IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 293–332. DOI:10.46586/tches.v2025.i2.293-332

A Code-Based ISE to Protect Boolean Masking in Software

Qi Tian[1,2], Hao Cheng[1,3*], Chun Guo[1,3], Daniel Page[4], Meiqin Wang[2,1,3] and Weijia Wang[2,1,3(✉)]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, China.
tianqi512@mail.sdu.edu.cn
{hao.cheng,chun.guo,mqwang,wjwang}@sdu.edu.cn
[2] Quan Cheng Laboratory, Jinan, China
[3] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao, China
[4] School of Computer Science, University of Bristol, Bristol, UK.
daniel.page@bristol.ac.uk

Abstract. Side-Channel Attacks (SCAs) pose a significant threat to data security in embedded environments. To counteract the power-based SCAs, masking is a widely used defense technique, that introduces randomness to obscure the side-channel information generated during the processing of secret data. However, in practice, some challenges exist when implementing masking schemes. For example, in the implementation of Boolean masking, they may refer to low noise level and implementation flaws. To address the said implementation challenges, we present an effective and efficient solution that incorporates the code-based masking technique: We mask the shares of Boolean masking with code-based masking and then use a self-designed Instruction Set Extension (ISE) to perform efficient private computations within this masked domain. Based on a 32-bit RISC-V Ibex core, we develop a prototype implementation of our ISE, whereby it mainly wraps the ALU with three code-based encoders/decoders and integrates a leakage-resilient pseudo-random generator (PRG). Compared to the base core (vanilla Ibex), the hardware overhead of the ISE implementation is only 8%. The security evaluation based on formal verification and practical evaluation demonstrates that our ISE can provide a more robust practical security guarantee. Furthermore, our approach significantly reduces the signal-to-noise ratio (SNR) of each share, decreasing it to just 2% of the original SNR on the base core.

Keywords: side-channel attack, code-based masking, RISC-V, ISE

1 Introduction

Side-channel attacks and masking. Modern embedded computing devices are ubiquitous in nearly every aspect of daily life and have brought significant convenience, with typical examples including home automation, healthcare, transportation, etc. However, the widespread adoption of embedded devices also poses new security challenges, since a great number of them deal with sensitive or secret data while simultaneously being deployed in an adversarial environment. In this context, Side-Channel Attack (SCA) is a major threat, which does not crack encryption algorithms *directly* to access secret information. Instead, SCA can obtain encryption keys or sensitive data *indirectly* through the analysis of the

[*]Part of this work was done while Hao Cheng was at the University of Luxembourg.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2024-10-15 Accepted: 2024-12-15 Published: 2025-03-04

physical side-channel information, which is produced during cryptographic operations on the device, such as power consumption, electromagnetic radiation, and execution time.

In this paper, we focus on Differential Power Analysis (DPA) [KJJ99] and its variants. To counteract DPA, various defense techniques have been explored and studied, which are usually categorized as being based on hiding [MOP07, Chapter 7] and/or masking [MOP07, Chapter 9]. Our focus is on masking, an encryption technique that introduces randomness to obscure the side-channel information generated during the processing of secret data. By interleaving the signals with randomness to counter potential breaches, masking therefore reduces the susceptibility of cryptographic implementations against side-channel attacks. In general, a masking scheme, sometimes called a private circuit compiler, comprises two components: *encoder* and *private computation*. The former encodes the secrets and some randomness into sharings, while the latter mainly computes the cryptographic algorithms over the sharings.

Boolean masking. The simplest and most popular encoder is the additive one, which is used in a d-th order Boolean masking scheme. It randomly encodes any secret variable x into a sharing comprising $d+1$ shares $\langle x_0, x_1, \ldots, x_d \rangle$ such that $x = x_0 \oplus x_1 \oplus \ldots \oplus x_d$, and we use $x \in \mathbb{F}_q$ as an example throughout the paper. To compute the cryptographic algorithms over the sharings, one shall transform each elementary operation (e.g., addition and multiplication over $\mathbb{F}_q$) into a masked and compatible version, which is called *gadget*. An advantage of the additive encoder is, due to its simple structure, the corresponding private computations (e.g., gadgets) can be quite efficient. In detail, the addition gadget with inputs $\langle x_0, x_1, \ldots, x_d \rangle$ and $\langle y_0, y_1, \ldots, y_d \rangle$ can be easily implemented by adding the corresponding shares with the same index, and outputs the result shares $z_i = x_i \oplus y_i$, for $i \in \{0, \ldots, d\}$. As for the multiplication gadget, it is not isomorphic to the additive sharings thus a bit more complex, and a well-known instance, namely ISW multiplication, is proposed by Ishai, Sahai, and Wagner [ISW03].

Challenges in practice. However, when implementing Boolean masking schemes in practice, several challenges arise, primarily related to implementation efficiency and security. In particular, given the target devices have (highly) constrained resources, these challenges may be amplified (terribly). In terms of efficiency, although offering demonstrable and adjustable security against side-channel attacks, the higher-order masking scheme often results in substantial computational and storage overhead. For example, the computational complexity of ISW multiplication, along with its many variants, exhibits a quadratic growth to the security order d. As for the implementation security, Boolean masking schemes on microprocessors confront at least two significant difficulties:

1. Lack of sufficient noise, and thus the masked implementation may suffer from some attacks such as horizontal attacks [BCPZ16] and side-channel dissection [BS20]. This stems mainly from the sequential circuits, because the data processed in the registers and data bus habitually leak (significantly) more than those in the combinational circuits.

2. Implementation flaws such as transitional leakage. Since the combinational circuits are stateless, most defects (excluding the glitches) also come from sequential circuits.

Code-based masking. Code-based masking is a very general type of masking scheme, including inner product masking [DF12, BFGV12], direct sum masking [BCC$^+$14], etc. It has two primary advantages. First, the higher algebraic complexity of the sharing function decreases the information leakage in the "low noise conditions" and may provide a better security than the Boolean masking. Second, compared to the simple additive encoder, its complicated encoder allows a smaller size of the sharing. Specifically, it can encode m-bit

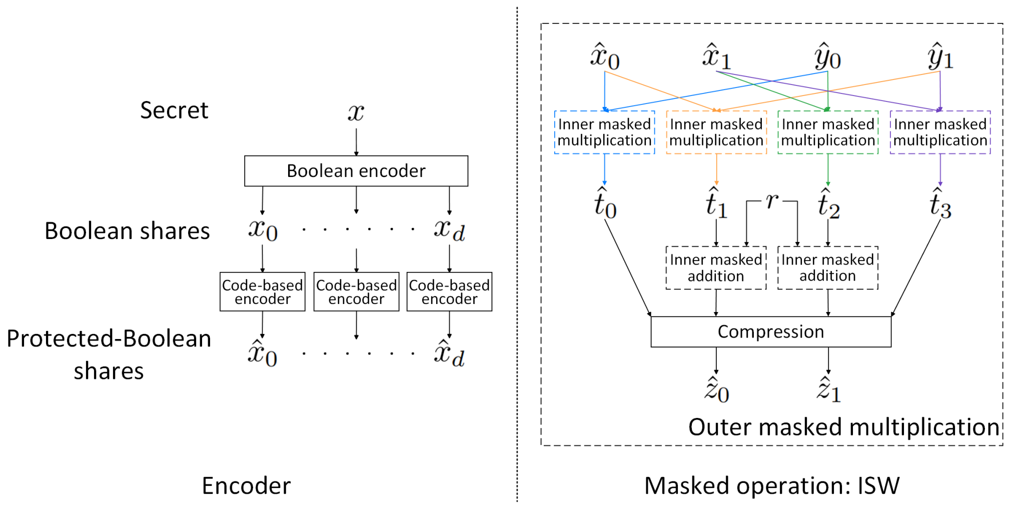


Figure 1: An example describing how to use code-based masking to protect a software implementation of Boolean masking.

secret[1] in $\mathbb{F}_q$, into n-bit ($n \geq d+m$) sharing in $\mathbb{F}_q$. In particular, if $q \leq m+d$, there exists an encoder such that $n = d+m$, while the additive encoder requires $n = dm$. However, the downside of code-based masking is that designing a corresponding private computation is challenging. The existing works only benefit the cryptographic algorithms over $\mathbb{F}_q$ with $q \geq 2^4$ [WMCS20].

Use of ISE to support masking. The implementation strategy of a masking scheme includes at least two options: hardware-only and software-only. When comparing the two, the former is believed to ensure a higher practical security, since it can address leakage via directly the micro-architecture, while the latter has only the architectural means to use. However, when coming to the cost of hardware resources, the former relies on a dedicated circuit, which implies a high overhead, but the latter is zero overhead. In addition, the latter offers greater flexibility, while the former is limited. Apart from them, the hybrid strategy, namely hardware/software co-design, is the third option, which combines the characteristics of the two extremes and attempts to offer a more attractive trade-off. Instruction Set Extension (ISE) is a promising approach in this scope, which extends the Instruction Set Architecture (ISA) by a small set of custom instructions to support the masking-related tasks. Various ISE proposals have been presented in literature aiming to improve the efficiency and/or security of masked implementation, which can be classified into *compute-oriented* (e.g., [TKS10, KS20, GGM+21, MP21, CKK+22, CB23, LT23, KLS+23]) and *data-oriented* (e.g., [GMPP20, CPW24]) two categories. According to definitions in [CPW24], the compute-oriented ISE means *"[in compute-oriented ISEs] software indicates that the micro-architecture should execute masking-specific computation on masking-specific data"*, and the data-oriented ISE means *"[in data-oriented ISEs] software indicates that the micro-architecture should execute generic computation on masking-specific data"*. Notably, the data-oriented ISE is currently less explored than the compute-oriented one.

1.1 Contributions

To address the challenges in software implementations of Boolean masking, we use code-based masking to mask the Boolean shares, and then present a data-oriented ISE to facilitate efficient private computations within the code-based masked domain. Figure 1 provides an overview of our concept. On the left side of the figure, we depict the process of encoding secret x into the corresponding $d + 1$ code-based sharings, we call them *protected-Boolean shares* of the secret x: First, the secret x is encoded into $d + 1$ Boolean shares using a Boolean encoder, and then each Boolean share is further encoded into the corresponding protected-Boolean share by a code-based encoder. The right side of the figure shows the computation in the masked domain, taking the execution of software-based ISW multiplication as an example. During this process, hardware-based code-based masking is used to resist micro-architectural leakage. We refer to the software-based ISW multiplication process as the *outer masked operations* and the hardware-based code-based masking as the *inner masked operations*. Throughout the process, Boolean shares remain invisible to software, while protected-Boolean shares are visible. During inner masked multiplication or addition, the two protected-Boolean shares are decoded into their respective Boolean shares, followed by performing multiplication or addition on the Boolean shares, and finally encoding the resulting value to obtain the protected-Boolean share. Specifically, our contributions are threefold:

1. Design of a data-oriented ISE. Our ISE includes two classes of instructions, namely 1) computation and 2) pseudorandom generator (PRG) management. Aligned with the design philosophy of data-oriented ISE, one computation instruction performs a single bit-manipulation operation in the code-based masked domain, which is trivially constructed in hardware (by decoding the sharing, computing and encoding the result). Although this trivial construction ensures probing security for only input/output code-based sharings not intermediates (since hardware computations in the code-based masked domain do not have probing security), it is sufficient to secure the data stored in the general-purpose registers or memory. Consequently, it can prevent the transitional leakage in registers and increase the noise level. It is important to emphasize that in our design, the primary sources of noise are: 1) the use of code-based masking and the introduction of randomness; and 2) the use of hardware instead of software, as hardware often introduces higher levels of noise.

2. Implementation of the ISE. Based on a 32-bit RISC-V Ibex[2] core, we demonstrate that our ISE can be efficiently implemented by simply integrating two (code-based) decoders, one encoder, four multiplexers, and four demultiplexers. Besides, we also integrate a *leakage-resilient pseudorandom generator* to produce the necessary random bits, which are essential for the masked operations in both code-based and Boolean masking. To further mitigate combinational leakage, such as switching wires leakage, we integrate the register gating technique described in [GHP$^+$21] into our implementation. We also provide a formal verification of our ISE using Coco [GHP$^+$21, HB21].

3. Practical validation of the ISE implementation. The efficiency evaluation shows that the hardware overhead of our ISE is only 8% compared to the base core (vanilla Ibex). On the other hand, the security evaluation based on first-order ISW multiplication and bit-sliced AES S-box demonstrates that the security order is guaranteed on the core extended with our ISE, while the significant leakage is detected on the base core. Furthermore, our approach significantly reduces the signal-to-noise ratio (SNR) of each share, decreasing it to just 2% of the original SNR on the base core.

[1] Note that the term "secret" in this context differs from the secret mentioned later in the context. For example, in Figure 2, the "secret" refers to one of the $d + 1$ Boolean shares, rather than the secret x.

[2] https://github.com/lowRISC/ibex

1.2 Related works

First, we recall two works related to data-oriented ISE for masking. Gao et al. [GMPP20] present a special fence instruction, which prevents transitional leakage by flushing the micro-architectural resources specified by a CSR. By setting the value of this CSR, users can target different resources, which offers a high flexibility. This instruction is tightly integrated with the processor design. Cheng et al. [CPW24] focus on the transitional leakage stemming from architectural and micro-architectural overwriting. They equip a certain set of masking-related instructions with a special “hint”: the said instructions remain the same functionality, but the hint informs the micro-architecture to flush the destination resource before the actual writing of a result. Both approaches operate at the micro-architecture level, e.g., they propose to flush micro-architectural resources. In contrast, our approach considers the data level by encoding Boolean shares. In addition to eliminating implementation flaws, our approach significantly increases the noise in side-channel leakage, which is a critical requirement for the effectiveness of masking.

Besides, Arsath et al. [AGBR20] propose a SCA-resistant microprocessor design called PARAM, which utilizes obfuscation techniques to reduce or eliminate leakage. Before loading data from off-chip memory to the processor, PARAM employs a 4-round Feistel obfuscation function with a secret key to obfuscate (or encrypt) the data, thereby minimizing its associated leakage in the datapath (e.g., data cache, register file). Our ISE design bears some similarities to that of Arsath et al. For example, we also propose to obfuscate the data (i.e., Boolean shares in our case) in hardware. However, we emphasize that our concept is fundamentally different. The goal of our ISE is only to secure the software Boolean masking, allowing for a much simpler obfuscation approach that requires no key—specifically, i.e., a usage of linear code. In this respect, the hardware overhead is considerably small compared to that in Arsath et al.'s work, but also the approach can be verified more comprehensively through both formal verification and practical evaluation.

1.3 Organization

The paper is organized as follows. Section 2 presents various background information, namely RISC-V, notation, the challenges in practical implementation of Boolean masking, the benefits of code-based masking and stateful pseudorandom generator. Section 3 provides the details of our ISE design, where we introduce the design principles and define the functionality and encoding of custom instructions. In Section 4, we analyze our design, identify the requirements on processor and software implementation sides, and demonstrate that our design can address the challenges described in Section 2.3. Using the open-source 32-bit RISC-V Ibex core as the base core, in Section 5 we illustrate the prototype implementation of our ISE and the formal verification of the security. In Section 6 we evaluate our ISE implementation with regard to both efficiency and security, after which we finally draw the conclusion in Section 7.

2 Background

2.1 RISC-V

RISC-V is an ISA based on the RISC design principles, and has received massive interests from both academia and industry since its inception. RISC-V is provided under royalty-free open-source licenses, allowing anyone to use and implement it freely for any purpose. An important feature of RISC-V is that it adopts a modular design: the ISA, designed with a compact base integer instruction set, can be supplemented with standard and/or custom instruction set extensions. In this paper, our focus is the 32-bit integer RISC-V base ISA, namely RV32I. RISC-V uses the XLEN to denote the word size of the architecture, which in our case is XLEN=32.

2.2 Notation

The Boolean operators are represented in the following way: $\neg$ for NOT, $\wedge$ for AND, $\vee$ for OR, and $\oplus$ for XOR. In addition, $x \leftarrow y$ denotes the assignment of the value y to x. About bit extraction, $[x]_y$ extracts the lower y bits of x. The notation $x \parallel y$ means the concatenation of x and y. Related to the masking, we use $\langle x_0, x_1, \ldots, x_d \rangle$ to denote the *Boolean shares* of x. Related to the (micro-)architecture, GPR[i], where $0 \leq i < r$, denotes the i-th, w-bit entry in the r-entry general-purpose register files. Since we focus on RV32I, the parameters $w = \text{XLEN} = 32$ and $r = 32$ are instantiated with concrete values, and GPR[0] is always 0, meaning that any read from GPR[0] returns zero and any write to GPR[0] is ignored.

2.3 Implementation of Boolean Masking

Boolean masking is an effective and widely-used countermeasure against power side-channel attacks, in which each sensitive variable is divided into multiple shares. These shares are carefully manipulated to ensure that the original sensitive variable is not exposed during computation, thereby reducing the risk of side-channel leakage.

However, the Boolean masking has several implementation weaknesses. For example, knowing one bit of each share is sufficient to reveal one bit of the secret. This limitation makes it challenging to increase the noise level, and the software code must be carefully designed to avoid the known implementation flaws caused by architecture leakage, which can damage the independent assumption. These implementation flaws are generally categorized into two classes: transitional leakage and combinational leakage.

Transitional leakage. It is the leakage caused by the transition in a storage element such as register or memory, and leaks the change of stored value in consecutive cycles, with typical examples including overwrite leakage and operand leakage. Note that it covers all registers in a microprocessor, including general-purpose registers, pipeline registers, and etc. Balasch et al. [BGG+14] discuss how transitional leakage reduces the security of a masked implementation.

Combinational leakage. It is the leakage occurred in the combinational circuits and also includes the leakage caused by glitches. Balasch et al. [BGG+14] discuss also how glitches can reduce the security of a masked implementation. Marshall et al. [MPW21] provide the (simple) examples of the non-glitching and glitching cases. The possible types are listed below.

