Leakage-Resilience of Circuit Garbling

Ruiyang Li[1,2], Yiteng Sun[1,2], Chun Guo[1,2], François-Xavier Standaert[3], Weijia Wang[1,2], and Xiao Wang[4]

[1] School of Cyber Science and Technology, Shandong University, Qingdao, Shandong, 266237, China

[2] Key Laboratory of Cryptologic Technology and Information Security of Ministry of Education, Shandong University, Qingdao, Shandong, 266237, China

{ruiyang.li,sunyiteng}@mail.sdu.edu.cn, {chun.guo,wjwang}@sdu.edu.cn

[3] ICTEAM/ELEN/Crypto Group, UCL, Louvain-la-Neuve, Belgium

fstandae@uclouvain.be

[4] Northwestern University, Evanston, USA

wangxiao@northwestern.edu

Abstract. Due to the ubiquitous requirements and performance leap in the past decade, it has become feasible to execute garbling and secure computations in settings sensitive to side-channel attacks, including smartphones, IoTs and dedicated hardwares, and the possibilities have been demonstrated by recent works. To maintain security in the presence of a moderate amount of leaked information about internal secrets, we investigate *leakage-resilient garbling*. We augment the classical privacy, obliviousness and authenticity notions with leakages of the garbling function, and define their leakage-resilience analogues. We examine popular garbling schemes and unveil additional side-channel weaknesses due to wire label reuse and XOR leakages. We then incorporate the idea of label refreshing into the GLNP garbling scheme of Gueron et al. and propose a variant GLNPLR that provably satisfies our leakage-resilience definitions. Performance comparison indicates that GLNPLR is 60X (using AES-NI) or 5X (without AES-NI) faster than the HalfGates garbling with second order side-channel masking, for garbling AES circuit when the bandwidth is 2Gbps.

1 Introduction

Garbled circuits (GCs). The idea of *garbled circuits (GCs)* was proposed by Yao for constant-round secure two-party computation [44]. Yao's protocol was proven secure by Lindell and Pinkas [32]. Though, it was Bellare et al. [4] that firstly formalized garbling schemes as a sort of cryptographic primitive. Concretely, a garbling scheme $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode})$ mainly consists of four (randomized) algorithms. The garbling algorithm $(F, e, d) \leftarrow \mathsf{Garble}(f)$ transforms a function $f : \{0,1\}^n \to \{0,1\}^m$ into a *garbled circuit* F and a pair of associated input and output tables e and d. The encoding function $X := \mathsf{Encode}(e, x)$ uses e to turn an initial input $x \in \{0,1\}^n$ into a garbled input X. The evaluating

algorithm $Y := \mathsf{Eval}(F, X)$ uses F to map the garbled input X to the corresponding garbled output Y. Finally, the decoding function $y := \mathsf{Decode}(d, Y)$ uses d to translate Y into the function output $y \in \{0,1\}^m$, which must coincide with $f(x)$. Informally, this probabilistically factors f into $d \circ F \circ e$.

Bellare et al. [4] formalized security of garbling schemes as *privacy*, *obliviousness* and *authenticity*, and constructed provably secure schemes from blockciphers. Subsequently, spurred by initial implementations demonstrating the practicality [34,40,22], circuit garbling has received a considerable amount of attention and improvements, including the point-and-permute technique [37], garbled row reduction [37], Free-XOR [30], fleXOR [28], HalfGates garbling [46] and its improvements [18,42,20], and optimizations using pipelined AES-NI instructions [3,17]. These have brought in significant improvements in computational and communication costs as well as concrete security.

Side-channel attacks on GC. Due to both requirements and the various advances in the past decade, it has become feasible to execute garbling and secure computation on "low-performance" platforms such as advanced Systems-On-a Chip (SoCs), smartphones [9,1,24], automotive, and even IoTs [1] and dedicated hardwares. Such hardware devices are susceptible to *side-channel attacks* (SCA), which was not reflected in their classical security models. In addition, using advanced techniques such as [33], garbling implementations using AES-NI in Intel SGX can be victims of SCAs as well.

Concretely, every time a device executes an operation over its internal secrets (a cryptographic key, or a crucial state), it would produce some leakages (in the form of certain physically measurable quantities) about these secrets. SCAs collect and analyze such leakages to extract these secrets. Typical leakages include timing, power consumption and electromagnetic radiation measurements. SCAs can be mounted in two main flavors which, in the context of power consumption, are called Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [35]. In an SPA, an attacker takes advantage of the leakages resulting from *a few (typically one or two) inputs* of the operation. A DPA, on the other hand, exploits the leakages resulting from *multiple inputs* (typically hundreds, thousands or even more), which all provide new information about the same internal secret in the device, reducing the computational secrecy of this state at a rate that is exponential in the number of distinct inputs. We will use the term SPA-type, resp. DPA-type attacks, to refer to all SCAs using a few input leakages, resp. multiple input leakages.

Standard cryptographic constructions are typically susceptible to DPA-type attacks. Regarding garbling, a large proposition of existing constructions employed the Free-XOR optimization [29]. Roughly, it requires Garble to set a global *secret* offset Δ and assign a random label w^0 to each wire representing garbled wire value 0 (or FALSE), and then $w^1 = w^0 \oplus \Delta$ represents garbled wire value 1 (or TRUE). This enables computing XOR gates directly in Eval. However, using Δ for multiple wire labels exactly fulfills the condition of mounting DPA-type attacks, and Levi and Hazay (LH) [31] have leveraged this to mount

a practical attack recovering Δ. In all, designing side-channel secure garbling schemes has become an important question.

Leakage-resilience. Traditionally, the protection against SCAs is achieved by applying implementation-level techniques to the cryptographic primitive to limit the leakages as much as possible. One of the most popular techniques is *masking* [10], in which the internal state of the device is secret-shared into a number of pieces, which are then used for the computation. The strong protection of a cryptographic primitive (against DPA-type attacks) usually decreases the standard performance metrics of implementations by orders of magnitude compared to non-protected ones [15,16].

Another approach, initiated by Dziembowski and Pietrzak [14], is to design *leakage-resilient constructions*. Such constructions, which often come with some computational overheads in the black-box setting, aim at substantially reducing the possibility of mounting DPA-type attacks. A classical ingredient is to use some form of key evolving [25,26] or re-keying [14,45] to ensure that each execution of an underlying primitive (e.g., a blockcipher) leaks about a different secret, hence effectively leaving the adversary with the possibility of SPA-type attacks only. In this way, it will be tolerated that the underlying implementations continuously leak a certain amount of information to the adversary every time they are used, hence requiring very limited protections, or even no specific protection at all depending on the platform (we refer to [8,7] for detailed discussions). This approach brings in two benefits. First, as will be demonstrated in our experiments, it leads to more efficient implementations for a given level of side-channel security. Second, the security reductions that come with the definition of leakage-resilient schemes clarify requirements on the specific blocks to implement. This considerably simplifies the task of designers and security evaluation laboratories. Therefore, this approach has spurred a plenty of works on stream ciphers [39,45], signatures [25], MACs [36,38] and encryption schemes [26,38,2], and has been employed by several submissions [6,13] to NIST standardization.

1.1 Our contributions

Motivated by the above discussion, we initiate leakage-resilience of circuit garbling schemes, aiming at formal definitions and much more efficient solutions.

Leakage-resilience notions for garbling. Consider semi-honest adversaries. Recall that in the classical setting, the adversary can corrupt the garbler or evaluator, strictly follow the instructions of the protocol but are still curious to learn information of other inputs from the transactions. In our leakage setting, besides corrupting the evaluator, the adversary can also measure the side-channel leakages of the garbler procedure Garble. But it is completely passive and does not affect the evaluator's computation or communicated data. On the other hand, we do not consider leakages of Encode and Decode, because they only involve simple mappings.

With the above in mind, we augment the classical security notions with leakages of Garble. Considering executing $\mathsf{Garble}(f) \to (F, e, d, \mathsf{leak})$ and $\mathsf{Encode}(e, x) \to$

X, where leak is the side-channel leakage of $\mathsf{Garble}(f)$. Our notion *leakage-resilient privacy* requires the resulted 4-tuple (F, X, d, leak) does not reveal any information about x that cannot be learned directly from $f(x)$, while our notion *leakage-resilient obliviousness* requires the triple (F, X, leak) given by $\mathsf{Garble}(f) \to (F, e, d, \mathsf{leak})$ and $\mathsf{Encode}(e, x) \to X$ does not reveal any information about x. We formalize them by requiring a simulator $\mathcal{S}(f, f(x))$, resp. $\mathcal{S}(f)$, that outputs an indistinguishable 4-tuple (F, X, d, leak), resp. triple (F, X, leak). In this way, schemes satisfying our definitions allow secure composition in the presence of Garble leakages (which will be further justified by our application to Yao's SFE protocol [44]).

Finally, *leakage-resilient authenticity* requires that (F, X, leak) from $\mathsf{Garble}(f) \to (F, e, d, \mathsf{leak})$ and $\mathsf{Encode}(e, x) \to X$ does not help computing a garbled output $\tilde{Y}$ with $\mathsf{Decode}(d, \tilde{Y}) \neq f(x)$.

Discussion: LH, and evaluation leakages. By allowing to corrupt the evaluator and get all wire labels, security in our model implies security against LH's attack [31] (which did not consider corrupting evaluator). Furthermore, in our model, there are new SCAs using leakages of the XOR operations: please see our discussion below.

Our security definitions and subsequent constructions only consider leakages of the Garble algorithm. This is a relevant first step, since it has given rise to much more efficient Garble implementations in SCA sensitive scenarios (please see our performance report below). In addition, note that LH also focused on Garble [31], and a scheme satisfying our security definition would resist their attack. Yet, we admit the value and challenge in leakage-resilient Eval (see Appendix A) and leave it as an interesting open problem.

Examining popular schemes. Due to the attack in [31], none of the existing garbling schemes using Free-XOR is leakage-resilient. We then make a natural step and examine leakage security of popular "non-free-XOR" garbling schemes, i.e., those did not use the Free-XOR optimization. These include BHR [4], GRR3 and GLNP [17]. As mentioned, we identified two common side-channel weaknesses and excluded their possibilities of leakage-resilience.

1. DPA-type attacks due to label reuse. In a classical "non-free-XOR" garbling, the Garble algorithm typically assigns two *independent* and (pseudo)-random labels w^0 and w^1 to each wire representing garbled wire values 0 and 1. This seems to exclude common secrets. However, it depends on a generalized form of fan-out value of the circuit: in the to-be-garbled circuit, if a wire is connected to the inputs of multiple gates, then during garbling these gates, the Garble algorithm would (re)use the label of this wire to invoke the underlying blockcipher E multiple times. This enables another standard DPA-type attack.

For example, consider the function $f_{\text{seq}}(\alpha, \gamma_1, ..., \gamma_\ell) = (\alpha \oplus \gamma_1, \alpha \oplus \gamma_2, ..., \alpha \oplus \gamma_\ell)$ producing a sequence from an initial value α, which is used in, e.g., [19]. In its circuit, the wire for each input bit of α is input to ℓ XOR gates, and the corresponding (secret) wire labels are used in ℓ distinct XOR garbling processes. We exhibit a concrete DPA-type attack in garbling f_{seq} with the GLNP scheme [17]. We also serve more circuits with high fan-out in Sect. 4.1.

We stress that the reuse of wire labels (for high fan-out circuits) is common in virtually all garbling schemes, including Yao, BHR [4], GRR3, etc., giving rise to similar DPA-type attacks. To deploy garbling in SCA sensitive scenarios, this sort of DPA has to be eliminated by either side-channel protections or label refreshing. The latter will be used in our proposal GLNPLR.

2. XOR leaks wire labels. In virtually all "non-free-XOR" garbling schemes, if the XOR computations in their GbAND procedures (i.e., the procedures to garble AND gates) leak non-trivial information, then an SCA adversary corrupting the evaluator could recover some unknown wire labels.

Briefly speaking, in "non-free-XOR" schemes, the GbAND procedure consists of XORing the output wire labels w_c^0 and w_c^1 with some other internal secrets. Since the truth table of AND is not uniform, the label w_c^0 is involved in more XORs. Therefore, if the leakages allow the SCA adversary to distinguish XORing with w_c^0 from XORing with w_c^1, then it could collect XOR leakages for w_c^0 as "templates". Now, by corrupting the evaluator, the SCA adversary obtains $w_c^{v_c}$ with $v_c \in \{0,1\}$ unknown. By comparing $w_c^{v_c}$ with the above "templates", the unknown v_c can be determined.

We serve a complete discussion on GLNP [17]. Due to using point-and-permute technique, we found two other similar SCA weaknesses in GLNP.GbAND. We extend the discussion to BHR [4] and another GbAND design of [17] in Appendix B. These observations resemble the well-known fact that XOR leakages (may) break semantic security of encryption [38]. However, our finding here is far less trivial.

A leakage-resilient garbling scheme. We then investigate constructing leakage-resilient garbling schemes using a blockcipher E. We were unable to find constructions with Free-XOR optimizations. Therefore, we seek to improve GLNP [17], the state-of-the-art "non-free-XOR" scheme, and give the first construction GLNPLR that provably achieves our definitions of leakage-resilience.

To garble AND and XOR gates, our procedures GbXOR and GbAND generally follow the ideas of GLNP [17]. Our novelty mainly lies in counteracting the aforementioned weakness due to large fan-out. For this, every time our Garble algorithm assigned labels w_c^0 and w_c^1 to wire c (this happens during initializing labels to input wires or assigning labels to gate output wires), it counts the number ℓ_c of gate input wires connected to wire c (these are called *sub-wires* of wire c) and assigns ℓ_c independent pairs of (pseudo)random labels $(w_{i[1]}^0, w_{i[1]}^1), ..., (w_{i[\ell_c]}^0, w_{i[\ell_c]}^1)$ to them. In this way, the associated sub-wires are still using independent labels, excluding the mentioned DPA-type attacks. This idea of refreshing internal secrets is classical in leakage-resilient cryptography [14,39,45,38,8]. To avoid SCAs, the labels $(w_{i[1]}^0, w_{i[1]}^1), ..., (w_{i[\ell_c]}^0, w_{i[\ell_c]}^1)$ are generated using a blockcipher-based leakage-resilient PRG of Yu et al. [45] (denoted GSL), and we refer to Sect. 5 for technical details.

To prove security, we model leakages as PPT functions on the secret wire labels. To limit the amount of leaked information, we follow Yu et al. [45,8] and adopt the assumption of *hard-to-invert leakages* in *the ideal cipher model* (i.e., we model E as a public randomly picked blockcipher). To our knowledge,

leakage-resilient constructions that do not rely on this combination of models are far less efficient. We additionally remark that hard-to-invert leakages appear to be both theoretically minimal and practically measurable, which is essential for real-world deployments. Moreover, since we did not impose any apriori determined leakage bound, our assumption fits into the continuous leakage model that reflects actual side-channel attacks.

Regarding the XOR operations, we follow Pereira et al. [38,8] and show that the leakage security of GLNP reduces to the leakage security of the XOR operations (the advantage of which may not be negligible). By this, the designer is guaranteed that the security of the full construction reduces to the security of some basic building blocks, including the blockcipher and the XORs (whatever security he is able to achieve). With these models, we formally prove security of GLNPLR w.r.t. our definitions.

Performance comparison. Since GLNPLR aims for side-channel security, we compare with masked classical schemes. The plain implementation (without extra protections) of GLNPLR's Garble procedure has been proven side-channel secure under plausible leakage assumptions (see Sect. 5.3–5.6). We thus provide two "plain" AES-based GLNPLR implementations: one follows [17] and uses pipelined AES-NI, and the other uses the C implementation of AES from the OpenSSL library [43]. We choose HalfGates as the benchmark of classical schemes, and we provide an implementation using the second-order marked AES from [27] (built upon [41,12,47,11]). As long as the garbled circuit has 10% AND gates and bandwidth exceeds 300Mbps, GLNPLR is 2 (without AES-NI) to several hundred (with AES-NI) faster than the masked HalfGates.

Application. To show the usefulness of our formalism, we demonstrate an application to secure function evaluation (SFE). Since such protocols (as well as other garbling-based systems) are usually built upon multiple building blocks that may leak, we are only able to formally prove that *there are no SCA weaknesses due to garbling any more.* For this, we define the (weaker form of) leakage-resilience of SFE as simulatability of the protocol outputs *and the leakages of the* Garble *invocations.* We then show that Yao's garbling-based SFE protocol [44] built upon our leakage-resilient garbling scheme provably achieves our leakage-resilience definition.

A summary. Our results are summarized as follows.

1. We define leakage-resilient extensions of the classical security notions. For the simulation-based notions of privacy and obliviousness, we requires the simulator to emulate the Garble leakages. For authenticity, we require a similar level of unpredictability in the presence of Garble leakages.
2. We examine leakage security of popular garbling schemes, including BHR, GRR3 and GLNP that do not employ Free-XOR optimization. We identify two common side-channel weaknesses: (i) the reuse of secret wire labels during garbling different gates enables a standard DPA-type attack. Therefore, none of them is leakage-resilient. (ii) when the adversary corrupts the evaluator, the XOR operations could leak information about wire labels.

3. By combining a leakage-resilient PRG of [45] with the GLNP garbling [17], coupled with other tweaks, we propose the first scheme GLNPLR that provably achieves leakage-resilience. We implement GLNPLR and masked HalfGates using [27], confirming that GLNPLR significantly outperforms masked HalfGates for practical parameters.
4. We show that once built upon our leakage-resilient garbling scheme, Yao's Secure Function Evaluation (SFE) protocol is provably secure in the presence of Garble leakages.

Practical interpretations. As discussed, all existing garbling admit some side-channel weaknesses. Consequently, to deploy garbling schemes in SCA sensitive scenarios, one has to either add heavy side-channel protections or employ our solution GLNPLR. In the latter case, one still has to add SCA protections in some modules, including: (i) the Eval algorithm; (ii) the XOR operations. In this respect, the performance comparison has confirmed the advantage of GLNPLR in SCA sensitive scenarios.

Related works and Open problems. Our work *leakage-resilience of garbling* should be distinguished from existing elegant works of *constructing leakage-resilient crypto from garbling* [23]. Regarding attacks, Hashemi et al. [21] recently exhibited a timing attack against specific Garble implementations optimized for SFE.

The most important yet challenging open problem is to build leakage-resilient Eval algorithms from symmetric primitives. We briefly discuss the difficulty in Appendix A. Equally important is to tailor the generic principles of the paper to actual targets for garbled circuits that may be more noisy than small embedded devices. Finally, natural future works also include designing leakage-resilient garbling with better efficiency (in particular, restoring free XOR property), concrete security or adaptive security.

Organization. We first serve necessary notations and definitions in Sect. 2. We formalize leakage-resilient garbling in Sect. 3. We then discuss new side-channel weaknesses in Sect. 4. Our scheme GLNPLR and its leakage-resilience proof are given in Sect. 5, and experimental results on its performance are given in Sect. 6. We finally give the application to SFE in Sect. 7.

2 Preliminaries

A reader may skip this on first reading and refer back as necessary.

General notations. We denote $x \| y$ by concatenation of x with y. We denote by a (q, t)-bounded adversary a probabilistic algorithm with an oracle that can make at most q queries to its oracle and run times at most t. If A is a finite set, then $y \xleftarrow{\$} A$ denotes selecting an element of A uniformly at random and assigning it to y.

Code based game. Our security definitions are stated using code-based games [5]. A code-based game–see Fig. 1 for an example–consists of an INITIALIZE proce-

dure, procedures that respond to adversary oracle queries, and a FINALIZE procedure. All procedures are optional. Procedure INITIALIZE, if present, executes first, and its output is input to the adversary, who may now invoke other procedures. Each time $\mathcal{A}$ makes a query, the corresponding game procedure executes, and what it returns, if anything, is the response to $\mathcal{A}$'s query. The adversary's output is the input to FINALIZE, and the output of the latter, denoted $\mathrm{Gm}^{\mathcal{A}}$, is called the output of the game. We let "$\mathrm{Gm}^{\mathcal{A}} = c$" represent the event that this game output takes value c. In some cases, INITIALIZE samples a random bit $b \xleftarrow{\$} \{0,1\}$ for subsequent execution, and we denote by "$\mathrm{Gm}^{\mathcal{A},\beta}$" the execution with $b = \beta$.

Circuit: the classical formalism. Boolean circuits consist of AND and XOR gates with fan-in 2 and NOT gates with fan-in 1. Following [4], such a circuit is defined by a 4-tuple $f = (\ell_{in}, \ell_{out}, g, \mathsf{Gates})$, where $\ell_{in} \geq 2$ denotes the number of input wires, $\ell_{out} \geq 1$ is the number of output wires and g is the number of gates. Such a circuit has $\ell_{in}+g$ wires numbered $1, ..., \ell_{in}+g$; we let $\mathsf{Inputs} = \{1, \cdots, \ell_{in}\}$ and $\mathsf{Outputs} = \{\ell_{in} + g - \ell_{out} + 1, \ldots, \ell_{in} + g\}$. There are g tuples of the form (a, b, c, G) in Gates, where $a, b, c \in \{1, \cdots, \ell_{in} + g\}$ represents a gate of type $G \in \{\mathsf{XOR}, \mathsf{AND}, \mathsf{NOT}\}$ with input wires a, b and output wire c (if $G = \mathsf{NOT}$ then $b = \bot$).

Garbling schemes. Following [4,17], a garbling scheme consists of four algorithms:

- $\mathsf{Garble}(f) \to (F, e, d)$ is an algorithm that takes as input a description of a boolean circuit f and returns a triple (F, e, d), where F represents a garbled circuit, e represents input encoding information (i.e., all the labels on the input wires) and d represents output decoding information (i.e., all the labels on the output wires). Following [18], we focus on concrete security and do not use explicit security parameters.
- $\mathsf{Encode}(e, x) \to X$ is a function that computes the garbled input X of an input x according to the input encoding e.
- $\mathsf{Eval}(F, X) \to Y$ is a function that computes the garbled output Y of a garbled input X under a garbled circuit F.
- $\mathsf{Decode}(Y, d) \to y$ is a function that takes as input decoding information d and garbled output Y and returns either the real output y of the circuit or $\bot$.

Given $(F, e, d) = \mathsf{Garble}(f)$, *correctness* means $\mathsf{Decode}\big(d, \mathsf{Eval}(F, \mathsf{Encode}(e, x))\big) = f(x)$ for any $x \in \{0,1\}^n$.

The classical security notions for GC include *privacy*, *obliviousness*, and *authenticity* [4]:

- **Privacy:** the triple (F, X, d) does not reveal any information about x that cannot be learned directly from $f(x)$. More formally, there exists a simulator $\mathcal{S}$ that receives input $(f, f(x))$ and outputs a simulated garbled circuit F with garbled input X and decoding information d that is indistinguishable from (F, X, d) generated using the real garbling functions $(F, e, d) \leftarrow \mathsf{Garble}(f)$ and $X \leftarrow \mathsf{Encode}(e, x)$. We refer to the experiment $\mathsf{Expt}^{\mathcal{G},\mathcal{S}}_{\mathsf{PrvSim}}$ in Fig. 1 (Top).

```
Game Expt^{G,S}_{PrvSim}
  procedure Initialize                procedure Main(f, x)
    b <-$ {0,1}                          if b = 0 then
  procedure Finalize(b')                   (F, e, d) := Garble(f)
    return (b' = b)                        X := Encode(e, x)
                                         else
                                           (F, X, d) := S(f, f(x))
                                         return (F, X, d)
-----------------------------------------------------------------
Game Expt^{G,S}_{ObvSim}
  procedure Initialize                procedure Main(f, x)
    b <-$ {0,1}                          if b = 0 then
  procedure Finalize(b')                   (F, e, d) := Garble(f)
    return (b' = b)                        X := Encode(e, x)
                                         else (F, X) := S(f)
                                         return (F, X)
-----------------------------------------------------------------
Game Expt^{G}_{Aut}
  procedure Finalize(Y)               procedure Main(f, x)
    X~ := Decode(d, Y)                   (F, e, d) := Garble(f)
    if X~ ∉ {⊥, f(x)} then               X := Encode(e, x)
      return TRUE                        return (F, X)
    else return FALSE
```

Fig. 1: Experiments for defining the PrvSim, ObvSim and Aut security of a garbling scheme $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Decode}, \mathsf{Eval})$.

- **Obliviousness:** the pair (F, X) does not reveal any information about x, and this is also formalized via simulation [4]. Formally, there exists a simulator $\mathcal{S}$ that receives input f and outputs a simulated garbled circuit F with garbled input X that is indistinguishable from (F, X) generated using the real garbling functions Garble(c) and Encode(e, x). We refer to the experiment $\mathsf{Expt}^{\mathcal{G},\mathcal{S}}_{\mathsf{ObvSim}}$ in Fig. 1 (Middle).
  For $\mathsf{xx} \in \{\mathsf{PrvSim}, \mathsf{ObvSim}\}$, the advantage $\mathbf{Adv}^{\mathcal{G},\mathcal{S}}_{\mathsf{xx}}(\mathcal{A})$ is defined as $\mathbf{Adv}^{\mathcal{G},\mathcal{S}}_{\mathsf{xx}}(\mathcal{A}) = 2\Pr\left[\mathsf{Expt}^{\mathcal{G},\mathcal{S}}_{\mathsf{xx}} = 1\right] - 1$.
- **Authenticity:** given (F, X), it is difficult to produce a garbled output $\tilde{Y}$ that when decoded provides a value that does not equal $f(x)$ or abort. We refer to $\mathsf{Expt}^{\mathcal{G},\mathcal{S}}_{\mathsf{Aut}}$ in Fig. 1 (Bottom) and define $\mathbf{Adv}^{\mathcal{G}}_{\mathsf{Aut}}(\mathcal{A}) = \Pr\left[\mathsf{Expt}^{\mathcal{G}}_{\mathsf{Aut}} = 1\right]$.

