, A_\nu, \overline{C}_\nu) \quad \text{with} \quad \overline{C}_\nu = \mathsf{AEnc}_{K_{\mathrm{S}}}(\nu, A_\nu, \overline{f}_\nu)$$

where $\overline{f}_\nu$ is the zeroized-constants version of f_ν.

Since from Experiment 7, the calls $\mathsf{SE}(\text{``Eval''}, \ldots)$ do not decrypt the bytecode anymore, this change has no effect on the SE functionality. Moreover, the privacy of the AEAD scheme prevents the adversary from distinguishing between the previous experiment in which $\widehat{P}$ contains encryptions of the f_ν's and the current experiment in which $\widehat{P}$ contains encryptions of the $\overline{f}_\nu$'s. Let out_{10} denotes the output of Experiment 10, we thus have

$$out_{10} \approx_{(t, \varepsilon_{\mathrm{pr}})} out_9 \ .$$

Compiling the upper bounds of the computational closeness of the successive experiments, we finally get that Experiment 1 and Experiment 10 produce (t, ε)-close output distribution, that is

$$out_1 \approx_{(t, \varepsilon)} out_{10}$$

with

$$\varepsilon \leq 2\varepsilon_{\mathrm{pr}} + \varepsilon_{\mathrm{au}} + \varepsilon_{\mathrm{ind}} + \varepsilon_{\mathrm{cr}} \ .$$

We summarize Experiment 10 hereafter with SE' denoting the secure element oracle defined throughout the sequence of experiments and with Obf' defining the obfuscator of Experiment 9 (in which a random shared key is embedded into the header):

Experiment 10 (on input P and $\mathcal{A}$):

$(\mathsf{pub}_{\mathsf{SE}}, \mathsf{priv}_{\mathsf{SE}}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$\widehat{P} \leftarrow \mathsf{Obf}'(\overline{P}, \mathsf{pub}_{\mathsf{SE}})$
$out_{10} \leftarrow \mathcal{A}^{\mathsf{SE}'(\mathsf{priv}_{\mathsf{SE}}, \cdot)}(\mathsf{pub}_{\mathsf{SE}}, \widehat{P}, \overline{P})$
Return out_{10}

We note that Experiment 10 only depends on $\overline{P}$, the zeroized-constants version of P, except when it evaluates the program P on a reconstructed input and stores the corresponding output in $T[E_{\mathrm{ID}}][\text{``output''}]$. We note that this evaluation can be performed with an oracle access to P. We can then define our white-box VBB simulator $\mathcal{S}^{P(\cdot)}$ as the procedure of Experiment 10 with an oracle access to evaluate P. We thus obtain

$$\mathcal{A}^{\mathsf{SE}(\mathsf{priv}_{\mathsf{SE}}, \cdot)}(\mathsf{pub}_{\mathsf{SE}}, \widehat{P}, \overline{P}) \approx_{(t, \varepsilon)} \mathcal{S}^{P(\cdot)}(\overline{P}) \ ,$$

which concludes the proof.

Relaxing the input-output dependency assumption. This assumption is used in the proof to ensure that, for any given execution identity E_{ID}, an "Input" request with $i = 0$ must be made before any revealing "Eval" request can be made. Thus, the $T[E_{\mathrm{ID}}][\text{``output''}]$ field has well been assigned by a call the program evaluation oracle before any part of the output must be revealed. However, we do not formally need to do this. Namely, without the input-output dependency assumption, a revealing "Eval" request could be called before the "Input" request with $i = 0$. But in this case, by construction, the $T[E_{\mathrm{ID}}][\text{``}X_i\text{''}]$ fields would have been assigned for all the input batches X_i incoming the revealed output. In that case, the simulator could still answer this request by calling the program evaluation oracle with the assigned X_i's and any other values for the non-assigned X_i's (since the revealed output is independent of the latter input batches). $\qquad\square$

B Impossibility of Encoding Permutations with Duplicates using a Single Beneš Network

In this section, we will prove that extending a single swap-based Beneš network with arbitrary 2×2 gates is not sufficient for computing arbitrary orderings $\pi \in \{1, \ldots, n\}^n$.