- **Glitchy register read**. It can cause the leakage of the bits (including their combinations) in all general-purpose registers, regardless of the source or destination operands in the instructions.
- **Always-active computation units**. It continuously leaks all values, including the glitchy ones, which are connected to the ALU. In the case of at most two operands, if an implementation can prevent the glitchy register read, then the "always-active computation units" only leak the combination of the two operands.
- **Bitwise interaction leakage**. In some implementations of cryptographic algorithms, data processing involves bitwise interaction operations such as shifts and rotations, where a single bit occurred in the computation may depend on multiple bits from the operators. This type of leakage exposes all bits in a register.

- **Switching wires leakage**. The selection of operands from general-purpose registers is typically implemented using a multiplexer tree. When two secret shares appear at the output of a multiplexer, an attacker may observe leakage. For example, assuming that the secret shares are stored in registers `x1` and `x2`, reading register `x3` in the first cycle and `x4` in the second cycle causes the 5th bit of the read address to switch from 1 to 0. An attacker observes leakage on the output wire of the first L0 multiplexer, which switches from `x1` to `x2`. This type of leakage is first discussed by Gigerl et al. [GHP+21], where the authors propose using register gating (RG) as a solution. However, apart from the multiplexer tree, leakage from the switching wires can also occur within the ALU, which cannot be resolved by register gating.

We omit the coupling wire leakage which originates from coupling capacitors between adjacent wires [CBG+17, Dho21]. Addressing this type of leakage necessitates specific consideration during the routing process, which is beyond the scope of our work. It is still an interesting open problem that whether it can be captured with abstract models [CS21].

2.4 Code-based Masking

Code-based masking demonstrates significant advantages in resisting side-channel attacks. The main advantage brought from the generalization is its robustness against the implementation flaws. Wang et al. [WGY+22] have shown that the more complex algebraic structure of code-based masking makes it more robust to the transitional leakage. Additionally, it has been demonstrated that the code-based masking reduces information leakage in low-noise conditions, and may increase the "statistical security order" of an implementation (with linear leakages).

We consider the code-based masking over $\mathbb{F}_2$. That is, it encodes an m-bit secret into an n-bit sharing comprising n shares, with each bit of this sharing representing one share. For an n-bit variable a and an invertible bit matrix A of size $n \times n$. For an m-bit secret x, code-based masking defines the encoder $\mathsf{Enc}(x)$, which randomly encodes x into a code-based sharing $\hat{x}$ as follows: $\mathsf{Enc}(x) = A \cdot (r \parallel x)$, where $\cdot$ represents matrix multiplication with the left operand treated as a vertical bit vector and the n-bit variable $r \parallel x$ is a concatenation of an $(n-m)$-bit uniformly random variable r and x. The decoder $\mathsf{Dec}(\hat{x})$ is then defined as $\mathsf{Dec}(\hat{x}) = [A^{-1} \cdot \hat{x}]_m$, where A^{-1} is the inverse of A, and $[a]_m$ extracts the lower m bits of $A^{-1} \cdot \hat{x}$.

However, the code-based masking incurs larger implementation overheads in software, compared to Boolean masking. This is primarily because its masked operations are not directly supported by microprocessors, which significantly reduces the practical efficiency of code-based masking. As discussed in [GR17], this limitation poses a major challenge to the adoption of such masking techniques in real-world systems.

2.5 (Stateful) Pseudorandom Generator

As mentioned in Section 1, we utilize a leakage-resilient PRG. To formalize, we adopt a definition that is adapted from the stateful PRG proposed by Bellare and Yee [BY03].

Definition 1 (Stateful pseudorandom generator [BY03])**.** A stateful pseudorandom generator is a pair of algorithms $(\mathsf{Setup}, \mathsf{Request})$, where Setup is a probabilistic algorithm which takes a seed *seed* as input and initializes an internal state S, $\mathsf{Request}$ is a deterministic algorithm which computes and outputs y (that is expected to be pseudorandom) from the internal state S and updates S.

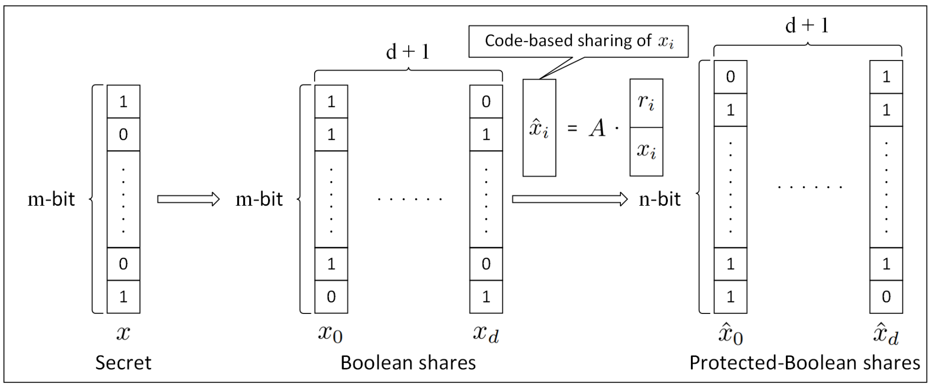


Figure 2: The process of splitting the secret x randomly into $d+1$ protected-Boolean shares, such that $x = x_0 \oplus x_1 \oplus \cdots \oplus x_d = \mathsf{Dec}(\hat{x}_0) \oplus \mathsf{Dec}(\hat{x}_1) \oplus \cdots \oplus \mathsf{Dec}(\hat{x}_d)$.

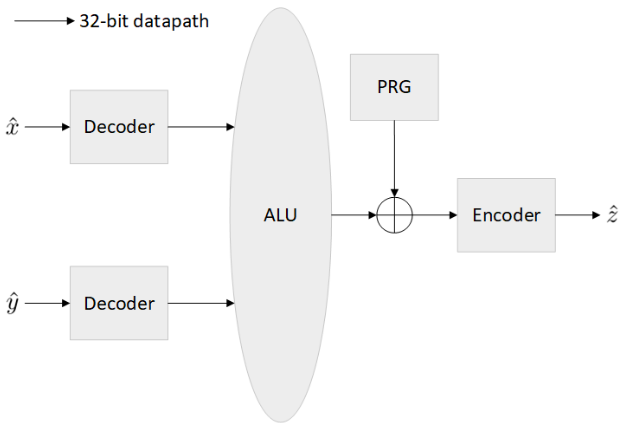


Figure 3: The (added) components and their connections used to support code-based masking.

3 Design

We present a robust yet efficient approach to address the challenges faced by software implementations of Boolean masking, by combining the code-based masking and an associated self-designed ISE. Conceptually, given the Boolean shares $\langle x_0, x_1, \ldots, x_d \rangle$ of the secret x, each share x_i where $i \in \{0, \ldots, d\}$ is randomly encoded into a sharing $\hat{x}_i \overset{\text{def}}{=} \mathsf{Enc}(x_i)$ with a code-based encoder $\mathsf{Enc}()$, ensuring that $x = \mathsf{Dec}(\hat{x}_0) \oplus \mathsf{Dec}(\hat{x}_1) \oplus \cdots \oplus \mathsf{Dec}(\hat{x}_d)$ with a code-based decoder $\mathsf{Dec}()$. As $\hat{x}_i$ where $i \in \{0, \ldots, d\}$ is the code-based sharing of the Boolean share x_i, we call it *protected-Boolean share* of the secret x hereafter in the paper, and Figure 2 illustrates the associated process. The inherent implementation weaknesses of Boolean masking can thus be addressed by manipulating the protected-Boolean shares instead of Boolean shares. Our ISE facilitates various operations between protected-Boolean shares by providing: 1) a series of computation instructions to support different bit-manipulations required by the code-based masking and 2) three instructions to manage a PRG to provide sufficient randomness.

Microarchitecture: components to support code-based masking. As per the description of our approach, several additional components are required, for example, encoder, decoder, and PRG. Figure 3 illustrates the related design that connects all of them. We assume $\hat{x}$, $\hat{y}$ are two code-based sharings (which can be regarded as protected-Boolean shares of two secret variables respectively), and we aim to perform an operation to get a result $\hat{z}$ which is also a code-based sharing. In Figure 3, $\hat{x}$ and $\hat{y}$ first go through the decoders individually to get their un-shared correspondences (which can be regarded as Boolean shares of two secret variables respectively). The un-shared variables are then computed by the ALU to get a result, which is finally encoded to obtain $\hat{z}$. To prevent transitional leakage in decoders, the decoders are activated only when executing CBM instructions. When CBM instructions are not being executed, the decoders remain disabled. In addition, we introduce a PRG to generate random numbers, and before the final encoding, the computation result from ALU can be optionally XORed with a random number thus to refresh the randomness in $\hat{z}$. Note that the associated computation is not required to be secure in the probing model, since the code-based masking is purposed for only 1) preventing the transitional leakage in registers and 2) increasing the noise level of Boolean shares.

Instruction functionality: computation in code-based masking. For the design of computation instructions, we take the same consideration and strategy described in [CPW24, Section 4.1]: *"[Gadgets] are implemented using a (short) sequence of bit-wise logical and shift instructions. As such, the goal of this instruction class is to provide a minimal set of such instructions to support the implementation of a maximal set of gadgets"*. Similarly, our computation instructions can serve as a secure version of the corresponding RV32I instructions (including `and[i]`, `or[i]`, `xor[i]`, `sll[i]`, `srl[i]`). In detail, a single register-register instruction performs an operation in a fashion of $\mathsf{GPR}[\texttt{rd}] = \mathsf{Enc}(\mathsf{Dec}(\mathsf{GPR}[\texttt{rs1}]) \odot \mathsf{Dec}(\mathsf{GPR}[\texttt{rs2}]))$, where $\odot$ is a bitwise logical or a shift operator. In the case of register-immediate instruction, it performs an operation $\mathsf{GPR}[\texttt{rd}] = \mathsf{Enc}(\mathsf{Dec}(\mathsf{GPR}[\texttt{rs1}]) \odot \texttt{imm})$, namely the second operand is used straightforwardly without decoding. In addition, to refresh a protected-Boolean share, one can XOR/OR the share with $\mathsf{GPR}[0]$,using `cbm.xor`/`cbm.or`.

Instruction functionality: PRG management. Given a PRG is added, we need a dedicated instruction to manage the PRG, e.g., to manually/automatically generate the pseudo-random numbers. In detail, we expect the following four PRG operations[3] to be available on the software side:

- **Operation #0:** resetting PRG with the current seed.
- **Operation #1:** manually generating a new random number.
- **Operation #2:** disabling the automatic generation of random numbers.
- **Operation #3:** enabling the automatic generation of random numbers.

Note that the operation #1 is only effective when the automatic generation of random numbers is disabled. Besides, the data transfer between (the state of) the PRG and the register file is needed, e.g., to refresh the PRG seed, to obtain a pseudo-random number for subsequent use, etc. Therefore, two associated instructions are added for two directions.

Instruction encoding. The detail of instruction encoding for our ISE is shown in Figure 4. Our ISE is called *CBM*, the acronym for *Code-Based Masking*, in a way that the name of all the custom instructions is with a prefix `cbm`, e.g., `cbm.and`. We obey the wider RISC-V design principles, e.g., we use only standard R-type and I-type formats for our custom instructions. In the encoding of computation instructions, 2 MSbs are used by a 2-bit

[3]The operations are numbered from 0 to align with the encoding of PRG management instruction.

Computation instructions

Instruction		31–30	29–25	24–20	19–15	14–12	11–7	6–2	1–0
`cbm.and rd, rs1, rs2, es`	:	es	00000	rs2	rs1	000	rd	00010	11
`cbm.or rd, rs1, rs2, es`	:	es	00000	rs2	rs1	001	rd	00010	11
`cbm.xor rd, rs1, rs2, es`	:	es	00000	rs2	rs1	010	rd	00010	11
`cbm.sll rd, rs1, rs2, es`	:	es	00000	rs2	rs1	011	rd	00010	11
`cbm.srl rd, rs1, rs2, es`	:	es	00000	rs2	rs1	100	rd	00010	11
`cbm.andi rd, rs1, imm, es`	:	es	imm		rs1	000	rd	01010	11
`cbm.ori rd, rs1, imm, es`	:	es	imm		rs1	001	rd	01010	11
`cbm.xori rd, rs1, imm, es`	:	es	imm		rs1	010	rd	01010	11
`cbm.slli rd, rs1, imm, es`	:	es	imm		rs1	011	rd	01010	11
`cbm.srli rd, rs1, imm, es`	:	es	imm		rs1	100	rd	01010	11

PRG management instructions

Instruction		31–27	26–25	24–20	19–15	14–12	11–7	6–2	1–0
`cbm.prg imm`	:	00000	imm	00000	00000	000	00000	10110	11
`cbm.s2r rd, imm`	:	00000	imm	00000	00000	001	rd	10110	11
`cbm.r2s rs1, imm`	:	00000	imm	00000	rs1	010	00000	10110	11

Figure 4: The encoding for instructions in our CBM ISE.

immediate `es` (meaning the Encoding Selector). In brief, the higher (resp. lower) bit of `es` selects whether the result generated by ALU will be encoded (resp. XORed) with the encoder (resp. a random number from PRG), and more detail is described in Section 5.2. As for `cbm.prg`, the 2-bit immediate selects a corresponding PRG management operation to perform, e.g., `imm = 0` indicates a PRG operation #0 is to be executed. Finally, an instruction `cbm.s2r` moves the data from PRG to a general-purpose register, while `cbm.r2s` does the opposite. The 2-bit immediate is used to select a specific part of the state of PRG, i.e., 32 bits from $32 \cdot \texttt{imm}$ to $32 \cdot \texttt{imm} + 31$.

4 Security Analysis and Discussions

The purpose of employing code-based masking is to prevent transitional leakage in the registers, thereby simplifying the prevention of implementation flaws. Section 3 has shown that, to support the code-based masked operations, it inserts combinational decoder/encoder circuits at the input/output of the ALU, however, this is not sufficient to prevent all known implementation flaws.

Processor modeling and overview of analysis. As shown in Figure 5, we model the processor as a combination of the ALU, general-purpose registers (GPRs), memory, and

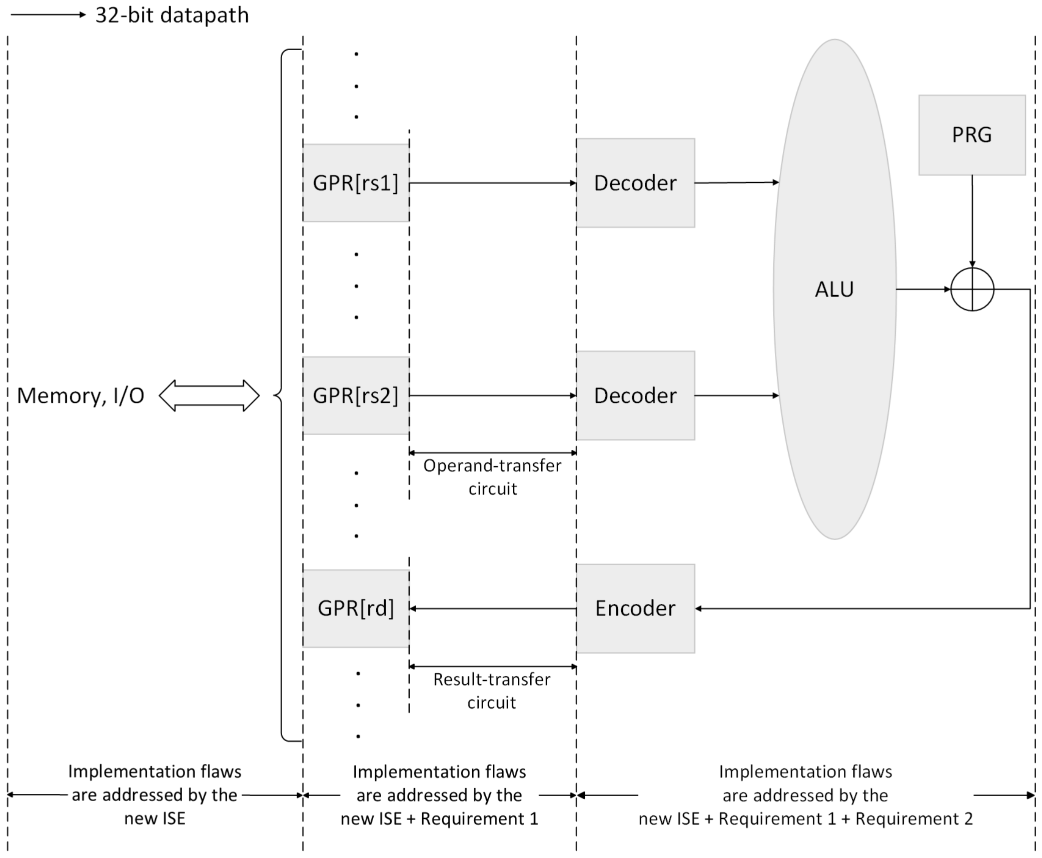

32-bit datapath
Memory, I/O
GPR[rs1]
GPR[rs2]
GPR[rd]
Decoder
Decoder
Encoder
ALU
PRG
Operand-transfer circuit
Result-transfer circuit
Implementation flaws are addressed by the new ISE
Implementation flaws are addressed by the new ISE + Requirement 1
Implementation flaws are addressed by the new ISE + Requirement 1 + Requirement 2

Figure 5: Processor modeling and overview of our analysis.

I/O. We consider the RISC-V base instruction set, where each instruction contains at most two operands, and the result is stored in a register. To implement our ISE, encoder and decoders are placed at the output and input of the ALU, respectively. The circuit connecting the general-purpose registers to the decoder is referred to as the *"operand-transfer circuit"*. This operand-transfer circuit may include the register-selection logic and several pipeline registers. Unlike the operand-transfer circuit, the circuit connecting the encoder to the general-purpose registers (referred as the *"result-transfer circuit"*) mainly consists of pipeline registers and does not involve any bitwise operations on the data. Similarly, neither memory nor I/O involves bitwise operations on the data. By encoding the Boolean shares, our ISE effectively addresses the implementation flaws in GPRs, memory and I/O. In this section, we describe the auxiliary requirements from processor and software implementation sides to address the implementation flaws in operand-transfer circuit, encoder/decoder and ALU, followed by the related security analysis.