Leaky implementations. A *leaky implementation* of an algorithm Algo is associated with a *probabilistic leakage function* $\mathsf{L}_{\mathsf{Algo}}$ that captures the additional information given by the implementation of Algo during its execution. The leakage functions will be parameters of relevant security definitions (e.g., see Sect. 3). Upon each execution of $\mathsf{Algo}(x)$, the corresponding leakage is $\mathsf{L}_{\mathsf{Algo}}(x, \iota)$, where ι is the internal randomness used during executing $\mathsf{Algo}(x)$ (when Algo is probabilistic). The use of ι is mainly for the probabilistic Garble. Another approach is to modify the syntax $\mathsf{Garble}(f)$ to $\mathsf{Garble}(f, \iota)$ to make internal randomness

Game $\mathsf{Expt}_{\mathsf{PrvSimL}}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}$

```
procedure INITIALIZE                procedure MAIN(f, x)
    b <-$ {0,1}                         if b = 0 then
                                            (F, e, d, leak) := LGarble(f)
procedure FINALIZE(b')                      X := Encode(e, x)
    return (b' = b)                     else
                                            (F, X, d, leak) := S(f, f(x))
                                        return (F, X, d, leak)
```

Game $\mathsf{Expt}_{\mathsf{ObvSimL}}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}$

```
procedure INITIALIZE                procedure MAIN(f, x)
    b <-$ {0,1}                         if b = 0 then
                                            (F, e, d, leak) := LGarble(f)
procedure FINALIZE(b')                      X := Encode(e, x)
    return (b' = b)                     else (F, X, leak) := S(f)
                                        return (F, X, leak)
```

Game $\mathsf{Expt}_{\mathsf{AutL}}^{\mathcal{G},\mathsf{L}_{\mathsf{Garble}}}$

```
procedure FINALIZE(Y)               procedure MAIN(f, x)
    X~ := Decode(d, Y)                  (F, e, d, leak) := LGarble(f)
    if X~ ∉ {⊥, f(x)} then              X := Encode(e, x)
        return TRUE                     return (F, X, leak)
    else return FALSE
```

Fig. 2: Experiments for defining the leakage-resilient $\mathsf{PrvSimL}$, $\mathsf{ObvSimL}$ and AutL security of a garbling scheme $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Decode}, \mathsf{Eval})$ with leaky implementation Garble.

explicit, but this may cause difficulty in understanding and we thus eschewed. To simplify notations, we define $\mathsf{LAlgo}(x) := \big(\mathsf{Algo}(x), \mathsf{L}_{\mathsf{Algo}}(x, \iota)\big)$ for the output of the leaky implementation of Algo.

3 Defining Leakage-Resilience of Garbling

In this section, we formalize the leakage-resilience of GC schemes (which are natural extensions of the classical notions *privacy*, *obliviousness*, and *authenticity* of [4] to the leakage setting).

Leakage-resilient privacy. Recall from Sect. 2 that the classical privacy notion requires that the triple (F, X, d) given by $(F, e, d) := \mathsf{Garble}(f)$ and $X := \mathsf{Encode}(e, x)$ is indistinguishable from a simulator's output. Since we concern with leakages of Garble, the leaky extension naturally requires that both (F, X, d) and the corresponding leakage $\mathsf{L}_{\mathsf{Garble}}(f, \iota)$ can be simulated, where $\mathsf{L}_{\mathsf{Garble}}$ is an added parameter denoting the leakage function of Garble and ι is the internal randomness used during executing $\mathsf{Garble}(f)$ (see Sect. 2). Therefore, even with the garbling leakages, the output does not reveal information about x.

To formalize, we define an experiment $\mathsf{Expt}_{\mathsf{PrvSimL}}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}$, where $\mathcal{G}$ is a garbling scheme, and $\mathcal{S}$ is a simulator. The simulator receives input $(f, f(x))$ and outputs

a simulated circuit F with encoding and decoding information e, d, *as well as the simulated leakage* that is indistinguishable from (F, X, d, leak) generated by the leaky implementation $\mathsf{Garble}(f)$ (see Sect. 2) and $\mathsf{Encode}(e, x)$. The advantage $\mathbf{Adv}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}}(\mathcal{A})$ of $\mathcal{A}$ is defined as

$$\mathbf{Adv}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}}(\mathcal{A}) = 2\Pr\left[\mathsf{Expt}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}} = 1\right] - 1.$$

Leakage-resilient obliviousness. Recall from Sect. 2 that the classical obliviousness requires that the triple (F, X) given by $(F, e, d) := \mathsf{Garble}(f)$ and $X := \mathsf{Encode}(e, x)$ is indistinguishable from the simulated. In a similar vein to privacy, our leaky extension requires that both (F, X) and the garbling leakage can be simulated. Concretely, we define an experiment $\mathsf{Expt}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{ObvSimL}}$, where $\mathcal{G}$ is a garbled scheme and $\mathcal{S}$ is a simulator. The simulator receives f as its input and outputs simulated F, simulated garbled input X of x and *the simulated leakage* leak, such that the simulated (F, X, leak) is indistinguishable from those given by $\mathsf{LGarble}(f)$ (see Sect. 2) and $\mathsf{Encode}(e, x)$. The advantage $\mathbf{Adv}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{ObvSimL}}(\mathcal{A})$ is defined as

$$\mathbf{Adv}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{ObvSimL}}(\mathcal{A}) = 2\Pr\left[\mathsf{Expt}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{ObvSimL}} = 1\right] - 1.$$

Leakage-resilient authenticity. Finally, leakage-resilient authenticity requires that the garbling leakage does not help in producing the "valid" garbled output $\tilde{Y}$ from (F, X). Formally, we introduce the experiment $\mathsf{Expt}^{\mathcal{G},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{AutL}}$ in Fig. 2 (Bottom) and define the advantage $\mathbf{Adv}^{\mathcal{G},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{AutL}}(\mathcal{A}) = \Pr\left[\mathsf{Expt}^{\mathcal{G},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{AutL}} = 1\right]$.

Informally, a garbling scheme $\mathcal{G}$ is $\mathsf{PrvSimL}$, resp. $\mathsf{ObvSimL}$ secure w.r.t the leakage function $\mathsf{L}_{\mathsf{Garble}}$, if for every efficient adversary $\mathcal{A}$ there exists an efficient simulator $\mathcal{S}$ such that $\mathbf{Adv}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}}(\mathcal{A})$, resp. $\mathbf{Adv}^{\mathcal{G},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{ObvSimL}}(\mathcal{A})$, is "sufficiently small". $\mathcal{G}$ is AutL secure w.r.t $\mathsf{L}_{\mathsf{Garble}}$, if $\mathbf{Adv}^{\mathcal{G},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{AutL}}$ is "sufficiently small" for every efficient adversary $\mathcal{A}$.

4 New Side-channel Weaknesses

As the Introduction mentions, Levi and Hazay [31] exhibited a concrete SCA against Free-XOR garbling schemes. In this section, we exhibit additional weaknesses in the other schemes, including a DPA-type attack in virtually all existing "non-free-XOR" schemes due to label reuse (in Sect. 4.1) and a weakness due to XOR leakages in GbAND functions (in Sect. 4.2).

4.1 DPA-type attacks due to label reuse

As discussed in the Introduction, if the to-be-garbled circuit f has a high fan-out (i.e., a wire connects to multiple gate inputs), then $\mathsf{Garble}(f)$ would (re)use the label of this wire to invoke E many times, and this enables a standard DPA-type attack. The function $f_{\mathrm{seq}}(\alpha, \gamma_1, ..., \gamma_\ell) = (\alpha \oplus \gamma_1, \alpha \oplus \gamma_2, ..., \alpha \oplus \gamma_\ell)$, which is used

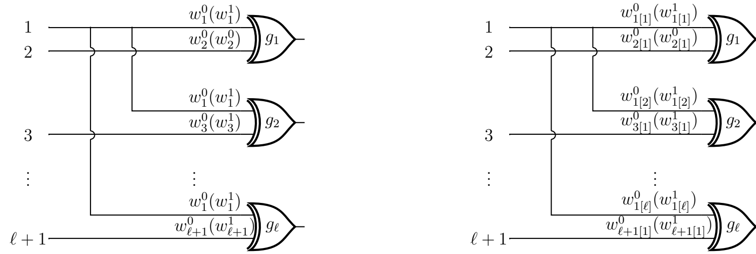

Fig. 3: Garbling $f_{\text{seq}}(\alpha, \gamma_1, ..., \gamma_\ell) = (\alpha \oplus \gamma_1, ..., \alpha \oplus \gamma_\ell)$, $\alpha, \gamma_1, ..., \gamma_\ell \in \{0,1\}$. All the ℓ XOR gates have their first input wires connected with 1. (Left) GLNP.Garble [17] assigns $\ell+1$ pairs of labels (w_i^0, w_i^1) to the $\ell+1$ input wires, so that (w_1^0, w_1^1) is used in ℓ calls to GLNP.GbXOR and can be recovered via a DPA-type attack. (Right) Our idea: our scheme GLNPLR view the ℓ gate input wires as *sub-wires* of 1, and assign ℓ pairs of *sub-labels* $(w_{1[1]}^0, w_{1[1]}^1), ..., (w_{1[\ell]}^0, w_{1[\ell]}^1)$ for the ℓ calls to GLNPLR.GbXOR to exclude the DPA.

in e.g., [19], to produce a sequence from an initial value α, is such a high fan-out function. For concreteness, here we consider garbling f_{seq} using the GLNP scheme and exhibit the attack.

We provide a high-level description of GLNP.Garble and refer to Fig. 4 for details. Roughly, GLNP.Garble begins by randomly choosing two n-bit strings (labels) w_i^0 and w_i^1 and a bit π_i for every input wire i. The garbled circuit is then generated gate-by-gate in topological order. For each NOT gate in the circuit with input wire a and output wire c (NOT gate only has one input wire and we write $b = \perp$), the garbler simply sets $w_c^0 = w_a^1$, $w_c^1 = w_a^0$ and $\pi_c = \overline{\pi}_a$. For each XOR gate (resp. AND gate) in the circuit with input wires a, b and output wire c, the garbler uses $w_a^0, w_a^1, w_b^0, w_a^0, \pi_a, \pi_b$ to compute the garbled table T (resp. (T_1, T_2, T_3)) as well as the labels w_c^0, w_c^1. This is done using a complicated procedure GbXOR or GbAND that is defined in Fig. 5. The garbled circuit consists of all the garbled AND gates and garbled XOR gates (garbling NOT gate only consists of three simple assignments, as shown in Fig. 4).

We now elaborate on the DPA against GLNP.Garble(f_{seq}). For simplicity, consider the case $\alpha, \gamma_1, ..., \gamma_\ell \in \{0,1\}$. Assuming that $\alpha, \gamma_1, ..., \gamma_\ell$ are input via wires $1, 2, ..., \ell+1$ respectively, and the XOR gate for $\alpha \oplus \gamma_i$ is indexed i, as illustrated in Fig. 3. Then, the execution of GLNP.Garble(f_{seq}) will make calls to GLNP.GbXOR($w_1^0, w_1^1, w_2^0, w_2^1, \pi_1, \pi_2$), ..., GLNP.GbXOR($w_1^0, w_1^1, w_{\ell+1}^0, w_{\ell+1}^1, \pi_1, \pi_{\ell+1}$) in turn. By design (see line 45 in Fig. 5), this induces ℓ calls to $E_{w_1^0}(1\|\pi_1)$, ..., $E_{w_1^0}(\ell\|\pi_1)$ and ℓ calls $E_{w_1^1}(1\|\overline{\pi}_1), ..., E_{w_1^1}(\ell\|\overline{\pi}_1)$. The labels w_1^0 and w_1^1 can thus be recovered via a standard DPA-type attack (as long as ℓ is large enough, e.g., hundreds to thousands).

It is crucial to highlight that the vulnerability of this attack does not depend on the details of the garbled scheme employed. The primary requirement for this attack is that the garbled scheme utilizes the input wire labels in the AES (or hash) encryption process. This usage provides the adversary $\mathcal{A}$ an opportu-

1: **procedure** Garble(f)
2: **for** $i \in$ Inputs **do**
3: $w_i^0, w_i^1 \xleftarrow{\$} \{0,1\}^n$, $\pi_i \xleftarrow{\$} \{0,1\}$, $e[i,0] := w_i^0 \| \pi_i$, $e[i,1] := w_i^1 \| \overline{\pi}_i$
4: $g := 0$
5: **for** $(a, b, c, G) \in$ Gates **do**
6: $g := g + 1$
7: PInputs $:= (w_a^0, w_a^1, w_b^0, w_b^1, \pi_a, \pi_b, g)$
8: **if** $G =$ XOR **then**
9: $(w_c^0, w_c^1, \pi_c, F[g]) :=$ GbXOR(PInputs)
10: **else if** $G =$ AND **then**
11: $(w_c^0, w_c^1, \pi_c, F[g]) :=$ GbAND(PInputs)
12: **else if** $G =$ NOT **then**
13: $w_c^0 := w_a^1$, $w_c^1 := w_a^0$, $\pi_c := \pi_a$
14: **for** $j \in$ Outputs **do**
15: $d[j,0] := E_{w_j^0}(out \| \pi_j)$, $d[j,1] := E_{w_j^1}(out \| \overline{\pi}_j)$
16: **return** (F, e, d)

17: **procedure** Eval(F, X)
18: $g := 0$
19: **for** $i \in$ Inputs **do** $w_i \| \lambda_i := X[i]$
20: **for** $(a[p], b[q], c, \ell_c, G) \in$ Gates **do**
21: $g := g + 1$
22: **if** $G =$ XOR **then**
23: $T := F[g]$, $\lambda_c := \lambda_a \oplus \lambda_b$
24: $w_c := E_{w_a}(g \| \lambda_a) \oplus E_{w_b}(g \| \lambda_b) \oplus \lambda_b \cdot T$
25: **else if** $G =$ AND **then**
26: $(T_1, T_2, T_3) := F[g]$
27: $w_c \| \lambda_c := T \oplus E_{w_a}(g \| \lambda_a \lambda_b) \oplus E_{w_b}(g \| \lambda_a \lambda_b)$
28: **else if** $G =$ NOT **then**
29: $w_c := w_a$, $\lambda_c := \lambda_a$
30: **for** $j \in$ Outputs **do**
31: $Y[j] := E_{w_j}(out \| \lambda_j)$
32: **return** Y

Fig. 4: GLNP garbling [17]. Its Encode and Decode procedures are the same as those in Fig. 6 (but are irrelevant to Sect. 4) and its GbAND and GbXOR are in Fig. 5 .

nity to gather leakage data and potentially recover the labels associated with a wire. Indeed, many garbled schemes, including notable examples such as the Yao scheme, GRR2, and GRR3, necessitate the participation of the input label in AES or hash function operations. Consequently, the attack described can be applied to these schemes as well.

33: **procedure** $\mathsf{GbAND}(w_a^0, w_a^1, w_b^0, w_b^1, \pi_a, \pi_b, g)$
34: $\quad W_0 := E_{w_a^{\pi_a}}(g\|00) \oplus E_{w_b^{\pi_b}}(g\|00)$
35: $\quad$ **if** $\pi_a = \pi_b = 1$ **then**
36: $\quad\quad w_c^0 \xleftarrow{\$} \{0,1\}^n,\ w_c^1\|\overline{\pi}_c := W_0$
37: $\quad$ **else**
38: $\quad\quad w_c^0\|\pi_c := W_0,\ w_c^1 \xleftarrow{\$} \{0,1\}^n$
39: $\quad W_c^0 := w_c^0\|\pi_c,\ W_c^1 := w_c^1\|\overline{\pi}_c$
40: $\quad T_1 := E_{w_a^{\pi_a}}(g\|01) \oplus E_{w_b^{\overline{\pi}_b}}(g\|01) \oplus W_c^{\pi_a \wedge \overline{\pi}_b}$
41: $\quad T_2 := E_{w_a^{\overline{\pi}_a}}(g\|10) \oplus E_{w_b^{\pi_b}}(g\|10) \oplus W_c^{\overline{\pi}_a \wedge \pi_b}$
42: $\quad T_3 := E_{w_a^{\overline{\pi}_a}}(g\|11) \oplus E_{w_b^{\overline{\pi}_b}}(g\|11) \oplus W_c^{\overline{\pi}_a \wedge \overline{\pi}_b}$
43: $\quad$ **return** $(w_c^0, w_c^1, \pi_c, (T_1, T_2, T_3))$

44: **procedure** $\mathsf{GbXOR}(w_a^0, w_a^1, w_b^0, w_b^1, \pi_a, \pi_b, g)$
45: $\quad \tilde{w}_a^0 := E_{w_a^0}(g\|\pi_a)[1 \cdots n],\ \tilde{w}_a^1 := E_{w_a^1}(g\|\overline{\pi}_a)[1 \cdots n]$
46: $\quad \pi_c := \pi_a \oplus \pi_b,\ \mathit{\Delta}_c := \tilde{w}_a^0 \oplus \tilde{w}_a^1$
47: $\quad$ **if** $\pi_b = 0$ **then**
48: $\quad\quad \tilde{w}_b^0 := E_{w_b}^0(g\|0)[1 \cdots n],\ \tilde{w}_b^{\overline{1}} := \tilde{w}_b^1 \oplus \mathit{\Delta}_c$
49: $\quad$ **else**
50: $\quad\quad \tilde{w}_b^1 := E_{w_b}^1(g\|0)[1 \cdots n],\ \tilde{w}_b^0 := \tilde{w}_b^1 \oplus \mathit{\Delta}_c$
51: $\quad T := E_{w_b^{\overline{\pi}_b}}(g\|1)[1 \cdots n] \oplus \tilde{w}_b^{\pi_b}$
52: $\quad$ **return** (w_c^0, w_c^1, π_c, T)

Fig. 5: The procedure GbAND and GbXOR of GLNP garbling [17].

A natural idea to remedy this weakness is to generate new labels for the connected gate input wires, as illustrated in Fig. 3 (Right). We refer to Sect. 5.1 and 5.2 for details.

Practical issues. The DPA-type attacks in this section rely on the existence of high fan-out wires. This *is* common in popular MPC tasks, including (matrix) multiplication, division, private set intersection, etc., and we list several examples in Table 1 to illustrate. It can be seen many wires have fan-out exceeding 1000, and there are thousands of wires with fan-out ≥ 200.

Although high fan-out wires do not account for a high proportion in these circuits, the attack remains devastating: the recovery of two labels of any wire not only breaks privacy in theory, but also helps deduce the other wire labels in the circuit. Moreover, in many cases including Array Search, multiplication, division and encryption, high fan-out wires are typically input wires. The compromise of such wires thus recovers the circuit input, which is the searched element in Array Search, the operator in multiplication/division and the key in encryption. In summary, DPA-type attacks due to high fan-out circuits are common and devastating.

Circuit	Wires	fan-out $\geq$ 200	fan-out $\geq$ 500	fan-out $\geq$ 1000
Multiplication	3140617	1649	1049	49
Division	4220908	1754	1304	554
Matrix multiplication	24954241	13185	8449	385
Exponential Modulus	41003946	7931	5981	2725
Private Set Intersection	50544545	52801	33601	1601
Longest-Common-Subsequence	36035250	8192	8192	0
Array Search	154661	32	32	32
Edit Distance	60762581	8192	8192	0
AES-CBC Encryption	29356000	1408	1408	1408

Table 1: Example circuits with high fan-out wires. The column "fan-out $\geq N$" indicates the number of wires with fan-out $\geq N$ in the corresponding circuit. Multiplication and division operations are performed on 1024-bit data. Matrix multiplication is performed on 128-bit data, and the matrix size is 8×8. The base of the exponential modulus operation is 1024 bits, and the exponent is 3 bits. The set size in the private set intersection is 1024, and the data size is 32 bits. The array size in the array search is 1024, and the data size is 32 bits. The length of the two sequences in the Longest-Common-Subsequence (LCS) is 1024, and the data size in the sequence is 8 bits. The length of each sequence in the edit distance is 1024, and the data size is 8 bits. AES-CBC Encryption refers to the AES blockcipher in CBC mode, where the number of encrypted message blocks is 1000.

4.2 Recovering labels using XOR leakages

Our second observation is a side-channel weakness in a series of garbling schemes if the XOR computations in their GbAND procedures leak non-trivial information. Again, we focus on GLNP [17] here—or its subprocedure $\mathsf{GLNP.GbAND}$ (described in Fig. 4) which used the GRR3 technique with optimizations (we will detail similar weaknesses in other schemes in Appendix B). Consider an adversary $\mathcal{A}$ collecting computation leakages from an execution of $\mathsf{GbAND}(w_a^0, w_a^1, w_b^0, w_b^1, \pi_a, \pi_b, g)$.

Following the semi-honest model of garbling, we assume that $\mathcal{A}$ corrupts (or colludes with) the evaluator. This means $\mathcal{A}$ has all the values $w_1, ..., w_{n+g}$ computed during GLNP.$\mathsf{Eval}(F, X)$. By the design of GLNP, for every $i \in \{1, .., n+g\}$, it actually has $w_i = w_i^{v_i}$ for an unknown bit $v_i \in \{0, 1\}$. Namely, for every wire i $\mathcal{A}$ "knows" one of w_i^0 and w_i^1, but it does not know which one it has—and we say *$\mathcal{A}$ breaks wire i if it determines v_i* (probably using the leakages).

With the above, consider an execution of $\mathsf{GbAND}(w_a^0, w_a^1, w_b^0, w_b^1, \pi_a, \pi_b, g)$. We found two issues as follows.

XOR leaks the result. First, at line 34, $\mathcal{A}$ would collect one leakage:

$$\mathsf{L}_{\oplus}(S_1, S_2): \text{ where } S_1 = E_{w_a^{\pi_a}}(00) \text{ and } S_2 = E_{w_b^{\pi_b}}(00), \quad (1)$$

where $\mathsf{L}_\oplus(a,b)$ is the leakage of $a \oplus b$ by our convention (see Sect. 2).

Simplified case: XOR leaks 1 bit of result. First, consider the case where the leakages leaked $\mathsf{msb}(W_0)$ $(S_1 \oplus S_2 = W_0)$. In GLNP.GbAND, the value W_0 computed at line 34 fulfills $W_0[1 \cdots n] = w_c^{\pi_a \wedge \pi_b}$ (this can be seen from the design). $\mathcal{A}$ compares $\mathsf{msb}(W_0)$ and $\mathsf{msb}(w_c^{v_c})$: $\mathcal{A}$ guesses $v_c = 1$ if $\mathsf{msb}(W_0) \neq \mathsf{msb}(w_c^{v_c})$, and guess $v_c = 0$ otherwise. This breaks wire c.

The main idea is that since $W_0[1 \cdots n] = w_c^{\pi_a \wedge \pi_b}$, $\mathsf{msb}(W_0) \neq \mathsf{msb}(w_c^{v_c})$ necessarily implies $\pi_a \wedge \pi_b \neq v_c$. Since $\pi_a, \pi_b \xleftarrow{\$} \{0,1\}$ are independently sampled in GLNP.Garble, $\Pr\big[\pi_a \wedge \pi_b = 0\big] = 3/4$. Therefore, it likely holds $v_c = 1$ in this case. Similarly, it likely holds $v_c = \pi_a \wedge \pi_b = 0$ in the other case. We present a detailed analysis of the success probability in Appendix B.1.

General case. Generally, assume that $\mathcal{A}$ can determine if a given value equals the result of an XOR operation $S_1 \oplus S_2$, using the leakage of the XOR. More formally, $\mathcal{A}$ could distinguish the pair $\big(\mathsf{L}_\oplus(S_1,S_2), S_1 \oplus S_2\big)$ from $\big(\mathsf{L}_\oplus(S_1,S_2), R\big)$, where S_1 or S_2 is an unknown secret and R is a random value independent from $S_1 \oplus S_2$. The above attack strategy easily extends to this general case.

XOR leaks the operand. Second, at line 5, $\mathcal{A}$ would collect three leakages:

$$\begin{aligned}
&\mathsf{L}_\oplus(S_1, W_c^{\pi_a \wedge \bar{\pi}_b}), \text{ where } S_1 = E_{w_a^{\pi_a}}(g\|01) \oplus E_{w_b^{\pi_b}}(g\|01),\\
&\mathsf{L}_\oplus(S_2, W_c^{\bar{\pi}_a \wedge \pi_b}), \text{ where } S_2 = E_{w_a^{\pi_a}}(g\|10) \oplus E_{w_b^{\pi_b}}(g\|10),\\
&\mathsf{L}_\oplus(S_3, W_c^{\bar{\pi}_a \wedge \bar{\pi}_b}), \text{ where } S_3 = E_{w_a^{\pi_a}}(g\|11) \oplus E_{w_b^{\pi_b}}(g\|11).
\end{aligned}$$

Simplified case: XOR leaks 1 bit of operand. First, consider the case where the three leakages leaked $\mathsf{msb}(W_c^{\pi_a \wedge \bar{\pi}_b}), \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \pi_b})$ and $\mathsf{msb}(W_c^{\bar{\pi}_a \wedge \bar{\pi}_b})$. $\mathcal{A}$ does not know values $W_c^{\pi_a \wedge \pi_b}$, $W_c^{\pi_a \wedge \bar{\pi}_b}$, $W_c^{\bar{\pi}_a \wedge \pi_b}$ and $W_c^{\bar{\pi}_a \wedge \bar{\pi}_b}$, but three of them are identical and are likely distinct from the fourth (since the four values $\pi_i \wedge \pi_j, \pi_i \wedge \bar{\pi}_j, \bar{\pi}_i \wedge \pi_j$ and $\bar{\pi}_i \wedge \bar{\pi}_j$ consist of three 0 and one 1).

When $\mathsf{msb}(W_c^0) \neq \mathsf{msb}(W_c^1)$ (the probability is $\approx 1/2$), $\mathcal{A}$ observes one of the follow two cases:

1. When $\pi_a \wedge \bar{\pi}_b = \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$, $\mathcal{A}$ observes $\mathsf{msb}\ (W_c^{\pi_a \wedge \bar{\pi}_b}) = \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \pi_b}) = \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \bar{\pi}_b})$. Then it necessarily be $\pi_a \wedge \bar{\pi}_b = ...\bar{\pi}_a \wedge \bar{\pi}_b = 0$ and $\mathcal{A}$ has $\mathsf{msb}(W_c^0)$. As discussed, $\mathcal{A}$ has the label $w_c^{v_c}$ with $v_c \in \{0,1\}$ by corrupting the evaluator. By comparing $\mathsf{msb}(W_c^0)$ with $\mathsf{msb}(w_c^{v_c})$, it could determine v_c and break wire c.
2. Otherwise, wlog assume $\pi_a \wedge \bar{\pi}_b \neq \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$. Then, it necessarily be $\pi_a \wedge \bar{\pi}_b = 1$ and $\bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b = 0$, and $\mathcal{A}$ observes $\mathsf{msb}(W_c^1)$ once while $\mathsf{msb}(W_c^0)$ twice from the leakages. $\mathcal{A}$ could then determine $\mathsf{msb}(W_c^0)$ and break the c-th wire as in the previous case.

We present a detailed analysis of the success probability in Appendix B.1.

General case. It can be seen that the above attack strategy easily extends to the general case where $\mathcal{A}$ could distinguish $\mathsf{L}_\oplus(S, W_c^0)$ from $\mathsf{L}_\oplus(S^*, W_c^1)$ (instead of

leaking 1 bit), where S and S^* are arbitrary unknown secrets. Anyway, as long as $\mathcal{A}$ can determine if $\pi_a \wedge \bar{\pi}_b = \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$ or not, the same attack follows.