Proposition 1. *Let $n = 2^m$ where $m \geq 3$ is a integer. Consider the Beneš permutation network for n elements, with the 2×2 controlled swap gates extended with the possibility of controlled copying of any of the inputs. Then, there exists an ordering $\pi \in \{1, \ldots, n\}^n$ that this network can not compute.*

Proof. Recall that the two outer layers of the Beneš network swap controllably the elements with 0-based index pairs $(j, j{\oplus}1)$. In the remaining middle layers, the odd- and even-indexed elements are processed independently. Consider 0-based ordering $(0, 1, 0, 2, 1, 2, 2, 3, \ldots,)$ (the ending is arbitrary but excludes indexes 0-3). Note that the input layer can not perform copying of the selected elements (indexed 0-3), since all 4 of them are present in the output and thus can not be overwritten. Thus, the elements indexed 0 and 1 have to go into different in-the-middle halves (by oddity), the same is true for 2 and 3. Therefore, the output layer induces constraint pairs $(0, 1), (0, 2), (1, 2), (2, 3)$, where the first element in each pair must belong to the other in-the-middle half than the other element (as each can belong only to one half, even although may be copied inside it). However, the three constraints $(0, 1), (0, 2), (1, 2)$ form a cycle of odd length: it impossible to distribute the elements with input indexes 0,1,2 across two in-the-middle oddity halves satisfying the constraint pairs. $\qquad\square$

C MNIST Neural Network

In this section, we present the discretization of Equation 4 and Equation 5 in details. The goal of the discretization is to transform these two equations into new equations working on unsigned 32-bit integers. This includes three passes described hereafter.

First pass: discretizing weights

In the following, we omit the superscript from the weight $w_{j,k}^{(i-1)}$ for the sake of simplicity and since there is no ambiguity on the layer. Let us choose a large enough ratio, denoted θ. Let $w'_{j,k}$ be the discretized weight, we do $w'_{j,k} = \lfloor \theta w_{j,k} \rceil$. We also multiply the ratio to the bias, $b'_{i,j} = \lfloor \theta b_{i,j} \rceil$. This way, we can rewrite Equation 4 as follows:

$$y'_{i,j} = \sum_k w'_{j,k} x_{i-1,k} + b'_{i,j} \tag{6}$$

where $y'_{i,j} \approx \theta y_{i,j}$. The activation function is then replaced by $f' = f/\theta$, so that we can rewrite Equation 5 as:

$$x_{i,j} := f'(y'_{i,j}) = \begin{cases} 0 & \text{if } y'_{i,j} \leq 0 \\ y'_{i,j}/\theta & \text{otherwise} \end{cases} \tag{7}$$

Second pass: encoding x and y

Let us denote by $\hat{y}_i^{\min}$ the minimal value of $\min\{y'_{i,1}, \ldots, y'_{i,N_i}\}$ and $\hat{y}_i^{\max}$ the maximal value $\max\{y'_{i,1}, \ldots, y'_{i,N_i}\}$, where both the min and the max are taken over several evaluations of the neural network (in practice we evaluate the $y'_{i,j}$ for all the images in the dataset). Then,

we compute the width of the interval $[\hat{y}_i^{\min}, \hat{y}_i^{\max}]$, that is $\hat{\Delta}_i = \hat{y}_i^{\max} - \hat{y}_i^{\min}$. To avoid a possible case $y'_{i,j} \notin [\hat{y}_i^{\min}, \hat{y}_i^{\max}]$ occurring for an input image outside the dataset, we increase the interval by a security ratio α to get $\Delta_i = (1+\alpha)\hat{\Delta}_i$, then define $y_i^{\max} = \hat{y}_i^{\max} + \alpha * \Delta_i/2$ and $y_i^{\min} = \hat{y}_i^{\min} - \alpha * \Delta_i/2$. We shall encode $y'_{i,j}$ on the interval $[y_i^{\min}, y_i^{\max})$. Namely, we define the following encoding $[y'_{i,j}]$ for $y'_{i,j}$:

$$[y'_{i,j}] = \frac{y'_{i,j} - y_i^{\min}}{\Delta_i} = \frac{y'_{i,j}}{\Delta_i} - \delta_i \text{ where } \delta_i = \frac{y_i^{\min}}{\Delta_i} \tag{8}$$

so that $y'_{i,j} \in [y_i^{\min}, y_i^{\max}) \Leftrightarrow [y'_{i,j}] \in [0,1)$. Then $x_{i-1,k}$ is also scaled by $1/\Delta_i$ and we simply define the encoding $[x_{i-1,k}]$ as:

$$[x_{i-1,k}] = \frac{x_{i-1,k}}{\Delta_i} \tag{9}$$

This way, we can rewrite Equation 6 as follows:

$$[y'_{i,j}] = \sum_k w'_{j,k}[x_{i-1,k}] + \frac{b'_{i,j}}{\Delta_i} - \delta_i \tag{10}$$

For the activation function, we still use Equation 7 with $y'_{i,j} = \Delta_i([y'_{i,j}] + \delta_i)$ derived from Equation 8.