4.1 Requirements of the Implementation

4.1.1 Preventing Flaws in the Operand-transfer and Result-transfer Circuit (hardware)

Recall that, in our approach, although the registers hold protected-Boolean shares instead of Boolean shares, the Boolean shares appear in the combinational circuits. Therefore, it is important that in pipeline registers there should be no combination of bits from general-purpose registers.

Furthermore, as illustrated in Section 3, the main purpose of code-based masking is to mitigate implementation flaws within (pipeline) registers. This underscores the necessity for additional efforts in averting such flaws within the combinational circuits. Specifically, it is imperative to avoid the intermingling of bit combinations across different general-purpose registers. To achieve this, additional attention must be directed towards the implementation of the operand-transfer circuit.

In the following, we commence by delineating a specific type of wire within the operand-transfer circuit that relates to the protected-Boolean shares.

Definition 2 (Sensitive wire)**.** A wire in the operand-transfer circuit is sensitive if it depends on the bits that appear in the general-purpose registers.

Then, the requirement of the operand-transfer circuit can be inferred from the characteristics of the sensitive wire.

Requirement 1. For each gate in the operand-transfer circuit, at most one input is sensitive.

It is pertinent to acknowledge that the Requirement 1 is feasible across numerous processor cores. In the context of the Ibex core, which is under consideration for implementation and practical evaluation, we advocate the adoption of the register gating technique proposed in [GHP+21]. This method entails storing the one-hot encoded enable signal within the register, with the combinatorial logic responsible for interpreting the register.

To better illustrate the rationale, we present an example as follows. Assuming x and y are bits from different general-purpose registers, there should exist a multiplexer in the operand-transfer circuit with a selection signal b to select x or y, and the output during the read operation is $z = xb \vee yb'$, where b and b' are stable complementary one-hot encoded selection signals, xb and yb' are two logical AND operations, and $\vee$ represents a logical OR operation. Requirement 1 ensures that each gate in the operand-transfer circuit has at most one input dependent on general-purpose registers. For the AND gate xb or yb', only the input x or y depends on general-purpose registers. Meanwhile, due to the complementary nature of b and b', at most one of the OR inputs is non-zero at any time. This ensures that only one input of the OR gate depends on general-purpose registers.

4.1.2 Preventing Switching Wires Leakage in Encoder/Decoder and ALU (software)

In addition to the (pipelined) registers, combinational circuits may also be susceptible to transitional leakage. This phenomenon occurs because the power consumption of a gate is influenced by changes in its output value, which is concluded as the switching wires discussed in Section 2.3. To address this, we propose the following requirements for using the new instructions on the software side.

Transitional leakage can also occur in the combinational circuits of the ALU. This may lead to the leakage of bitwise combinations information of the input operand a, which we denote as $g(a)$. Consequently, when the input operand changes across consecutive clock cycles, information about the bitwise combinations of these operands may be leaked.

For instance, consider two protected-Boolean shares of secret x, denoted as $\hat{x}_0$ and $\hat{x}_1$. The always-active computation units in the ALU, due to their complex structure (such as the non-linear structure in multiplication unit, etc.), may leak $g(\hat{x}_0) \oplus g(\hat{x}_1)$ if $\hat{x}_0$ and $\hat{x}_1$ are accessed in consecutive instructions, even if these instructions are not related to multiplication. Although this leakage may seem secure since it only reveals information about the bitwise combinations between protected-Boolean shares, there is a possibility that $g(\hat{x}_0)$ and $g(\hat{x}_1)$ may equal certain bits of the Boolean shares x_0 and x_1, which are Boolean shares of secret x, thereby failing to ensure security.

To address this, we propose a strict requirement in Requirement 2.a. This requirement can be easily met by inserting `nop` instructions to ensure that related sensitive data are not processed in consecutive cycles, thereby reducing leakage risks.

In practical software implementations of Boolean masking, due to the extremely low probability of the aforementioned issues occurring and the presence of noise further reducing the risk of leakage, we propose a relatively loose requirement in Requirement 2.b. This requirement does not imply that any two consecutive CBM instructions are disallowed; instead it means that accessing two or more related sensitive data in two consecutive CBM instructions is impermissible. This requirement can be met by appropriately arranging the execution order of instructions with out-of-order execution disabled. For example, in d-th order ISW multiplication, which requires $(d+1)^2$ `cbm.and` operations and more than $(d+1)^2$ `xor` operations, interleaving `cbm.and` and `xor` instructions is sufficient to fulfill this requirement.

Requirement 2.

a (Strict). Related sensitive data, such as shares of the same secret, etc. must not be accessed within two consecutive instructions.

b (Loose). Related sensitive data, such as shares of the same secret, etc. must not be accessed within two consecutive CBM instructions.

In practical software implementations of Boolean masking, it is sufficient to meet Requirement 2.b (Loose), while for scenarios with higher security requirements (e.g., formal verification in Section 5.5) it is necessary to satisfy Requirement 2.a (Strict).

4.1.3 Increasing the Noise Level of Leakage (hardware)

In addition to preventing implementation flaws, our approach also aims to elevate the noise level of leakage. This is achieved by ensuring that both the (pipeline) registers and memory solely retain protected-Boolean shares, ensured by Requirement 1. Furthermore, the augmentation of the noise level hinges on the fulfillment of the subsequent requirement.

Requirement 3. The SNR of combinational circuits should be notably lower than that of registers and memory.

This requirement applies to most circuits because combinational circuits only exhibit instantaneous power consumption, without maintaining a state over time. Consequently, the frequency and amplitude of power variations are relatively low. In contrast, registers and memory retain data states, leading to more pronounced power fluctuations during data reads and writes, which results in a higher SNR.

4.2 Security Analysis

In this subsection, we demonstrate how the implementation flaws discussed in Section 2.3 can be mitigated within the CBM instructions by ensuring the fulfillment of Requirement 1 and 2.

Combinational leakage naturally occurs in the combinational circuits. To delve into the analysis of this phenomenon, we initially establish a corollary of Requirement 1 through the lemma presented below.

Lemma 1. Requirement 1 guarantees that for every cycle, each wire within the operand-transfer circuit depends to at most one bit of the general-purpose register.

Proof. We employ proof by contradiction, initially supposing the existence of a wire, denoted as w, which connects to multiple bits within the protected-Boolean shares. We can see that w cannot function as an output wire of a general-purpose register, given that protected-Boolean shares are typically housed within such registers. Consequently, we separate our analysis into two cases below.

1. If this wire is an output of a pipeline register, then its value is equivalent to the input wire (referred to as w') of the pipeline register in the preceding clock cycle.

2. If this wire is the output of a gate G, according to Requirement 1, at most one input of G is associated with the bits appeared in the general-purpose registers. Then, we further separate our analysis into two cases below.

   (a) If none of the inputs of G are associated with the bits appeared in the general-purpose registers, then w is similarly not associated with the bits within these registers, thus contradicting the initial assumption.

   (b) If one of the inputs of G, denoted as w', corresponds to the bits in the general-purpose registers, then w' should correspond to multiple bits within the code-based sharing.

For the cases 1 and 2(b), we proceed to examine the wire w'. This process of analysis can iteratively continue until we reach a stage where the current wire under examination either serves as the output of a general-purpose register, or it bears no relevance to any bit within the general-purpose registers, thus contradicting the initial assumption. □

By Lemma 1, for every cycle, each wire in the operand-transfer circuit corresponds to only one bit of the protected-Boolean shares. Besides, since result-transfer circuit does not involve any bitwise operations on the data, each wire within the operand-transfer circuit corresponds to at most one bit of the encoder output. This directly eliminates the possibility of implementation flaws in operand-transfer and result-transfer circuits caused by combinational leakage except for the switching wires.

For the switching wires in the operand-transfer (resp., result-transfer) circuit, as each wire corresponds to at most one bit of a general-purpose register (resp., the encoder output), when a wire switches its value from a to b, the transitional leakage actually leaks $a \oplus b$. As we illustrated in Section 3, for any two independently masked shares (say, $\hat{x}_1$ and $\hat{x}_2$), any bit in $\hat{x}_1 \oplus \hat{x}_2$ is independent of x_1 and x_2. This ensures the security order in the presence of leaking wires in the operand-transfer circuit.

For the switching wires in the ALU and encoder/decoder, Requirement 2.a (Strict) ensures that sensitive data, such as shares of the same secret, cannot be accessed by the ALU and encoder/decoder in consecutive cycles. This prevents the wires in the ALU and encoder/decoder from holding related bits during consecutive operations, thereby reducing the risk of leakage and ensuring the security order is maintained even in the presence of potentially leaking wires. For Requirement 2.b (Loose), it ensures that related sensitive data should not be accessed by consecutive CBM instructions in the ALU and encoder/decoder, thereby reducing the risk of leakage while still allowing more flexibility compared to Requirement 2.a (Strict). This looser requirement not only ensures security but also offers greater practicality in real-world applications.

To analyze implementation flaws caused by transitional leakage, we need to demonstrate that each pipeline register in the operand-transfer (resp., result-transfer) does not combine bits from general-purpose registers (resp., the encoder output). This indicates that each pipeline register stores only the value from a single bit of a protected-Boolean share, thereby ensuring the security order in the presence of transitional leakage.

Remark. Formal proofs within the transition-glitch leakage model [HHB+24, FGDP+18] are known to provide strong and reliable guarantee of the security order in hardware. However, this paper does not include such rigorous proofs, as they require extensive preliminaries on the transition-glitch leakage model, which could render the paper overly cumbersome. In this respect, our analysis cannot make strong claims regarding the security order. We leave the formal proof/discussion within the transition-glitch leakage model for future work.

5 Implementation and Formal Verification

5.1 Base core: Ibex

General overview. Ibex is an open-source 32-bit RISC-V core designed for use in embedded environments. The micro-architectural design of Ibex core is highly configurable. Specifically, Ibex core supports either integer RV32I [RIS19, Chapter 2] or embedded RV32E [RIS19, Chapter 4] base ISA, which can be supplemented by the standard M (Multiplication) [RIS19, Chapter 7], C (Compressed) [RIS19, Chapter 16], or B (Bit-manipulation) [RIS19, Chapter 17] extensions. The core can be configured to use either a 2- or 3-stage pipeline, by enabling or disabling a dedicated Write Back (WB) stage independent of an Instruction Fetch (IF) stage and an Instruction Decode and Execute (ID/EX) stage. Additionally, different types of multipliers (e.g., single-cycle, fast/slow multi-cycle) and register files (e.g., flip-flop-based, FPGA, latch-based) are also available.

Specific configuration. We develop the prototype ISE implementation and perform associated experiments on Ibex Demo System[4], which consists of the Ibex core and some peripherals (e.g., UART, GPIO, SPI). We select to use the flip-flop-based register file, with all other options set to the default configuration. The base ISA is RV32IMC; 2-stage pipeline is used; a fast multi-cycle multiplier is used; PMP and the instruction cache are disabled.

5.2 Encoder and Decoder

Given an *abstract* design of our approach has been provided in Figure 3, here Figure 6 attempts to explain the corresponding *specific* implementation: an additional circuit is inserted to wrap the ALU, containing mainly an encoder, two decoders, four multiplexers, and four demultiplexers. The workflow is straightforward. There are four pairs of multiplexers and demultiplexers, and each pair shares the same selector signal. Before entering the ALU, the operand `operand_a` (resp. `operand_b`) is processed by an associated decoder, depending on a selector signal `cbm_opa_sel` (resp. `cbm_opb_sel`) whose value will be 1 if a CBM instruction is currently being executed. In the other direction, the 16 MSbs of the ALU output can be optionally XORed with 16 random bits from PRG, selected by `cbm_enc_sel[0]` whose value is from the lower bit of `es` in a CBM instruction (see, e.g., Figure 4). This allows the subsequent final encoding, which is selected by `cbm_enc_sel[1]` (determined by the higher bit of `es`), to be dynamically refreshed (i.e., adding noise), thereby enhancing the flexibility of the overall implementation.

Matrix multiplication. The input/output of the linear encoder/decoder is 32 bits, and, more specifically, an element in $\mathbb{F}_{2^4}^8$. Using $\boldsymbol{A}$ to denote an 8×8 MDS matrix over $\mathbb{F}_{2^4}$ and given $x \in \mathbb{F}_{2^4}^8$, the encoder and decoder are formally defined as:

$$\mathsf{Enc}(x) = \boldsymbol{A} \times x,$$

[4] https://github.com/lowRISC/ibex-demo-system

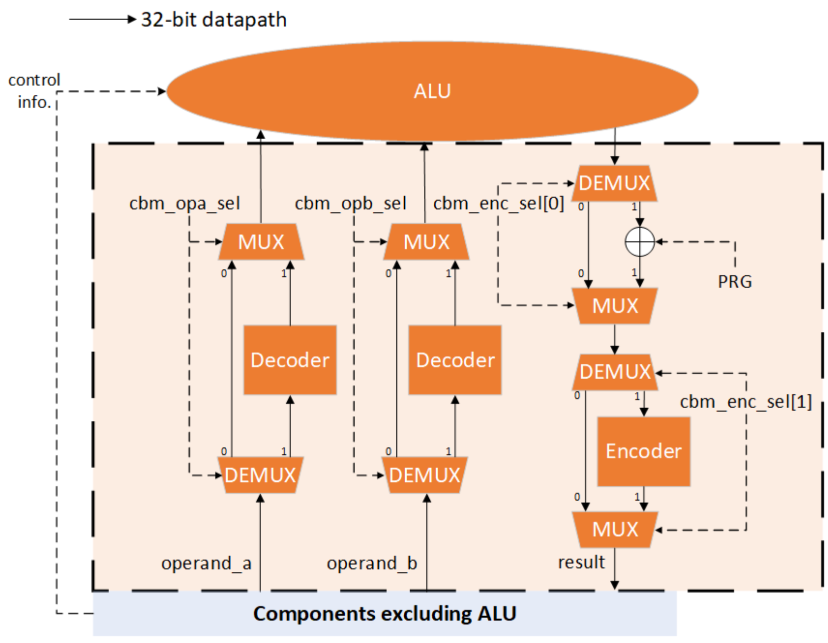


Figure 6: The overview of the implementation of our approach.

$$\mathsf{Dec}\,(x) = \boldsymbol{A}^{-1} \times x.$$

Concretely, we use the involutory MDS matrix in [SKOP15], namely

$$\boldsymbol{A} = \boldsymbol{A}^{-1} = \begin{pmatrix} 2 & 3 & 4 & 12 & 5 & 10 & 8 & 15 \\ 3 & 2 & 12 & 4 & 10 & 5 & 15 & 8 \\ 4 & 12 & 2 & 3 & 8 & 15 & 5 & 10 \\ 12 & 4 & 3 & 2 & 15 & 8 & 10 & 5 \\ 5 & 10 & 8 & 15 & 2 & 3 & 4 & 12 \\ 10 & 5 & 15 & 8 & 3 & 2 & 12 & 4 \\ 8 & 15 & 5 & 10 & 4 & 12 & 2 & 3 \\ 15 & 8 & 10 & 5 & 12 & 4 & 3 & 2 \end{pmatrix}.$$

The problem of optimizing the implementation of matrix multiplication can be translated into "how to compute a set of linear expressions with a minimum number of linear operations". This is known as the Shortest Linear Program (SLP) and is essentially an NP-hard problem. We use the framework[5] proposed by Lin et al. [LXZZ21] to optimize the matrix multiplication implementation. Specifically, using an example where given a matrix

$$\boldsymbol{B} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

and $t = (t_0, t_1, t_2, t_3)$ in $\mathbb{F}_2^4$, when performing the multiplication $r = \boldsymbol{B} \times t$ over $\mathbb{F}_2$, we can compute in sequence: 1) $r_1 = t_1 \oplus t_2$; 2) $r_0 = t_0 \oplus r_1$; 3) $r_3 = r_1 \oplus t_3$; 4) $r_2 = r_1 \oplus t_3$. With this approach, only four XOR operations are needed for the entire matrix multiplication process.

Although finding the minimum number of XORs for matrix multiplication is an NP-hard problem, after a careful optimization, we have found a computation path that requires

[5] https://github.com/DaLin10512/framework

```
Variable: internal state $S \in \{0,1\}^b$

procedure Setup($seed$)        // seed length $|seed| = \kappa$
    1. $S \leftarrow \mathsf{P}([0]_{b-\kappa} \| seed)$
procedure Request()
    1. $y \leftarrow \mathsf{msb}_r(S)$, $S \leftarrow \mathsf{P}(S)$
    2. return $y$
```

Figure 7: The duplex-based leakage-resilient PRG $\mathsf{DPRG}^{\mathsf{P}}$, where $\mathsf{P} : \{0,1\}^b \to \{0,1\}^b$ and $S \in \{0,1\}^b$.

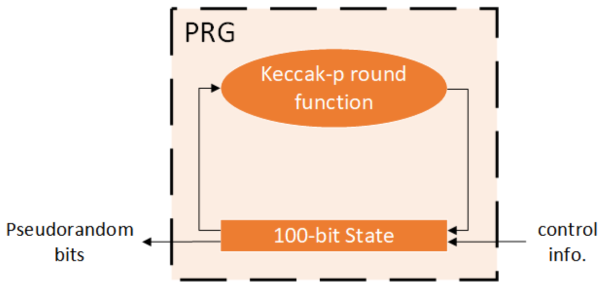


Figure 8: PRG module.

only 182 XORs for the matrix multiplication in encoder/decoder, refer to Appendix A. We note that this implementation results in a relatively larger reduction in the maximal core frequency (but relatively lower area overhead). Therefore, we also develop an implementation optimized for frequency by using the tool of Liu et al. [LWF+22], which requires 251 XORs, refer to Appendix B. To make it clear, we refer to the former implementation (182 XORs) as "area-optimized", while the latter (251 XORs) as "frequency-optimized". In this paper, our primary focus is the area-optimized vision, meaning the evaluation results in Section 6 reflect the implementation using the area-optimized matrix multiplication. For reference, we also present the evaluation results related to the frequency-optimized version in Appendix B.