Discussion. In practice, leakages of uncareful XOR implementations do satisfy our assumption. E.g., it is common that computing $W_0 := S_1 \oplus S_2$ leaks the Hamming weight of W_0, which enables distinguishing. Given the leakage trace of a Garble execution, it requires non-trivial efforts to extract the above leakages (but this may be feasible with the help of recent machine learning techniques, and is an interesting future work). Importantly to us, provable security does become vague without assumptions on XOR leakages, and this provides a valuable guidance by emphasizing caution on XORs in cryptographic engineering. In Sect. 5, we will follow the approach of [38] to characterize the influence of XOR leakages on the leakage security of the whole scheme.

4.3 The leakage of Permute bits

The permute bit is the secret value in garbled circuits and cannot be leaked. Thus, if the operation of permute bits leaks the permute bit, the security will be broken.

1. In GLNP.GbXOR, it computes $\pi_c = \pi_a \oplus \pi_b$. If this XOR leaks any bits π_a, π_b and π_c then $\mathcal{A}$ breaks the corresponding wire.
2. In GLNP.GbXOR, it computes $\pi_c = \pi_a \oplus \pi_b$. If this XOR leaks any bits π_a, π_b and π_c then $\mathcal{A}$ breaks the corresponding wire.

Discussion. At a theoretical level, it is possible to extract information about the permuted bits using leakage. However, in practice, permute bits typically involve only 1-bit XOR or AND operations, making it difficult to extract information about the permuted bits using this leakage. In our subsequent proof, we assume these operations are strongly protected and leak-free.

5 Improved GLNP variant and Its Leakage-resilience

We first introduce a refined model of circuits with explicit sub-wires in Sect. 5.1. With these additional preliminaries, we describe our improved GLNP variant GLNPLR in Sect. 5.2. We then present our leakage model in Sect. 5.3 and leakage assumptions regarding blockcipher evaluations and XORs in Sect. 5.4 and 5.5 resp. After these, we prove leakage-resilience of GLNPLR in Sect. 5.6.

5.1 General settings

Circuit with explicit sub-wires. As discussed in Sect. 4.1, we need to assign more than one label for the high fan-out setting. The fan-out information is reflected by the topological structure of a circuit, but we need to refine the classical circuit model in Sect. 2 to make it explicit. To this end, when a wire i is connected to ℓ gate input wires, we call the latter *sub-wires* of wire i, and

we number them $i[1], ..., i[\ell]$ (as shown in Fig. 3 Right). With this, we refine our circuit model as a tuple $f = (\ell_{in}, \ell_{out}, g_1, g_2, g_3, \mathsf{Gates}, \mathsf{Wires})$ consisting of $\ell_{in} \geq 2$ input wires, $\ell_{out} \geq 1$ output wires, g_1 AND gates, g_2 XOR gates and g_3 NOT gates. Let $g = g_1 + g_2 + g_3$. It thus has $\ell_{in} + g$ wires numbered $1, ..., \ell_{in} + g$. We let $\mathsf{Inputs} = \{1, \ldots, \ell_{in}\}$ and $\mathsf{Outputs} = \{\ell_{in}+g-\ell_{out}+1, \ldots, \ell_{in}+g\}$ denoting the sets of input and output wires. The set $\mathsf{Gates} = \big\{(a[p], b[q], c, G)\big\}$ containing g tuples, specifies the wiring of the circuit; a tuple $(a[p], b[q], c,\ G) \in \mathsf{Gates}$ with $a, b, c \in \{1, \ldots, \ell_{in} + g\}$ represents a gate of type $G \in \{\mathsf{XOR}, \mathsf{AND}, \mathsf{NOT}\}$ with input sub-wires $a[p], b[q]$ and output wire c ($b[q] = \perp$ if $G = \mathsf{NOT}$). The set Wires containing $(\ell_1, \ldots, \ell_{n+g})$ where ℓ_i represents the sub-wires number of wire i.

5.2 Description of GLNPLR

Our scheme is built upon a blockcipher $E : \{0,1\}^n \times \{0,1\}^{n+1} \to \{0,1\}^{n+1}$, and is formally described in Fig. 6. As mentioned in the Introduction, our procedures GbXOR and GbAND generally follow [17], and our novelty lies in assigning independent wire labels to all gate input wires to avoid the issue due to high fan-out circuits. Below we provide high-level descriptions for Garble, GSL, GbXOR, GbAND and Eval in turn.

Overview of Garble. Our Garble algorithm begins by assigning wire labels for the input wires. In detail, for every input wire i:

1. Garble first randomly pick a permutation bit π_i and a pair of n-bit wire labels $w_i^{\pi_i}, w_i^{\overline{\pi}_i}$. The use of permutation bit π_i is intended to apply the point-and-permutation method [37] to reduce garbled table size. As will be seen, the bit π_i also "fixes" the order of computations with w_i^0 and w_i^1. Namely, during addressing wire i, computations with $w_i^{\pi_i}$ always go ahead of those with $w_i^{\overline{\pi}_i}$ (e.g., see Fig. 6. Implementations of GLNPLR have to follow this order as well.). In contrast, traditional designs seldom enforce such a fixed order of computations. This "order-fixing" is crucial for our simulator (it needs to know which label is "faked"), and we believe it also improves the leakage property of the scheme.
2. Garble then generates the associated "sub-labels" using GSL. Concretely, assume that the (input) wire i is connected to ℓ_i different gate inputs. Garble then uses $w_i^{\pi_i}$ and $w_i^{\overline{\pi}_i}$ as seeds to invoke the leakage-resilient PRG GSL to produce two sequence of ℓ_i "sub-labels".

Then, Garble generates the garbled circuit F gate-by-gate in topological order. Concretely, for every gate $(a[p], b[q], c, G)$ in circuit f, with input sub-wires $a[p]$, $b[q]$ and output wire c, it uses input sub-labels $w_{a[p]}^{\pi_a}$, $w_{a[p]}^{\overline{\pi}_a}$, $w_{b[q]}^{\pi_b}$, $w_{b[q]}^{\overline{\pi}_b}$ and input permutation bits π_a, π_b to compute the output sub-labels $w_c^{\pi_c}$, $w_c^{\overline{\pi}_c}$ permutation bit π_c as well as the garbled table which is: (i) T if $G = \mathsf{XOR}$; (ii) T_1, T_2, T_3 if $G = \mathsf{AND}$; or (iii) $\perp$ if $G = \mathsf{NOT}$.Then, for every output label c, it generates the associated "sub-labels" using GSL.

Generate sub-labels (procedure GSL). GSL, as illustrated in Fig. 8, is a blockcipher-based leakage-resilient PRG of Yu et al. [45,38]. It takes a seed k_0,

procedure $\mathsf{Garble}(f)$
  for $i \in \mathsf{Inputs}$ **do**
    $w_i^{\pi_i} \xleftarrow{\$} \{0,1\}^n, w_i^{\overline{\pi}_i} \xleftarrow{\$} \{0,1\}^n,\ \pi_i \xleftarrow{\$} \{0,1\}$
    $e[i, \pi_i] := w_i^{\pi_i} \| 0, e[i, \overline{\pi}_i] := w_i^{\overline{\pi}_i} \| 1$
    $(w_{i[0]}^{\pi_i}, \cdots, w_{i[\ell_i]}^{\pi_i}) := \mathsf{GSL}(w_i^{\pi_i}, \ell_i)$
    $(w_{i[0]}^{\overline{\pi}_i}, \cdots, w_{i[\ell_i]}^{\overline{\pi}_i}) := \mathsf{GSL}(w_i^{\overline{\pi}_i}, \ell_i)$
  for $(a[p], b[q], c, G) \in \mathsf{Gates}$ **do**
    $\mathsf{PInputs} := (w_{a[p]}^{\pi_a}, w_{a[p]}^{\overline{\pi}_a}, w_{b[q]}^{\pi_b}, w_{b[q]}^{\overline{\pi}_b}, \pi_a, \pi_b)$
    if $G = \mathsf{XOR}$ **then**
      $(w_c^{\pi_c}, w_c^{\overline{\pi}_c}, \pi_c, F[c]) := \mathsf{GbXOR}(\mathsf{PInputs})$
    else if $G = \mathsf{AND}$ **then**
      $(w_c^{\pi_c}, w_c^{\overline{\pi}_c}, \pi_c, F[c]) := \mathsf{GbAND}(\mathsf{PInputs})$
    else if $G = \mathsf{NOT}$ **then**
      $w_c^{\pi_c} = w_{a[p]}^{\overline{\pi}_a},\ w_c^{\overline{\pi}_c} = w_{a[p]}^{\pi_a},\ \pi_c = \pi_a$
    $(w_{c[0]}^{\pi_c}, \cdots, w_{c[\ell_c]}^{\pi_c}) := \mathsf{GSL}(w_c^{\pi_c}, \ell_c)$
    $(w_{c[0]}^{\overline{\pi}_c}, \cdots, w_{c[\ell_c]}^{\overline{\pi}_c}) := \mathsf{GSL}(w_c^{\overline{\pi}_c}, \ell_c)$
  for $j \in \mathsf{Outputs}$ **do**
    $d[j, \pi_j] := E_{w_j^{\pi_j}}(0),\ d[j, \overline{\pi}_j] := E_{w_j^{\overline{\pi}_j}}(1)$
  return (F, e, d)

procedure $\mathsf{Eval}(F, X)$
  for $i \in \mathsf{Inputs}$ **do**
    $w_i \| \lambda_i := X[i]$
    $(w_{i[0]}, \cdots, w_{i[\ell_i]}) := \mathsf{GSL}(w_i, \ell_i)$
  for $(a[p], b[q], c, G) \in \mathsf{Gates}$ **do**
    if $G = \mathsf{XOR}$ **then**
      $T := F[c]$
      $\hat{w}_c := E_{w_{a[p]}}(\lambda_a)[1 \cdots n] \oplus E_{w_{b[q]}}(\lambda_b)[1 \cdots n] \oplus \lambda_b \cdot T$
      $\lambda_c := \lambda_a \oplus \lambda_b,\ w_c := E_{\hat{w}_c}(\lambda_c)[1 \cdots n]$
    else if $G = \mathsf{AND}$ **then**
      $(T_1, T_2, T_3) := F[c]$
      $w_c \| \lambda_c := T \oplus E_{w_{a[p]}}(\lambda_a \lambda_b) \oplus E_{w_{b[q]}}(\lambda_a \lambda_b)$
    else
      $w_c = w_{a[p]},\ \lambda_c = \lambda_a$
    $(w_{c[0]}, \cdots, w_{c[\ell_c]}) := \mathsf{GSL}(w_c, \ell_c)$
  for $j \in \mathsf{Outputs}$ **do**
    $Y[j] := E_{w_j}(\lambda_j)$
  return Y

Fig. 6: our garbling scheme GLNPLR (continued in Fig. 7)

which is supposed to be a label of some wire c, and an integer ℓ as inputs, uses two public distinct constants $P_A, P_B \in \{0,1\}^{n+1}$, and makes (roughly) $\ell - 1$ pairs of calls to E to generate $\ell - 1$ pseudorandom strings as sub-labels for $\ell - 1$

procedure $\mathsf{GSL}(k_0, \ell)$
 for $i := 1$ **to** $\ell - 1$ **do**
  $w_i := E_{k_{i-1}}(P_B)[1 \cdots n]$
  $k_i := E_{k_{i-1}}(P_A)[1 \cdots n]$
 $w_\ell := k_{\ell-1}$
 return $(w_1, \cdots, w_\ell)$

procedure $\mathsf{Encode}(e, x)$
 for $i := 1$ to $|x|$ **do**
  $X[i] := e[i, x_i]$
 return (X)

procedure $\mathsf{GbXOR}(w_a^{\pi_a}, w_a^{\overline{\pi}_a}, w_b^{\pi_b}, w_b^{\overline{\pi}_b}, \pi_a, \pi_b)$
 $\Delta_a := w_a^{\pi_a} \oplus w_a^{\overline{\pi}_a}$, $\Delta_b := w_b^{\pi_b} \oplus w_b^{\overline{\pi}_b}$
 $\pi_c := \pi_a \oplus \pi_b$, $T := \Delta_a \oplus \Delta_b$
 $\hat{w}_c^{\pi_c} := w_a^{\pi_a} \oplus w_b^{\pi_b}$, $\hat{w}_c^{\overline{\pi}_c} := w_a^{\overline{\pi}_a} \oplus w_b^{\pi_b}$
 $w_c^{\pi_c} := E_{\hat{w}_c^{\pi_c}}(0)[1 \cdots n]$
 $w_c^{\overline{\pi}_c} := E_{\hat{w}_c^{\overline{\pi}_c}}(1)[1 \cdots n]$
 return $(w_c^{\pi_c}, w_c^{\overline{\pi}_c}, \pi_c, T)$

procedure $\mathsf{GbAND}(w_a^{\pi_a}, w_a^{\overline{\pi}_a}, w_b^{\pi_b}, w_b^{\overline{\pi}_b}, \pi_a, \pi_b)$
 $W_0 := E_{w_a^{\pi_a}}(00) \oplus E_{w_b^{\pi_b}}(00)$
 if $\pi_a = \pi_b = 1$ **then**
  $w_c^0 \xleftarrow{\$} \{0,1\}^n$, $w_c^1 \| \overline{\pi}_c := W_0$
 else
  $w_c^0 \| \pi_c := W_0$, $w_c^1 \xleftarrow{\$} \{0,1\}^n$
 $W_c^0 := w_c^0 \| \pi_c$, $W_c^1 := w_c^1 \| \overline{\pi}_c$
 $T_1 := E_{w_a^{\pi_a}}(01) \oplus E_{w_b^{\overline{\pi}_b}}(01) \oplus W_c^{g(\pi_a, \overline{\pi}_b)}$
 $T_2 := E_{w_a^{\overline{\pi}_a}}(10) \oplus E_{w_b^{\pi_b}}(10) \oplus W_c^{g(\overline{\pi}_a, \pi_b)}$
 $T_3 := E_{w_a^{\overline{\pi}_a}}(11) \oplus E_{w_b^{\overline{\pi}_b}}(11) \oplus W_c^{g(\overline{\pi}_a, \overline{\pi}_b)}$
 return $(w_c^{\pi_c}, w_c^{\overline{\pi}_c}, \pi_c, (T_1, T_2, T_3))$

procedure $\mathsf{Decode}(Y, d)$
 for i := 1 to $|Y|$ **do**
  if $Y[i] := d[i, 0]$ **then** $y[i] := 0$
  else if $Y[i] := d[i, 1]$ **then**
   $y[i] := 1$
  else return $\perp$
 return y

Fig. 7: our garbling scheme GLNPLR (continued from Fig. 6)

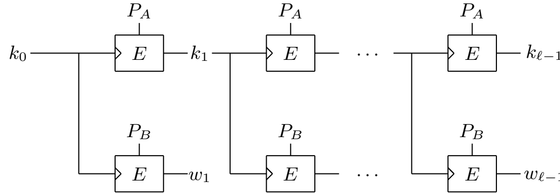

Fig. 8: The structure of $\mathsf{GSL}(k_0, \ell)$, where P_A and P_B are two arbitrary distinct constants.

gate input wires connected with wire c (In addition, we let the $\ell - 1$th sub-key to be w_ℓ). Importantly, the cipher key is kept refreshed between every pair of calls to E to avoid leaking too much.

Garble XOR gate (procedure GbXOR). We basically follow the GbXOR of GLNP [17]. However, we enforce an order (which slightly deviates from [17]) for the sequence of operations to refine its leakage property. Concretely, GbXOR proceeds as follows.

1. Compute the offset of sub-wires a and sub-wire b: $\Delta_a = w_a^{\pi_a} \oplus w_a^{\overline{\pi}_a}$, $\Delta_b = w_b^{\pi_b} \oplus w_b^{\overline{\pi}_b}$
2. Set the ciphertext: $T = \Delta_a \oplus \Delta_b$ and set $\pi_c = \pi_a \oplus \pi_b$
3. Compute output pre-labels on wire c: $\hat{w}_c^{\pi_c} = w_a^{\pi_a} \oplus w_b^{\pi_b}$ and $\hat{w}_c^{\overline{\pi}_c} = w_a^{\overline{\pi}_a} \oplus w_b^{\pi_b}$
4. Compute output labels on wire c: $w_c^{\pi_c} = E_{\hat{w}_c^{\pi_c}}(0)$, $w_c^{\overline{\pi}_c} = E_{\hat{w}_c^{\overline{\pi}_c}}(1)$

Garble AND gate. We adopt the Garbled Row Reduction (GRR3) approach from Naor, Pinkas, and Sumner [37] (also used in [17]), to decrease garbled gates from four to three ciphertexts by setting the first ciphertext to zero. Further details are provided in Fig 7.

Garble NOT gate. We follow the NOT gate in [17], which can still be computed for free without using free-XOR. Consider an NOT gate g with input wire a and output wire c, we can simply define $w_c^0 := w_a^1$ and $w_c^1 := w_a^0$. During the garbling of the circuit, gates receiving wire c as input will use these "reversed" values. Furthermore, when evaluating the circuit, if the value k_a^0 is given on wire a, then the result of the NOT gate is k_c^1 which *equals* k_a^0. Thus, nothing needs to be done.

Evaluation Eval. To evaluate a garbled circuit, starting with labels $\{w_i\}_{i \in \mathsf{Inputs}}$ (where the evaluator does not know if $w_i = w_i^0$ or $w_i = w_i^1$) and signal bit λ_i, the evaluator proceeds as follows. He computes the input sub-labels using procedure GSL at first. Then, the garbled circuit is evaluated gate-by-gate in topological order. For an XOR or AND gate with input sub-wires a, b, output wire c, and ℓ_c output sub-wires, the evaluator computes the output sub-label w_c using the corresponding garbled table (see Fig. 6) and then use procedure GSL to generate the according sub-labels)

5.3 Modeling leakages

Following [45,8], we model the blockcipher E as a (publicly accessible) ideal cipher (so we are in the ideal cipher model), and the leakages as *probabilistic efficient functions* manipulating and/or computing (partially) secret values. Concretely, each computation of E (resp. $\oplus$) comes with some additional (internal) information given by L_E (resp. $\mathsf{L}_\oplus$). We model the leakage as a leak list generated by *probabilistic efficient* functions manipulating and/or computing (partially) secret values. Such leakages might contain "future" calls to E and enable the well-known "future computation attack" [14,45]. To overcome, we follow Yu et al. [45,8] and assume oracle-free leakage functions, i.e., the leakage

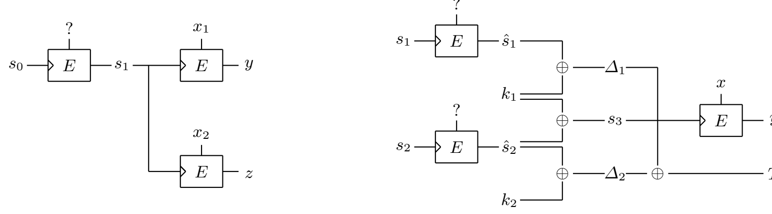

Fig. 9: Values and E-calls involved in: (Left) the 2-up$[q]$ assumption; (Right) the XOR-inv$[q]$ assumption.

functions cannot call E, since it is natural for an implementation not to evaluate computations that are unrelated to its current state.

Furthermore, we split the leakage resulting from a leaky execution of the blockcipher E between its *input* and *output* parts: if $E_k(x) \to y$, $L_E(k,x,y) := (\mathsf{L}_E^{in}(k,x), \mathsf{L}_E^{out}(k,y))$. For XOR leakage, we define $\widehat{\mathsf{L}_\oplus}(m_0,m_1) = [\mathsf{L}_\oplus(m_0,m_1), \mathsf{L}_\oplus(m_1,m_0)]$ for simplicity.

As discussed in [45], this leakage model appears to have a natural correspondence with concrete attacks on circuits implementing blockciphers, where the measured leakages can be interpreted as a simple function of the cipher's input and key during the first few rounds of the computation, and/or as a simple function of the cipher's output and key during the last few rounds of the computation, but where any useful function of the cipher input and output remains elusive (unless the implementation is fully broken).

5.4 Non-invertible leakage assumptions

We still have to limit the amount of information in leakages, which is orthogonal to oracle-freeness. To this end, we also follow [45,8] and assume leakages are non-invertible, i.e., it is hard to recover a secret internal state from a small number of relevant leakages. We give two concrete definitions.

2-up$[q]$ assumption. We first adapt the "2-up$[q]$" assumption of [8] to our setting. Concretely, we define

$$\mathbf{Adv}^{\text{2-up}[q]}(\mathcal{A}) := \Pr_{E,s_1,r}\big(y = E_{s_1}(x_1), z = E_{s_1}(x_2), \\ \text{Guesses} \leftarrow \mathcal{A}^E(y,z,\mathsf{leak}) : s_2 \in \text{Guesses}\big), \quad (2)$$

where $|\text{Guesses}| = q$, and $\mathcal{A}^E$'s input leak depends on values s_0, x_1, x_2 specified by $\mathcal{A}^E$:

$$\mathsf{leak} = \big[\mathsf{L}^{out}(s_0, s_1\|r), \mathsf{L}^{in}(s_1,x_1), \mathsf{L}^{out}(s_1,y), \mathsf{L}^{in}(s_1,x_2), \mathsf{L}^{out}(s_1,z)\big]$$

This emulates the real-world scenario in which a blockcipher-call $E_{s_0}(\star) \to s_1\|r$ yields (nearly) uniform output $s_1\|r$ where s_1 is the key of two other blockcipher-calls (see Fig. 9 Left), and the leakage of these *three* calls may contain information

Fig. 10: Values and E-calls involved in: (Left) the XOR-ope assumption, and (Right) the XOR-res assumption.

about the target secret s_1. $\mathcal{A}^E$ then outputs a set of q guesses and wins as long as s_1 is in these guesses. Note that we follow [8] and allow $\mathcal{A}^E$ to choose the "previous" key s_0 for composability purposes in the security proof.

XOR-inv[q] assumption. For the second assumption, we define

$$\begin{aligned}&\mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q]}(\mathcal{A}) = \qquad (3)\\ &\Pr_{E,\hat{s_1},\hat{s_2},r,r'}\big(\varDelta_1 = \hat{s}_1 \oplus k_1, \varDelta_2 = \hat{s}_2 \oplus k_2, T \leftarrow \varDelta_1 \oplus \varDelta_2, s_3 \leftarrow (k_1 \oplus \hat{s}_2),\\ &\qquad y \leftarrow E_{s_3}(x), \text{Guesses} \leftarrow \mathcal{A}^E(T, y, \mathsf{leak}) : s_3 \in \text{Guesses})\big),\end{aligned}$$

where $|\text{Guesses}| = q$, and $\mathcal{A}^E$'s input leak depending on values k_1, s_1, k_2, s_2, x specified by $\mathcal{A}^E$:

$$\begin{aligned}\mathsf{leak} = \big[&\mathsf{L}^{out}(s_1, \hat{s}_1\|r), \mathsf{L}^{out}(s_2, \hat{s}_2\|r'), \widehat{\mathsf{L}_\oplus}(\hat{s}_1, k_1), \widehat{\mathsf{L}_\oplus}(\hat{s}_2, k_2),\\ &\mathsf{L}_\oplus(\varDelta_1, \varDelta_2), \widehat{\mathsf{L}_\oplus}(k_1, \hat{s}_2), \widehat{\mathsf{L}_\oplus}(\hat{s}_1, \hat{s}_2), \mathsf{L}^{in}(s_3, x), \mathsf{L}^{out}(s_3, y)\big]. \qquad (4)\end{aligned}$$

We refer to Fig. 9 (Right) for the involved values and operations. Recall from Sect. 5.3 that $\widehat{\mathsf{L}_\oplus}(m_0, m_1) = [\mathsf{L}_\oplus(m_0, m_1), \mathsf{L}_\oplus(m_1, m_0)]$.

Unstanding XOR-inv[q] Assumption. Eq. (3) defines a leakage property of a small "unit" of XOR gate in the garbled scheme. Concretely, it captures that the temporary key (secret value) s_3 cannot be recovered from the involved leakages.

Although this assumption appears complex, the probability of an adversary breaking this assumption is similar to breaking the 2-up[q] assumption. This is because the leakage from XOR operations has a much smaller impact on adversary attacks compared to leakage from blockcipher operations. Therefore, we argue that this assumption is easy to satisfy in the real world despite its apparent complexity.

5.5 Limiting the leakages of the XOR operation

As discussed in Sect. 4.2, without limitation on the leakages from the XOR operations, security of GbAND could not be achieved. For this, we adapt the "LORL2" assumption from [8], which (informally) captures that the XOR operations do not leak "too much" information about their operands. We also give a new assumption XOR-res to capture that the XOR operations do not leak "too much" about their results.

XOR-ope assumption

$$\mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(\mathcal{A}) := \tag{5}$$
$$\Big|\Pr_{E,s_1}[y_0 \leftarrow s_1 \oplus m_0^0 \oplus m_1^0 : \mathcal{A}^E(y_0, \mathsf{leak}_0) \Rightarrow 1] -$$
$$\Pr_{E,s_1}[y_1 \leftarrow s_1 \oplus m_0^1 \oplus m_1^1 : \mathcal{A}^E(y_1, \mathsf{leak}_1) \Rightarrow 1]\Big|$$

where leak_b again depends on values s_0, m_0^b, m_1^b specified by $\mathcal{A}^E$ (Note that s_0 is n bits and s_1, m_0^b, m_1^b are $n+1$ bits).

$$\mathsf{leak}_b = \left[\mathsf{L}^{out}(s_0, s_1), \mathsf{L}_{\oplus}(s_1, m_0^b, m_1^b), \mathsf{L}_{\oplus}(m_0^b, s_1, m_1^b)\right]$$

XOR-res assumption

$$\mathbf{Adv}^{\mathsf{XOR\text{-}res}}(\mathcal{A}) := \Big|\Pr_{E,s_1}[y \leftarrow s_1 \oplus m_0 : \mathcal{A}^E(y, \mathsf{leak}) \Rightarrow 1] -$$
$$\Pr_{E,s_1}[R \xleftarrow{\$} \{0,1\}^{n+1} : \mathcal{A}^E(R, \mathsf{leak}) \Rightarrow 1]\Big| \tag{6}$$

where leak again depends on values s_0, m_0 specified by $\mathcal{A}^E$. (Note that s_0 is n bits and s_1, m_0 are $n+1$ bits).

$$\mathsf{leak} = \left[\mathsf{L}^{out}(s_0, s_1), \mathsf{L}_{\oplus}^{b}(s_1, m_0)\right]$$

Understanding XOR-ope and XOR-res assumptions. Intuitively,

- The XOR-ope assumption is adapted from "LORL2" in [8]. Intuitively, the XOR-ope Assumption captures that the adversary cannot distinguish the output and leakage from tuple m_0^0, m_0^1 or tuple m_1^0, m_1^1, which can capture that the XOR operation cannot leak one bit of one operand.
- The XOR-res assumption captures that the adversary cannot distinguish the output y, leak and R, leak, where y is from the secret value s_1 XORing m_0, m_1, R is a random value and leak is described above, which can capture that the XOR computation leakage cannot leak one bit of result.