Third pass: rescaling to 32-bit integers

We rescale the encodings of $x_{i-1,k}$ (Equation 9) and $y'_{i,j}$ (Equation 8) to 32-bit integers as follows:

$$[\![x_{i-1,k}]\!] = \left\lfloor \frac{x_{i-1,k}}{\Delta_i} \times 2^{32} \right\rfloor \pmod{2^{32}} \tag{11}$$

$$[\![y'_{i,j}]\!] = \left\lfloor \left(\frac{y'_{i,j}}{\Delta_i} - \delta_i \right) \times 2^{32} \right\rfloor \pmod{2^{32}} \tag{12}$$

Note that for the above encoding of $x_{i-1,k}$ to be sound, we require $[x_{i-1,k}] = x_{i-1,k}/\Delta_i < 1$ (otherwise some information is lost by the modular reduction). We checked that this always occurred in our case. In case this would not occur, one would need to define a different encoding ensuing $[x_{i-1,k}] \in [0,1)$ with further scaling.

The bias $b'_{i,j}$ is also rescaled:

$$[\![b'_{i,j}]\!] = \left\lfloor \left(\frac{b'_{i,j}}{\Delta_i} - \delta_i \right) \times 2^{32} \right\rfloor \pmod{2^{32}} \tag{13}$$

The weighted sum (Equation 10) can be now rewritten as:

$$[\![y'_{i,j}]\!] = \sum_k w'_{j,k}[\![x_{i-1,k}]\!] + [\![b'_i]\!] \tag{14}$$

For the activation function, we consider Equation 7 with $y'_{i,j}$ derived from Equation 12 as follows:

$$y'_{i,j} = ([\![y'_{i,j}]\!] + \delta_i \times 2^{32}) \times \frac{\Delta_i}{2^{32}} \tag{15}$$

The comparison $y'_{i,j} \leq 0$ in Equation 7 can be replaced with $[\![y'_{i,j}]\!] \leq -\delta_i \times 2^{32}$ (from Equation 15). We notice that if this comparison does not hold, the activation function should return the rescaled encoding $[\![x_{i,j}]\!]$ (for the computation in the neurons of the layer

$i + 1$) which is encoded using Δ_{i+1}. From Equation 12, Equation 5 and the notice of $y_{i,j} = y'_{i,j} \times \theta$, we derive $x_{i,j}$ as follows:

$$x_{i,j} = \frac{(\llbracket y'_{i,j} \rrbracket + \delta_i \times 2^{32}) \times \Delta_i}{\theta \times 2^{32}} \tag{16}$$

Similar to Equation 11, we encode and rescale $x_{i,j}$ from Equation 16:

$$\llbracket x_{i,j} \rrbracket = \left(\llbracket y'_{i,j} \rrbracket - (-\delta_i \times 2^{32}) \right) \times \frac{\Delta_i \times 2^{32}}{\Delta_{i+1} \times \theta} \times \frac{1}{2^{32}} \tag{17}$$

Denote $D_i = \left\lfloor -\delta_i \times 2^{32} \right\rceil$ and $T_i = \left\lfloor \frac{\Delta_i \times 2^{32}}{\Delta_{i+1} \times \theta} \right\rfloor$, we have that D_i and T_i are unsigned 32-bit integers ($D_i > 0$ as $y_i^{\min} < 0$ in practice, see Equation 8) which can be precomputed for a pre-trained neural network. Now, we can rewrite the activation function (Equation 7) as

$$\llbracket x_{i,j} \rrbracket = f''(\llbracket y'_{i,j} \rrbracket) = \begin{cases} 0 & \text{if } \llbracket y'_{i,j} \rrbracket \leq D_i \\ \mathrm{msb}_{32} \left(\left(\llbracket y'_{i,j} \rrbracket - D_i \right) T_i \right) & \text{otherwise} \end{cases} \tag{18}$$

where $\mathrm{msb}_{32}(\cdot)$ denote the function returning the upper 32 bits of a multiplication of two 32-bit integers.

Finally, we use Equation 14 and Equation 18 as the weighted sum and activation function which are adapted to computations on 32-bit integers.

After this transformation, we obtain a discretized neural network working on unsigned 32-bit integers. On our validation set, the accuracy of the obtained network is similar to that of the original network, namely of 97.22%. In other words, the discretization of the neural network to make it compatible with OBSCURE did not imply any loss in terms of accuracy. Section 5 presents the obtained results while applying OBSCURE to this discretized neural network.