5.3 Pseudorandom Generator

As mentioned in the Section 1, we utilize a leakage-resilient PRG $\mathsf{DPRG}^{\mathsf{P}}$. Concretely, $\mathsf{DPRG}^{\mathsf{P}}$ is the standard duplex construction-based PRG built upon a keyless permutation $\mathsf{P} : \{0,1\}^n \to \{0,1\}^n$, as described in Figure 7 (following the formalism of Definition 1). To justify its theoretical soundness, in Appendix C we formally prove the leakage-resilience of $\mathsf{DPRG}^{\mathsf{P}}$ under the assumptions that leakages are non-invertible and P is a public random permutation. More specifically, given the outputs and leakages of q executions of $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{Request}$, the next non-leaky execution of $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{Request}$ still yields an r-bit pseudorandom string. The parameter q can be determined by the side-channel adversary and can be large. Therefore, $\mathsf{DPRG}^{\mathsf{P}}$ is sound in the *continuous leakage model.* We instantiate the permutation P with Keccak-p[100] permutation. As shown in Figure 8, the round function of Keccak-p is executed once per clock cycle, thereby updating the PRG state.

This design aims to provide an efficient and scalable pseudorandom generation scheme, suitable for a diverse range of security applications. However, if random numbers are requested too frequently before a sufficient number of hash function rounds have been called, the generated bits may not strictly adhere to a uniform distribution. Despite this, we propose the following viewpoints:

- For the randomness used in software, the frequency of randomness calls should be low enough to sufficiently generate random numbers. We validate this intuition by conducting evaluation based on ISW multiplication and bit-sliced AES S-Box. The results in Section 6.2 confirm the secure order, indicating that the randomness is adequate for practical use.

- For the randomness used to refresh the hardware code-based masking, since the leakage caused by implementation flaws (e.g., transitional leakage) is highly noisy, the security should still be maintained in practice even if the acquisition of random numbers is very frequent.

In conclusion, although frequent acquisition of random numbers may pose the risk of inhomogeneity, in practice, our new design still provides a highly secure random number generation capability to meet security requirements.

5.4 Register Gating

As discussed in [GHP$^+$21, Chapter 3], register gating addresses the problems of wire switching in the multiplexer tree, unintended reads, and glitchy signals by placing an AND gate after each register. This mechanism works by using the register value as the first input to the AND gate, and a 32-bit one-hot encoded read address as the second input. Consequently, the entire multiplexer tree can be replaced by a simpler tree of OR gates [GHP$^+$21, Figure 2]. This not only prevents the leakage of secret share data during register transitions but also ensures that reads are only enabled when necessary, reducing the likelihood of unintended reads. To avoid glitchy signals, the one-hot signal is computed during the IF stage and stored in an intermediate register to ensure its stability when reaching the ID/EX stage. In our implementation, to balance efficiency and security, we only apply the gating technique to both read ports, but not to the write port. As stated in Section 4.1.1, the Requirement 1 can be fulfilled in Ibex core with register gating.

5.5 Formal Verification

To verify the security of the design and requirements, we conduct a formal verification. Coco [GHP$^+$21, HB21] is used as the verification tool. Coco is a co-design and co-verification tool for masked software implementations on CPUs, capable of verifying leakage security of masked assembly code down to the gate level. It can detect any CPU design flaws that may reduce the protection order of masked software implementations.

We employ first-order ISW multiplication on our ISE-extended core, separately using CBM instructions and standard RV32I instructions. Our ISE-extended core implementation fulfills Requirements 1 (because of the register gating) and 3, and our software implementation of the ISW multiplication fulfills Requirement 2.a (Strict). The result from the Coco shows that no leakage is present when using our CBM instructions, whereas leakage is captured when using standard RV32I instructions, as detailed in Appendix D. A practical security evaluation concerning the ISW multiplication fulfilling Requirement 2.b (loose) can be found in Section 6.2.

Table 1: Comparison of area, stemming from FPGA synthesis of the base core plus implementation of the CBM.

	Registers	LUTs
Base core	2362 (1.00×)	3951 (1.00×)
Base core + CBM (excl. RG and PRG)	2369 (1.00×)	3964 (1.00×)
Base core + CBM (excl. PRG)	2425 (1.03×)	4626 (1.17×)
Base core + CBM (excl. RG)	2476 (1.05×)	4250 (1.08×)
Base core + CBM	2530 (1.07×)	4795 (1.21×)

Table 2: Comparison of area, stemming from ASIC synthesis of the base core plus implementation of the CBM.

	Cells	GE
Base core	19445 (1.00×)	30627 (1.00×)
Base core + CBM (excl. RG and PRG)	19593 (1.01×)	31407 (1.03×)
Base core + CBM (excl. PRG)	19546 (1.01×)	31604 (1.03×)
Base core + CBM (excl. RG)	20730 (1.07×)	33616 (1.10×)
Base core + CBM	20306 (1.05×)	33057 (1.08×)

6 Evaluation in Practice

To evaluate our ISE, we use a NewAE ChipWhisperer CW305[6] board, which hosts a Xilinx Artix-7 FPGA (model XC7A100T2FTG256). Xilinx Vivado 2022.2 is used to synthesize the implementations of the base core and the core extended with our ISE. Once programmed into the FPGA, the core is provided with an 8 MHz clock frequency via the CW305 Phase Locked Loop (PLL). To measure power consumption, we connect a NewAE ChipWhisperer CW1173 (or ChipWhisperer-Lite)[7] board to the CW305 (via its X4 pin). This setup amplifies the measured power signal by Low-Noise Amplifiers (LNA) on both CW305 (fixed to 20 dB gain) and CW1173 (configured to 20 dB gain) boards.

6.1 Efficiency

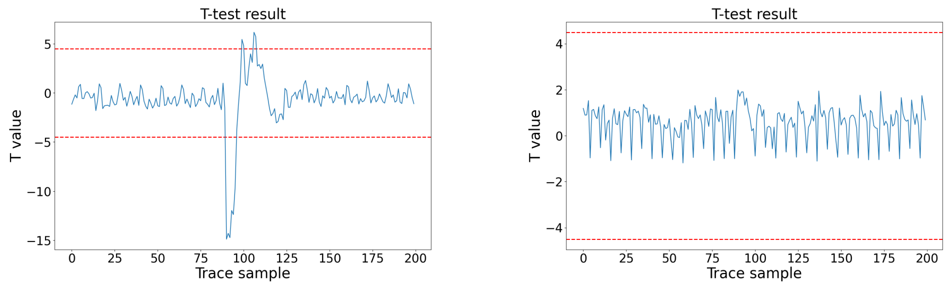
Area. As shown in Table 1, we summarize the area overhead of the ISE, presenting the hardware overhead for both the base core and the ISE-extended core. We take an incremental approach to illustrate the overhead introduced by each component. By analyzing these data, we identify two main sources of additional area overhead in our design. The first is register gating, which increases the overall resource usage. The second is the PRG module, in which the primary contributors to the overhead are the state registers and the associated 1-round Keccak-$p[100]$ permutation. These components determine the resources and area requirements of our design. By default, our implementation includes register gating, which contributes to a 2% increase in registers and a 13% increase in LUTs compared to the hardware implementation without register gating. Overall, our ISE introduces up to 7% more registers and 21% more LUTs compared to the base core.

In addition to the FPGA synthesis, we also synthesize our implementation using ASIC technology. For ASIC evaluation, we adhere to the guidelines provided in the official Ibex documentation[8], selecting the 45nm Nangate45 open library and using the Yosys tool for synthesis. The evaluation result demonstrates that our implementation achieves excellent efficiency, with only a 5% increase in cells and an 8% increase in GE (per Table 2).

[6] https://rtfm.newae.com/Targets/CW305ArtixFPGA
[7] https://rtfm.newae.com/Capture/ChipWhisperer-Lite
[8] https://github.com/lowRISC/ibex/tree/master/syn

Table 3: Comparison of frequency, stemming from FPGA/ASIC synthesis of the base core plus implementation of the CBM.

	FPGA	ASIC
Base core	72.03 MHz (1.00×)	347.22 MHz (1.00×)
Base core + CBM	62.43 MHz (0.87×)	289.85 MHz (0.83×)



(a) Base core: RV32I instructions (b) ISE-extended core: CBM instructions

Figure 9: T-test results of experiments regarding overwriting on base core and the ISE-extended core. Each experiment uses 10 million traces.

Frequency. Through the FPGA and ASIC synthesis, we evaluate the impact of our implementation on the maximum achievable frequency. As shown in Table 3, the implementation of CBM reduces the maximum frequency. Specifically, the maximum frequency obtained from FPGA synthesis decreases from 72.03 MHz to 62.43 MHz, representing 87% of the base core. Similarly, in ASIC synthesis, the maximum frequency decreases from 347.22 MHz to 289.85 MHz, which is 83% of the original maximum frequency. These results highlight the trade-off between security and the associated performance cost in terms of frequency.

Latency. The execution latency of each instruction in CBM is 1 clock cycle.

6.2 Security

To validate the effectiveness of our proposed solution in addressing the issues discussed in Section 2.3, we perform a series of empirical tests.

Empritical tests regarding overwriting. To verify that our design can address the transitional leakage in registers, we conduct the experiments of executing an overwrite operation on both the base core and the core extended with CBM. We use the `mv` instruction to overwrite a general-purpose register and apply the T-test [Wel47] to assess the presence of leakage. Specifically, we overwrite $\mathsf{Enc}(y)$ with $\mathsf{Enc}(x)$ using the "`mv`" instruction and collect 10 million traces during the process. These traces are classified into two groups based on the distance between x and y (i.e., $x \oplus y$): 1) one group in which x and y are random, i.e., the distance is random, and 2) the other group in which the distance between x and y is fixed. Finally, we perform a T-test analysis of these traces. These tests are crucial for determining whether our extended core provides the expected security enhancements over the base core. Figure 9 shows the T-test results of 10 million traces during the overwrite process. The significant peaks above 4.5 are present on the base core, but these peaks no longer exist on our ISE-extended core.

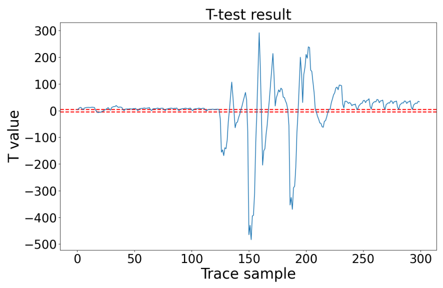
```
# Using standard RV32I instructions
# rX0, rX1:     hold $x_0$, $x_1$
# rY0, rY1:     hold $y_0$, $y_1$
# rRM:          holds random mask
# rZ0, rZ1: to hold $z_0$, $z_1$

and     rZ0, rX0, rY0
xor     rZ0, rZ0, rRM
and     rZ1, rX0, rY1
xor     rZ1, rZ1, rRM
and     tmp, rX1, rY0
xor     rZ1, rZ1, tmp
and     tmp, rX1, rY1
xor     rZ1, rZ1, tmp
```

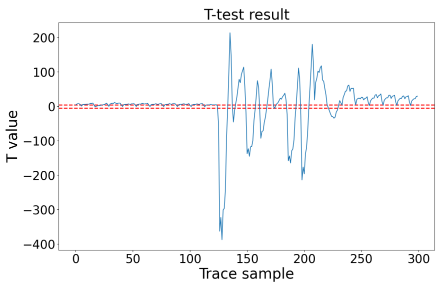
```
# Using CBM instructions
# rX0,rX1:     hold $\hat{x}_0$, $\hat{x}_1$
# rY0,rY1:     hold $\hat{y}_0$, $\hat{y}_1$
# rRM:         holds random mask
# rZ0,rZ1: to hold $\hat{z}_0$, $\hat{z}_1$

cbm.and     rZ0, rX0, rY0, 3
xor         rZ0, rZ0, rRM
cbm.and     rZ1, rX0, rY1, 3
xor         rZ1, rZ1, rRM
cbm.and     tmp, rX1, rY0, 3
xor         rZ1, rZ1, tmp
cbm.and     tmp, rX1, rY1, 3
xor         rZ1, rZ1, tmp
```

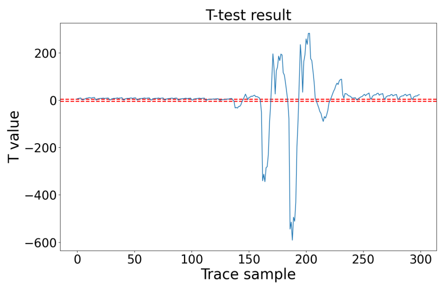
Figure 10: Micro-benchmarks of first-order ISW multiplication.



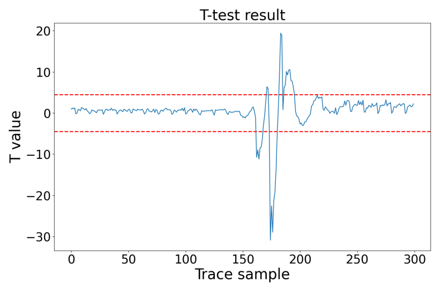
(a) Base core: RV32I instructions



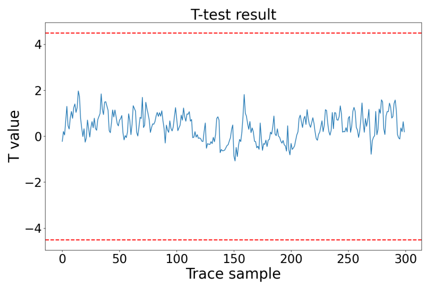
(b) ISE-extended core (excl. RG): RV32I instructions



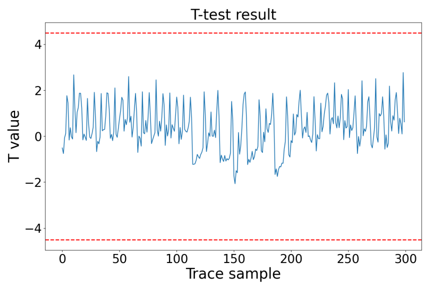
(c) ISE-extended core: RV32I instructions



(d) ISE-extended core: CBM instructions without Requirement 2 satisfied

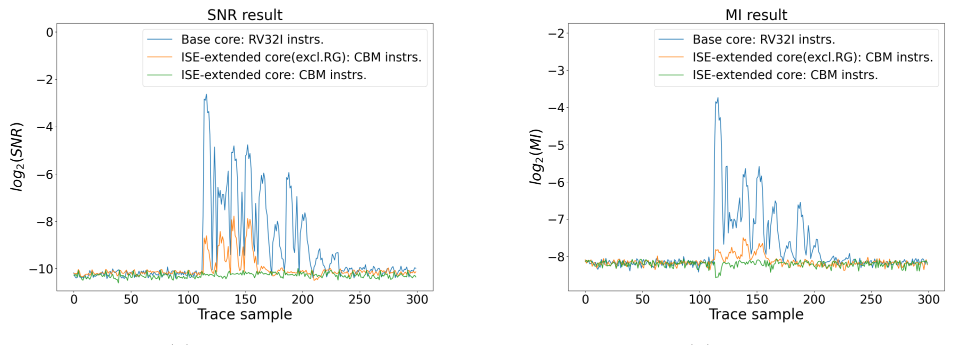


(e) ISE-extended core (excl. RG): CBM instructions



(f) ISE-extended core: CBM instructions

Figure 11: T-test results of experiments regarding ISW multiplication on base core and the ISE-extended core, using standard RV32I instructions or CBM instructions. Each experiment uses 10 million traces.



(a) SNR result (b) MI result

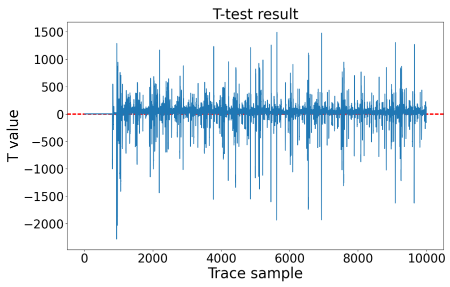
Figure 12: SNR and MI results of experiments regarding ISW multiplication on base core and the ISE-extended core, using standard RV32I instructions or CBM instructions. Each experiment uses 300 thousand traces.

Empritical tests regarding first-order ISW multiplication. To comprehensively evaluate the presence of leakage, we utilize several statistical techniques, including the T-test, Signal-to-Noise Ratio (SNR), and Mutual Information (MI) [Sha48], while conducting comparative tests under various conditions. Figure 10 illustrates the micro-benchmarks used in our testing. The code on the left-hand side uses only standard RV32I instructions, while the code on the right-hand side uses our CBM instructions. Figure 11 and Figure 12 show the results of our empirical tests. For the T-test analysis, we use 10 million traces per test, while for the SNR and MI analyses, we use 300 thousand traces. As expected, significant peaks in the T-test values above 4.5 are observed when using standard RV32I instructions, indicating the presence of potential leakage. However, when using our CBM instructions, these peaks are eliminated regardless of whether register gating is used, demonstrating the effectiveness of CBM instructions in reducing side-channel leakage. Furthermore, by comparing Figures 11(d) and 11(f), we demonstrate the necessity of Requirement 2.b. It is a bit surprised that the ISE-extended core without register gating also exhibits no leakage. However, we still believe that Requirement 1 is necessary, since we latter show that ISE-extended core exhibits less SNR when register gating is excluded.