Remark. Similarly to [8,38], we aim to reduce the security of our full construction GLNPLR to the security of several considerably simpler structures (which may not offer exponential security). The purpose is to simplify hardware designers' (arguably difficult) tasks: with such a reduction, they can focus on protecting and testing some small structures (See Appendix C). In other words, we still rely on implementation-level countermeasures to protect the small structures, but in a less demanding way.

5.6 Leakage-resilience of GLNPLR

Let $\mathsf{L}_{\mathrm{GLNPLR}}$ be the leakages generated during executing the procedure GLNPLR.Garble. In detail, $\mathsf{L}_{\mathrm{GLNPLR}}$ consists of:

- $\mathsf{L}^{in}(k,x)$ & $\mathsf{L}^{out}(k,y)$ generated by each internal call (including those made by sub-procedures) to $E_k(x) \to y$.
- $\mathsf{L}_{\oplus}(x_1,\ldots,x_n)$ generated by the internal computations (including those made in sub-procedures) of $x_1 \oplus x_2 \oplus \ldots \oplus x_n$.
- We don't model the leakage of the permute bit operation, and we assume that the permutation bit XOR computations are leak-free (strongly protected in practice).

With $\mathsf{L}_{\mathrm{GLNPLR}}$, we define

$$\mathbf{Adv}^{\mathrm{GLNPLR},\mathcal{S},\mathsf{L}_{\mathrm{GLNPLR}}}_{\mathsf{PrvSimL}}(q_E,t) = \max\left\{\mathbf{Adv}^{\mathrm{GLNPLR},\mathcal{S},\mathsf{L}_{\mathrm{GLNPLR}}}_{\mathsf{PrvSimL}}(\mathcal{A})\right\},$$

where the maximum is taken over all (q_E,t)-bound adversaries.

For simplicity, for each assumption $\mathsf{xx} \in \{\mathsf{2\text{-}up}[q], \mathsf{XOR\text{-}inv}[q], \mathsf{XOR\text{-}ope}, \mathsf{XOR\text{-}res}\}$, we define

$$\mathbf{Adv}^{\mathsf{xx}}(q_E,t) \overset{\mathrm{def}}{=} \max\left\{\mathbf{Adv}^{\mathsf{xx}}(\mathcal{A})\right\}, \tag{7}$$

where the maximum is taken over all adversaries that make q_E queries to E and runs in time t. Our main result is then as follows.

Theorem 1. *In the ideal cipher model, consider the aforementioned leakage function L_{GLNPLR}. If the involved leakage functions $\mathsf{L}^{in}, \mathsf{L}^{out}, \mathsf{L}_{\oplus}$ satisfy the assumptions specified by Eq. (2), Eq. (3), Eq. (5) and Eq. (6), then it holds*

$$\begin{aligned}\mathbf{Adv}^{GLNPLR,\mathcal{S},\mathsf{L}_{GLNPLR}}_{\mathsf{PrvSimL}}(q_E,t) \le (3g_1+g_2+2\ell_{out}-\ell_{in})\mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*,t^*)\\ +g_2\mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q^*]}(q^*,t^*) + g_1\cdot\mathbf{Adv}^{\mathsf{XOR\text{-}res}}(q^*,t^*)\\ +3g_1\mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(q^*,t^*) + \frac{3g_1}{2^{n+1}}\end{aligned} \tag{8}$$

where $q^* = (12g_1+6g_2-4\ell_{in}+6\ell_{out}+q_E)$, $t^* = O(t+(12g_1+6g_2-4\ell_{in}+6\ell_{out}+q_E)t_l)$, g_1, g_2 is the number of AND gate and XOR gate respectively, ℓ_{in} is the number of input wires, ℓ_{out} is the number of output wires and t_l is the total time needed for evaluating L^{in}, L^{out}.

Interpreting the bound. The terms $g_1\cdot\mathbf{Adv}^{\mathsf{XOR\text{-}res}}(q^*,t^*)$ and $3g_1\mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(q^*,t^*)$ correspond to the reduction to the "minimal" assumption on XOR leakages: unsurprisingly, more GbAND-calls indicate less security. The terms $O(g)\mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*,t^*)$ and $g_2\mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q^*]}(q^*,t^*)$ capture the hardness of side-channel key recovery, and it is roughly of

$$O\left(\sigma\cdot\frac{q_E+g+t}{c\cdot 2^n}\right) = O\left(\frac{(q_E+g+t)\sigma}{c\cdot 2^n}\right),$$

for some parameter c that depends on the concrete conditions. Yet, it is nowadays a common assumption that with such a small data complexity, the value of c should not be significant [35,39].

Garbled scheme	NI support?	5 Gbps	2 Gbps	300Mbps	50Mbps
HalfGate AND	Y	21.9	7.99	1.18	0.19
Masked HG AND	N	0.036	0.036	0.036	0.036
GLNPLR AND	Y	7.8	5.62	0.79	0.13
GLNPLR XOR	Y	9.9	9.01	2.37	0.39
GLNPLR AND	N	0.63	0.62	0.64	0.13
GLNPLR XOR	N	1.00	0.98	1.01	0.39

Table 2: Performance comparison with HalfGates (HG) in different bandwidths. All reported numbers are in 10^6 gates per second. The length of the wire labels is 128. "NI" indicates whether the implementation utilizes AES-NI instructions.

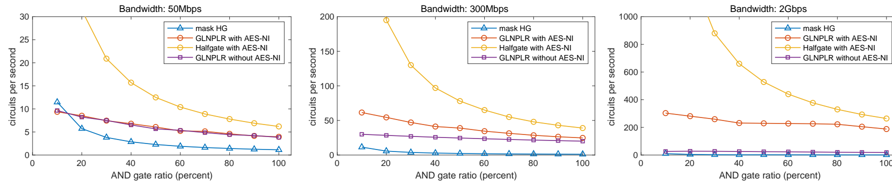

Fig. 11: Performance comparison of GLNPLR (with and without AES-NI) versus the HalfGates scheme (both masked and unprotected) at various AND gate ratios within a 31,924-gate circuit. Sub-figures display performance data at bandwidths of 50 Mbps, 300 Mbps, and 2 Gbps, demonstrating the performance advantage of GLNPLR at higher bandwidths.

Proof (Proof sketch). We first construct a plausible simulator. We begin with designing procedures SimGSL, SimXOR and SimAND to simulate the outputs of LGSL, LGbXOR and LGbAND respectively. In particular, the simulated internal secrets are randomly picked "fake" values, and the leakages are simulated by running the leakage function on the simulated secrets. We then combine SimGSL, SimXOR and SimAND to have a complete simulator.

We then prove indistinguishability of the simulated outputs. The main step is to prove SimGSL $\approx$ LGSL, SimXOR $\approx$ LGbXOR and SimAND $\approx$ LGbAND, and this relies on the leakage assumptions. Briefly speaking, the unpredictability assumptions ensure that certain internal values (supposed to be secret in the non-leaky setting) would not be involved in adversarial ideal cipher queries (so that E-calls in Garble yield random outputs that resemble the simulated values), while the assumptions on XOR leakages ensure that SimAND would not significantly deviate from LGbAND (otherwise, see Sect. 4.2). We refer to Appendix D for the full proof.

Obliviousness and authenticity of GLNPLR can be proven similarly to Theorem 1. We defer the proofs to Appendix D.5.

Theorem 2. *In the ideal cipher model, with the* Garble *leakage functions* $\mathsf{L}_{\mathsf{Garble}}$*, the following holds:*

$$\mathbf{Adv}_{\mathsf{ObvSimL}}^{GLNPLR,\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}(q_E,t) \leq \mathbf{Adv}_{\mathsf{PrvSimL}}^{GLNPLR,\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}(q_E,t),$$
$$\mathbf{Adv}_{\mathsf{AutL}}^{GLNPLR,\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}(q_E,t) \leq \mathbf{Adv}_{\mathsf{PrvSimL}}^{GLNPLR,\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}(q_E,t) + \frac{\ell_{out}}{2^{n+1}},$$

where ℓ_{out} *is the number of output wires.*

6 Experimental Results

Experimental settings. Admittedly, GLNPLR consumes more computational and communication resources than the classical schemes HalfGates and GLNP. However, given its provable leakage-resilience and suitability for side-channel attack (SCA) sensitive scenarios, we believe the performance trade-off is acceptable. To substantiate this, we implement GLNPLR and side-channel masked classical schemes and compare performances. This resembles leakage-resilient encryption [38,8].

We consider AES-based GLNPLR, i.e., setting E = AES-128 in Fig. 6. We consider two (SCA sensitive) hardware environments, i.e., the ARM environments with and without AES-NI support (AES-NI-based garbling implementations can also be victims of SCAs [33]). For this, we provide two implementations of AES-based GLNPLR. Our first implementation used AES-NI, and we employ the optimizations of [17] (in both Garble and GSL). Our second implementation relies on the OpenSSL C implementation of AES [43]. Thanks to its leakage-resilience, we can directly employ existing optimized AES implementations without side-channel protections.

We choose the (original) HalfGates [46] as the benchmark of classical schemes (which is *faster* than [18] and has comparable concrete security with GLNPLR), and we provide an implementation using the C implementation of second-order marked AES from [27] (which implements the masking scheme of [41] optimized with CPRR method [12], the Common Shares method [47] and the Random Reduction method [11]).

Our experiments were conducted on a virtualized platform powered by the Apple M1 processor, featuring the ARMv8-A architecture. We utilized VMware Fusion Professional 13.5.1 to create a virtual environment that ran Ubuntu 64-bit ARM Server 22.04.4. The virtual machine was provisioned with 2 CPU cores and 4 GB of RAM, specifically configured to evaluate the performance and compatibility of applications within an ARM-based setting.

Performance. In Table 2, we list the performance evaluations of GLNPLR and masked HalfGates in different bandwidths. Unsurprisingly, GLNPLR (4th—7th columns) is inferior to the (unprotected) HalfGates (2nd column). But with masking, it can be seen:

1. The bottleneck of masked HalfGates.GbAND is computation.

Game $\mathsf{Expt}_{\mathsf{sfe}}^{\mathcal{F},i,\mathcal{S}}$	
procedure INITIALIZE $b \xleftarrow{\$} \{0,1\}$ **procedure** FINALIZE(b') **return** $(b' = b)$	**procedure** $\mathsf{GetView}(f, x, y)$ **if** $b = 1$ **then return** $view := \mathsf{View}_{\Pi}^{i}(f, x, y)$ **if** $i = 1$ **then return** $view := \mathcal{S}(x, f_1(x, y), f)$ **if** $i = 2$ **then return** $view := \mathcal{S}(y, f_2(x, y), f)$
Game $\mathsf{Expt}_{\mathsf{sfeLR}}^{\mathcal{F},i,\mathcal{S},\mathsf{L}_1,\mathsf{L}_2}$	

procedure INITIALIZE
$b \xleftarrow{\$} \{0,1\}$
procedure FINALIZE(b')
return $(b' = b)$
	procedure $\mathsf{GetView}(f, x, y)$
	if $b = 1$ **then**
	$(view, \mathsf{leak}) := \mathsf{ViewL}_{\Pi}^{i}(f, x, y, \mathsf{L}_1, \mathsf{L}_2)$
	if $i = 1$ **then**
	$(view, \mathsf{leak}) := \mathcal{S}(x, f_1(x, y), f, \mathsf{L}_1, \mathsf{L}_2)$
	if $i = 2$ **then**
	$(view, \mathsf{leak}) := \mathcal{S}(y, f_2(x, y), f, \mathsf{L}_1, \mathsf{L}_2)$
	return $(view, \mathsf{leak})$

Fig. 12: **Games for defining the sfe and sfeLR security of an SFE scheme** Π. The leakage is computed $\mathsf{leak} := \big(\mathsf{L}_1(x, \omega_1), \mathsf{L}_2(y, \omega_2)\big)$ using the two parties' inputs and their randomness (x, ω_1) and (y, ω_2).

2. With a 50 Mbps network, the AES-NI-based GLNP.GbAND is 3.6 times faster than the masked HalfGates.GbAND, and the advantage increases to 21, 156 and > 200 times at 300 Mbps, 2 Gbps and 5 Gbps respectively.
3. Without AES-NI, GLNPLR.GbAND remains 3 to 17 times faster than the masked HalfGates.GbAND with bandwidth ranging from 50 Mbps to 5 Gbps.

HalfGates.GbXOR is free even with masking. To have a complete comparison, we vary the ratio of AND gates in the garbled circuits, and detail the performance curves in Fig. 11 (for different bandwidths). It can be seen GLNPLR outperforms masked HalfGates as long as AND ratio exceeds 10% and bandwidth exceeds 300Mbps. In particular, since AES has 31,924 gates and $\approx 21.3\%$ AND gates, GLNPLR is 60X (using AES-NI) or 5X (without AES-NI) faster than the masked HalfGates when the bandwidth is 2Gbps.

We remark that while our primary experiments demonstrated the advantage of GLNPLR over the traditional masking approach, further optimized performances are beyond the scope of this paper.

7 Application

We believe GLNPLR can be plugged into most garbling-based protocols to improve robustness against SCAs. One would hope to yield "fully" leakage-resilient protocols. However, since such protocols are usually built upon multiple building blocks, a complete leakage-resilience characterization is beyond the scope of this paper, and we are only able to formally prove the SCA weaknesses due to garbling are eliminated.

We take Yao's Secure Function Evaluation (SFE) protocol [44] for example. Below we first serve necessary preliminaries. We then give our leakage-resilience definition for SFE. We finally describe the SFE protocol and prove its leakage-resilience.

Secure Function Evaluation (SFE). An SFE is a two-party protocol defined via a pair $\Pi = (\Pi_1, \Pi_2)$ of polynomial time (PT) algorithms. The secret inputs of party 1 and 2 are $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$ respectively, while the public input is a function $f = (f_1, f_2)$. The interaction leads to party i learning $f_i(x, y)$, but they cannot get any information about other secret inputs.

We can define a PT algorithm $\mathrm{View}_I^i(f, x, y)$ that returns the view of party i in an execution of Π with inputs (f, x, y). Concretely, $\mathrm{View}_I^i(f, x, y)$ picks at random coins ω_1, ω_2, executes the interaction between the parties as defined by Π with the public input f and coins of party $j \in \{1, 2\}$ being (x, ω_1) and (y, ω_2) respectively, and returns $(conv, \omega_i)$ where the conversation $conv$ is the messages received by party i.

To model leakages, we define another PT algorithm $\mathsf{ViewL}_I^i(f, x, y,\ \mathsf{L}_1, \mathsf{L}_2)$ that on input f, x, y and two leakage functions L_1 and L_2 and returns the view of party i as well as the leakages $\mathsf{L}_1(x, \omega_1)$ and $\mathsf{L}_2(y, \omega_2)$ of the two parties during executing Π. $\mathrm{ViewL}_I^i(f, x, y)$ returns $(conv, \mathsf{leak}, \omega_i)$ with $\mathsf{leak} = \big(\mathsf{L}_1(x, \omega_1), \mathsf{L}_2(y, \omega_2)\big)$, which resembles $\mathrm{View}_I^i(f, x, y)$.

The conventional privacy of SFE in the semi-honest setting means the parties follow the protocol, and their views do not allow the computation of any undesired information. This is typically formalized via a simulation-based approach [32], and we present a game-based description. Concretely, we use the game $\mathsf{Expt}_{\mathsf{sfe}}^{\mathcal{F}, i, \mathcal{S}}$ defined in Fig. 12 that is defined upon a protocol $\mathcal{F}$ and a simulator $\mathcal{S}$. The adversary $\mathcal{B}$ makes a query to the $\mathsf{GetView}$ procedure in $\mathsf{Expt}_{\mathsf{sfe}}^{\mathcal{F}, i, \mathcal{S}}$ to collect information, and its advantage is defined as

$$\mathbf{Adv}_{\mathsf{sfe}}^{\Pi, i, f, \mathcal{S}}(\mathcal{B}) = 2\Pr\big[\mathsf{Expt}_{\mathsf{sfe}}^{\mathcal{F}, i, \mathcal{S}}\big] - 1$$

To define its leakage extension, consider the game $\mathsf{Expt}_{\mathsf{sfeLR}}^{\mathcal{F}, i, \mathcal{S}, \mathsf{L}_1, \mathsf{L}_2}$ in Fig. 12 parameterized by leakage functions L_1 and L_2. The adversary $\mathcal{B}$ queries $\mathsf{GetView}$ in $\mathsf{Expt}_{\mathsf{sfeLR}}^{\mathcal{F}, i, \mathcal{S}, \mathsf{L}_1, \mathsf{L}_2}$ to collect information, and $\mathsf{GetView}$ includes leakages of the two parties in its return list. The advantage of $\mathcal{B}$ is defined as

$$\mathbf{Adv}_{\mathsf{sfeLR}}^{\Pi, i, f, \mathcal{S}, \mathsf{L}_1, \mathsf{L}_2}(\mathcal{B}) = 2\Pr\big[\mathsf{Expt}_{\mathsf{sfeLR}}^{\mathcal{F}, i, \mathcal{S}, \mathsf{L}_1, \mathsf{L}_2}\big] - 1.$$

We say that Π is sfeLR secure w.r.t. the leakage $(\mathsf{L}_1, \mathsf{L}_2)$, if for each $i \in \{0, 1\}$ and each PT adversary $\mathcal{B}$ there is a PT simulator $\mathcal{S}$ such that $\mathbf{Adv}_{\mathsf{sfeLR}}^{\Pi, i, f, \mathcal{S}, \mathsf{L}_1, \mathsf{L}_2}(\mathcal{B})$ is negligible.

The protocol. We consider the simplest case: $f = (f_1, f_2)$ where $f_1 = f_2 := \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^\ell$ (which actually suffices for constructing secure protocols for arbitrary function f [44]). Let $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode})$ be a garbling scheme. Let f be a PT function where $f = \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^\ell$.

The construction needs an n-oblivious transfer (n-OT) protocol, where party 1 has inputs $X_1^0, X_1^1, \cdots, X_n^0, X_n^1$, party 2 has n selection bits $y_1, \cdots, y_n$, and

the result is that party 1 gets nothing while party 2 gets $X_1^{y_1}, \cdots, X_n^{y_n}$. Due to page limits, we defer the formal definition to Appendix E.

We define an SFE scheme YaoSFE = (Π_1, Π_2) for securely computing the functions f that can be garbled by $\mathcal{G}$. In detail, Π_2 (also called the *garbler*), on inputs y, begins by letting $(F, e, d) := \mathsf{Garble}(f)$ and parsing e as $(X_1^0, X_1^1, \cdots, X_{n+m}^0, X_{n+m}^1) := e$. Π_2 then sends $F, d, X_{n+1}^{y_1}, \cdots, X_{n+m}^{y_n}$ to Π_1 (also called the *evaluator*). Now, Π_1 and Π_2 execute the n-OT protocol $\mathcal{OT}$, where Π_1 takes $x = (x_1, ..., x_n)$ as the n selection bits and Π_2 takes $(X_1^0, X_1^1, \cdots, X_n^0, X_n^1)$ as inputs. This allows Π_1 to obtain $(X_1^{x_1}, \cdots, X_n^{x_n})$. Π_1 sets $X := (X_1^{x_1}, \cdots, X_n^{x_n}, X_{n+1}^{y_1}, \cdots, X_{n+m}^{y_m})$, outputs $z := \mathsf{Decode}(d, \mathsf{Eval}(F, X))$ and sends z to Π_2.

While our leakage-resilience definition for SFE is general, our positive result only considers garbling leakages (as mentioned in the Introduction). Namely, we focus on $\mathsf{L}_1 = \perp$ and $\mathsf{L}_2 = \mathsf{L}_{\mathsf{Garble}}$.

Theorem 3 (Informal). *Assume $\mathcal{OT}$ is a secure n-OT protocol and the garbling scheme $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode})$ is* PrvSimL *secure w.r.t the leakage $\mathsf{L}_{\mathsf{Garble}}$. Then the above SFE scheme YaoSFE is* sfeLR*-secure w.r.t. the leakage $(\perp, \mathsf{L}_{\mathsf{Garble}})$.*

Due to page limits, the full proof is deferred to Appendix F.

Acknowledgments

Ruiyang Li and Chun Guo are supported by the National Key Research and Development Program of China (grant 2022YFA1004900), the National Natural Science Foundation of China (grant 62372274) and the Taishan Scholars Program (for Young Scientists) of Shandong. François-Xavier Standaert is a senior research associate of the Belgian Fund for Scientific Research (F.R.S.-FNRS), and has been funded in parts by the ERC Advanced Grant 101096871 (acronym BRIDGE) and the Horizon Europe project 1010706275 (acronym REWIRE). Weijia Wang is supported by the National Natural Science Foundation of China (grant 62372273), the Taishan Scholars Program (for Young Scientists) of Shandong. Views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

A Leakage-resilient evaluation and its difficulty

Recall the SFE protocol:

1. The garbler generate an encoded function (F), the encode information e, and the decode information d: $(F, e, d) := \mathsf{Garble}(f)$.
2. The garbler sends the encoded function (F), the decode information d, and the input labels corresponding to its input bits.

3. The garbler and the evaluator engage in an oblivious transfer (OT) protocol, through which the evaluator gets the input labels corresponding to its input bits.
4. Then the garbler uses all input labels and garbled function (F) to compute the output label Y ($Y := \mathsf{Eval}(F, X)$), and use the decode information d to get the function output $y := \mathsf{Decode}(d, Y)$.
5. Finally, the garbler sends y to the garbler.

SFE security requires that the garbler not know the input labels corresponding to the evaluator's input bits. In other words, the garbler cannot get $w_i = w_i^0$ or $w_i = w_i^1$ for every wire i, which is attached by evaluator input bits. But the garbler holds the encoded information e (which includes all the input labels w_i^0, w_i^1 for all the input wire i). If the leakage of Eval leaks one bit of w_i, then the garbler can obtain whether $w_i = w_i^0$ or $w_i = w_i^1$ (Similar to attack in 4).

In Eval, the label w_i will participate in AES calculations and participate in some XOR operations. It is not difficult for an adversary to obtain 1 bit of information from these leaks. Theoretically, this w_i will participate in various operations, and it is difficult to give a unified and simple assumption to limit this part of the leakage. In addition, from a practical level, it is relatively difficult to require given protection to prevent all these leaks from leaking any information (because it is relatively simple to recover 1 bit of information from AES leaks, even with masking technology).

B Detailed Discussion on the Weakness Of AND Gate

B.1 Success probabilities of SCAs in Sect. 4.2

This section provides detailed analysis of success probabilities of the SCAs in Sect. 4.2.

XOR leaks 1 bit of the result. Recall from Sect. 4.2 that $\mathcal{A}$ collects the leakage $\mathsf{L}_{\oplus}(S_1, S_2)$ at line 34, where $S_1 = E_{w_a^{\pi_a}}(00)$ and $S_2 = E_{w_b^{\pi_b}}(00)$. As assumed, $\mathsf{L}_{\oplus}(S_1, S_2)$ leaks $\mathsf{msb}(W_0)$ ($S_1 \oplus S_2 = W_0$) where $\mathsf{msb}(W_0)$ is the first bit of W_0. In GLNP.GbAND, the value W_0 computed at line 34 fulfills $W_0[1 \cdots n] = w_c^{\pi_a \wedge \pi_b}$ (this can be seen from the design). $\mathcal{A}$ compares $\mathsf{msb}(W_0)$ and $\mathsf{msb}(w_c^{v_c})$: if $\mathsf{msb}(W_0) = \mathsf{msb}(w_c^{v_c})$, $\mathcal{A}$ guess $v_c = 0$; Else, quit attacking.

As discussed in Sect. 4.2, since $W_0[1 \cdots n] = w_c^{\pi_a \wedge \pi_b}$, $\mathsf{msb}(W_0) \neq \mathsf{msb}(w_c^{v_c})$ necessarily implies $\pi_a \wedge \pi_b \neq v_c$. Since $\pi_a, \pi_b \xleftarrow{\$} \{0, 1\}$ are independently sampled in GLNP.Garble, $\Pr\big[\pi_a \wedge \pi_b = 0\big] = 3/4$.

Here we assume the AND gate input bits v_a, v_b are uniform, this will make $\Pr[v_c = 0] = \frac{3}{4}$ and $\Pr[v_c = 1] = \frac{1}{4}$. Of course, the attack will succeed in the general case. Here, we only discuss the uniform input bits case in a simple way.

With the above, there are two cases where $\mathcal{A}$ could observe $\mathsf{msb}(W_0) = \mathsf{msb}(w_c^{v_c})$, i.e.,

– $\pi_a \wedge \pi_b = 0$, $v_c = 0$, and $\mathsf{msb}(W_0) = \mathsf{msb}(w_c^{v_c})$;

– $\pi_a \wedge \pi_b = 1$, $v_c = 1$, and $\mathsf{msb}(W_0) = \mathsf{msb}(w_c^{v_c})$.

Therefore, the probability that $\mathcal{A}$ observes $\mathsf{msb}(W_0) \neq \mathsf{msb}(w_c^{v_c})$ is approximately $\frac{3}{4} \times \frac{3}{4} + \frac{1}{4} \times \frac{1}{4} = \frac{5}{8}$. Furthermore, conditioned on observing $\mathsf{msb}(W_0) = \mathsf{msb}(w_c^{v_c})$, $\mathcal{A}$ succeeds with probability $\frac{\frac{3}{4} \times \frac{3}{4}}{\frac{5}{8}} = \frac{9}{10}$, which is bigger than the probability of 3/4.