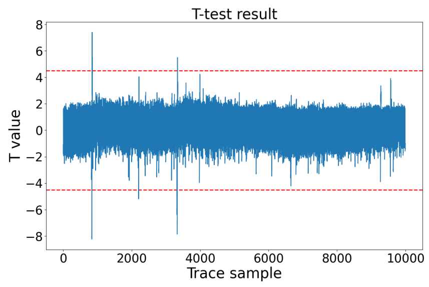
For the computation of SNR and MI, we first randomly generate four shares x_0, x_1, y_0 and y_1, with their [15 : 8] bits fixed to 0 to simplify analysis during the data collection phase. These shares serve as input values, and both the shares and the corresponding traces are saved for later processing. During the SNR calculation, one of the four shares is selected as the signal data, and the traces are categorized into 256 classes based on the selected share. The average trace of each class is then incrementally calculated. Using these averages, the noise mean and variance are determined by computing the deviations of individual trace samples from their respective class averages. Finally, the SNR is calculated as the ratio of the variance of the signal to the variance of the noise. For MI calculation, one of the four shares is selected as the input variable. The MI is then computed using the `mutual_info_score` function from the Python library `scikit-learn`[9]. This function calculates MI by determining the joint probability distribution of the given input and comparing it with its marginal distributions.

The results of SNR and MI analyses reveal significant differences between using standard RV32I instructions and CBM instructions. When using RV32I instructions, prominent peaks are observed, indicating areas of high signal compared to noise and potential leakage. In contrast, these peaks are significantly reduced or even completely eliminated when CBM

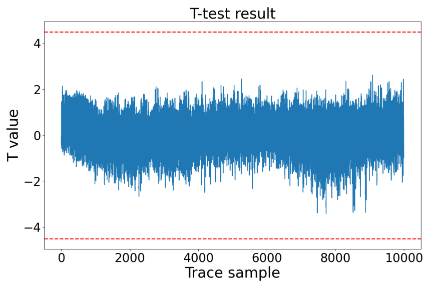
[9] https://scikit-learn.org/



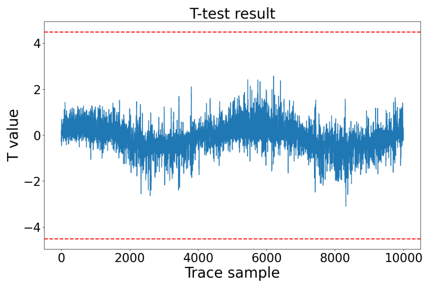
(a) Base core: RV32I instructions



(b) ISE-extended core: CBM instructions without Requirement 2.b satisfied



(c) ISE-extended core (excl. RG): CBM instructions



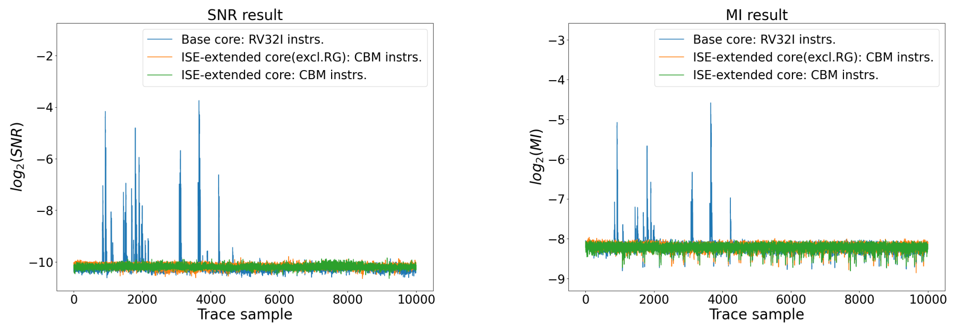
(d) ISE-extended core: CBM instructions

Figure 13: T-test results of experiments regarding bit-sliced AES S-box on base core and the ISE-extended core, using standard RV32I instructions or CBM instructions. Each experiment uses 10 million traces.

instructions are used, demonstrating the effectiveness of the ISE in mitigating sensitive information leakage. Furthermore, a comparison of the results with and without register gating shows that enabling register gating further reduces SNR and MI, offering enhanced leakage suppression. It is important to note that the data presented in the Figure 12 have been logarithmically transformed.

Empritical tests regarding first-order bit-sliced AES S-Box. To further validate the effectiveness of our ISE, we extend our evaluation to include the bit-sliced AES S-Box proposed by Goudarzi and Rivain [GR17]. For this test, we use the first-order bit-sliced AES S-Box and perform a series of analyses, including the T-test, SNR, and MI, to assess potential side-channel leakage. Figure 13 and Figure 14 show the results of these tests. Similar to our previous evaluation, we use 10 million traces per test for the T-test analysis, while the SNR and MI analyses use 300 thousand traces. During these tests, the T-test results for standard RV32I instructions exhibit significant peaks again, indicating the presence of side-channel leakage. In contrast, applying our ISE effectively eliminates these peaks, aligning with the observations from earlier experiments. Additionally, the comparison between Figure 13(b) and 13(d) further corroborates the necessity of Requirement 2.b. This recurring pattern across different test cases highlights the robustness and reliability of our ISE in reducing side-channel leakage. Moreover, the reduction of SNR and MI peaks indicates that our ISE not only eliminates potential leakage points but also reduces overall information leakage, thereby enhancing security. Our ISE consistently reduces these leakages, regardless of the underlying cryptographic implementation, showcasing its versatility.

These consistent results across different methods and cryptographic components demon-



(a) SNR result (b) MI result

Figure 14: SNR and MI results of experiments regarding bit-sliced AES S-box on base core and the ISE-extended core, using standard RV32I instructions or CBM instructions. Each experiment uses 300 thousand traces.

strate the robustness of our proposed approach. They strongly indicate that our ISE can be an important addition to secure embedded systems, especially in environments where preventing side-channel attacks is crucial.

7 Conclusion and Future Works

7.1 Summary

This paper focuses on designing an ISE for the RISC-V architecture, aimed at improving resistance against side-channel attacks while balancing efficiency and security. Specifically, we introduce an ISE named CBM, which employs code-based masking to safeguard the shares of software Boolean masking. By masking Boolean shares with code-based masking, we ensure that the Boolean shares are not directly stored in registers or memory. This approach addresses two key issues: 1) it mitigates implementation flaws caused by transitional leakage, and 2) it significantly reduces the SNR of Boolean shares. Nevertheless, it's crucial to recognize that our design serves as a foundational principle that necessitates further extensions depending on the specific base core used in practical implementations. For example, when utilizing the Ibex core, we integrate the register gating technique to counteract other implementation flaws associated with combinational leakage. Finally, our evaluation results indicate that our design and implementation achieve a favorable balance between efficiency and security, offering a practical solution for enhancing side-channel security in the RISC-V architecture.

7.2 Future Work

Increasing computational capacity. In Section 5, we propose a scheme where 16 bits of the 32-bit result from the ALU are XORed with a 16-bit random number generated by the PRG module. This scheme aims to increase the noise level and reduce the leakage of sensitive information, thereby improving resistance against side-channel attacks. Although this scheme has demonstrated initial effectiveness, the limitation in resource efficiency cannot be ignored due to the fact that only 16 bits of the 32-bit register are used, potentially leading to underutilization of the core's computational capacity. To address this limitation and further enhance robustness, future work could focus on developing a dynamic bit-selection mechanism for the XOR operation. By dynamically adjusting the number of bits to be XORed in response to varying security requirements, a more flexible and resource-efficient solution could be achieved. This would not only allow for a more precise control of the efficiency-security trade-off but also enable real-time adjustments in response to evolving threat landscapes. Additionally, exploring more advanced noise injection methods may also provide significant efficiency and security gains.

Fault detection. In addition to addressing resource efficiency, another important direction for future work involves the introduction of fault detection into the design. Given the critical role of fault detection in code-based masking schemes, we could split the 16-bit XOR operation into two parts: one part for randomness and the other for fault detection. The fault detection bits would be fixed and encoded into protected-Boolean shares. During operations, after decoding, these bits could be verified to detect any faults, ensuring the integrity of the masked values and providing protection against fault injection attacks. For instance, consider a 16-bit Boolean share x_i. Its corresponding protected Boolean share, with fault detection capability, can be represented as $\hat{x}'_i = A \cdot (1^8 \parallel r' \parallel x_i)$, where 1^8 consists of eight bits of 1's, and r' is an 8-bit random number. A fault detection component can then be added between the decoder and the ALU to check whether the higher 8 bits of the decoded value are all ones. Therefore, we view the exploration of the above design as a promising future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions. This work was supported by the Key R&D Program of Shandong Province, China (Grant No. 2024ZLGX05), National Key R&D Program of China (Grant Nos. 2023YFA1009500, 2021YFA1000600, 2024YFA1013000), the National Natural Science Foundation of China (Grant Nos. 62372273, 62032014, 62372274, U2336207), the Program of Taishan Young Scholars of the Shandong Province, Department of Science & Technology of Shandong Province (Grant No. SYS202201), Quan Cheng Laboratory (Grant Nos. QCLZD202301, QCLZD202306), the Luxembourg National Research Fund (FNR) via the CORE project ImPAKT (C21/IS/16221219/ImPAKT), and the Innovate UK via project 10065634 (SCHEME: Safety Critical Harsh Environment Micro-processing Evolution).

References

[AGBR20] Muhammad Arsath K.F., Vinod Ganesan, Rahul Bodduna, and Chester Rebeiro. PARAM: A microprocessor hardened for power side-channel attack resistance. In *Hardware Oriented Security and Trust (HOST)*, pages 23–34. IEEE, 2020. https://doi.org/10.1109/HOST45689.2020.9300263.

[BCC+14] Julien Bringer, Claude Carlet, Hervé Chabanne, Sylvain Guilley, and Houssem Maghrebi. Orthogonal direct sum masking: A smartcard friendly computation paradigm in a code, with builtin protection against side-channel and fault attacks. In *Workshop on Information Security Theory and Practices (WISTP)*, LNCS 8501, pages 40–56. Springer-Verlag, 2014. https://doi.org/10.1007/978-3-662-43826-8_4.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 9813, pages 23–39. Springer-Verlag, 2016. https://doi.org/10.1007/978-3-662-53140-2_2.

[BFGV12] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. Theory and practice of a leakage resilient masking scheme. In *Advances in Cryptology (ASIACRYPT)*, LNCS 7658, pages 758–775. Springer-Verlag, 2012. https://doi.org/10.1007/978-3-642-34961-4_45.

[BGG$^+$14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 8968, pages 64–81. Springer-Verlag, 2014. https://doi.org/10.1007/978-3-319-16763-3_5.

[BS20] Olivier Bronchain and François-Xavier Standaert. Side-channel countermeasures' dissection and the limits of closed source security evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):1–25, 2020. https://doi.org/10.13154/tches.v2020.i2.1-25.

[BY03] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In *Topics in Cryptology (CT-RSA)*, LNCS 2612, pages 1–18. Springer-Verlag, 2003. https://doi.org/10.1007/3-540-36563-X_1.

[CB23] Songqiao Cui and Josep Balasch. Efficient software masking of AES through instruction set extensions. In *Design, Automation & Test in Europe (DATE)*, pages 1–6, 2023. https://doi.org/10.23919/DATE56975.2023.10137150.

[CBG$^+$17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, pages 1–18, 2017. https://doi.org/10.1007/978-3-319-64647-3_1.

[CKK$^+$22] Piljoo Choi, Won Bae Kong, Ji-Hoon Kim, Mun-Kyu Lee, and Dong Kyue Kim. Architectural supports for block ciphers in a RISC CPU core by instruction overloading. *IEEE Transactions on Computers*, 71(11):2844–2857, 2022. https://doi.org/10.1109/TC.2021.3050515.

[CPW24] Hao Cheng, Daniel Page, and Weijia Wang. eLIMInate: a leakage-focused ISE for masked implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2024(2):329–358, 2024. https://doi.org/10.46586/tches.v2024.i2.329-358.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):136–158, 2021. https://doi.org/10.46586/tches.v2021.i2.136-158.

[DF12] Stefan Dziembowski and Sebastian Faust. Leakage-resilient circuits without computational assumptions. In *Theory of Cryptography Conference (TCC)*, LNCS 7194, pages 230–247. Springer-Verlag, 2012. https://doi.org/10.1007/978-3-642-28914-9_13.

[Dho21] Siemen Dhooghe. Analyzing masked ciphers against transition and coupling effects. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 201–223. Springer, 2021. https://doi.org/10.1007/978-3-030-92518-5_10.

[DM19] Christoph Dobraunig and Bart Mennink. Leakage resilience of the duplex construction. In *Advances in Cryptology (ASIACRYPT)*, LNCS 11923, pages 225–255. Springer-Verlag, 2019. https://doi.org/10.1007/978-3-030-34618-8_8.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *Foundations of Computer Science (FOCS)*, pages 293–302. IEEE Computer Society, 2008. https://doi.org/10.1109/FOCS.2008.56.

[FGDP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 89–120, 2018. https://doi.org/10.13154/tches.v2018.i3.89-120.

[GGM+21] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):283–325, 2021. https://doi.org/10.46586/tches.v2021.i4.283-325.

[GHP+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In *USENIX Security Symposium*, pages 1469–1468, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl.

[GMPP20] Si Gao, Ben Marshall, Dan Page, and Thinh Pham. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):73–98, 2020. https://doi.org/10.13154/tches.v2020.i2.73-98.

[GPPS20] Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Towards low-energy leakage-resistant authenticated encryption from the duplex sponge construction. *IACR Transactions on Symmetric Cryptology (ToSC)*, 2020(1):6–42, 2020. https://doi.org/10.13154/tosc.v2020.i1.6-42.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In *Advances in Cryptology (EUROCRYPT)*, LNCS 10210, pages 567–597. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-56620-7_20.

[HB21] Vedad Hadzic and Roderick Bloem. Cocoalma: A versatile masking verifier. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2021*, page 14, 2021. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_9.

[HHB+24] Johannes Haring, Vedad Hadži, Roderick Bloem, et al. Closing the gap: Leakage contracts for processors with transitions and glitches. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):110–132, 2024. https://doi.org/10.46586/tches.v2024.i4.110-132.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology (CRYPTO)*, LNCS 2729, pages 463–481. Springer-Verlag, 2003. https://doi.org/10.1007/978-3-540-45146-4_27.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, pages 388–397. Springer-Verlag, 1999. https://doi.org/10.1007/3-540-48405-1_25.

[KLS+23] Markus Krausz, Georg Land, Florian Stolz, Dennis Naujoks, Jan Richter-Brockmann, Tim Güneysu, and Lucie Kogelheide. To extend or not to extend: Agile masking instructions for PQC. Cryptology ePrint Archive, Paper 2023/1287, 2023. https://eprint.iacr.org/2023/1287.

[KS20] Pantea Kiaei and Patrick Schaumont. Domain-oriented masked instruction set architecture for RISC-V. Cryptology ePrint Archive, Report 2020/465, 2020. https://eprint.iacr.org/2020/465.

[LT23] Fabrice Lozachmeur and Arnaud Tisserand. A RISC-V instruction set extension for flexible hardware/software protection of cryptosystems masked at high orders. In *International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 360–364. IEEE, 2023. https://doi.org/10.1109/MWSCAS57524.2023.10405991.

[LWF+22] Qun Liu, Weijia Wang, Yanhong Fan, Lixuan Wu, Ling Sun, and Meiqin Wang. Towards low-latency implementation of linear layers. *Cryptology ePrint Archive*, 2022. https://eprint.iacr.org/2022/231.

[LXZZ21] Da Lin, Zejun Xiang, Xiangyong Zeng, and Shasha Zhang. A framework to optimize implementations of matrices. In *Topics in Cryptology (CT-RSA)*, LNCS 12704, pages 609–632. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-75539-3_25.

[Men23] Bart Mennink. Understanding the duplex and its security. *IACR Transactions on Symmetric Cryptology (ToSC)*, 2023(2):1–46, 2023. https://doi.org/10.46586/tosc.v2023.i2.1-46.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. https://doi.org/10.1007/978-0-387-38162-6.

[MP21] Ben Marshall and Dan Page. SME: Scalable masking extensions. Cryptology ePrint Archive, Report 2021/1416, 2021. https://eprint.iacr.org/2021/1416.

[MPW21] B. Marshall, D. Page, and J. Webb. MIRACLE: MIcRo-ArChitectural Leakage Evaluation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(1):175–220, 2021. https://doi.org/10.46586/tches.v2022.i1.175-220.

[Pie09] Krzysztof Pietrzak. A leakage-resilient mode of operation. In *Advances in Cryptology (EUROCRYPT)*, LNCS 5479, pages 462–482. Springer-Verlag, 2009. https://doi.org/10.1007/978-3-642-01001-9_27.

[RIS19] The RISC-V instruction set manual, Volume I: User-level ISA (version 20191213-base-ratified). Technical report, 2019. http://riscv.org/specifications.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x.

[SKOP15] Siang Meng Sim, Khoongming Khoo, Frédérique E. Oggier, and Thomas Peyrin. Lightweight MDS involution matrices. In *Fast Software Encryption (FSE)*, LNCS 9054, pages 471–493. Springer-Verlag, 2015. https://doi.org/10.1007/978-3-662-48116-5_23.

[TKS10] Stefan Tillich, Mario Kirschbaum, and Alexander Szekely. SCA-resistant embedded processors: The next generation. In *Annual Computer Security Applications Conference (ACSAC)*, pages 211–220, 2010. https://doi.org/10.1145/1920261.1920293.

[Wel47] Bernard L. Welch. The generalization of 'STUDENT'S' problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947. https://doi.org/10.1093/biomet/34.1-2.28.

[WGY$^+$22] Weijia Wang, Chun Guo, Yu Yu, Fanjie Ji, and Yang Su. Side-channel masking with common shares. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 290–329, 2022. https://doi.org/10.46586/tches.v2022.i3.290-329.

[WMCS20] Weijia Wang, Pierrick Méaux, Gatan Cassiers, and Franois Xavier Standaert. Efficient and private computations with code-based masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020:128–171, 2020. https://doi.org/10.13154/tches.v2020.i2.128-171.

[YSPY10] Yu Yu, François-Xavier Standaert, Olivier Pereira, and Moti Yung. Practical leakage-resilient pseudorandom generators. In *Computer and Communications Security (CCS)*, pages 141–151. ACM, 2010. https://doi.org/10.1145/1866307.1866324.

A Area-optimized Matrix Multiplication

Figure 15 illustrates the area-optimized matrix multiplication computation process for the encoder and decoder. The optimization of matrix multiplication translates to finding the minimum number of linear operations needed to compute a set of linear expressions. This is known as the Shortest Linear Program (SLP) problem, which is essentially NP-hard.

We utilize the framework[10] designed by Lin et al. [LXZZ21] to optimize the implementation of matrix multiplication, embedding several state-of-the-art algorithms, including Paar1, RPaar1, BP, RNBP, A1, and A2. The **Paar1 algorithm** enhances efficiency by decomposing a matrix into smaller submatrices, optimizing the linear operations of matrices, and improving processing speed for large-scale data. This approach is particularly effective in cryptography and matrix optimization. The **RPaar1 algorithm**, a randomized version of Paar1, introduces randomness to improve flexibility and adaptability by selecting operation paths randomly, which helps avoid local optimal solutions and yields better optimization for specific matrix structures. The **BP algorithm** simplifies subsequent optimization by employing two randomly generated substitution matrices to alter the original matrix structure, thereby reducing the complexity and time required for operations. This technique is particularly advantageous in data encryption and algorithm optimization. The **RNBP algorithm**, a randomized non-block localization optimization method, focuses on determining the optimal sequence of matrix operations. By randomly selecting operation paths and sequences, RNBP can identify the optimal solution within complex matrix structures, making it highly suitable for scenarios requiring efficient optimization. The **A1 and A2 algorithms** reduce the complexity of matrix computations by optimizing XOR operations, thereby significantly lowering computational costs, which is particularly beneficial for cryptographic algorithms and matrix operation optimization.

The framework includes two key functions: “Further Reduction” and “Iterative Reduction”, and supports both single-threaded and multi-threaded execution. By default, we use the multi-threaded mode with the “Iterative Reduction” function, which reads a matrix M and searches for its optimized implementation. Each time the framework is employed, an algorithm is randomly selected to find an optimized implementation of the given matrix. Eventually, using this framework, matrix multiplication requires only 182 XOR operations.

B Frequency-optimized matrix multiplication

B.1 Computation process

Figure 16 illustrates the frequency-optimized matrix multiplication computation process for the encoder and decoder. We utilize the framework[11] designed by Liu et al. [LWF+22] to optimize the implementation of matrix multiplication, which is a new low-latency framework to implement linear layers in lightweight cryptography. The proposed method employs a backward search strategy that iteratively splits the output bits until all input bits are present, thus optimizing the circuit depth and achieving low latency.

B.2 Efficiency

As shown in Table 4 and 5, in terms of hardware overhead, the implementation of CBM with frequency-optimized matrix multiplication shows a slight increase compared to the area-optimized version, but the overall resource overhead remains at a low level. Specifically, the frequency-optimized implementation only increases about 2% more LUTs and GEs than the area-optimized version.

[10]https://github.com/DaLin10512/framework
[11]https://github.com/QunLiu-sdu/Towards-Low-Latency-Implementation

$t[32] = t[6] \oplus t[5]$	$t[78] = t[52] \oplus t[70]$	$t[124] = t[32] \oplus t[15]$	$t[170] = t[138] \oplus t[96]$
$t[33] = t[17] \oplus t[22]$	$t[79] = t[70] \oplus t[65]$	$t[125] = t[15] \oplus t[72]$	$t[171] = t[134] \oplus t[141]$
$t[34] = t[10] \oplus t[25]$	$t[80] = t[78] \oplus t[62]$	$t[126] = t[107] \oplus t[30]$	$t[172] = t[160] \oplus t[99]$
$t[35] = t[20] \oplus t[21]$	$t[81] = t[25] \oplus t[72]$	$t[127] = t[3] \oplus t[89]$	$t[173] = t[149] \oplus t[116]$
$t[36] = t[26] \oplus t[29]$	$t[82] = t[34] \oplus t[13]$	$t[128] = t[30] \oplus t[125]$	$t[174] = t[168] \oplus t[93]$
$t[37] = t[31] \oplus t[33]$	$t[83] = t[13] \oplus t[67]$	$t[129] = t[125] \oplus t[89]$	$t[175] = t[96] \oplus t[172]$
$t[38] = t[36] \oplus t[9]$	$t[84] = t[67] \oplus t[54]$	$t[130] = t[94] \oplus t[105]$	$t[176] = t[175] \oplus t[162]$
$t[39] = t[23] \oplus t[30]$	$t[85] = t[84] \oplus t[63]$	$t[131] = t[105] \oplus t[82]$	$t[177] = t[109] \oplus t[159]$
$t[40] = t[37] \oplus t[24]$	$t[86] = t[79] \oplus t[14]$	$t[132] = t[100] \oplus t[111]$	$t[178] = t[93] \oplus t[155]$
$t[41] = t[12] \oplus t[35]$	$t[87] = t[86] \oplus t[77]$	$t[133] = t[111] \oplus t[123]$	$t[179] = t[169] \oplus t[170]$
$t[42] = t[11] \oplus t[2]$	$t[88] = t[77] \oplus t[66]$	$t[134] = t[123] \oplus t[89]$	$t[180] = t[159] \oplus t[166]$
$t[43] = t[8] \oplus t[16]$	$t[89] = t[50] \oplus t[60]$	$t[135] = t[72] \oplus t[38]$	$t[181] = t[170] \oplus t[147]$
$t[44] = t[42] \oplus t[43]$	$t[90] = t[88] \oplus t[49]$	$t[136] = t[128] \oplus t[120]$	$t[182] = t[141] \oplus t[163]$
$t[45] = t[41] \oplus t[39]$	$t[91] = t[66] \oplus t[46]$	$t[137] = t[120] \oplus t[71]$	$t[183] = t[116] \oplus t[161]$
$t[46] = t[24] \oplus t[0]$	$t[92] = t[49] \oplus t[59]$	$t[138] = t[129] \oplus t[82]$	$t[184] = t[161] \oplus t[179]$
$t[47] = t[40] \oplus t[39]$	$t[93] = t[91] \oplus t[44]$	$t[139] = t[89] \oplus t[116]$	$t[185] = t[147] \oplus t[156]$
$t[48] = t[4] \oplus t[14]$	$t[94] = t[92] \oplus t[54]$	$t[140] = t[102] \oplus t[80]$	$t[186] = t[157] \oplus t[167]$
$t[49] = t[22] \oplus t[15]$	$t[95] = t[46] \oplus t[54]$	$t[141] = t[113] \oplus t[140]$	$t[187] = t[182] \oplus t[185]$
$t[50] = t[47] \oplus t[32]$	$t[96] = t[54] \oplus t[18]$	$t[142] = t[140] \oplus t[104]$	$t[188] = t[115] \oplus t[178]$
$t[51] = t[28] \oplus t[29]$	$t[97] = t[18] \oplus t[62]$	$t[143] = t[118] \oplus t[124]$	$t[189] = t[183] \oplus t[176]$
$t[52] = t[29] \oplus t[48]$	$t[98] = t[97] \oplus t[44]$	$t[144] = t[135] \oplus t[71]$	$t[190] = t[133] \oplus t[142]$
$t[53] = t[48] \oplus t[27]$	$t[99] = t[90] \oplus t[98]$	$t[145] = t[71] \oplus t[127]$	$t[191] = t[142] \oplus t[167]$
$t[54] = t[16] \oplus t[52]$	$t[100] = t[98] \oplus t[56]$	$t[146] = t[124] \oplus t[87]$	$t[192] = t[99] \oplus t[174]$
$t[55] = t[53] \oplus t[35]$	$t[101] = t[62] \oplus t[69]$	$t[147] = t[104] \oplus t[127]$	$t[193] = t[172] \oplus t[184]$
$t[56] = t[1] \oplus t[32]$	$t[102] = t[69] \oplus t[14]$	$t[148] = t[127] \oplus t[80]$	$t[194] = t[163] \oplus t[158]$
$t[57] = t[19] \oplus t[35]$	$t[103] = t[14] \oplus t[73]$	$t[149] = t[112] \oplus t[145]$	$t[195] = t[174] \oplus t[193]$
$t[58] = t[57] \oplus t[34]$	$t[104] = t[73] \oplus t[56]$	$t[150] = t[119] \oplus t[145]$	$t[196] = t[158] \oplus t[154]$
$t[59] = t[55] \oplus t[2]$	$t[105] = t[56] \oplus t[65]$	$t[151] = t[87] \oplus t[117]$	$t[197] = t[167] \oplus t[194]$
$t[60] = t[58] \oplus t[39]$	$t[106] = t[65] \oplus t[44]$	$t[152] = t[137] \oplus t[121]$	$t[198] = t[194] \oplus t[188]$
$t[61] = t[39] \oplus t[5]$	$t[107] = t[44] \oplus t[81]$	$t[153] = t[117] \oplus t[109]$	$t[199] = t[188] \oplus t[180]$
$t[62] = t[51] \oplus t[45]$	$t[108] = t[101] \oplus t[81]$	$t[154] = t[152] \oplus t[139]$	$t[200] = t[155] \oplus t[173]$
$t[63] = t[9] \oplus t[56]$	$t[109] = t[108] \oplus t[74]$	$t[155] = t[132] \oplus t[131]$	$t[201] = t[180] \oplus t[184]$
$t[64] = t[27] \oplus t[18]$	$t[110] = t[83] \oplus t[72]$	$t[156] = t[80] \oplus t[144]$	$t[202] = t[173] \oplus t[181]$
$t[65] = t[2] \oplus t[45]$	$t[111] = t[103] \oplus t[110]$	$t[157] = t[121] \oplus t[133]$	$t[203] = t[136] \oplus t[166]$
$t[66] = t[64] \oplus t[38]$	$t[112] = t[81] \oplus t[110]$	$t[158] = t[153] \oplus t[145]$	$t[204] = t[171] \oplus t[164]$
$t[67] = t[61] \oplus t[38]$	$t[113] = t[110] \oplus t[59]$	$t[159] = t[139] \oplus t[130]$	$t[205] = t[162] \oplus t[177]$
$t[68] = t[35] \oplus t[50]$	$t[114] = t[59] \oplus t[74]$	$t[160] = t[145] \oplus t[148]$	$t[206] = t[131] \oplus t[151]$
$t[69] = t[68] \oplus t[43]$	$t[115] = t[114] \oplus t[30]$	$t[161] = t[150] \oplus t[144]$	$t[207] = t[193] \oplus t[154]$
$t[70] = t[7] \oplus t[21]$	$t[116] = t[74] \oplus t[60]$	$t[162] = t[144] \oplus t[126]$	$t[208] = t[192] \oplus t[143]$
$t[71] = t[33] \oplus t[5]$	$t[117] = t[106] \oplus t[60]$	$t[163] = t[130] \oplus t[38]$	$t[209] = t[178] \oplus t[166]$
$t[72] = t[45] \oplus t[18]$	$t[118] = t[60] \oplus t[63]$	$t[164] = t[126] \oplus t[138]$	$t[210] = t[176] \oplus t[187]$
$t[73] = t[5] \oplus t[21]$	$t[119] = t[95] \oplus t[76]$	$t[165] = t[38] \oplus t[82]$	$t[211] = t[156] \oplus t[186]$
$t[74] = t[21] \oplus t[66]$	$t[120] = t[76] \oplus t[32]$	$t[166] = t[82] \oplus t[134]$	$t[212] = t[203] \oplus t[189]$
$t[75] = t[43] \oplus t[0]$	$t[121] = t[85] \oplus t[3]$	$t[167] = t[165] \oplus t[93]$	$t[213] = t[143] \oplus t[197]$
$t[76] = t[75] \oplus t[49]$	$t[122] = t[63] \oplus t[112]$	$t[168] = t[146] \oplus t[96]$	
$t[77] = t[0] \oplus t[67]$	$t[123] = t[122] \oplus t[107]$	$t[169] = t[148] \oplus t[136]$	

Figure 15: Matrix multiplication computation process for encoder/decoder. In the figure, $t[0]$ to $t[31]$ denote 32-bit input, $\{ t[202], t[164], t[189], t[184], t[209], t[205], t[210], t[186], t[195], t[211], t[196], t[177], t[213], t[185], t[199], t[166], t[201], t[154], t[208], t[181], t[198], t[191], t[207], t[151], t[179], t[206], t[204], t[190], t[212], t[187], t[200], t[197]\}$ denote 32-bit output.