XOR leaks 1 bit of the operand. Recall from Sect. 4.2 that $\mathcal{A}$ collects three leakages at line 5 in Fig. 4:

$$\mathsf{L}_{\oplus}(S_1, W_c^{\pi_a \wedge \bar{\pi}_b}), \text{ where } S_1 = E_{w_a^{\pi_a}}(g\|01) \oplus E_{w_b^{\pi_b}}(g\|01),$$
$$\mathsf{L}_{\oplus}(S_2, W_c^{\bar{\pi}_a \wedge \pi_b}), \text{ where } S_2 = E_{w_a^{\pi_a}}(g\|10) \oplus E_{w_b^{\pi_b}}(g\|10),$$
$$\mathsf{L}_{\oplus}(S_3, W_c^{\bar{\pi}_a \wedge \bar{\pi}_b}), \text{ where } S_3 = E_{w_a^{\pi_a}}(g\|11) \oplus E_{w_b^{\pi_b}}(g\|11).$$

When $\mathsf{msb}(W_c^0) \neq \mathsf{msb}(W_c^1)$ (the probability is nearly 1/2 since they are pseudorandom), $\mathcal{A}$ observes one of the follow two cases:

1. When $\pi_a \wedge \bar{\pi}_b = \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$, $\mathcal{A}$ observes $\mathsf{msb}\ (W_c^{\pi_a \wedge \bar{\pi}_b}) = \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \pi_b}) = \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \bar{\pi}_b})$. Then it necessarily be $\pi_a \wedge \bar{\pi}_b = ...\bar{\pi}_a \wedge \bar{\pi}_b = 0$ and $\mathcal{A}$ has $\mathsf{msb}(W_c^0)$. By colluding with the evaluator, $\mathcal{A}$ could have the label w_c^b with $b \in \{0, 1\}$. By comparing $\mathsf{msb}(W_c^0)$ with $\mathsf{msb}(w_c^b)$, it could determine the value of b and break the c-th wire.
2. Otherwise, wlog assume $\pi_a \wedge \bar{\pi}_b \neq \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$. Then, it necessarily be $\pi_a \wedge \bar{\pi}_b = 1$ and $\bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b = 0$, and $\mathcal{A}$ observes $\mathsf{msb}(W_c^1)$ once while $\mathsf{msb}(W_c^0)$ twice from the leakages. $\mathcal{A}$ could then determine $\mathsf{msb}(W_c^0)$ and break the c-th wire as in the previous case.

This attack succeeds with probability 1 when $\mathsf{msb}(W_c^0) \neq \mathsf{msb}(W_c^1)$. In fact, $\mathcal{A}$ don't know whether $\mathsf{msb}(W_c^0) = \mathsf{msb}(W_c^1)$ or $\mathsf{msb}(W_c^0) \neq \mathsf{msb}(W_c^1)$. The $\mathcal{A}$ can only need to act as mentioned above. If $\mathsf{msb}(W_c^0) = \mathsf{msb}(W_c^1)$, the $\mathcal{A}$ must observe $\mathsf{msb}(W_c^{\pi_a \wedge \bar{\pi}_b}) = \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \pi_b}) = \mathsf{msb}(W_c^{\bar{\pi}_a \wedge \bar{\pi}_b})$. In addition, $\mathsf{msb}(W_c^0) = \mathsf{msb}(w_c^{v_c})$ and $\mathcal{A}$ must guess $v_c = 0$. This succeeds with probability p when $w_c^0 = w_c^1$ where p is the probability of $w_c^{v_c} = w_c^0$. Thus, the attack succeeds with probability $1/2 + 1/2p \gg p$.

B.2 BHR's scheme

The scheme is described in Fig. 13.

The XOR leaks one bit of one operand. Second, at line 5, $\mathcal{A}$ would collect four leakages:

$$\mathsf{L}_{\oplus}(S_1, W_c^{\pi_a \wedge \pi_b}), \text{ where } S_1 = E_{w_a^{\pi_a}}(g\|00) \oplus E_{w_b^{\pi_b}}(g\|00),$$
$$\mathsf{L}_{\oplus}(S_2, W_c^{\pi_a \wedge \bar{\pi}_b}), \text{ where } S_2 = F_{w_b^{\pi_a}}(g\|01) \oplus F_{w_b^{\bar{\pi}_b}}(g\|01),$$
$$\mathsf{L}_{\oplus}(S_3, W_c^{\bar{\pi}_a \wedge \pi_b}), \text{ where } S_3 = F_{k_a^{\bar{\pi}_a}}(g\|10) \oplus F_{k_b^{\pi_b}}(g\|10),$$
$$\mathsf{L}_{\oplus}(S_4, W_c^{\bar{\pi}_a \wedge \bar{\pi}_b}), \text{ where } S_3 = F_{k_a^{\bar{\pi}_a}}(g\|11) \oplus F_{k_b^{\bar{\pi}_b}}(g\|11).$$

procedure $\mathsf{GbAND}(w_a^0, w_a^1, w_b^0, w_b^1, \pi_a, \pi_b, g)$
$w_c^0 \xleftarrow{\$} \{0,1\}^n$, $w_c^1 \| \overline{\pi}_c := W_0$, $\pi_c \xleftarrow{\$} \{0,1\}$
$W_c^0 := w_c^0 \| \pi_c$, $W_c^1 := w_c^1 \| \overline{\pi}_c$
$T_0 := E_{w_a^{\pi_a}}(g\|00) \oplus E_{w_b^{\pi_b}}(g\|00) \oplus W_c^{\pi_a \wedge \pi_b}$
$T_1 := E_{w_a^{\pi_a}}(g\|01) \oplus E_{w_b^{\overline{\pi}_b}}(g\|01) \oplus W_c^{\pi_a \wedge \overline{\pi}_b}$
$T_2 := E_{w_a^{\overline{\pi}_a}}(g\|10) \oplus E_{w_b^{\pi_b}}(g\|10) \oplus W_c^{\overline{\pi}_a \wedge \pi_b}$
$T_3 := E_{w_a^{\overline{\pi}_a}}(g\|11) \oplus E_{w_b^{\overline{\pi}_b}}(g\|11) \oplus W_c^{\overline{\pi}_a \wedge \overline{\pi}_b}$
return $(w_c^0, w_c^1, \pi_c, (T_0, T_1, T_2, T_3))$

Fig. 13: The GbAND of BHR garbling scheme [4]

Recall from our convention in Sect. 2 that $\mathsf{L}_{\oplus}(a, b)$ is the leakage of $a \oplus b$.

XOR leaks 1 bit of operand. First, consider the case where the four leakages leaked $\mathsf{msb}(W_c^{\pi_a \wedge \pi_b})$, $\mathsf{msb}(W_c^{\pi_a \wedge \overline{\pi}_b})$, $\mathsf{msb}(W_c^{\overline{\pi}_a \wedge \pi_b})$ and $\mathsf{msb}(W_c^{\overline{\pi}_a \wedge \overline{\pi}_b})$. $\mathcal{A}$ does not know the values $W_c^{\pi_a \wedge \pi_b}$, $W_c^{\pi_a \wedge \overline{\pi}_b}$, $W_c^{\overline{\pi}_a \wedge \pi_b}$ and $W_c^{\overline{\pi}_a \wedge \overline{\pi}_b}$, but when g represents AND, three of them are identical while another is distinct from them (since the four values $\pi_i \wedge \pi_j, \pi_i \wedge \bar{\pi}_j, \bar{\pi}_i \wedge \pi_j$ and $\bar{\pi}_i \wedge \bar{\pi}_j$ consist of three 0 and one 1).

1. When $\mathsf{msb}(w_c^0) \neq \mathsf{msb}(w_c^1)$, wlog assume $\pi_a \wedge \pi_b \neq \pi_a \wedge \bar{\pi}_b = \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$. Then, it necessarily be $\pi_a \wedge \pi_b = 1$ and $\pi_a \wedge \bar{\pi}_b = \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b = 0$. $\mathcal{A}$ could then determine $\mathsf{msb}(w_c^0)$. Then, $\mathcal{A}$ colludes with the evaluator and obtains $w_c^{v_c}$ with $v_c \in \{0, 1\}$(The $\mathcal{A}$ does not know whether $v_c = 0$ or $v_c = 1$). By comparing $\mathsf{msb}(w_c^0)$ with $\mathsf{msb}(w_c^{v_c})$, it could determine the value of b and break the c-th wire.
2. When $\mathsf{msb}(w_c^0) = \mathsf{msb}(w_c^1)$, then $\pi_a \wedge \pi_b = \pi_a \wedge \bar{\pi}_b = \bar{\pi}_a \wedge \pi_b = \bar{\pi}_a \wedge \bar{\pi}_b$. Then, the $\mathcal{A}$ cannot use the attack.

B.3 Another design of GbAND of GLNP

In this design, $E_{w_a^{\pi_a}}(g\|00) \oplus E_{w_b^{\pi_b}}(g\|00)$ is assigned by one output label similar to GRR3 technique. Thus, the attack described in Section 4.2 can succeed, too.

C Testers for assumption

In practice, the value of different assumptions can be similarly measured by a tester and it is easier to study (and to reduce with relevant protections) than to study entire modes.

Tester for 2-$\mathsf{up}[q]$ $\mathbf{Adv}^{2\text{-}\mathsf{up}[q]}$

```
procedure GbAND(w_a^0, w_a^1, w_b^0, w_b^1, π_a, π_b, g)
    K_0‖m_0 := E_{w_a^{π_a}}(g‖00) ⊕ E_{w_b^{π_b}}(g‖00)
    K_1‖m_1 := E_{w_a^{π_a}}(g‖01) ⊕ E_{w_b^{π̄_b}}(g‖01)
    K_2‖m_2 := E_{w_a^{π̄_a}}(g‖10) ⊕ E_{w_b^{π_b}}(g‖10)
    K_3‖m_3 := E_{w_a^{π̄_a}}(g‖11) ⊕ E_{w_b^{π̄_b}}(g‖11)
    s := 2π_a + π_b, π_l ←$ {0,1}
    if s ≠ 0 then
        w_c^0 = K_0, w_c^1 = K_1 ⊕ K_2 ⊕ K_3
    else
        w_c^0 = K_1 ⊕ K_2 ⊕ K_3, w_c^1 = K_0
    if s = 3 then T_1 = K_0 ⊕ K_1, T_2 = K_0 ⊕ K_2
    if s = 2 then T_1 = K_0 ⊕ K_1, T_2 = K_1 ⊕ K_3
    if s = 1 then T_1 = K_2 ⊕ K_3, T_2 = K_0 ⊕ K_2
    if s = 0 then T_1 = K_2 ⊕ K_3, T_2 = K_1 ⊕ K_3
    t_s = m_s ⊕ π̄_c
    for α ∈ {0,1,2,3}\{s} do t_α = m_α ⊕ π_c
    return (w_c^0, w_c^1, π_c, T_1, T_2, t_0, t_1, t_2, t_3)
```

Fig. 14: Another design of GbAND of GLNP [17].

1. Let the challenging adversary $\mathcal{A}$ specified s_0, x_1, x_2
2. Pick the secret $s_1 \xleftarrow{\$} \{0,1\}^n$, $r \xleftarrow{\$} \{0,1\}$ and compute $P_{pre} := (E_{s_0})^{-1}(s_1\|r)$
3. Compute $s_1\|r := E_{s_0}(P_{pre})$, $y := E_{s_1}(x_1)$, and $z := E_{s_1}(x_2)$.
4. Pass the leakage of Step (3) to $\mathcal{A}$. In our model, this means [$\mathsf{L}^{out}(s_0, s_1\|r)$, $\mathsf{L}^{in}(s_1, x_1)$, $\mathsf{L}^{out}(s_1, y)$, $\mathsf{L}^{in}(s_1, x_2)$, $\mathsf{L}^{out}(s_1, z)$] are returned to $\mathcal{A}$.
5. Let the challenging adversary $\mathcal{A}$ output q guesses $k_1, \ldots, k_q$, the adversary wins as long as $s_1 \in \{k_1, \ldots, k_q\}$.

Tester for 2-up[q] $\mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q]}$

1. Let the challenging adversary $\mathcal{A}$ specified k_1, s_1, k_2, s_2, x.
2. Pick the secret $\hat{s}_1 \xleftarrow{\$} \{0,1\}^n$, $\hat{s}_2 \xleftarrow{\$} \{0,1\}^n$, $r_1 \xleftarrow{\$} \{0,1\}$, $r' \xleftarrow{\$} \{0,1\}$ and compute $P^1_{pre} := (E_{s_1})^{-1}(\hat{s}_1\|r)$, $P^2_{pre} := (E_{s_2})^{-1}(\hat{s}_2\|r')$.
3. Compute $\hat{s}_1 := E_{s_1}(P^1_{pre})$, $\hat{s}_2 := E_{s_2}(P^2_{pre})$, $\Delta_1 := \hat{s}_1 \oplus k_1$, $\Delta_1 := k_1 \oplus \hat{s}_1$, $T := \Delta_1 \oplus \Delta_2$, $s_3 = k_1 \oplus \hat{s}_2$, $s_3 = \hat{s}_2 \oplus k_1$, $temp = \hat{s}_1 \oplus \hat{s}_2$, $temp = \hat{s}_1 \oplus \hat{s}_1$, $y = E_{s_3}(x)$. $y := E_{s_1}(P_A)$, and $z := E_{s_1}(P_B)$.
4. Pass the leakage of Step (3) to $\mathcal{A}$. In our model, this means [$\mathsf{L}^{out}(s_1, \hat{s}_1\|r)$, $\mathsf{L}^{out}(s_2, \hat{s}_2\|r')$, $\widehat{\mathsf{L}_\oplus}(\hat{s}_1, k_1)$, $\widehat{\mathsf{L}_\oplus}(\hat{s}_2, k_2)$, $\mathsf{L}_\oplus(\Delta_1, \Delta_2)$, $\widehat{\mathsf{L}_\oplus}(k_1, \hat{s}_2)$, $\widehat{\mathsf{L}_\oplus}(\hat{s}_1, \hat{s}_2)$, $\mathsf{L}^{in}(s_3, x)$, $\mathsf{L}^{out}(s_3, y)$] are returned to $\mathcal{A}$.
5. Let the challenging adversary $\mathcal{A}$ output q guesses $k_1, \ldots, k_q$, the adversary wins as long as $s_3 \in \{k_1, \ldots, k_q\}$.

Tester for XOR-ope $\mathbf{Adv}^{\mathsf{XOR\text{-}ope}}$

1. Let the challenging adversary $\mathcal{A}$ specified $s_0, m_0^0, m_1^0, m_0^1, m_1^1$.
2. Pick the secret $s_1 \xleftarrow{\$} \{0,1\}^{n+1}$, $b \xleftarrow{\$} \{0,1\}$ and compute $P_{pre} := (E_{s_0})^{-1}(s_1)$.
3. Compute $s_1 := E_{s_0}(P_{pre})$, $y := s_1 \oplus m_0^b \oplus m_1^b$, $y = m_0^b \oplus s_1 \oplus m_1^b$.
4. Pass the leakage of Step (3) to $\mathcal{A}$. In our model, this means $[\mathsf{L}^{out}(s_0, s_1)$, $\mathsf{L}_{\oplus}(s_1, m_0^b, m_1^b)$, $\mathsf{L}_{\oplus}(m_0^b, s_1, m_1^b)$ are returned to $\mathcal{A}$.
5. Let the challenging adversary $\mathcal{A}$ output the guess b'.

Tester for XOR-ope $\mathbf{Adv}^{\text{XOR-ope}}$

1. Let the challenging adversary $\mathcal{A}$ specified s_0, m_0
2. Pick the secret $s_1 \xleftarrow{\$} \{0,1\}^{n+1}$, $b \xleftarrow{\$} \{0,1\}$ and compute $P_{pre} := (E_{s_0})^{-1}(s_1)$
3. Compute $s_1 := E_{s_0}(P_{pre})$. If $b = 0$, $y := s_1 \oplus m_0$, otherwise, $y \xleftarrow{\$} \{0,1\}^{n+1}$.
4. Pass the leakage of Step (3) to $\mathcal{A}$. In our model, this means $[\mathsf{L}^{out}(s_0, s_1)$, $\mathsf{L}^{in}(s_1, P_A)$, $\widehat{\mathsf{L}_{\oplus}}(s_1, m_0)]$ are returned to $\mathcal{A}$.
5. Let the challenging adversary $\mathcal{A}$ output the guess b'.

D Proof of Theorem 1

For our proof, we need additional notations: we denote by $\mathsf{leak} := \mathsf{EmptyList}$ initializing a list leak to empty, and by $\mathsf{leak} \xleftarrow{\mathsf{add}} B$ adding an element B to a list leak. Recall that the evaluator will obtain one label for every wire. For each wire i, the label obtained is called the active label, denoted by w_i^v, while the label not obtained is referred to as the inactive label, denoted by $w_i^{\overline{v}}$.

Then, in Appendices D.1, D.2, D.3, we give lemmas for each modules of GLNPLR. We then "combine" them to yield a proof for the entire GLNPLR scheme in Appendix D.4. We finally prove leakage-resilient obliviousness and authenticity in Appendix D.5.

D.1 GSL vs SimGSL

The indistinguishability proof of GSL and SimGSL follows [8], and we include it here for completeness. Recall the procedure $\mathsf{GSL}(k_0, \ell)$ will generate different sub-wires using a series of blockciphers. The procedure uses k_0 as the first key of two blockciphers (whose input is two different constants) and denotes the outputs as the first sub-key k_1 and the first sub-label w_1. Then, the procedure repeats this process until w_ℓ is generated. In addition, here we define a simulator for procedure $\mathsf{GSL}(k_0, \ell)$, denoted by $\mathsf{SimGSL}(k_0, \ell)$. The simulator procedure is the same as procedure $\mathsf{GSL}(k_0, \ell)$ except that it chooses the sub-key and sub-label randomly rather than generated by the blockcipher. The detail is in Fig. 16.

We define $\mathsf{LGSL} := (\mathsf{GSL}(k_0, \ell), \mathsf{leak})$ (resp. $\mathsf{LSimGSL} := (\mathsf{GSL}(k_0, \ell), \mathsf{leak})$) where leak is a list containing the leakage generated in procedure $\mathsf{GSL}(k_0, \ell)$ (resp. $\mathsf{SimGSL}(k_0, \ell)$).

For their indistinguishability, we give an experiment in Fig. 16 and have the following lemma.

```
procedure GSL(k_0, ℓ)/SimGSL(k_0, ℓ)
    leak := EmptyList
    for i = 1, ..., ℓ − 1 do
        K_i := E_{k_{i−1}}(P_A), W_i := E_{k_{i−1}}(P_A)                ▷ GSL
        K_i ←$ {0,1}^{n+1}, W_i ←$ {0,1}^{n+1}                          ▷ SimGSL
        leak ←add {L^in(k_{i−1}, P_A), L^out(k_{i−1}, K_i)},
        leak ←add {L^in(k_{i−1}, P_B), L^out(k_{i−1}, W_i)}
        w_i := W_i[1...n], k_i := K_i[1...n]
    w_ℓ := k_{ℓ−1}
    return w_1, ..., w_ℓ
```

Fig. 15: Procedure $\mathsf{GSL}(k_0, \ell)$ and $\mathsf{SimGSL}(k_0, \ell)$

```
procedure Expt(k^−, ℓ)                                   Game Expt_LGSL(k^−, ℓ)
    leak := EmptyList, leak ←add L^out(k^−, K_0)
    if b = 0 then
        (w_1, ..., w_ℓ, leak') := LGSL(k_0, ℓ)
    else
        (w_1, ..., w_ℓ, leak') := LSimGSL(k_0, ℓ)
    leak ←add leak'
    return (w_1, ..., w_ℓ, leak)
```

Fig. 16: Experiment $\mathsf{Expt}_{\mathsf{LGSL}}$, the INITIALIZE is choose the challenge bit $b \xleftarrow{\$} \{0,1\}$, choose $K_0 \xleftarrow{\$} \{0,1\}^{n+1}$ and set $k_0 := K_0[1\dots n]$. There is no FINALIZE procedure.

Lemma 1. *For every (q_E, t)-bounded adversary $\mathcal{A}^E$ and every (k^-, ℓ) specified by $\mathcal{A}^E$, it holds*

$$\left|\Pr[\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0} = 1] - \Pr[\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,1} = 1]\right| \leq (\ell - 1) \cdot \mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*, t^*) \quad (9)$$

where $q^* = q_E + 2\ell - 2$, $t^* = O(t + (2\ell - 2) \cdot t_l)$, t_l is the total time needed for evaluating L^{in} and L^{out}, ℓ is the number of sub-wires.

Proof. Consider the execution of $\mathcal{A}^E$ for $\mathsf{Expt}_{\mathsf{LGSL}}$. We define a bad event $\mathsf{BadQuery}$, which occurs when any of the internal keys $k_0, k_1, \dots, k_{\ell-1}$ appears in the key field of a blockcipher E query made by $\mathcal{A}^E$. This event, once happens, would cause the key stream blocks to lose randomness. So, here we just need to adapt Yu, Bertil et al.'s argument to our setting. In detail, given an adversary $\mathcal{A}^E$, we

construct an adversary $\mathcal{A}_2^E$ such that

$$\mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(\mathcal{A}_2^E) \le \Pr\left[\mathsf{BadQuery}\text{ in }\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}\right] \tag{10}$$

Concretely, $\mathcal{A}_2^E$ runs an instance of $\mathcal{A}^E$ and keeps a record of $\mathcal{A}^E$'s queries to E in a set τ_E. $\mathcal{A}_2^E$ simulates the following process against $\mathcal{A}^E$:

1. $\mathcal{A}_2^E$ invoke $\mathcal{A}^E$, get its input k^-, ℓ.
2. $\mathcal{A}_2^E$ randomly guesses an index $i \xleftarrow{\$} [0, \ell-2]$ and initializes an empty list leak
3. $\mathcal{A}_2^E$ samples an initial key $K_0 \xleftarrow{\$} \{0,1\}^{n+1}$, set $k_0 = K_0[1\dots n]$, and add $\mathsf{L}^{out}(k^-, K_0)$ to leak.
4. For $j = 1, \dots, i-1$, $\mathcal{A}_2^E$ queries E to obtain $K_j := E_{k_{j-1}}(P_A)$, $k_j = K_j[1\dots n]$, $W_j = E_{k_{j-1}}(P_B)$ and $w_j = W_j[1\dots n]$. $\mathcal{A}^E$ then add the leakage traces [L^{in} (k_{j-1}, P_A), $\mathsf{L}^{out}(k_{j-1}, K_j)$], $\mathsf{L}^{in}(k_{j-1}, P_B)$, $\mathsf{L}^{out}(k_{j-1}, W_j)$ to leak.
5. $\mathcal{A}_2^E$ queries $W_i := E_{k_{i-1}}(P_B)$ and let $w_i = W_i[1\dots n]$. Then $\mathcal{A}_2^E$ submits $s_0 = k_{i-1}$, $x_1 = P_A$ and $x_2 = P_B$ to its $\mathsf{2\text{-}up}[q]$ challenger, and (according to our convention) this results in the outputs (y, z, leak'), where leak' $= [\mathsf{L}^{out}(k_{i-1}, s_1)$, $\mathsf{L}^{in}(s_1, P_A)$, $\mathsf{L}^{out}(s_1, y)$, $\mathsf{L}^{in}(s_1, P_B)$, $\mathsf{L}^{out}(s_1, z)]$. Then $\mathcal{A}_2^E$ adds $\mathsf{L}^{in}(k_{i-1}, P_A)$, $\mathsf{L}^{out}(k_{i-1}, s_1)$, $\mathsf{L}^{in}(k_{i-1}, P_B)$, $\mathsf{L}^{out}(k_{i-1}, W_i)$ to leak as the leakage of the i-th iteration.
6. Then $\mathcal{A}_2^E$ set $k_{i+1} := y[1\dots n]$, $w_{i+1} := z[1\dots n]$. It (conceptually) takes the challenge s_1, y, z as the key k_i, K_{i+1}, W_{i+1} and adds $\mathsf{L}^{in}(s_1, P_A)$, $\mathsf{L}^{out}(s_1, y)$, $\mathsf{L}^{in}(s_1, P_B), \mathsf{L}^{out}(s_1, z)$ to leak as the leakage of the $i+1$-th iteration.
7. Then $\mathcal{A}_2^E$ starts from k_{i+1} to emulate the remaining actions of $\mathsf{GSL}_{k_0}(k, \sigma)$ to obtain $w_{i+1}, \dots, w_\ell$. Eventually $\mathcal{A}_2^E$ serves the output $w_1, \dots, w_\ell$ as well as the leakage list leak to $\mathcal{A}^E$, and output the set Guesses $= \{k : (k, x, y) \in \tau_E\}$ for some x, y.

The strategy of $\mathcal{A}_2^E$ is obvious: if $\mathcal{A}^E$ triggers the event $\mathsf{BadQuery}$, then the key k_i being queried must be in τ_E. Therefore, $\mathcal{A}_2^E$ makes a uniform guess on the position of the first key on which such a query is made; guessing the first queried key ensures that key will only be correlated to one thing: the corresponding leakages (and not any previous call on E). The guess will be correct with probability $1/\ell$. Then, $\mathcal{A}_2^E$ emulates the process of $\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}$ and provides the leakages to $\mathcal{A}^E$, except for the i index, for which the leakages and E output are replaced by those obtained from a challenger for the seed-preserving property. If the guess on the index i is correct, all the inputs sent to $\mathcal{A}^E$ are distributed exactly as what the adversary $\mathcal{A}^E$ obtains in $\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}$. Therefore, when $\mathcal{A}_2^E$ halts, if $\mathcal{A}_2^E$ made a query on s_1, then simply outputting τ_E would break the game. So we have $\Pr\left[s_1 \in \text{Guesses}|\ \text{BadQuery in } \mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}\right] = \frac{1}{\ell-1}$. Now, we observe that

$$\begin{aligned}\Pr&\left[s_1 \in \text{Guesses}|\mathsf{BadQuery}\text{ in }\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}\right]\\ &\le \frac{\Pr[s_1 \in \text{Guesses}]}{\Pr\left[\mathsf{BadQuery}\text{ in }\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}\right]}\end{aligned}$$

procedure HXOR(PInputs)/SimXOR(PInputs)
    leak := EmptyList
    $\Delta_a := w_a^v \oplus w_a^{\overline{v}}$, leak $\xleftarrow{\text{add}}$ $\mathsf{L}_\oplus(w_a^v, w_a^{\overline{v}}; \lambda_a)$
    $\Delta_b := w_b^v \oplus w_b^{\overline{v}}$, leak $\xleftarrow{\text{add}}$ $\mathsf{L}_\oplus(w_b^v, w_b^{\overline{v}}; \lambda_b)$
    $T := \Delta_a \oplus \Delta_b$, leak $\xleftarrow{\text{add}}$ $\mathsf{L}_\oplus(\Delta_a, \Delta_b)$
    $\lambda_c := \lambda_a \oplus \lambda_b$
    if $\lambda_b = 0$ **then**
        $\hat{w}_c^v := w_a^v \oplus w_b^v$, $\mathsf{leak}_1 := \mathsf{L}_\oplus(w_a^v, w_b^v)$
        $\hat{w}_c^{\overline{v}} := w_a^{\overline{v}} \oplus w_b^v$, $\mathsf{leak}_2 := \mathsf{L}_\oplus(w_a^{\overline{v}}, w_b^v)$
    else
        $\hat{w}_c^v := w_a^{\overline{v}} \oplus w_b^{\overline{v}}$, $\mathsf{leak}_1 := \mathsf{L}_\oplus(w_a^{\overline{v}}, w_b^{\overline{v}})$
        $\hat{w}_c^{\overline{v}} := w_a^v \oplus w_b^{\overline{v}}$, $\mathsf{leak}_2 := \mathsf{L}_\oplus(w_a^v, w_b^{\overline{v}})$
    leak $\xleftarrow{\text{add}}$ $L(\mathsf{leak}_1, \mathsf{leak}_2; \lambda_c)$
    $W_c^{v_c} := E_{\hat{w}_c^v}(\lambda_c)$, $w_c^{v_c} := W_c^{v_c}[1 \cdots n]$
    $\mathsf{leak}_3 := [\mathsf{L}^{in}(\hat{w}_c^v, \lambda_c), \mathsf{L}^{out}(\hat{w}_c^v, W_c^v)]$
    $W_c^{\overline{v}} := E_{\hat{w}_c^v}(\overline{\lambda}_c)$, $w_c^{\overline{v}_c} := W_c^{\overline{v}}[1 \cdots n]$ ▷ HXOR
    $W_c^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}$, $w_c^{\overline{v}} := W_c^{\overline{v}}[1 \cdots n]$ ▷ SimXOR
    $\mathsf{leak}_4 := \mathsf{L}^{in}(\hat{w}_c^{\overline{v}}, \overline{\lambda}_c), \mathsf{L}^{out}(\hat{w}_c^{\overline{v}}, \overline{W}_c^v)]$
    leak $\xleftarrow{\text{add}}$ $L(\mathsf{leak}_3, \mathsf{leak}_4; \lambda_c)$
    return $w_c^v, w_c^{\overline{v}}, \lambda_c, T$

procedure $L(\mathsf{leak}_1, \mathsf{leak}_2; \lambda_c)$
    if $\lambda_c = 0$ **then**
        return $[\mathsf{leak}_1, \mathsf{leak}_2]$
    else
        return $[\mathsf{leak}_2, \mathsf{leak}_1]$

procedure $L_\oplus(a, b; \lambda_c)$
    if $\lambda_c = 0$ **then**
        return $\mathsf{L}_\oplus(a, b)$
    else
        return $\mathsf{L}_\oplus(b, a)$

Fig. 17: Procedure HXOR(PInputs) and SimXOR(PInputs) as well as procedure LHXOR and LSimXOR where Inputs $= (w_a^v, w_a^{\overline{v}}, w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b)$.