$t[32] = t[2] \oplus t[18]$
$t[33] = t[12] \oplus t[29]$
$t[34] = t[21] \oplus t[27]$
$t[35] = t[34] \oplus t[19]$
$t[36] = t[6] \oplus t[10]$
$t[37] = t[34] \oplus t[36]$
$t[38] = t[24] \oplus t[26]$
$t[39] = t[38] \oplus t[19]$
$t[40] = t[0] \oplus t[15]$
$t[41] = t[40] \oplus t[19]$
$t[42] = t[13] \oplus t[40]$
$t[43] = t[5] \oplus t[23]$
$t[44] = t[43] \oplus t[40]$
$t[45] = t[3] \oplus t[13]$
$t[46] = t[45] \oplus t[34]$
$t[47] = t[8] \oplus t[21]$
$t[48] = t[47] \oplus t[40]$
$t[49] = t[32] \oplus t[48]$
$t[50] = t[24] \oplus t[31]$
$t[51] = t[50] \oplus t[13]$
$t[52] = t[14] \oplus t[25]$
$t[53] = t[52] \oplus t[13]$
$t[54] = t[39] \oplus t[53]$
$t[55] = t[15] \oplus t[22]$
$t[56] = t[55] \oplus t[47]$
$t[57] = t[16] \oplus t[20]$
$t[58] = t[57] \oplus t[34]$
$t[59] = t[4] \oplus t[30]$
$t[60] = t[14] \oplus t[23]$
$t[61] = t[59] \oplus t[60]$
$t[62] = t[13] \oplus t[19]$
$t[63] = t[59] \oplus t[62]$
$t[64] = t[63] \oplus t[58]$
$t[65] = t[10] \oplus t[25]$
$t[66] = t[65] \oplus t[59]$
$t[67] = t[66] \oplus t[46]$
$t[68] = t[0] \oplus t[20]$
$t[69] = t[68] \oplus t[65]$
$t[70] = t[69] \oplus t[33]$
$t[71] = t[7] \oplus t[28]$
$t[72] = t[30] \oplus t[71]$
$t[73] = t[71] \oplus t[68]$
$t[74] = t[3] \oplus t[16]$
$t[75] = t[74] \oplus t[60]$
$t[76] = t[18] \oplus t[28]$
$t[77] = t[76] \oplus t[30]$
$t[78] = t[3] \oplus t[22]$
$t[79] = t[78] \oplus t[76]$
$t[80] = t[1] \oplus t[29]$
$t[81] = t[20] \oplus t[80]$
$t[82] = t[66] \oplus t[81]$
$t[83] = t[80] \oplus t[78]$
$t[84] = t[83] \oplus t[72]$
$t[85] = t[84] \oplus t[49]$
$t[86] = t[84] \oplus t[54]$
$t[87] = t[17] \oplus t[31]$
$t[88] = t[12] \oplus t[18]$
$t[89] = t[87] \oplus t[88]$
$t[90] = t[9] \oplus t[20]$
$t[91] = t[87] \oplus t[90]$
$t[92] = t[11] \oplus t[24]$
$t[93] = t[4] \oplus t[21]$
$t[94] = t[92] \oplus t[93]$
$t[95] = t[1] \oplus t[22]$
$t[96] = t[92] \oplus t[95]$
$t[97] = t[6] \oplus t[28]$
$t[98] = t[97] \oplus t[15]$
$t[99] = t[61] \oplus t[98]$
$t[100] = t[3] \oplus t[27]$
$t[101] = t[97] \oplus t[100]$
$t[102] = t[9] \oplus t[26]$
$t[103] = t[102] \oplus t[1]$
$t[104] = t[0] \oplus t[16]$
$t[105] = t[102] \oplus t[104]$
$t[106] = t[11] \oplus t[18]$
$t[107] = t[22] \oplus t[26]$
$t[108] = t[106] \oplus t[107]$
$t[109] = t[1] \oplus t[3]$
$t[110] = t[106] \oplus t[109]$
$t[111] = t[17] \oplus t[29]$
$t[112] = t[111] \oplus t[107]$
$t[113] = t[63] \oplus t[112]$
$t[114] = t[18] \oplus t[24]$
$t[115] = t[114] \oplus t[5]$
$t[116] = t[41] \oplus t[115]$
$t[117] = t[114] \oplus t[23]$
$t[118] = t[117] \oplus t[105]$
$t[119] = t[118] \oplus t[67]$
$t[120] = t[7] \oplus t[31]$
$t[121] = t[120] \oplus t[5]$
$t[122] = t[25] \oplus t[29]$
$t[123] = t[122] \oplus t[120]$
$t[124] = t[3] \oplus t[21]$
$t[125] = t[124] \oplus t[120]$
$t[126] = t[125] \oplus t[94]$
$t[127] = t[3] \oplus t[23]$
$t[128] = t[127] \oplus t[122]$
$t[129] = t[41] \oplus t[128]$
$t[130] = t[8] \oplus t[13]$
$t[131] = t[16] \oplus t[30]$
$t[132] = t[130] \oplus t[131]$
$t[133] = t[132] \oplus t[103]$
$t[134] = t[4] \oplus t[6]$
$t[135] = t[130] \oplus t[134]$
$t[136] = t[91] \oplus t[135]$
$t[137] = t[116] \oplus t[136]$
$t[138] = t[4] \oplus t[22]$
$t[139] = t[12] \oplus t[19]$
$t[140] = t[138] \oplus t[139]$
$t[141] = t[132] \oplus t[140]$
$t[142] = t[9] \oplus t[23]$
$t[143] = t[138] \oplus t[142]$
$t[144] = t[123] \oplus t[143]$
$t[145] = t[5] \oplus t[31]$
$t[146] = t[145] \oplus t[138]$
$t[147] = t[146] \oplus t[108]$
$t[148] = t[1] \oplus t[10]$
$t[149] = t[18] \oplus t[21]$
$t[150] = t[148] \oplus t[149]$
$t[151] = t[150] \oplus t[42]$
$t[152] = t[8] \oplus t[23]$
$t[153] = t[148] \oplus t[152]$
$t[154] = t[25] \oplus t[26]$
$t[155] = t[28] \oplus t[30]$
$t[156] = t[154] \oplus t[155]$
$t[157] = t[11] \oplus t[17]$
$t[158] = t[154] \oplus t[157]$
$t[159] = t[14] \oplus t[28]$
$t[160] = t[24] \oplus t[27]$
$t[161] = t[159] \oplus t[160]$
$t[162] = t[32] \oplus t[161]$
$t[163] = t[162] \oplus t[70]$
$t[164] = t[162] \oplus t[133]$
$t[165] = t[12] \oplus t[22]$
$t[166] = t[159] \oplus t[165]$
$t[167] = t[166] \oplus t[44]$
$t[168] = t[166] \oplus t[94]$
$t[169] = t[168] \oplus t[129]$
$t[170] = t[2] \oplus t[9]$
$t[171] = t[6] \oplus t[17]$
$t[172] = t[170] \oplus t[171]$
$t[173] = t[172] \oplus t[121]$
$t[174] = t[168] \oplus t[173]$
$t[175] = t[172] \oplus t[153]$
$t[176] = t[19] \oplus t[28]$
$t[177] = t[170] \oplus t[176]$
$t[178] = t[177] \oplus t[108]$
$t[179] = t[141] \oplus t[178]$
$t[180] = t[86] \oplus t[179]$
$t[181] = t[177] \oplus t[37]$
$t[182] = t[7] \oplus t[14]$
$t[183] = t[182] \oplus t[170]$
$t[184] = t[5] \oplus t[11]$
$t[185] = t[16] \oplus t[22]$
$t[186] = t[184] \oplus t[185]$
$t[187] = t[61] \oplus t[186]$
$t[188] = t[187] \oplus t[113]$
$t[189] = t[188] \oplus t[174]$
$t[190] = t[51] \oplus t[186]$
$t[191] = t[175] \oplus t[190]$
$t[192] = t[13] \oplus t[15]$
$t[193] = t[184] \oplus t[192]$
$t[194] = t[16] \oplus t[19]$
$t[195] = t[194] \oplus t[184]$
$t[196] = t[69] \oplus t[195]$
$t[197] = t[6] \oplus t[12]$
$t[198] = t[197] \oplus t[192]$
$t[199] = t[75] \oplus t[198]$
$t[200] = t[199] \oplus t[82]$
$t[201] = t[199] \oplus t[178]$
$t[202] = t[85] \oplus t[201]$
$t[203] = t[5] \oplus t[25]$
$t[204] = t[203] \oplus t[197]$
$t[205] = t[204] \oplus t[56]$
$t[206] = t[205] \oplus t[126]$
$t[207] = t[204] \oplus t[110]$
$t[208] = t[113] \oplus t[207]$
$t[209] = t[2] \oplus t[5]$
$t[210] = t[209] \oplus t[194]$
$t[211] = t[89] \oplus t[210]$
$t[212] = t[2] \oplus t[8]$
$t[213] = t[3] \oplus t[10]$
$t[214] = t[212] \oplus t[213]$
$t[215] = t[123] \oplus t[214]$
$t[216] = t[215] \oplus t[64]$
$t[217] = t[164] \oplus t[216]$
$t[218] = t[215] \oplus t[167]$
$t[219] = t[17] \oplus t[24]$
$t[220] = t[212] \oplus t[219]$
$t[221] = t[101] \oplus t[220]$
$t[222] = t[221] \oplus t[151]$
$t[223] = t[208] \oplus t[222]$
$t[224] = t[179] \oplus t[222]$
$t[225] = t[206] \oplus t[222]$
$t[226] = t[221] \oplus t[144]$
$t[227] = t[226] \oplus t[200]$
$t[228] = t[1] \oplus t[27]$
$t[229] = t[20] \oplus t[23]$
$t[230] = t[228] \oplus t[229]$
$t[231] = t[230] \oplus t[77]$
$t[232] = t[205] \oplus t[231]$
$t[233] = t[230] \oplus t[158]$
$t[234] = t[233] \oplus t[196]$
$t[235] = t[234] \oplus t[86]$
$t[236] = t[30] \oplus t[31]$
$t[237] = t[228] \oplus t[236]$
$t[238] = t[79] \oplus t[237]$
$t[239] = t[196] \oplus t[238]$
$t[240] = t[169] \oplus t[239]$
$t[241] = t[239] \oplus t[222]$
$t[242] = t[191] \oplus t[239]$
$t[243] = t[2] \oplus t[10]$
$t[244] = t[11] \oplus t[12]$
$t[245] = t[243] \oplus t[244]$
$t[246] = t[245] \oplus t[156]$
$t[247] = t[116] \oplus t[246]$
$t[248] = t[247] \oplus t[200]$
$t[249] = t[26] \oplus t[29]$
$t[250] = t[249] \oplus t[8]$
$t[251] = t[250] \oplus t[183]$
$t[252] = t[196] \oplus t[251]$
$t[253] = t[201] \oplus t[252]$
$t[254] = t[250] \oplus t[96]$
$t[255] = t[254] \oplus t[211]$
$t[256] = t[163] \oplus t[255]$
$t[257] = t[254] \oplus t[99]$
$t[258] = t[257] \oplus t[252]$
$t[259] = t[14] \oplus t[20]$
$t[260] = t[13] \oplus t[17]$
$t[261] = t[259] \oplus t[260]$
$t[262] = t[83] \oplus t[261]$
$t[263] = t[35] \oplus t[262]$
$t[264] = t[263] \oplus t[201]$
$t[265] = t[137] \oplus t[263]$
$t[266] = t[262] \oplus t[147]$
$t[267] = t[266] \oplus t[85]$
$t[268] = t[262] \oplus t[181]$
$t[269] = t[268] \oplus t[255]$
$t[270] = t[193] \oplus t[261]$
$t[271] = t[118] \oplus t[270]$
$t[272] = t[271] \oplus t[206]$
$t[273] = t[9] \oplus t[14]$
$t[274] = t[31] \oplus t[273]$
$t[275] = t[274] \oplus t[73]$
$t[276] = t[113] \oplus t[275]$
$t[277] = t[218] \oplus t[276]$
$t[278] = t[22] \oplus t[25]$
$t[279] = t[0] \oplus t[278]$
$t[280] = t[89] \oplus t[279]$
$t[281] = t[216] \oplus t[280]$
$t[282] = t[280] \oplus t[257]$

Figure 16: Matrix multiplication computation process for frequency-optimized. In the figure, $t[0]$ to $t[31]$ denote 32-bit input, $\{ t[240], t[241], t[272], t[276], t[256], t[264], t[281], t[119], t[189], t[235], t[282], t[232], t[242], t[217], t[223], t[255], t[267], t[265], t[258], t[200], t[224], t[225], t[248], t[252], t[253], t[202], t[227], t[174], t[180], t[277], t[269], t[222]\}$ denote 32-bit output.

Table 4: Comparison of area, stemming from FPGA synthesis of the base core plus implementation of the CBM, with frequency-optimized matrix multiplication.

	Registers	LUTs
Base core	2362 (1.00×)	3951 (1.00×)
Base core + CBM (excl. RG and PRG)	2365 (1.00×)	3931 (1.00×)
Base core + CBM (excl. PRG)	2425 (1.03×)	4738 (1.20×)
Base core + CBM (excl. RG)	2472 (1.05×)	4317 (1.09×)
Base core + CBM	2530 (1.07×)	4865 (1.23×)

Table 5: Comparison of area, stemming from ASIC synthesis of the base core plus implementation of the CBM, with frequency-optimized matrix multiplication.

	Cells	GE
Base core	19445 (1.00×)	30627 (1.00×)
Base core + CBM (excl. RG and PRG)	20415 (1.05×)	31785 (1.04×)
Base core + CBM (excl. PRG)	19475 (1.00×)	31988 (1.04×)
Base core + CBM (excl. RG)	21102 (1.09×)	33646 (1.10×)
Base core + CBM	20501 (1.06×)	33772 (1.10×)

In terms of performance, the frequency-optimized matrix multiplication exhibits significant advantages. For maximum frequency, the CBM with frequency-optimized version demonstrates a significant improvement over the area-optimized one. In FPGA synthesis, the maximum frequency has increased from 86% to 87%; while in ASIC synthesis, it is significantly increased from 83% to 98% (per Table 6).

B.3 Security

In terms of security, the results in Figures 17 and 18 show that both implementations perform equally well, meeting the expected security requirements.

C Leakage-Resilience of the Duplex-based PRG

Before we formally prove our claim on $\mathsf{DPRG}^{\mathsf{P}}$ in C.3, we first introduce our leakage model in C.1 and define leakage-resilience of stateful PRGs and our leakage assumptions in C.2.

C.1 Oracle-free Leakages: Preventing Future Computation Attacks

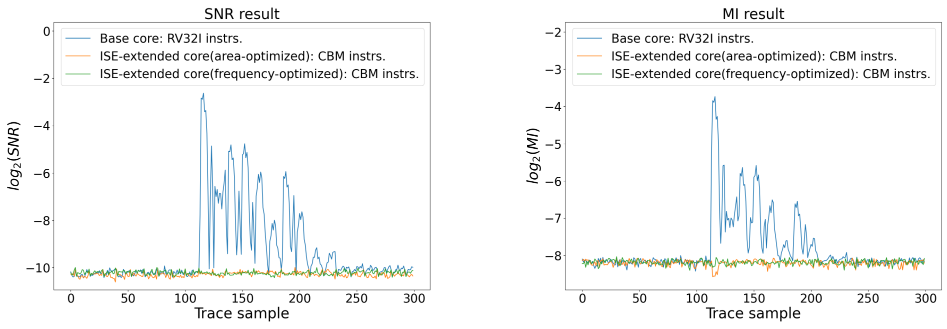
An n-bit cryptographic permutation sampled uniformly at random from the set of all n-bit permutations is called a *random permutation.* We follow [GPPS20] and analyze the duplex-based PRG in the random permutation model, i.e., we model P as a public random permutation. Note that this is also the common approach in most research papers on the duplex construction (see e.g., [Men23]).

The execution of $\mathsf{DPRG}^{\mathsf{P}}$ only consists of invocations of the permutation P and the corresponding leakages. We model these leakages as PPT functions of the involved values. Following [GPPS20], we further split the leakage into an input and an output part, i.e., we write $(\mathsf{L}^{in}(S), \mathsf{L}^{out}(S'))$ for the leakage due to evaluating $\mathsf{P}(S) \to S'$. This distinction between L^{in} and L^{out} allows to independently quantify the secrecy of the input and the output which better reflects the designers implementation goals for each functions/calls.

We require that the leakage functions L^{in} and L^{out} have no access to the random permutation (oracle) P. This restriction effectively excludes the artificial future computation attacks from the model: it guarantees that $\mathsf{L}^{in}(S)$ and $\mathsf{L}^{out}(S')$ only leak information

Table 6: Comparison of frequency, stemming from FPGA/ASIC synthesis of the base core plus implementation of the CBM, with frequency-optimized matrix multiplication.

	FPGA	ASIC
Base core	72.03 MHz (1.00×)	347.22 MHz (1.00×)
Base core + CBM	64.11 MHz (0.89×)	340.64 MHz (0.98×)



(a) SNR result (b) MI result

Figure 17: SNR and MI results of experiments regarding ISW multiplication on the ISE-extended core with the frequency-optimized matrix multiplication. Each experiment uses 300 thousand traces.

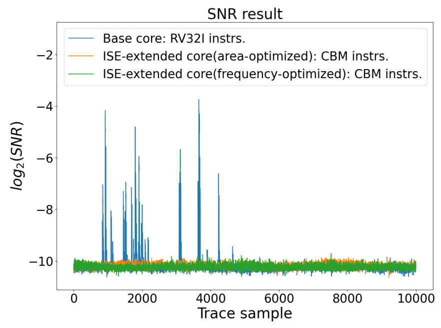
about the computation that is happening in the device rather than the computation that may happen in "future" invocations of P. Oracle-freeness restriction was first used by Yu et al. [YSPY10]. For the sake of space, we refer to [YSPY10] or earlier [DP08] for detailed discussion about future computation attacks.

Based on $\mathsf{L} = (\mathsf{L}^{in}, \mathsf{L}^{out})$, we consider a leaky implementation of $\mathsf{DPRG}^{\mathsf{P}}$, which consists of two leaky procedures $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{SetupL}(seed)$ and $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{RequestL}()$. The leaky procedure $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{SetupL}(seed)$ invokes $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{Setup}(seed)$ and outputs the corresponding leakages $\mathsf{L}^{in}([0]_{b-\kappa}\|seed)$ and $\mathsf{L}^{out}(S)$ for $S = \mathsf{P}([0]_{b-\kappa}\|seed)$. Similarly, the procedure $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{RequestL}()$ invokes $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{Request}()$ and outputs the corresponding output and leakages $\mathsf{L}^{in}(S)$ and $\mathsf{L}^{out}(\mathsf{P}(S))$, where S is the value of the internal state before this call to $\mathsf{RequestL}$. These two procedures will be used in Figure 19.

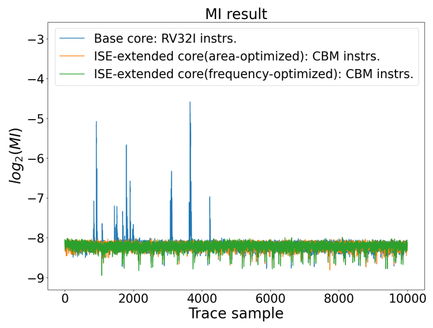
C.2 Leakage-Resilience Definition for Stateful PRGs

With the above notations, we consider the experiment $\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},\beta}(\mathcal{A})$ introduced by Yu et al. [YSPY10] (adapted to the random permutation model) to define the security of stateful PRG. It is parameterized by a leakage function $\mathsf{L} = (\mathsf{L}^{in}, \mathsf{L}^{out})$ chosen by the adversary at the very beginning of the experiment (therefore, it is the non-adaptively chosen leakage setting).

The adversary $\mathcal{A}^{\mathsf{P}}$ makes q queries to $\mathsf{DPRG}.\mathsf{RequestL}$, the leaky implementation of $\mathsf{DPRG}.\mathsf{Request}$, and then makes 1 query to the non-leaky $\mathsf{DPRG}.\mathsf{Request}$. The goal of $\mathcal{A}^{\mathsf{P}}$ is to distinguish the output of the final call to $\mathsf{DPRG}.\mathsf{Request}$ from a uniformly distributed random value, while given the outputs and leakages of the first q leaky queries.



(a) SNR result



(b) MI result

Figure 18: SNR and MI results of experiments regarding bit-sliced AES S-Box on the ISE-extended core with the frequency-optimized matrix multiplication. Each experiment uses 300 thousand traces.

Experiment $\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},\beta}(\mathcal{A})$: // $\mathsf{L} = (\mathsf{L}^{in}, \mathsf{L}^{out})$

1. A seed $seed \xleftarrow{\$} \{0,1\}^\kappa$ is sampled and $\mathsf{DPRG}^\mathsf{P}.\mathsf{SetupL}(seed)$ is invoked.
2. When $\mathcal{A}^\mathsf{P}$ submits a test query, a value $z \xleftarrow{\$} \{0,1\}^r$ is chosen uniformly at random, and z is given to $\mathcal{A}^\mathsf{P}$ if $\beta = 0$ or the honest output $\mathsf{DPRG}^\mathsf{P}.\mathsf{Request}()$ is given otherwise.
3. $\mathcal{A}^\mathsf{P}$ outputs its guess β'.

Figure 19: The experiment $\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},\beta}(\mathcal{A})$ formalizing leakage security of PRGs. The leaky procedures SetupL and RequestL are defined in C.1.

Definition 3. For any stateful DPRG^P built upon a random permutation P, the advantage of $\mathcal{A}$ against DPRG^P is

$$\mathbf{Adv}^{\mathrm{LRPRG}}_{\mathsf{DPRG},\mathsf{L}}(\mathcal{A}) := \Big|\Pr\big[\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},0}(\mathcal{A}) = 1\big] - \Pr\big[\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},1}(\mathcal{A}) = 1\big]\Big|. \quad (1)$$

Asymptotically, if $\mathbf{Adv}^{\mathrm{LRPRG}}_{\mathsf{DPRG},\mathsf{L}}(\mathcal{A})$ is a negligible function of the security parameter, then DPRG is physically unpredictable w.r.t the leakage function L.

C.2.1 Non-invertible leakage: Bounding leakage per permutation-calls

To have meaningful security, we have to assume that the leakages are somewhat bounded. To this end, we adapt [GPPS20] and assume that the leakages of the following three actions do not enable recovering the critical internal secret (i.e., *non-invertible*): (1) deriving a critical secret b-bit state S from the previous state S_{pre}: $S \leftarrow \mathsf{P}(S_{pre})$; (2) squeezing and outputting r bits from S: $y \leftarrow \mathsf{msb}_r(S)$; (3) deriving a new state: $S' \leftarrow \mathsf{P}(S)$. We assume that the side-channel adversary cannot predict the value of $\mathsf{lsb}_\omega(S)$ within a limited number of guesses, even if all the other involved $(b-\omega)$-bit values are revealed to him.

Definition. Formally, we define

$$\mathbf{Adv}^{\mathrm{Inv}[\omega]}_{\mathsf{L}}(\mathcal{A}) := \Pr\left[s_{ch} \xleftarrow{\$} \{0,1\}^\omega, \mathcal{G} \leftarrow \mathcal{A}^\mathsf{P}(\mathsf{leak}) : s_{ch} \in \mathcal{G}\right],$$

where $\mathcal{G}$ is a finite set of guesses, and $\mathcal{A}$'s input leak is a list of leakages depending on a value $y \in \{0,1\}^{b-\omega}$ chosen by $\mathcal{A}$, i.e.,

$$\mathsf{leak} = \left[\mathsf{L}^{out}(y \| s_{ch}), \mathsf{L}^{in}(y \| s_{ch})\right].$$

Further Insights. To clarify, the random state s_{ch} is the secret that is to be challenged by $\mathcal{A}$. $\mathcal{A}$ is required to choose $y \in \{0,1\}^r$ to "fill in the gap" and gets the leakages, as if $y \| s_{ch} \in \{0,1\}^b$ is the value of the state S upon the query to $\mathsf{RequestL}$.

In such an invertibility game, the power of $\mathcal{A}$ is quantified along four dimensions, i.e., the number p of queries to P, the number q of queries to $\mathsf{DPRG}^{\mathsf{P}}.\mathsf{RequestL}$, the running time t, and the number N_G of allowed guesses (i.e., $|\mathcal{G}| \le N_G$; clearly the larger N_G, the higher $\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[\omega]}(\mathcal{A})$). To simplify, we call such adversaries (p,q,t,N_G)-bounded, and further define

$$\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[\omega]}(p,q,t,N_G) := \max_{(p,q,t,N_G)\text{-bounded } \mathcal{A}} \left\{\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[\omega]}(\mathcal{A})\right\}.$$

As discussed [YSPY10, GPPS20], while the use of random permutation model is very strong, the assumption of non-invertibility is the weakest leakage assumption.

C.3 Leakage-Resilience of $\mathsf{DPRG}^{\mathsf{P}}$

We now show that the duplex-based PRG $\mathsf{DPRG}^{\mathsf{P}}$ (see Figure 7) is leakage-resilient in the sense of Definition 3, as long as the chosen leakage function L is oracle-free and non-invertible.

Theorem 1. *For any (p,q,t,N_G)-bounded adversary $\mathcal{A}^{\mathsf{P}}$ and any oracle-free leakage function $\mathsf{L} = (\mathsf{L}^{in}, \mathsf{L}^{out})$, it holds*

$$\mathbf{Adv}_{\mathsf{DPRG},\mathsf{L}}^{\mathrm{LRPRG}}(\mathcal{A}) \le \frac{2(q+2)^2}{2^{c+1}} + 2\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[\kappa]}(p,t',2p) + (q+1)\cdot 2\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[c]}(p,t',2p),$$

where $t' = O(t + qt_l)$ and t_l is the total time needed for evaluating L^{in} and L^{out}.

Interpreting the bound. Consider the parameter $c = 84$ of our concrete instantiation. The first term indicates that the number of requested pseudorandom outputs has to be $q \ll 2^{84/2} = 2^{42}$. In our scenario q would not be too large, and this limitation is fulfilled.

On the one hand, the terms $2\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[\kappa]}(p,t',2p)$ and $(q+1)\cdot 2\mathbf{Adv}_{\mathsf{L}}^{\mathrm{Inv}[84]}(p,t',2p)$ capture the influences of side-channel state recovery attacks, and they are roughly of some birthday type, namely

$$O\left(\frac{p+t}{\lambda\cdot 2^{\kappa}}\right) + O\left(q\cdot\frac{p+t}{\lambda\cdot 2^{84}}\right) = O\left(\frac{p+t}{\lambda\cdot 2^{\kappa}}\right) + O\left(q\cdot\frac{(p+t)q}{\lambda\cdot 2^{84}}\right)$$

for some parameter λ that depends on the concrete implementation and attack techniques. Yet, it is nowadays a common assumption that with such a small data complexity (only 2 relevant leakage traces), the value of λ should be very small [Pie09].

We remark that when leakages are entropy-preserving, leakage-resilience of $\mathsf{DPRG}^{\mathsf{P}}$ can also be derived from [DM19]. As complementary, our Theorem 1 provides a positive result under the weaker assumption of non-invertible leakages.

Proof. Instead of proving from the scratch, we invoke a technical lemma of Guo et al. [GPPS20]. Concretely, [GPPS20, Lemma 2] defined two leaky processes LDuStr and LIdealS, which are given in Figure 20 and Figure 21 respectively. Briefly speaking, the process LDuStr in Figure 20 consists of computing a duplex-based stream cipher to have ℓ

ciphertext blocks $C[1], ..., C[\ell]$ and the corresponding leakages, while the process LIdealS in Figure 21 consists of sampling a sequence of random state values to have ℓ random ciphertext blocks and the leakages computed from these random state values. To closely mimic the encryption of a duplex-based AEAD of [GPPS20, Lemma 2], the two processes have a number of XOR operations that are actually irrelevant to us. Guo et al. [GPPS20, Lemma 2] proved that the two processes are indistinguishable.

Lemma 2. *For every (p,t)-bounded distinguisher $\mathcal{D}^{\mathsf{P}}$ and every adversary-chosen input triple (IV, A, M) such that (A, M) has ℓ blocks in total, it holds*

$$\begin{aligned}&\left|\Pr\left[\mathcal{D}^{\mathsf{P}}(\mathrm{LDuStr}^{\mathsf{P}}_{seed}(IV,A,M)) \Rightarrow 1\right] - \Pr\left[\mathcal{D}^{\mathsf{P}}(\mathrm{LIdealS}(IV,A,M)) \Rightarrow 1\right]\right| \\ \leq & \frac{(\ell+2)^2}{2^{c+1}} + \mathbf{Adv}^{\mathrm{Inv}[\kappa]}_{\mathsf{L}}(p,t^*,2p) + (\ell+1)\cdot\mathbf{Adv}^{\mathrm{Inv}[c]}_{\mathsf{L}}(p,t^*,2p),\end{aligned}$$

where $t^* = O(t+\ell t_l)$, and t_l is the total time needed for evaluating $\mathsf{L}^{in}, \mathsf{L}^{out}, \mathrm{L}_{\oplus}$, and the xor of two r-bit values.

As mentioned, LDuStr and LIdealS have a number of XOR operations and their leakages. The information gained by $\mathcal{A}^{\mathsf{P}}$ in our real world game $\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},0}$ is almost the same as those provided by $\mathrm{LDuStr}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)$ with $\delta_1 = [0]_c$. By Lemma 2, the distance between $\mathrm{LDuStr}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)$ and $\mathrm{IdealS}(IV,\perp,([0]_r)^q)$ is bounded by

$$\frac{(q+2)^2}{2^{c+1}} + \mathbf{Adv}^{\mathrm{Inv}[\kappa]}_{\mathsf{L}}(p,t',2p) + (q+1)\cdot\mathbf{Adv}^{\mathrm{Inv}[c]}_{\mathsf{L}}(p,t',2p),$$

where $t' = O(t+qt_l)$ and t_l is the total time needed for evaluating L^{in} and L^{out}.

On the other hand, the information gained by $\mathcal{A}^{\mathsf{P}}$ in the ideal game $\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},1}(\mathcal{A})$ is the same as those provided by the modified stream cipher $\mathrm{LDuStr2}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)$ in Figure 22. $\mathrm{LDuStr2}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)$ only deviates from $\mathrm{IdealS}(IV,\perp,([0]_r)^q)$ in the first $q-1$ output blocks. Therefore, the difference between $\mathrm{LDuStr2}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)$ and $\mathrm{IdealS}(IV,\perp,([0]_r)^q)$ is bounded by

$$\frac{(q+1)^2}{2^{c+1}} + \mathbf{Adv}^{\mathrm{Inv}[\kappa]}_{\mathsf{L}}(p,t',2p) + q\cdot\mathbf{Adv}^{\mathrm{Inv}[c]}_{\mathsf{L}}(p,t',2p).$$

Therefore,

$$\begin{aligned}&\mathbf{Adv}^{\mathrm{LRPRG}}_{\mathsf{DPRG},\mathsf{L}}(\mathcal{A}) \\ = & \left|\Pr\left[\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},0}(\mathcal{A})=1\right] - \Pr\left[\mathbf{Pred}_{\mathsf{DPRG},\mathsf{L},1}(\mathcal{A})=1\right]\right| \\ \leq & \left|\Pr\left[\mathcal{A}^{\mathsf{P}}(\mathrm{LDuStr}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)) \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\mathsf{P}}(\mathrm{LDuStr2}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)) \Rightarrow 1\right]\right| \\ & + \left|\Pr\left[\mathcal{D}^{\mathsf{P}}\left(\mathrm{LDuStr2}^{\mathsf{P}}_{seed}(IV,\perp,([0]_r)^q)\right) \Rightarrow 1\right] - \Pr\left[\mathcal{D}^{\mathsf{P}}(\mathrm{LIdealS}(IV,\perp,([0]_r)^q)) \Rightarrow 1\right]\right| \\ \leq & \frac{2(q+2)^2}{2^{c+1}} + 2\mathbf{Adv}^{\mathrm{Inv}[\kappa]}_{\mathsf{L}}(p,t^*,2p) + 2(q+1)\cdot\mathbf{Adv}^{\mathrm{Inv}[c]}_{\mathsf{L}}(p,t^*,2p)\end{aligned}$$

as claimed. □

The duplex-based leaky stream cipher $\mathrm{LDuStr}^{\mathsf{P}}_{seed}(IV, A, M)$, $|seed| = \kappa$:

1. Computes $S'_0 \leftarrow IV \| seed, S_1 \leftarrow \mathsf{P}(S'_0)$. The leakages of this step are $\mathsf{L}^{in}(S'_0)$ and $\mathsf{L}^{out}(S_1)$;
2. For $i = 1, \dots, \nu, \nu = |A|/r$, computes $S'_i \leftarrow (A[i] \| [0]_c) \oplus S_i$ and $S_{i+1} \leftarrow \mathsf{P}(S'_i)$. The leakages are $\mathsf{L}^{in}(S'_i), \mathsf{L}^{out}(S_{i+1}), \mathsf{leak}_{\oplus}(\mathrm{msb}_r(S_i), A[i])$;
3. $S_{\nu+1} \leftarrow S_{\nu+1} \oplus ([0]_r \| \delta_1)$ for a fixed offset δ_1. The leakages are $\mathsf{leak}_{\oplus}(\mathrm{lsb}_c(S_{\nu+1}), \delta_1)$;
4. For $i = 1, \dots, \ell, \ell = |M|/r$, computes $j \leftarrow i + \nu, C[i] \leftarrow \mathrm{msb}_r(S_j) \oplus M[i], S'_j \leftarrow C[i] \| \mathrm{lsb}_c(S_j)$, $S_{j+1} \leftarrow \mathsf{P}(S'_j)$. Leakages are $\mathsf{leak}_{\oplus}(\mathrm{msb}_r(S_j), M[i])$ and $\mathsf{L}^{in}(S'_j)$, $\mathsf{L}^{out}(S_{j+1})$;
5. Returns $C[1] \| \dots \| C[\ell]$.

Figure 20: The duplex-based leaky stream cipher $\mathrm{LDuStr}^{\mathsf{P}}_{seed}$.

The ideal stream cipher $\mathrm{IdealS}(IV, A, M)$:

1. Samples $seed \xleftarrow{\$} \{0,1\}^{\kappa}$;
2. Computes $S'_0 \leftarrow IV \| seed$ and samples $S_1 \xleftarrow{\$} \{0,1\}^b$. The leakages of this step are $\mathsf{L}^{in}(S'_0)$ and $\mathsf{L}^{out}(S_1)$;
3. For $i = 1, \dots, \nu, \nu = |A|/r$, computes $S'_i \leftarrow (A[i] \| [0]_c) \oplus S_i$ and samples $S_{i+1} \xleftarrow{\$} \{0,1\}^b$. The leakages are $\mathsf{L}^{in}(S'_i), \mathsf{L}^{out}(S_{i+1}), \mathsf{leak}_{\oplus}(\mathrm{msb}_r(S_i), A[i])$;
4. $S_{\nu+1} \leftarrow S_{\nu+1} \oplus ([0]_r \| \delta_1)$. The leakages are $\mathsf{leak}_{\oplus}(\mathrm{lsb}_c(S_{\nu+1}), \delta_1)$;
5. For $i = 1, \dots, \ell, \ell = |M|/r$, computes $j \leftarrow i + \nu, C[i] \leftarrow \mathrm{msb}_r(S_j) \oplus M[i], S'_j \leftarrow C[i] \| \mathrm{lsb}_c(S_j)$, and samples $S_{j+1} \xleftarrow{\$} \{0,1\}^b$. The leakages are $\mathsf{leak}_{\oplus}(\mathrm{msb}_r(S_j), M[i])$ and $\mathsf{L}^{in}(S'_j), \mathsf{L}^{out}(S_{j+1})$
6. Returns $C[1] \| \dots \| C[\ell]$.

Figure 21: The ideal leaky stream cipher IdealS.

The leaky stream cipher $\mathrm{LDuStr2}^{\mathsf{P}}_{seed}(IV, A, M)$, $|seed| = \kappa$:

1. Computes $S'_0 \leftarrow IV\|seed$, $S_1 \leftarrow \mathsf{P}(S'_0)$. The leakages of this step are $\mathsf{L}^{in}(S'_0)$ and $\mathsf{L}^{out}(S_1)$;
2. For $i = 1, \dots, \nu, \nu = |A|/r$, computes $S'_i \leftarrow (A[i]\|[0]_c) \oplus S_i$ and $S_{i+1} \leftarrow \mathsf{P}(S'_i)$. The leakages are $\mathsf{L}^{in}(S'_i), \mathsf{L}^{out}(S_{i+1}), \mathsf{leak}_{\oplus}(\mathrm{msb}_r(S_i), A[i])$;
3. $S_{\nu+1} \leftarrow S_{\nu+1} \oplus ([0]_r\|\delta_1)$ for a fixed offset δ_1. The leakages are $\mathsf{leak}_{\oplus}(\mathrm{lsb}_c(S_{\nu+1}), \delta_1)$;
4. For $i = 1, \dots, \ell - 1, \ell = |M|/r$, computes $j \leftarrow i + \nu, C[i] \leftarrow \mathrm{msb}_r(S_j) \oplus M[i]$, $S'_j \leftarrow C[i]\|\,\mathrm{lsb}_c(S_j)$, $S_{j+1} \leftarrow \mathsf{P}(S'_j)$. Leakages are $\mathsf{leak}_{\oplus}(\mathrm{msb}_r(S_j), M[i])$ and $\mathsf{L}^{in}(S'_j)$, $\mathsf{L}^{out}(S_{j+1})$;
5. Finally, $C[\ell] \xleftarrow{\$} \{0,1\}^r$. Namely, in contrast to $\mathrm{LDuStr}^{\mathsf{P}}_{seed}$, the last output block is truly random.
6. Returns $C[1]\| \dots \|C[\ell]$.

Figure 22: $\mathrm{LDuStr2}^{\mathsf{P}}_{seed}$: replacing the last r-bit block of output of LDuStr with a random block.

D Formal verification

```
 1 # Using standard RV32I instructions
 2 # x31:    holds random mask
 3 nop
 4 cbm.or     x31, x31, x0 , 3
 5 cbm.srli   x31, x31, 16 , 3
 6 nop
 7 and        x20, x5 , x6
 8 nop
 9 xor        x20, x20, x31
10 nop
11 and        x21, x5 , x13
12 nop
13 xor        x21, x21, x31
14 nop
15 and        x22, x12, x6
16 nop
17 xor        x21, x21, x22
18 nop
19 and        x23, x12, x13
20 nop
21 xor        x21, x21, x23
22 nop
23 nop
24 nop
25 nop
26 # Leakage captured
```

```
 1 # Using CBM instructions
 2 # x31:    holds random mask
 3 nop
 4 cbm.or     x5 , x5 , x0 , 0
 5 cbm.or     x5 , x5 , x0 , 3
 6 cbm.or     x6 , x6 , x0 , 0
 7 cbm.or     x6 , x6 , x0 , 3
 8 cbm.or     x12, x12, x0 , 0
 9 cbm.or     x12, x12, x0 , 3
10 cbm.or     x13, x13, x0 , 0
11 cbm.or     x13, x13, x0 , 3
12 cbm.or     x31, x31, x0 , 3
13 cbm.srli   x31, x31, 16 , 3
14 cbm.and    x20, x5 , x6 , 3
15 xor        x20, x20, x31
16 nop
17 cbm.and    x21, x5 , x13, 3
18 xor        x21, x21, x31
19 nop
20 cbm.and    x22, x12, x6 , 3
21 xor        x21, x21, x22
22 nop
23 cbm.and    x23, x12, x13, 3
24 xor        x21, x21, x23
25 nop
26 # No leakage
```

Figure 23: Micro-benchmarks of first-order ISW multiplication for formal verification.

Figure 23 shows the micro-benchmarks used in our formal verification. Note that the first 12 lines on the right are dedicated to transforming shares into protected shares, while lines 4/5 on the left and lines 13/14 on the right are used to generate the random numbers required for the execution of ISW multiplication. During the verification process, GPR[5] and GPR[12] are labeled as two shares of the same secret, while GPR[6] and GPR[13] are labeled as two shares of another secret. For the verification, we set the key parameters as follows: the cycles is set to 50, and the mode is TRANSIENT.

The verification results are as follows: when using standard RV32I instructions, leakage between GPR[6] and GPR[13] is captured at line 11, with the leakage location identified at the general-purpose register write port, and the verification takes approximately 10 seconds. In contrast, when using CBM instructions, no leakage is detected, and the verification takes approximately 180 seconds.