And it can be seen $\mathcal{A}^E$ is $(q_E + 2\ell, t^*)$-bounded for $t^* = O(t + \ell \cdot t_l)$. By this,

$$\begin{aligned}\Pr\left[\mathsf{BadQuery} \text{ in } \mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}\right] &\leq (\ell - 1) \cdot \Pr[s_1 \in \text{Guesses}] \\ &\leq (\ell - 1) \cdot \mathbf{Adv}^{\text{2-up}[q^*]}(\mathcal{A}_2^E) \\ &\leq (\ell - 1) \cdot \mathbf{Adv}^{\text{2-up}[q^*]}(q^*, t^*).\end{aligned}$$

During the real execution $\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,0}$, as long as the event BadQuery never happens, all the keys and key stream blocks are fresh random values independent from τ_E the transcript of E queries of $\mathcal{A}^E$, and have the same distribution as those in the ideal execution $\mathsf{Expt}_{\mathsf{LGSL}}^{\mathcal{A}^E,1}$. Therefore, this proof is finished.

D.2 HXOR vs SimXOR

Let $\mathsf{PInputs} = (w_a^v, w_a^{\overline{v}}, w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b)$. We describe two procedures, $\mathsf{HXOR}(\mathsf{PInputs})$ and $\mathsf{SimXOR}(\mathsf{PInputs})$, in Fig 17. Here, SimXOR $(\mathsf{PInputs})$ is an XOR gate garbling simulator, while $\mathsf{HXOR}(\mathsf{PInputs})$ is an intermediate procedure designed to facilitate the final proof. $\mathsf{HXOR}(\mathsf{PInputs})$ consists of the same computational steps as GbXOR $(w_a^{\pi_a}, w_a^{\overline{\pi}_a}, w_b^{\pi_b}, w_b^{\overline{\pi}_b}, \pi_a, \pi_b)$, but its input $\mathsf{PInputs}$ follows the format of $\mathsf{SimXOR}(\mathsf{PInputs})$. In a sense, $\mathsf{HXOR}(\mathsf{Inputs})$ rearranges the representation of the inputs of GbXOR to aid in understanding and comparison with $\mathsf{SimXOR}(\mathsf{PInputs})$.

In addition, we define $\mathsf{LHXOR} := (\mathsf{HXOR}(\mathsf{PInputs}), \mathsf{leak})$ and $\mathsf{LSimXOR} := (\mathsf{SimXOR}(\mathsf{PInputs}), \mathsf{leak})$ where leak is a list containing the leakages generated in the procedure. We give an experiment in Fig. 18 and have the following lemma for their indistinguishability. We define $\mathbf{Adv}_{\mathsf{XOR}}(q_E, t) \stackrel{\text{def}}{=} \max\big\{\big|\Pr[\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0} = 1] - \Pr[\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,1} = 1]\big|\big\}$ where the maximum is taken over all the (q_E, t)-bounded adversary $\mathcal{A}^E$.

Lemma 2. *For every $(q_E,$ $t)$-bounded adversary $\mathcal{A}^E$ and every $(w_a^-, w_b^-, w_a^v,$ $w_b^v,$ $\lambda_a, \lambda_b)$ specified by $\mathcal{A}^E$, It holds*

$$\mathbf{Adv}_{\mathsf{XOR}}(q_E, t) \leq \mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q^*]}(q^*, t^*)$$

where $q^* = q_E + 2$, $t^* = O(t + 2 \cdot t_l)$, t_l is the total time needed for evaluating L^{in} and L^{out}.

Proof. Consider the execution of $\mathcal{A}^E$ against $\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}$. We define a bad event $\mathsf{BadQuery}$, which occurs when the internal key (label) $\hat{w}_c^{\overline{v}_c}$ appears in the key field of a blockcipher query made by $\mathcal{A}^E$. This event, once happens, would cause the blockcipher output $W_c^{\overline{v}_c}$ to lose randomness. In detail, given an adversary $\mathcal{A}^E$, we build a distinguisher $\mathcal{D}^E$ such that

$$\mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q^*]}(\mathcal{D}^E) \leq \Pr\left[\mathsf{BadQuery}\text{ in }\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}\right] \tag{11}$$

where $q^* = q_E + 2$. Concretely, $\mathcal{D}^E$ runs an instance of $\mathcal{A}^E$ and keeps a record of $\mathcal{A}^E$'s queries to E in a set τ_E. $\mathcal{D}^E$ simulates the following process against $\mathcal{A}^E$:

1. $\mathcal{D}^E$ initialize an empty list leak.
2. $\mathcal{D}^E$ involve $\mathcal{A}^E$, get its input $(w_a^-, w_b^-, w_a^v, w_b^v,$ $\lambda_a, \lambda_b)$ and set $\lambda_c = \lambda_a \oplus \lambda_b$.
3. If $\lambda_b = 0$:
   - $\mathcal{D}^E$ submit $s_1 = w_b^-$, $k_1 = w_b^v$, $s_2 = w_a^-$, $k_2 = w_a^v$, $x = \overline{\lambda}_c$ to its $\mathsf{XOR\text{-}inv}[q]$ challenger (according to our convention) this results in the outputs (T, y, leak) where $\mathsf{leak} = [\mathsf{L}^{out}(w_b^-, \hat{s}_1||r)$, $\mathsf{L}^{out}(w_a^-, \hat{s}_2||r')$, $\widehat{\mathsf{L}_\oplus}(\hat{s}_1, w_b^v)$, $\widehat{\mathsf{L}_\oplus}(\hat{s}_2, w_a^v)$, $\mathsf{L}_\oplus$ (Δ_1, Δ_2), $\widehat{\mathsf{L}_\oplus}(w_b^v, \hat{s}_2)$, $\widehat{\mathsf{L}_\oplus}(\hat{s}_1, \hat{s}_2)$, $\mathsf{L}^{in}(s_3, \overline{\lambda}_c)$, $\mathsf{L}^{out}(s_3, y)]$.
   - It (conceptually) takes the secret $\hat{s}_1||r$, $\hat{s}_1$, $\hat{s}_2||r'$, $\hat{s}_2$, s_3, y as $W_b^{\overline{v}}$, $w_b^{\overline{v}}$, $W_a^{\overline{v}}$, $w_a^{\overline{v}}$, $\hat{w}_c^{\overline{v}}$, $W_c^{\overline{v}}$ and adds $\mathsf{L}^{out}(w_a^-, \hat{s}_2||r')$, $\mathsf{L}^{out}(w_b^-, \hat{s}_1||r)$, $\mathsf{L}_\oplus(w_a^v, \hat{s}_2; \lambda_a)$, $\mathsf{L}_\oplus(w_b^v, \hat{s}_1; \lambda_b)$, $\mathsf{L}_\oplus(\Delta_1, \Delta_2)$ to leak.

procedure $\mathsf{Expt}(w_a^-, w_b^-, w_a^v, w_b^v, \lambda_a, \lambda_b)$
  $\mathsf{PInputs} := (w_a^v, w_a^{\overline{v}}, w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b)$
  $\mathsf{leak} \xleftarrow{\mathsf{add}} \mathsf{EmptyList}$
  if $\mathrm{b} = 0$ **then**
    $(\mathsf{Outputs}, \mathsf{leak}') := \mathsf{LHXOR}(\mathsf{PInputs})$
  else
    $(\mathsf{Outputs}, \mathsf{leak}') := \mathsf{LSimXOR}(\mathsf{PInputs})$
  $\mathsf{leak} \xleftarrow{\mathsf{add}} [\mathsf{L}^{out}(w_a^-, W_a^{\overline{v}}), \mathsf{L}^{out}(w_b^-, W_b^{\overline{v}}), \mathsf{leak}']$
  return $(\mathsf{Outputs}, \mathsf{leak})$

Fig. 18: Experiment $\mathsf{Expt}_{\mathsf{XOR}}$. The INITIALIZE procedure is that choose $W_a^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}, W_b^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}$ randomly and let $w_a^{\overline{v}} = W_a^{\overline{v}}[1 \ldots n], w_b^{\overline{v}} = W_b^{\overline{v}}[1 \ldots n]$. There is no FINALIZE procedure

   - It (conceptually) takes s_3 as $\hat{w}_c^{\overline{v}}$ and compute $\hat{w}_c^v = w_a^v \oplus w_b^v$. Let $\mathsf{leak}_1 = \mathsf{L}_{\oplus}(w_a^v, w_b^v)$, $\mathsf{leak}_2 = \mathsf{L}_{\oplus}(\hat{s}_2, w_b^v)$ and add $L(\mathsf{leak}_1, \mathsf{leak}_2; \lambda_c)$ to leak.
   - It (conceptually) takes y as $W_c^{\overline{v}}$ and compute $W_c^v = E_{\hat{w}_c^v}(\lambda_c)$, $w_c^v = W_c^v[1 \ldots n]$, $w_c^{\overline{v}} = y[1 \ldots n]$. Let $\mathsf{leak}_3 = [\mathsf{L}^{in}(\hat{w}_c^v, \lambda_c), \mathsf{L}^{out}(\hat{w}_c^v, W_c^v)]$, $\mathsf{leak}_4 = [\mathsf{L}^{in}(s_3, \overline{\lambda}_c), \mathsf{L}^{out}(s_3, y)]$ and add $L(\mathsf{leak}_3, \mathsf{leak}_4; \lambda_c)$ to leak.
4. If $\lambda_b = 1$
   - $\mathcal{D}^E$ submit $s_1 = w_a^-$, $k_1 = w_a^v$, $s_2 = w_b^-$, $k_2 = w_b^v$, $x = \overline{\lambda}_c$ to its $\mathsf{XOR\text{-}inv}[q]$ challenger (according to our convention) this results in the outputs (T, y, leak) where $\mathsf{leak} = [\mathsf{L}^{out}(w_a^-, \hat{s}_1||r), \mathsf{L}^{out}(w_b^-, \hat{s}_2||r'), \widehat{\mathsf{L}_{\oplus}}(\hat{s}_1, w_a^v), \widehat{\mathsf{L}_{\oplus}}(\hat{s}_2, w_b^v), \mathsf{L}_{\oplus}(\Delta_1, \Delta_2), \widehat{\mathsf{L}_{\oplus}}(w_a^v, \hat{s}_2), \widehat{\mathsf{L}_{\oplus}}(\hat{s}_1, \hat{s}_2), \mathsf{L}^{in}(s_3, \overline{\lambda}_c), \mathsf{L}^{out}(s_3, y)$.
   - It (conceptually) takes the secret $\hat{s}_1||r$, $\hat{s}_1$, $\hat{s}_2||r'$, $\hat{s}_2$, s_3, y as $W_a^{\overline{v}}$, $w_a^{\overline{v}}$, $W_b^{\overline{v}}$, $w_b^{\overline{v}}$, $\hat{w}_c^{\overline{v}}$, $W_c^{\overline{v}}$.
   - Add $[\mathsf{L}^{out}(w_a^-, \hat{s}_1||r), \mathsf{L}^{out}(w_b^-, \hat{s}_2||r'), \mathsf{L}_{\oplus}(w_a^v, \hat{s}_1, \lambda_a), \mathsf{L}_{\oplus}(w_b^v, \hat{s}_2, \lambda_b), \mathsf{L}_{\oplus}(\Delta_1, \Delta_2)$ to leak.
   - It (conceptually) takes s_3 as $\hat{w}_c^{\overline{v}}$ and compute $\hat{w}_c^v = w_a^v \oplus w_b^v \oplus T$.Let $\mathsf{leak}_1 = \mathsf{L}_{\oplus}(\hat{s}_1, \hat{s}_2)$, $\mathsf{leak}_2 = \mathsf{L}_{\oplus}(w_a^v, \hat{s}_2)$ and add $L(\mathsf{leak}_1, \mathsf{leak}_2; \lambda_c)$ to leak.
   - It (conceptually) takes y as $W_c^{\overline{v}}$ and compute $W_c^v = E_{\hat{w}_c^v}(\lambda_c)$, $w_c^v = W_c^v[1 \ldots n]$, $w_c^{\overline{v}} = y[1 \ldots n]$. Let $\mathsf{leak}_3 = [\mathsf{L}^{in}(\hat{w}_c^v, \lambda_c), \mathsf{L}^{out}(\hat{w}_c^v, W_c^v)]$, $\mathsf{leak}_4 = [\mathsf{L}^{in}(s_3, \overline{\lambda}_c), \mathsf{L}^{out}(s_3, y)]$ and add $L(\mathsf{leak}_3, \mathsf{leak}_4; \lambda_c)$ to leak.
5. Then $\mathcal{D}^E$ starts from w_c^v, $w_c^{\overline{v}}$ to emulate the remaining actions HXOR. Eventually, $\mathcal{D}^E$ return $(w_c^{v_c}, w_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak})$.

The strategy of $\mathcal{D}^E$ is obvious: if $\mathcal{A}^E$ triggers the event $\mathsf{BadQuery}$ then the key $\hat{w}_c^{\overline{v}}$ must be in τ_E. Therefore, $\mathcal{D}^E$ emulates the $\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}$ and provides the leakages to $\mathcal{A}^E$, except for the XOR gate output pre-label $\hat{w}_c^{\overline{v}}$, for which the related leakages and E output are replaced by those obtained from the $\mathsf{XOR\text{-}inv}[q]$ challenger for the seed-preserving property. As we can see, all the inputs sent to $\mathcal{A}^E$ are distributed exactly as those produced by $\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}$. Therefore, when $\mathcal{D}^E$ halts, if $\mathcal{A}^E$ made a query on s_3, then simply outputting τ_E would break

the game. So we have $\Pr\left[\hat{w}_c^{\overline{v}} \in \text{Guesses} \mid \text{BadQuery in } \mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}\right] = 1$. Now, we observe that

$$\begin{aligned}\Pr\left[\hat{w}^{\overline{v}_c} \in \text{Guesses}|\mathsf{BadQuery}\text{ in } \mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}\right] \\ \leq \frac{\Pr[\hat{w}^{\overline{v}_c} \in \text{Guesses}]}{\Pr\left[\mathsf{BadQuery}\text{ in } \mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}\right]}\end{aligned}$$

And it can be seen $\mathcal{D}^E$ is $(q_E + 4\ell, t^*)$-bounded for $t^* = O(t + 2\ell \cdot t_l)$. By this,

$$\begin{aligned}\Pr\left[\mathsf{BadQuery}\text{in } \mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}\right] \leq & \Pr[\hat{w}_c^{\overline{v}} \in \text{Guesses}] \\ \leq & \mathbf{Adv}^{\mathsf{XOR\text{-}inv}[q]}(\mathcal{D}^E)\end{aligned}$$

where $q^* = q_E + 2$, $t^* = O(t + 2 \cdot t_l)$, t_l is the total time needed for evaluating L^{in} and L^{out}.

During the real execution $\mathsf{BadQuery}$ in $\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,0}$, as long as the event $\mathsf{BadQuery}$ never happens, $w_c^{\overline{v}_c}$ is fresh random values independent from τ_E the transcript of blockcipher queries of $\mathcal{A}^E$, and have the same distribution as those in $\mathsf{Expt}_{\mathsf{XOR}}^{\mathcal{A}^E,1}$.

D.3 HAND vs SimAND

Let $\mathsf{PInputs1} = (w_a^v, w_a^{\overline{v}}, w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b, v_a, v_b)$ and let $\mathsf{PInputs2} = (w_a^v, w_a^{\overline{v}}, w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b)$. We describe an intermediate procedure $\mathsf{HAND}(\mathsf{PInputs1})$ and AND gate simulator $\mathsf{SimAND}(\mathsf{PInputs2})$ in Fig. 20. Similarly, HAND $(\mathsf{PInputs1})$ consists of the same computation steps as $\mathsf{GbAND}(w_a^{\pi_a}, w_a^{\overline{\pi}_a}, w_b^{\pi_b}, w_b^{\overline{\pi}_b}, \pi_a, \pi_b)$. To explain this, we give an overview of $\mathsf{HAND}(\mathsf{PInputs1})$.

1. Compute $W_0 = E_{w_a^{\pi_a}}(00) \oplus E_{w_b^{\pi_b}}(00)$ and add its related leakage to list leak (Line 3—6).
   - Note the dependency of $E_{w_a^{\pi_a}}(00)$ on λ_a. Specifically, $E_{w_a^{\pi_a}}(00)$ equals $E_{w_a^{v_a}}(00)$ when $\lambda_a = 0$, and $E_{w_a^{\overline{v_a}}}(00)$ when $\lambda_a = 1$. To accurately compute $E_{w_a^{\pi_a}}(00)$, we define $M_a^{00} := \mathsf{LEnc}(w_a^{v_a}, w_a^{\overline{v_a}}, \lambda_a, 00)$. Similarly, for input wire b, $E_{w_b^{\pi_b}}(00)$ is correctly computed using $M_b^{00} := \mathsf{LEnc}(w_b^{v_b}, w_b^{\overline{v_b}}, \lambda_b, 00)$. Finally, W_0 is computed by taking the XOR of M_a^{00} and M_b^{00}.
2. Assign $w_c^0 \xleftarrow{\$} \{0,1\}^n$ and $w_c^1 \| \overline{\pi}_c := W_0$ if $\pi_a = \pi_b = 1$, or assign $w_c^0 \| \pi_c := W_0$ and $w_c^1 \xleftarrow{\$} \{0,1\}^n$ (lines 8—17).
3. Compute the garbled tables T_1, T_2, T_3 (lines 18—25). Here, we explain how to compute T_1. T_2 and T_3 are computed similarly to T_1.
   - $T_1 = E_{w_a^{\pi_a}}(01) \oplus E_{w_b^{\overline{\pi}_b}}(01) \oplus W_c^{g(\pi_a, \overline{\pi}_b)}$. Similar to the above, we use $M_a^{01} = \mathsf{LEnc}(w_a^{v_a}, w_a^{\overline{v}_a}, \lambda_a, 01)$ to compute $E_{w_a^{\pi_a}}(01)$ and use $M_b^{01} = \mathsf{LEnc}(w_b^{v_a}, w_b^{\overline{v}_a}, \overline{\lambda}_b, 01)$ to compute $E_{w_b^{\overline{\pi}_b}}(01)$. In addition, compute $W_c^{g(\pi_a, \overline{\pi}_b)} = W_c^{g(\lambda_a \oplus v_a, \overline{\lambda}_b \oplus v_b)}$

Then, let's demonstrate the difference between $\mathsf{HAND}(\mathsf{PInputs1})$ and $\mathsf{SimAND}(\mathsf{PInputs2})$, which can help to understand our proof.

1: **procedure** $\mathsf{HAND}(\mathsf{PInputs1})$
2: $\quad$ $\mathsf{leak} := \mathsf{EmptyList}$
3: $\quad$ $(M_a^{00}, \mathsf{leak}_a^{00}) := \mathsf{LEnc}(w_a^v, w_a^{\overline{v}}, \lambda_a, 00)$
4: $\quad$ $(M_b^{00}, \mathsf{leak}_b^{00}) := \mathsf{LEnc}(w_b^v, w_b^{\overline{v}}, \lambda_b, 00)$
5: $\quad$ $\mathsf{leak} \xleftarrow{\mathsf{add}} [\mathsf{leak}_a^{00}, \mathsf{leak}_b^{00}]$
6: $\quad$ $W_0 := M_a^{00} \oplus M_b^{00}$, $\mathsf{leak} \xleftarrow{\mathsf{add}} L^{\oplus}(M_a^{00}, M_b^{00})$
7: $\quad$ $\pi_a := \lambda_a \oplus v_a$, $\pi_b := \lambda_b \oplus v_b$
8: $\quad$ **if** $v_a = v_b = 1$ **then**
9: $\quad\quad$ **if** $\pi_a = \pi_b = 1$ **then**
10: $\quad\quad\quad$ $w_c^{\overline{v}} := \{0,1\}^n$, $w_c^v \| \lambda_c := W_0$
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ $w_c^{\overline{v}} \| \overline{\lambda}_c := W_0$, $w_c^v \xleftarrow{\$} \{0,1\}^n$
13: $\quad$ **else**
14: $\quad\quad$ **if** $\pi_a = \pi_b = 1$ **then**
15: $\quad\quad\quad$ $w_c^v \xleftarrow{\$} \{0,1\}^n$, $w_c^{\overline{v}} \| \overline{\lambda}_c := W_0$
16: $\quad\quad$ **else**
17: $\quad\quad\quad$ $w_c^v \| \lambda_c := W_0$, $w_c^{\overline{v}} \xleftarrow{\$} \{0,1\}^n$
18: $\quad$ $W_c^v = w_c^v \| \lambda_c$, $W_c^{\overline{v}} = w_c^{\overline{v}} \| \overline{\lambda}_c$
19: $\quad$ **for** $2\alpha + \beta \in \{1,2,3\}$ **do**
20: $\quad\quad$ $(M_a^{\alpha\beta}, \mathsf{leak}_a^{\alpha\beta}) := \mathsf{LEnc}(w_a^v, w_a^{\overline{v}}, \lambda_a \oplus \alpha, \alpha\|\beta)$
21: $\quad\quad$ $(M_b^{\alpha\beta}, \mathsf{leak}_b^{\alpha\beta}) := \mathsf{LEnc}(w_b^v, w_b^{\overline{v}}, \lambda_b \oplus \beta, \alpha\|\beta)$
22: $\quad\quad$ $\mathsf{leak} \xleftarrow{\mathsf{add}} [\mathsf{leak}_a^{\alpha\beta}, \mathsf{leak}_b^{\alpha\beta}]$
23: $\quad\quad$ $W_c = W_c^{g(\alpha\oplus\lambda_a\oplus v_a, \beta\oplus\lambda_b\oplus v_b)}$
24: $\quad\quad$ $T_{2\alpha+\beta} := M_a^{\alpha\beta} \oplus M_b^{\alpha\beta} \oplus W_c$
25: $\quad\quad$ $\mathsf{leak} \xleftarrow{\mathsf{add}} L^{\oplus}(M_a^{\alpha\beta}, M_b^{\alpha\beta}, W_c)$
26: $\quad$ **return** $(w_c^v, w_c^{\overline{v}}, \lambda_c, (T_1, T_2, T_3))$
50: **procedure** $\mathsf{LEnc}(w_1, w_2, b, x)$ $\boxed{\mathsf{SimLEnc}(w_1, w_2, b, x)}$
51: $\quad$ **if** $b = 0$ **then**
52: $\quad\quad$ $w_3 = E_{w_1}(x)$
53: $\quad\quad$ $\mathsf{leak}' := [\mathsf{L}^{in}(w_1, x), \mathsf{L}^{out}(w_1, w_3)]$
54: $\quad$ **else**
55: $\quad\quad$ $w_3 := E_{w_2}(x)$ $\boxed{w_c := \{0,1\}^n}$
56: $\quad\quad$ $\mathsf{leak}' := [\mathsf{L}^{in}(w_2, x), \mathsf{L}^{out}(w_1, w_3)]$
57: $\quad$ **return** (w_3, leak')

Fig. 19: Procedure $\mathsf{HAND}(\mathsf{PInputs1})$ where $\mathsf{PInputs1} = (w_a^v, w_a^{\overline{v}} w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b, v_a, v_b)$

1. For all blockciphers E with keys $w_a^{\overline{v}}$ or $w_b^{\overline{v}}$, outputs are randomly sampled, whereas with w_a^v or w_b^v, outputs are computed using the blockcipher. Thus, we use $\mathsf{SimLEnc}$ to compute the blockcipher E.
2. The output labels w_c^v and $w_c^{\overline{v}}$ are assigned differently (The details are in line 36 — line 39).

```
30: procedure SimAND(PInputs2)
31:     leak := EmptyList
32:     (M_a^00, leak_1) := SimLEnc(w_a^v, w_a^v̄, λ_a, 00)
33:     (M_b^00, leak_2) := SimLEnc(w_b^v, w_b^v̄, λ_b, 00)
34:     leak ←add− [leak_a^00, leak_b^00]
35:     W_0 := M_a^00 ⊕ M_b^00, leak ←add− L^⊕(M_a^00, M_b^00)
36:     if λ_a = λ_b = 0 then
37:         w_c^v‖λ_c := W_0, w_c^v̄ ←$− {0,1}^n
38:     else
39:         w_c^v‖λ_c ←$− {0,1}^(n+1), w_c^v̄ ←$− {0,1}^n
40:     for 2α + β ∈ {1,2,3} do
41:         (M_a^00, leak_a^αβ) := SimLEnc(w_a^v, w_a^v̄, λ_a ⊕ α, α‖β)
42:         (M_b^00, leak_b^αβ) := SimLEnc(w_b^v, w_b^v̄, λ_b ⊕ β, α‖β)
43:         leak ←add− [leak_3, leak_4]
44:         if 2α + β = 2λ_a + λ_b then W_c := w_c^v‖λ_c
45:         else w_c ←$− {0,1}^(n+1)
46:         T_(2α+β) := M_a^αβ ⊕ M_b^αβ ⊕ W_c
47:         leak ←add− L^⊕(M_a^αβ, M_b^αβ, W_c)
48:     return (λ_c, w_c^v, w_c^v̄, (T_1, T_2, T_3))
```

Fig. 20: Procedure SimAND(PInputs2) where PInputs2 $= (w_a^v, w_a^{\overline{v}} w_b^v, w_b^{\overline{v}}, \lambda_a, \lambda_b)$.

```
procedure Expt(w_a^-, w_b^-, w_a^v, w_b^v, v_a, v_b, λ_a, λ_b)
    Inputs1 := (w_a^v, w_a^v̄, w_b^v, w_b^v̄, v_a, v_b, λ_a, λ_b)
    Inputs2 := (w_a^v, w_a^v̄, w_b^v, w_b^v̄, λ_a, λ_b)
    if b = 0 then
        (Outputs, leak) := LHAND(PInputs1)
    else
        (Outputs, leak) := LSimAND(PInputs2)
    leak ←add− L^out(w_a^-, W_a^v̄), L^out(w_b^-, W_b^v̄)
    return (Outputs, leak)
```

Fig. 21: Experiment $\mathsf{Expt}_{\mathsf{AND}}$. The Initialize procedure is that choose $W_a^{\overline{v}} := \{0,1\}^{n+1}, W_b^{\overline{v}} := \{0,1\}^{n+1}$ randomly and let $w_a^{\overline{v}} = W_a^{\overline{v}}[1 \cdots n], w_b^{\overline{v}} = W_b^{\overline{v}}[1 \cdots n]$. There is no Finalize procedure

- If $\lambda_a = \lambda_b = 0$, the evaluator will compute $w_c^v \| \lambda_c = E_{w_a^v} \oplus E_{w_b^v}$ (namely W_0). Thus, the simulator set $w_c^v \| \lambda_c := W_0$ and choose $w_c^{\overline{v}}$ randomly.
- Else, the evaluator will compute $w_c^v \| \lambda_c$ using the garbled table. Thus, the simulator choose $w_c^v \| \lambda_c$ and $w_c^{\overline{v}}$ randomly.

3. The garbled table is computed differently (The details are in line 44 — line 47). the evaluator will compute $w_c^v \| \lambda_c$ using the garbled table $T_{2\lambda_a+\lambda_b}$ (think $T_0 = 0$). Thus, the evaluator will compute $T_{2\lambda_a+\lambda_b}$ "correctly" whereas computing another garbled table "wrongly". Concretely:
   - Compute $T_{2\lambda_a+\lambda_b} = E_{w_a^v}(g\|\lambda_a\lambda_b) \oplus E_{w_b^v}(g\|\lambda_a\lambda_b) \oplus W_c^v$ where $W_c^v = w_c^v\|\lambda_c$.
   - Compute $T_{2\alpha+\beta} = M_a^{\alpha\beta} \oplus M_b^{\alpha\beta} \oplus R$ where $M_a^{\alpha\beta} = E_{w_a^v}(g\|\lambda_a\lambda_b)$ if $\lambda_a = 0$, $M_a^{\alpha\beta}$ is choosed randomly, $M_b^{\alpha\beta}$ is computed similarly, and R is a ramdom $n+1$ bits string.

For the indistinguishability of procedure $\mathsf{HAND}(\mathsf{PInputs}1)$ and $\mathsf{SimAND}(\mathsf{PInputs}2)$, we give a game(experiment) in Fig. 21 and define $\mathbf{Adv}_{\mathsf{AND}}(q_E, t) = \Pr[\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,0} = 1] - \Pr[\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,1} = 1]$ where adversary is (q_E, t) bound. Then we have the following lemma.

Lemma 3. *For every $(q_E,\ t)$-bounded adversary $\mathcal{A}^E$ and $(w_a^-,\ w_b^-, w_a^v, w_b^v, v_a, v_b, \lambda_a, \lambda_b)$ specified by $\mathcal{A}^E$, It holds*

$$\mathbf{Adv}_{\mathsf{AND}}^{\mathcal{A}^E}(q_E, t) \leq 2 \cdot \mathbf{Adv}^{\text{2-up}[q^*]}(q^*, t^*) + \mathbf{Adv}^{\mathsf{XOR\text{-}res}}(q^*, t^*) + 3\mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(q^*, t^*) + \frac{3}{2^{n+1}} \tag{12}$$

where $q^* = (q_E+8)$, $t^* = O(t+8\cdot t_l)$, t_l is the total time needed for evaluating L^{in} and L^{out}.

Proof. Consider an adversary $\mathcal{A}^E$ against $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,0}$. We define seven games, $\mathsf{G}_0, \ldots, \mathsf{G}_6$, each capturing the interaction between $\mathcal{A}^E$ and a corresponding intermediate world H_i. Game G_0 is identical to $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,0}$. Using a hybrid argument, we progressively modify the experiment through these games, culminating in G_6, which aligns with $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,1}$. Specifically, G_1 and G_2 alter the output of the blockcipher with an 'unknown' key from $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,0}$, replacing it with a random value. G_3 modifies how the output label is assigned. Games G_4, G_5, and G_6 introduce changes in the computation of the garbled table. We detail the specifics of each intermediate world H_i and explain how differences between successive worlds can be reduced to our underlying assumptions.

H_0 : This is identical to what the adversary obtains in $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,0}$.

H_1 : For the blockcipher with key $w_a^{\overline{v}}$, sample the output randomly. Concretely, we let $(M_a^{\overline{\lambda}_a 0}, \mathsf{leak}_a^{\overline{\lambda}_a 0}) := \mathsf{SimLEnc}(w_a^v, w_a^{\overline{v}}, 1, \overline{\lambda}_a\|0)$, $(M_a^{\overline{\lambda}_a 1}, \mathsf{leak}_a^{\overline{\lambda}_a 1}) := \mathsf{SimLEnc}(w_a^v, w_a^{\overline{v}}, 1, \overline{\lambda}_a\|1)$. This difference can be easily reduced to assumption $\text{2-up}[q^*]$ (Similar to the proof of lemma.). Concretely, we have

$$\left|\Pr[\mathcal{A}^E(\mathsf{H}_1) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathsf{H}_0) \Rightarrow 1]\right| \leq \mathbf{Adv}^{\text{2-up}[q^*]}(q^*, t^*)$$

H_2 : For the blockcipher with key $w_a^{\overline{v}}$, sample the output randomly. Concretely, we let $(M_b^{0,\overline{\lambda}_b}, \mathsf{leak}_b^{0,\overline{\lambda}_b}) := \mathsf{SimLEnc}(w_b^v, w_b^{\overline{v}}, 1, 0\|\overline{\lambda}_b)$, $(M_b^{1,\overline{\lambda}_b}, \mathsf{leak}_b^{1,\overline{\lambda}_b}) := \mathsf{SimLEnc}(w_b^v, w_b^{\overline{v}}, \overline{\lambda}_b, 1\|\lambda_b)$.

As discussed above, this difference can be reduced to an assumption $\mathsf{2\text{-}up}[q^*]$. Concretely, we have

$$\left|\Pr[\mathcal{A}^E(\mathsf{H}_2) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathsf{H}_1) \Rightarrow 1]\right| \leq \mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*, t^*)$$

H_3 : We modify the assignment of w_c^v and $w_c^{\overline{v}}$ to align with SimAND ($\mathsf{PInputs2}$). Specifically, if $\lambda_a = \lambda_b = 0$, there are no changes. Otherwise, we sample $w_c^v \| \lambda_c$ uniformly from $\{0,1\}^{n+1}$ and $w_c^{\overline{v}}$ from $\{0,1\}^n$. We can reduce this difference to $\mathsf{XOR\text{-}res}$ Assumption. Concretely, we have

$$\left|\Pr[\mathcal{A}^E(\mathsf{H}_3) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathsf{H}_2) \Rightarrow 1]\right| \leq \mathbf{Adv}^{\mathsf{XOR\text{-}res}}(q^*, t^*) + \frac{1}{2^{n+1}}$$

Proof. If $\lambda_a = \lambda_b = 0$, $\mathsf{H}_2 \equiv \mathsf{H}_3$. Otherwise, consider a (q_E, t)-bounded adversary $\mathcal{A}^E$ against H_2 and H_3, we build an adversary $\mathcal{A}_2^E$ against the distribution defined in Eq. (6). Concretely, $\mathcal{A}_2^E$ proceeds with the following steps:

1. $\mathcal{A}_2^E$ initialize an empty list leak. Sample $W_a^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}$, $W_b^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}$. Let $w_a^{\overline{v}} := W_a^{\overline{v}}[1 \ldots n]$, $w_b^{\overline{v}} := W_b^{\overline{v}}[1 \ldots n]$.
2. $\mathcal{A}_2^E$ involve the adversary $\mathcal{A}^E$ and get its input w_a^-, w_b^-, w_a^v, w_b^v, v_a, v_b, λ_a, λ_b. Then, $\mathcal{A}_2^E$ adds $\mathsf{L}^{out}(w_a^-, W_a^{\overline{v}})$, $\mathsf{L}^{out}(w_b^-, W_b^{\overline{v}})$ to leak.
3. If $\lambda_b = 1$ ($\lambda_a = 0$ or $\lambda_a = 1$)
   - $(M_a^{00}, \mathsf{leak}_a^{0,0}) := \mathsf{SimLEnc}(w_a^v, w_a^{\overline{v}}, \lambda_a, 00)$. Add $\mathsf{leak}_a^{0,0}$ to leak.
   - submit $s_0 = w_b^{\overline{v}}, m_0 = M_a^{00}$ to its $\mathsf{XOR\text{-}res}$ challenger. According to our convention, $\mathcal{A}_2^E$ will get $(y(or\ R), \mathsf{leak}')$ where $\mathsf{leak}' = \mathsf{L}^{out}(w_b^{\overline{v}}, s_1), \widehat{\mathsf{L}_\oplus}(s_1, M_a^{00})$. Add $\mathsf{L}^{in}(w_b^{\overline{v}}, 00)$, $\mathsf{L}^{out}(w_b^{\overline{v}}, s_1)$, $\mathsf{L}_\oplus(M_a^{00}, s_1)$ to leak.
   - If $v_a = v_b = 1$
     - If $\pi_a = \pi_b = 1$: $w_c^{\overline{v}} := \{0,1\}^n$, $w_c^v \| \lambda_c := (y\ or\ R)$
     - Else: $w_c^{\overline{v}} \| \overline{\lambda}_c := (y\ or\ R)$, $w_c^v \xleftarrow{\$} \{0,1\}^n$
   - Else ($v_a \neq 1$ or $v_b \neq 1$)
     - If $\pi_a = \pi_b = 1$: $w_c^v := \{0,1\}^n$, $w_c^{\overline{v}} \| \overline{\lambda}_c := (y\ or\ R)$
     - Else $w_c^v \| \lambda_c := (y\ or\ R)$, $w_c^{\overline{v}} \xleftarrow{\$} \{0,1\}^n$
4. If $\lambda_b = 0$, ($\lambda_a = 1$):
   - $M_b^{00}, \mathsf{leak}_b^{00} := \mathsf{SimLEnc}(w_b^v, w_b^{\overline{v}}, \lambda_b, 00)$.
   - submit $s_0 = w_a^{\overline{v}}, m_0 = M_b^{00}$ to its $\mathsf{XOR\text{-}res}$ challenger. According to our convention, $\mathcal{A}_2^E$ will get $(y(or\ R), \mathsf{leak}')$ where $\mathsf{leak}' = \mathsf{L}^{out}(w_a^{\overline{v}}, s_1), \widehat{\mathsf{L}_\oplus}(s_1, M_b^{00})$. Add $\mathsf{L}^{in}(w_a^{\overline{v}}, 00)$, $\mathsf{L}^{out}(w_a^{\overline{v}}, s_1)$, leak_b^{00}, $\mathsf{L}_\oplus(s_1, M_b^{00})$ to leak.
   - If $v_a = v_b = 1$ (must be $\pi_a = 0, \pi_b = 1$): $w_c^{\overline{v}} \| \overline{\lambda}_c := y\ or\ R$, $w_c^v \xleftarrow{\$} \{0,1\}^n$.
   - Else:
     - If $\pi_a = \pi_b = 1$: $w_c^v \xleftarrow{\$} \{0,1\}^n$, $w_c^{\overline{v}} \| \overline{\lambda}_c := (y\ or\ R)$.
     - Else, $w_c^v \| \lambda_c := (y\ or\ R)$, $w_c^{\overline{v}} \xleftarrow{\$} \{0,1\}^n$
   - If $\pi_a = \pi_b = 1$: let $w_c^{\overline{v}} \| \overline{\lambda}_c = (y\ or\ R)$ and $w_c^v := \{0,1\}^n$).
   - Else, let $w_c^v \| \lambda_c = (y\ or\ R)$ and $w_c^{\overline{v}} := \{0,1\}^n$).
5. Then emulate the remaining action of H_2.

It can be seen that, as long as $M_a^{0,0} \neq s_1$ when $\lambda_b = 1$ or $M_b^{0,0} \neq s_1$ when $\lambda_b = 0$, depending on whether the input tuple received by $\mathcal{A}_2^E$ is (y, leak) or (R, leak_1) outputted by the challenger $\mathsf{XOR\text{-}res}$, the inputs to $\mathcal{A}^E$ is identical to H_2 and H_3. Note that $\Pr[M_{1[p]}^{0,0} = s_1] = 1/2^{n+1}$. Moreover, $\mathcal{D}^E$ is (q^*, t^*)-bounded if adversary $\mathcal{A}^E$ is (q_F, t) bounded. Then, the proof is finished.

H_4 : When $\lambda_a = 0$ and $\lambda_b = 1$, there are no changes. Otherwise, let $T_1 = M_a^{01} \oplus M_b^{01} \oplus R$. We can reduce this difference to $\mathsf{XOR\text{-}ope}$ Assumption. Concretely, we have

$$\left| \Pr[\mathcal{A}^E(\mathsf{H}_4) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathsf{H}_3) \Rightarrow 1] \right| \\ \leq \mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(q^*, t^*) + \frac{1}{2^{n+1}}$$

Proof. If $\lambda_a = 0$, $\lambda_b = 1$, $\mathsf{H}_4 \equiv \mathsf{H}_3$. Otherwise, consider an adversary $\mathcal{A}^E$ against H_4 and H_5, we construct a distinguisher $\mathcal{D}^E$ against the distribution defined in Eq. (5). Concretely

1. $\mathcal{D}^E$ initialize an empty list leak, sample $W_a^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}$, $W_b^{\overline{v}} \xleftarrow{\$} \{0,1\}^{n+1}$, let $w_a^{\overline{v}} := W_a^{\overline{v}}[1 \dots n]$, $w_b^{\overline{v}} := W_b^{\overline{v}}[1 \dots n]$.
2. $\mathcal{D}^E$ involve the adversary $\mathcal{A}^E$ and get its input $w_a^-, w_b^-, w_a^v, w_b^v, v_a, v_b, \lambda_a, \lambda_b$. Then, $\mathcal{D}^E$ add $\mathsf{L}^{out}(w_a^-, W_a^{\overline{v}})$, $\mathsf{L}^{out}(w_b^-, W_b^{\overline{v}})$ to leak.
3. Compute $(M_a^{00}, \mathsf{leak}_a^{00}) := \mathsf{SimLEnc}\ (w_a^v,\ w_a^{\overline{v}},\ \lambda_a, 00)$ and $(M_b^{00},\ \mathsf{leak}_b^{00}) := \mathsf{SimLEnc}(w_b^v, w_b^{\overline{v}}, \lambda_b, 00)$.
4. Compute $W_0 = M_a^{00} \oplus M_b^{00}$ and add $\mathsf{leak}_a^{00}, \mathsf{leak}_b^{00}, \mathsf{L}_\oplus(M_a^{00}, M_b^{00})$ to leak.
5. If $\lambda_a = \lambda_b = 0$, then $w_c^v || \lambda_c = W_0$, $w_c^{\overline{v}} \xleftarrow{\$} \{0,1\}^n$
6. Otherwise, $w_c^v || \lambda_c \xleftarrow{\$} \{0,1\}^{n+1}$, $w_c^{\overline{v}} \xleftarrow{\$} \{0,1\}^n$
7. Set $W_c^v := w_c^v \| \lambda_c$, $W_c^{\overline{v}} := w_c^{\overline{v}} \| \overline{\lambda}_c$
8. If $\lambda_b = 0$ ($\lambda_a = 0$ or $\lambda_a = 1$):
   - Compute $(M_a^{01}, \mathsf{leak}_a^{01}) := \mathsf{SimLEnc}(w_a^v, w_a^{\overline{v}}, \lambda_a, 00)$ and add leak_a^{01} to leak.
   - Submit $s_0 = w_b^{\overline{v}_b}$, $m_0^0 = M_a^{0,1}$ and $m_0^1 := W_c^{g(\lambda_a \oplus v_a, 1 \oplus \lambda_b \oplus v_b)}$, $m_1^0 := M_a^{0,1}$, $m_1^1 \xleftarrow{\$} \{0,1\}^{n+1}$ to its challenger $\mathsf{XOR\text{-}ope}$. Then, the distinguisher $\mathcal{D}^E$ will get y_b and leak_b where $\mathsf{leak}_b = [\mathsf{L}^{out}(w_b^{\overline{v}_b}, s_1),\ \mathsf{L}_\oplus(s_1, m_0^b, m_1^b)$, $\mathsf{L}_\oplus(m_0^b, s_1, m_1^b)]$. Let $T_1 := y_b$ and add $\mathsf{L}^{in}(w_b^{\overline{v}_b}, 01)$, $\mathsf{L}^{out}(w_b^{\overline{v}_b}, s_1)$, $\mathsf{L}_\oplus(m_0^b, s_1, m_1^b)$ to leak.
9. If $\lambda_b = 1$, ($\lambda_a = 1$):
   - Compute $(M_b^{01}, \mathsf{leak}_b^{01}) := \mathsf{SimLEnc}(w_b^v, w_b^{\overline{v}}, 0, 00)$
   - Submit $s_0 = w_a^{\overline{v}_b}$, $m_0^0 = M_b^{0,1}$ and $m_0^1 := W_c^{\overline{v}}$, $m_1^0 := M_a^{0,1}$, $m_1^1 \xleftarrow{\$} \{0,1\}^{n+1}$ to its challenger $\mathsf{XOR\text{-}ope}$. Then, the distinguisher $\mathcal{D}^E$ will get y_b and leak_b where $\mathsf{leak}_b = [\mathsf{L}^{out}(w_a^{\overline{v}_a}, s_1), \mathsf{L}_\oplus(s_1, m_0^b, m_1^b), \mathsf{L}_\oplus(m_0^b, s_1, m_1^b)]$. Let $T_1 := y_b$ and add $\mathsf{L}^{in}(w_a^{\overline{v}}, 01)$, $\mathsf{L}^{out}(w_a^{\overline{v}_a}, s_1)$, leak_b^{01}, $\mathsf{L}_\oplus(s_1, m_0^b, m_1^b)$ to leak.
10. Emulates the remaining action of H_3, return the output to $\mathcal{A}^E$ and output the guess of $\mathcal{A}^E$ b'.

It can be seen that, as long as $M_a^{1,1} \neq s_1$ (if $\lambda_b = 0$) or $M_b^{1,1} \neq s_1$ (if $\lambda_b = 1$), depending on whether the input tuple received by $\mathcal{D}^E$ is (y_0, leak_0) or (y_1, leak_1) outputted by the challenger $\mathsf{XOR\text{-}ope}$, the inputs to $\mathcal{A}^E$ is identical to H_0 and H_1. Note that $\Pr[M_b^{1,1} = s_1] = 1/2^{n+1}$. Moreover, $\mathcal{D}^E$ is (q^*, t^*)-bounded if adversary $\mathcal{A}^E$ is (q_F, t) bounded. Then, the proof is finished.

H_5 : If $\lambda_a = 1$ and $\lambda_b = 0$, there are no changes, and $\mathsf{H}_5 \equiv \mathsf{H}_4$. Otherwise, let $T_2 = M_a^{10} \oplus M_b^{10} \oplus R$ and we can reduce this difference to $\mathsf{XOR\text{-}ope}$ (The proof is similar to the above). Concretely, we have

$$\left| \Pr[\mathcal{A}^E(\mathsf{H}_5) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathsf{H}_4) \Rightarrow 1] \right| \leq \mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(q^*, t^*) + \frac{1}{2^{n+1}}$$

H_6 : If $\lambda_a = 1, \lambda_b = 1$, there are no changes, and $\mathsf{H}_6 \equiv \mathsf{H}_5$. Otherwise, let $T_3 = M_a^{11} \oplus M_b^{11} \oplus R$ and we can reduce this difference to $\mathsf{XOR\text{-}ope}$ (The proof is similar to the above). Concretely, we have

$$\left| \Pr[\mathcal{A}^E(\mathsf{H}_6) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathsf{H}_5) \Rightarrow 1] \right| \leq \mathbf{Adv}^{\mathsf{XOR\text{-}ope}}(q^*, t^*) + \frac{1}{2^{n+1}}$$

It's easy to see that G_0 and G_6 capture the game $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,0}$ and $\mathsf{Expt}_{\mathsf{AND}}^{\mathcal{A}^E,1}$ respectively. Then, using a union bound, we can get Eq. (12) (Note that one of the cases must meet $\mathsf{H}_i \equiv \mathsf{H}_{i+1} (i \in \{2,3,4\})$, so there is a term $\frac{3}{2^{n+1}}$ in this formula instead of $\frac{4}{2^{n+1}}$).

Then, using a union bound, we can get this Lemma.

D.4 Security proof of GLNPLR

First, We describe a simulator $\mathcal{S}$ and hybrid schemes $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4$ for the prvL experiment. $\mathcal{S}$ is invoked with input $(f, f(x))$ and works in Fig. 23 . As we will show, $\mathcal{S}$ will define an *active label* on a wire a and denote it by w_a^v. This label will be the one "obtained" by the evaluator. The other label is *inactive* and is denoted by $k_a^{\overline{v}}$. Note that the simulator doesn't know the value of v_a.

We begin by proving that our garbling scheme achieves privacy.

1. $\mathcal{G}_1(f, x)$: The procedure generate (F, X, d) is same as the adversary obtained from $\mathsf{Expt}_{\mathsf{PrvSimL}}^{\mathrm{GLNPLR},\mathcal{S},0}$.
2. $\mathcal{G}_2(f, x) \equiv \mathcal{G}_1(f, x)$. We just use the active label $w_i^{v_i}$ and inactive label $w_i^{\overline{v}_i}$ for all wire i, but these computations and the leakage distribution are the same.
3. $\mathcal{G}_3(f, x) \approx \mathcal{G}_2(f, x)$. We replace $\mathsf{GSL}(w_i, \ell_i)$ with SimGSL (w_i, ℓ_i) for all circuit input wire i. For the differences between $\mathcal{G}_2(f, x)$ and $\mathcal{G}_3(f, x)$, we have:

$$\left| \Pr[\mathcal{A}^E(\mathcal{G}_2(f, x)) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathcal{G}_3(f, x)) \Rightarrow 1] \right| \leq \sum^{i \in \mathsf{Inputs}} (\ell_i - 1) \cdot \mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*, t^*)$$

procedure $\mathcal{G}_1(f, x)$
  for $i \in \mathsf{Inputs}$ **do**
    $w_i^{\pi_i}, w_i^{\overline{\pi}_i} \xleftarrow{\$} \{0,1\}^n$, $\pi_i \xleftarrow{\$} \{0,1\}$, $e[i, \pi_i] := w_i^{\pi_i} \| 0$, $e[i, \overline{\pi}_i] := w_i^{\overline{\pi}_i} \| 1$
    $(w_{i[0]}^{\pi_i}, \cdots, w_{i[\ell]}^{\pi_i}, \mathsf{leak}_{\mathsf{GSL}}^{\pi_i}) := \mathsf{LGSL}(w_i^{\pi_i}, 0)$
    $(w_{i[0]}^{\overline{\pi}_i}, \cdots, w_{i[\ell]}^{\overline{\pi}_i}, \mathsf{leak}_{\mathsf{GSL}}^{\overline{\pi}_i}) := \mathsf{LGSL}(w_i^{\overline{\pi}_i}, 1)$
  for $(a[p], b[q], c, G) \in \mathsf{Gates}$ **do**
    $\mathsf{PInputs} = (w_{a[p]}^{\pi_a}, w_{a[p]}^{\overline{\pi}_a}, w_{b[q]}^{\pi_b}, w_{b[q]}^{\overline{\pi}_b}, \pi_a, \pi_b)$
    if $G = \mathsf{XOR}$ **then**
      $(w_c^{\pi_c}, w_c^{\overline{\pi}_c}, \pi_c, F[c], \mathsf{leak}) := \mathsf{LGbXOR}(\mathsf{PInputs})$
    else if $G = \mathsf{AND}$ **then**
      $(w_c^{\pi_c}, w_c^{\overline{\pi}_c}, \pi_c, F[c], \mathsf{leak}) := \mathsf{LGbAND}(\mathsf{PInputs})$
    else $w_c^{\pi_c} := w_a^{\overline{\pi}_a}$, $w_c^{\overline{\pi}_c} := w_a^{\pi_a}$, $\pi_c = \overline{\pi}_a$
    $(w_{c[0]}^{v_c}, \cdots, w_{c[\ell]}^{v_c}, \mathsf{leak}) := \mathsf{LGSL}(w_c^{v_c}, \lambda_c)$
    $(w_{c[0]}^{\overline{v}_c}, \cdots, w_{c[\ell]}^{\overline{v}_c}, \mathsf{leak}) := \mathsf{LGSL}(w_c^{\overline{v}_c}, \overline{\lambda}_c)$
  for $i \in \mathsf{Outputs}$ **do**
    $d[i, \pi_i] := F_{k_i^{\pi_i}}(0)$, $d[i, 1] := F_{k_i^{\overline{\pi}_i}}(1)$
  return (F, X, d, leak)

procedure $\mathcal{G}_2(f, x)$ / $\mathcal{G}_3(f, x)$
  for $i \in \mathsf{Inputs}$ **do**
    $w_i^{v_i}, w_i^{\overline{v}_i} \xleftarrow{\$} \{0,1\}^n$, $\lambda_i \xleftarrow{\$} \{0,1\}$, $e[i] := w_i^{v_i} \| \lambda_i$
    $(w_{i[0]}^{v_i}, \cdots, w_{i[\ell]}^{v_i}, \mathsf{leak}) := \mathsf{LGSL}(w_i^{v_i}, \lambda_i)$
    $(w_{i[0]}^{\overline{v}_i}, \cdots, w_{i[\ell]}^{\overline{v}_i}, \mathsf{leak}) := \mathsf{LGSL}(w_i^{\overline{v}_i}, \overline{\lambda}_i)$ ▷ $\mathcal{G}_2$
    $(w_{i[0]}^{\overline{v}_i}, \cdots, w_{i[\ell]}^{\overline{v}_i}, \mathsf{leak}) := \mathsf{LSimGSL}(w_i^{\overline{v}_i}, \overline{\lambda}_i)$ ▷ $\mathcal{G}_3$
  for $(a, b, c, G) \in \mathsf{Gates}$ **do**
    $\mathsf{PInputs}\ 1 = (w_a^{v_a}, w_a^{\overline{v}_a}, w_b^{v_b}, w_b^{\overline{v}_b}, \lambda_a, \lambda_b)$
    $\mathsf{PInputs}\ 2 = (w_a^{v_a}, w_a^{\overline{v}_a}, w_b^{v_b}, w_b^{\overline{v}_b}, \lambda_a, \lambda_b, v_a, v_b)$
    if $G = \mathsf{XOR}$ **then**
      $(w_c^{v_c}, w_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak}_{\mathsf{HXOR}}) := \mathsf{LHXOR}(\mathsf{PInputs1})$
    else if $G = \mathsf{AND}$ **then**
      $(k_c^{v_c}, k_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak}_{\mathsf{HAND}}) := \mathsf{LHAND}(\mathsf{PInputs2})$
    else $w_c^{\lambda_c} := w_a^{\overline{\lambda}_a}$, $w_c^{\overline{\lambda}_c} := w_a^{\lambda_a}$, $\lambda_c = \overline{\lambda}_a$
    $v_c = G(v_a, v_b)$ (if G = NOT, think $v_b = \perp$)
    $(w_{c[0]}^{v_c}, \cdots, w_{c[\ell]}^{v_c}, \mathsf{leak}) := \mathsf{LGSL}(w_c^{v_c}, \lambda_c)$
    $(w_{c[0]}^{\overline{v}_c}, \cdots, w_{c[\ell]}^{\overline{v}_c}, \mathsf{leak}) := \mathsf{LGSL}(w_c^{\overline{v}_c}, \overline{\lambda}_c)$
  for $i \in \mathsf{Outputs}$ **do**
    $d[i, f(x)_i] := F_{k_i^{v_i}}(\lambda_i)$, $d[i, \overline{f(x)_i}] := E_{k_i^{\overline{v}_i}}(\overline{\lambda}_i)$
  return (F, X, d, leak)

Fig. 22: Procedures for the proof of GLNPLR privacy. Additionally, each procedure incorporates an implicit input $\mathsf{L}_{\mathsf{Garble}}$. (Continued in Fig. 23)

procedure $\mathcal{G}_4(f, f(x))$

  for $i \in$ Inputs **do**

    $k_i^{v_i}, k_i^{\overline{v}_i} \xleftarrow{\$} \{0,1\}^n,\ \lambda_i \xleftarrow{\$} \{0,1\},\ X[i] := k_i^{v_i} \| \lambda_i$

    $(w_{i[0]}^{v_i}, \cdots, w_{i[\ell]}^{v_i}, \mathsf{leak}_{\mathsf{GSL}}) := \mathsf{GSL}(w_i^{v_i}, \lambda_i)$

    $(w_{i[0]}^{\overline{v}_i}, \cdots, w_{i[\ell]}^{\overline{v}_i}, \mathsf{leak}_{\mathsf{SimGSL}}) := \mathsf{SimGSL}(w_i^{\overline{v}_i}, \overline{\lambda}_i)$

  for $(a[p], b[q], c, G) \in$ Gates **do**

    $\mathsf{PInputs} = (w_a^{v_a}, w_a^{\overline{v}_a}, w_b^{v_b}, w_b^{\overline{v}_b}, \lambda_a, \lambda_b)$

    if $G =$ XOR **then**

      $(w_c^{v_c}, w_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak}_{\mathsf{XOR}}) := \mathsf{LSimXOR}(\mathsf{PInputs})$

    else if $G =$ AND **then**

      $(w_c^{v_c}, w_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak}_{\mathsf{AND}}) := \mathsf{LSimAND}(\mathsf{PInputs})$

    else

      $w_c^{\lambda_c} := w_a^{\overline{\lambda}_a},\ w_c^{\overline{\lambda}_c} := w_a^{\lambda_a},\ \lambda_c = \overline{\lambda}_a$

    $(w_{c[0]}^{v_c}, \cdots, w_{c[\ell]}^{v_c}, \mathsf{leak}) := \mathsf{LGSL}(w_c^{v_c}, \lambda_c)$

    $(w_{c[0]}^{\overline{v}_c}, \cdots, w_{c[\ell]}^{\overline{v}_c}, \mathsf{leak}) := \mathsf{LSimGSL}(w_c^{\overline{v}_c}, \overline{\lambda}_c)$

  for $i \in$ Outputs **do**

    $d[i, f(x)_i] := F_{k_i^{v_i}}(\lambda_i),\ d[i, \overline{f(x)_i}] := E_{k_i^{\overline{v}_i}}(\overline{\lambda}_i)$

  return (F, X, d, leak)

procedure $\mathcal{S}(f, f(x))$

  for $i \in$ Inputs **do**

    $k_i^{v_i}, k_i^{\overline{v}_i} \xleftarrow{\$} \{0,1\}^n,\ \lambda_i \xleftarrow{\$} \{0,1\},\ X[i] := k_i^{v_i} \| \lambda_i$

    $(w_{i[0]}^{v_i}, \cdots, w_{i[\ell]}^{v_i}, \mathsf{leak}_{\mathsf{GSL}}) := \mathsf{GSL}(w_i^{v_i}, \lambda_i)$

    $(w_{i[0]}^{\overline{v}_i}, \cdots, w_{i[\ell]}^{\overline{v}_i}, \mathsf{leak}_{\mathsf{SimGSL}}) := \mathsf{SimGSL}(w_i^{\overline{v}_i}, \overline{\lambda}_i)$

  for $(a[p], b[q], c, G) \in$ Gates **do**

    $\mathsf{PInputs} = (w_a^{v_a}, w_a^{\overline{v}_a}, w_b^{v_b}, w_b^{\overline{v}_b}, \lambda_a, \lambda_b)$

    if $G =$ XOR **then**

      $(w_c^{v_c}, w_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak}_{\mathsf{XOR}}) := \mathsf{LSimXOR}(\mathsf{PInputs})$

    else if $G =$ AND **then**

      $(w_c^{v_c}, w_c^{\overline{v}_c}, \lambda_c, T, \mathsf{leak}_{\mathsf{AND}}) := \mathsf{LSimAND}(\mathsf{PInputs})$

    $(w_{c[0]}^{v_c}, \cdots, w_{c[\ell]}^{v_c}, \mathsf{leak}) := \mathsf{LGSL}(w_c^{v_c}, \lambda_c)$

    $(w_{c[0]}^{\overline{v}_c}, \cdots, w_{c[\ell]}^{\overline{v}_c}, \mathsf{leak}) := \mathsf{LSimGSL}(w_c^{\overline{v}_c}, \overline{\lambda}_c)$

  for $i \in$ Outputs **do**

    $d[i, f(x)_i] := F_{w_i^{v_i}}(\lambda_i),\ d[i, \overline{f(x)_i}] \xleftarrow{\$} \{0,1\}^{n+1}$

  return (F, X, d, leak)

Fig. 23: Procedures for the proof. Additionally, each procedure incorporates an implicit input $\mathsf{L}_{\mathsf{Garble}}$. (Continued from Fig. 22)

where $q^* = (12g_1 + 6g_2 - 4\ell_{in} + 6\ell_{out} + q_E)$, $t^* = O(t + (12g_1 + 6g_2 - 4\ell_{in} + 6\ell_{out} + q_E)t_l)$, g_1, g_2, g_3 is the number of AND gate, XOR gate and XOR gate respectively, ℓ_{in} is the number of input wires, ℓ_{out} is the number of output wires and t_l is the total time needed for evaluating L^{in}, L^{out}.

4. $\mathcal{G}_4(f, f(x)) \approx \mathcal{G}_3(f, f(x))$. We replace LHXOR, LHAND and GSL with LSimXOR, LSimAND and SimGSL. For the differences between $\mathcal{G}_4(f, f(x))$ and $\mathcal{G}_3(f, f(x))$, we can use a hybrid argument easily and can obtain:

$$\begin{aligned} &\left|\Pr[\mathcal{A}^E(\mathcal{G}_4(f, f(x))) = 1] - \Pr[\mathcal{A}^E(\mathcal{G}_3(f, x)) = 1]\right| \\ &\le g_1 \mathbf{Adv}_{\mathsf{AND}}(q_E, t) + g_2 \mathbf{Adv}_{\mathsf{XOR}}(q_E, t) \\ &+ \sum^{i \in \{\ell_{in}+1, \ldots, \ell_{in}+g\}} (\ell_i - 1) \cdot \mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*, t^*) \end{aligned} \tag{13}$$

where $q^* = (12g_1 + 6g_2 - 4\ell_{in} + 6\ell_{out} + q_E)$, $t^* = O(t + (12g_1 + 6g_2 - 4\ell_{in} + 6\ell_{out} + q_E)t_l)$, g_1, g_2, g_3 is the number of AND gate, XOR gate and XOR gate respectively, ℓ_{in} is the number of input wires, ℓ_{out} is the number of output wires and t_l is the total time needed for evaluating L^{in}, L^{out}.

5. $\mathcal{S}(f, f(x)) \approx \mathcal{G}_4(f, x)$. We replace inactive output labels $d[i, \overline{f(x)_i}] \xleftarrow{\$} \{0,1\}^{n+1}$.We have the following equation:

$$\begin{aligned} &\left|\Pr[\mathcal{A}^E(\mathcal{S}(f, x)) \Rightarrow 1] - \Pr[\mathcal{A}^E(\mathcal{G}_4(f, f(x))) \Rightarrow 1]\right| \\ &\le \ell_{out} \cdot \mathbf{Adv}^{\mathsf{2\text{-}up}[q^*]}(q^*, t^*, \ell_i) \end{aligned} \tag{14}$$

where $q^* = (12g_1 + 6g_2 - 4\ell_{in} + 6\ell_{out} + q_E)$, $t^* = O(t + (12g_1 + 6g_2 - 4\ell_{in} + 6\ell_{out} + q_E)t_l)$, g_1, g_2, g_3 is the number of AND gate, XOR gate and XOR gate respectively, ℓ_{in} is the number of input wires, ℓ_{out} is the number of output wires and t_l is the total time needed for evaluating L^{in}, L^{out}.

Then, using a union bound, we can get the Theorem 1. Note that $(\sum^{i \in \{1, \ldots, \ell_{in}+g\}} \ell_i = 2g_1 + 2g_2 + g_3 + \ell_{out})$.

D.5 Proofs of Theorem 2

Proof of leakage-resilient obliviousness We construct a simulator that outputs (F, X, leak) given only circuit f as an input to satisfy the obliviousness requirement. Note that the simulator $\mathcal{S}$ completed above for the privacy requirement outputs the triple (F, X, d, leak). However, $\mathcal{S}$ uses circuit f only for generating (F, X, leak), particularly the output $f(x)$, used only for generating d. Thus, we can remove the generation of the decoding information from $\mathcal{S}$'s instruction and obtain a simulator that produces only (F, X, leak) as required. Proving that this simulator's output is indistinguishable from (F, X, leak) generated by the real scheme is the same as in the proof of privacy.

Proof of leakage-resilient authenticity Regarding authenticity, we need to show that an adversary $\mathcal{A}^E$ that is given (F, X) as input can output $\tilde{Y}$ such that Decode $(\tilde{Y}, d \neq \{f(x), \perp\})$ with at most probability $\frac{m}{2^{n+1}} + \mathbf{Adv}_{\mathsf{prvL}}(\mathcal{A})$.

At first, we claim that if we give $\mathcal{A}^E$ the pair (F, X) generated by our simulator, it can succeed only with probability at most $\frac{m}{2^{n+1}}$. This is because in the simulator, for each output wire corresponding to the jth output bit, $d[j, \overline{f(x)}_j]$

is a random string, and it can succeed only with probability at most $1/2^{n+1}$. Then, using a union bound can get the claim.

Now, if given the real (F, X, leak), the adversary can output such a $\tilde{Y}$ with probability more than $\frac{m}{2^{n+1}} + \mathbf{Adv}^{\mathrm{GLNPLR},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}}(q_E, t)$. It could be used by an adversary given (F, X, d) to break the privacy property with probability more than $\mathbf{Adv}^{\mathrm{GLNPLR},\mathcal{S},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}}(q_E, t)$. Observe that since the adversary in the privacy experiment is given all of the decoding information d, it can efficiently verify if $\mathcal{A}^E$ output a $\tilde{Y}$ with the property that $\mathsf{Decode}(\tilde{Y}, d) \notin \{f(x), \perp\}$.

E Definition of OT Protocols

Formally, an n-OT protocol $\mathcal{OT} = (\Pi_1, \Pi_2)$ is an SFE scheme for function $f^{ot} = (f_1^{ot}, f_2^{ot})$, where $f_1^{ot}(x, y) = \perp$ and $f_2^{ot}(x, y) = (X_1^{y_1}, \cdots, X_n^{y_n})$. In this context, x comprises the vectors $X_1^0, X_1^1, \cdots, X_n^0, X_n^1$, and y is an n-bit string with y_i as its ith bit.

F Proof of Theorem 4

Using the notations in Appendix E, we first provide a more formal presentation of Theorem 3.

Theorem 4. *Assume $\mathcal{OT}$ is an* sfe *security n-OT protocol and the garbling scheme $\mathcal{G} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Decode})$ is* PrvSimL *secure w.r.t the leakage* $\mathsf{L}_{\mathsf{Garble}}$*. Then the above SFE scheme YaoSFE is* sfeLR*-secure w.r.t. the leakage* $(\perp, \mathsf{L}_{\mathsf{Garble}})$.

Proof. Let $i \in \{1, 2\}$ and let $\mathcal{B}$ be a *PT* adversary attacking $\mathcal{F}$. We build a PT adversary $\mathcal{B}_{\mathcal{G}}$ attack $\mathcal{G}$ and a PT adversary $\mathcal{B}_{\mathcal{OT}}$ attack $\mathcal{OT}$. By assumptions, these have simulators, respectively $\mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{OT}}$. We then use these simulators to build a simulator $\mathcal{S}$ for $\mathcal{B}$ such that for $i = 1$, we have

$$\mathbf{Adv}^{\mathcal{F},\mathcal{S},\mathsf{L}_{\mathsf{Garble}},1}_{\mathsf{sfeLR}}(\mathcal{B}) \leq \mathbf{Adv}^{\mathcal{G},\mathcal{S}_{\mathcal{G}},\mathsf{L}_{\mathsf{Garble}}}_{\mathsf{PrvSimL}}(\mathcal{B}_{\mathcal{G}}) + \mathbf{Adv}^{\mathcal{OT},\mathcal{S}_{\mathcal{OT}},1}_{\mathsf{sfe}}(\mathcal{B}_{\mathcal{OT}})$$

Case 1: $i = 1$. Adversary $\mathcal{B}_{\mathcal{G}}$ runs $\mathcal{B}$ to get its $\mathsf{GetView}$ query f, x, y. It will compute and return a reply *view* as well as the Garble leakage $\mathsf{L}_{\mathsf{Garble}}$ to this query as follows. Adversary $\mathcal{B}_{\mathcal{G}}$ queries its $\mathsf{LGarble}$ oracle with $f, x\|y$ to get back $(F, (X_1, \ldots, X_{n+m}), d, \mathsf{leak}_{\mathsf{Gb}})$. It records (F, d) as well as $X_{n+1}, \ldots, X_{n+m}$ as the first message in *conv*. (This message is from party 2 to party 1.) Now, for $i = 1, \ldots, n$, it lets $X_i^{x_i} := X_{n+i}$ and $X_i^{1-x_i} \overset{\$}{\leftarrow} \{0,1\}^{|X_{n+i}|}$.

It then lets $view^{ot} := View^1_{\Pi^{ot}}(f^{ot}, x,\ (X_1^0, X_1^1, \ldots, X_n^0, X_n^1))$. It obtains this by direct execution of the 2-party protocol Π^{ot} on inputs x for party 1 and $(X_1^0, X_1^1, \ldots, X_n^0, X_n^1)$ for party 2, it appends $conv^{ot}$ to *conv*. Finally, compute the output $y = \mathsf{De}(\mathsf{Eval}(F, X))$ and append y to *conv* and return $(conv, w_1, \mathsf{leak}_{\mathsf{Gb}})$ to $\mathcal{B}$'s query. Adversary $\mathcal{B}$ now output a bit b', and $\mathcal{B}$ adopt this as its own output as well.

Adversary $\mathcal{B}_{\mathcal{OT}}$ runs $\mathcal{B}$ to get its $\mathsf{GetView}$ query f, x, y. It will compute and return a reply $view$ to this query as follows. Adversary $\mathcal{B}_{\mathcal{OT}}$ lets $(F, e, d, \mathsf{leak}_{\mathsf{Gb}}) :=$ $\mathsf{LGarble}(f)$ and parses $(X_1^0, X_1^1, \ \ldots, X_{n+m}^0, X_{n+m}^1) := e$. It records (F, d) as well as $X_{n+1}^{y_1} \ldots, X_{n+m}^{y_m}$ as the first message in $conv$. It makes query $view^{ot} :=$ $\mathsf{GetView}(f^{ot}, x, \ (X_1^0, X_1^1, \ldots, X_n^0, X_n^1))$. Parsing $view^{ot}$ as $(conv^{ot}, w_1^{ot})$, it appends $conv^{ot}$ to $conv$. Finally, compute the output $y = \mathsf{De}(\mathsf{Eval}(F, X))$, append y to $conv$ and return $(conv, w_1, \mathsf{leak}_{\mathsf{Gb}})$ to $\mathcal{B}$'s query. Adversary $\mathcal{B}$ now output a bit b', and $\mathcal{B}$ adopt this as its own output as well.

By assumption, the two adversaries we have just built have simulators, respectively $\mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{OT}}$. We define simulator $\mathcal{S}$ for $\mathcal{B}$. On input f, x, it lets $(F, (X_1, \ldots,$ $X_{m+n}), d, \mathsf{leak}_{\mathsf{Gb}}) := \mathcal{S}_{\mathcal{G}}(f(x, y), f)$ and records $(F, d), X_{n+1}, \ldots, X_{n+m}$ as the first message in $conv$. It lets $view^{ot} := \mathcal{S}_{\mathcal{OT}}(f^{ot}, x, (X_1, \ldots, X_n))$. Parsing $view^{ot}$ as $(conv^{ot}, \ w_1^{ot})$, it appends $conv^{ot}$ to $conv$ and then return $view = (conv, w_1^{ot},$ $\mathsf{leak}_{\mathsf{Gb}})$

We then use these simulators to build a simulator $\mathcal{S}$ for $\mathcal{B}$ such that for $i = 2$, we have

$$\mathbf{Adv}_{\mathsf{sfeLR}}^{\mathcal{F}, \mathcal{S}, \mathsf{LGarble}, 2}(\mathcal{B}) \leq \mathbf{Adv}_{\mathsf{sfe}}^{\mathcal{OT}, \mathcal{S}_{\mathcal{OT}}, 2}(\mathcal{B}_{\mathcal{OT}})$$

Case 2: $i = 2$. Adversary $\mathcal{B}_{\mathcal{OT}}(1^k)$ runs $\mathcal{B}(1^k)$ to get its $\mathsf{GetView}$ query f, x, y. It will compute and return a reply $view$ to this query as follows. Adversary $\mathcal{B}_{OT}$ lets $(F, e, d) := \mathsf{Garble}(f)$ and parses $(X_1^0, X_1^1, \ldots, \ X_{n+m}^0, X_{n+m}^1) := e$. It makes query $view^{ot} := \mathsf{GetView}\ (f^{ot}, x, \ (X_1^0, X_1^1, \ \ldots, X_n^0, X_n^1))$. Parsing $view^{ot}$ as $(conv^{ot}, w_2^{ot})$, it appends $conv^{ot}$ to $conv$. Finally, compute the output $(Y) =$ $\mathsf{Eval}(F, X)$. Compute $y = \mathsf{De}(Y)$ and append y to $conv$ and return $(conv, w_2)$ to $\mathcal{B}$'s query. Adversary $\mathcal{B}$ now output a bit b', and $\mathcal{B}$ adopt this as its own output as well.

By assumption, the adversary we have just built has simulator $\mathcal{S}_{\mathcal{OT}}$. We define simulator $\mathcal{S}$ for $\mathcal{B}$. On input f, x, y, it lets $(F, e, d) := \mathsf{Gb}(f)$ and parses $(X_1^0, X_1^1, \ \ldots, X_{n+m}^0, X_{n+m}^1) := e$. It lets $view^{ot} := \mathcal{S}_{\mathcal{OT}}(x, (X_1^{x_1}, \ldots, X_n^{x_n}),$ $(|X_1^0|, |X_1^1|, \ \ldots, \ |X_n^0|, |X_n^1|))$. Parsing $view^{ot}$ as $(conv^{ot}, w_1^{ot})$, it appends $conv^{ot}$ to $conv$. Then compute $Y := \mathsf{Eval}(F, X)$, let $y := \mathsf{Decode}(d, Y)$ and add y to $conv$. Then return $view = (conv, w_1^{ot})$. ☐

References

1. A. Barak, M. Hirt, L. Koskas, and Y. Lindell. An end-to-end system for large scale P2P MPC-as-a-service and low-bandwidth MPC for weak participants. In *ACM Conf. on Computer and Communications Security (CCS) 2018*, pages 695–712. ACM Press, 2018.
2. G. Barwell, D. P. Martin, E. Oswald, and M. Stam. Authenticated encryption in the face of protocol and side channel leakage. In *Advances in Cryptology—Asiacrypt 2017, Part I*, LNCS, pages 693–723. Springer, 2017.
3. M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy (S&P) 2013*, pages 478–492, 2013.

4. M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *ACM Conf. on Computer and Communications Security (CCS) 2012*, pages 784–796. ACM Press, 2012.
5. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 409–426. Springer, 2006.
6. D. Bellizia, F. Berti, O. Bronchain, G. Cassiers, S. Duval, C. Guo, G. Leander, G. Leurent, I. Levi, C. Momin, et al. Spook: Sponge-based leakage-resistant authenticated encryption with a masked tweakable block cipher. 2020. Submission to NIST LWC.
7. D. Bellizia, O. Bronchain, G. Cassiers, V. Grosso, C. Guo, C. Momin, O. Pereira, T. Peters, and F.-X. Standaert. Mode-level vs. implementation-level physical security in symmetric cryptography - A practical guide through the leakage-resistance jungle. LNCS, pages 369–400. Springer, 2020.
8. F. Berti, C. Guo, O. Pereira, T. Peters, and F.-X. Standaert. TEDT: a leakage-resistant AEAD mode. 2020(1):256–320, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8400.
9. P. Chapman, D. Evans, Y. Huang, and S. Koo. Secure Computation on Smartphones. http://mightbeevil.org/mobile/, 2017.
10. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology—Crypto 1999*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
11. J.-S. Coron, A. Greuet, E. Prouff, and R. Zeitoun. Faster evaluation of sboxes via common shares. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 498–514. Springer, 2016.
12. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In *Fast Software Encryption (FSE)*, LNCS, pages 410–424. Springer, 2014.
13. C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer. Isap v2.0. 2020. Submission to NIST LWC.
14. S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *49th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 293–302. IEEE, 2008.
15. D. Goudarzi and M. Rivain. How fast can higher-order masking be in software? In *Advances in Cryptology—Eurocrypt 2017, Part I*, LNCS, pages 567–597. Springer, 2017.
16. H. Groß, S. Mangard, and T. Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In *Cryptographers' Track—RSA*, LNCS, pages 95–112. Springer, 2017.
17. S. Gueron, Y. Lindell, A. Nof, and B. Pinkas. Fast garbling of circuits under standard assumptions. *J. Cryptology*, 31(3):798–844, July 2018.
18. C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu. Better concrete security for half-gates garbling (in the multi-instance setting). LNCS, pages 793–822. Springer, 2020.
19. C. Guo, X. Wang, K. Yang, and Y. Yu. On tweakable correlation robust hashing against key leakages. Cryptology ePrint Archive, Paper 2024/163, 2024.
20. X. Guo, K. Yang, X. Wang, W. Zhang, X. Xie, J. Zhang, and Z. Liu. Half-tree: Halving the cost of tree expansion in COT and DPF. LNCS, pages 330–362. Springer, 2023.

21. M. Hashemi, D. Forte, and F. Ganji. Time is money, friend! timing side-channel attack against garbled circuit constructions. In C. Pöpper and L. Batina, editors, *Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part III*, volume 14585 of *Lecture Notes in Computer Science*, pages 325–354. Springer, 2024.
22. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium 2011*. USENIX Association, 2011.
23. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs - (full version). In *Cryptographic Hardware and Embedded Systems – CHES 2010*, LNCS, pages 383–397. Springer, 2010.
24. D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert. Mobile private contact discovery at scale. In *USENIX Security Symposium 2019*, pages 1447–1464. USENIX Association, 2019.
25. J. Katz and V. Vaikuntanathan. Signature schemes with bounded leakage resilience. In *Advances in Cryptology—Asiacrypt 2009*, volume 5912 of *LNCS*, pages 703–720. Springer, 2009.
26. E. Kiltz and K. Pietrzak. Leakage resilient ElGamal encryption. In *Advances in Cryptology—Asiacrypt 2010*, LNCS, pages 595–612. Springer, 2010.
27. knarfrank. Higher-Order-Masked-AES-128. https://github.com/knarfrank/Higher-Order-Masked-AES-128, 2016.
28. V. Kolesnikov, P. Mohassel, and M. Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, 2014.
29. V. Kolesnikov and C. Rackoff. Password mistyping in two-factor-authenticated key exchange. In *Intl. Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5126 of *LNCS*, pages 702–714. Springer, 2008.
30. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *Intl. Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.
31. I. Levi and C. Hazay. Garbled circuits from an SCA perspective free XOR can be quite expensive. . 2023(2):54–79, 2023.
32. Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, Apr. 2009.
33. M. Lipp, A. Kogler, D. F. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. PLATYPUS: Software-based power side-channel attacks on x86. pages 355–371, 2021.
34. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium 2004*, pages 287–302. USENIX Association, 2004.
35. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.
36. D. P. Martin, E. Oswald, M. Stam, and M. Wójcik. A leakage resilient MAC. LNCS, pages 295–310, 2015.
37. M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 129–139, 1999.

38. O. Pereira, F.-X. Standaert, and S. Vivek. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *ACM Conf. on Computer and Communications Security (CCS) 2015*, pages 96–108. ACM Press, 2015.
39. K. Pietrzak. A leakage-resilient mode of operation. In *Advances in Cryptology—Eurocrypt 2009*, LNCS, pages 462–482. Springer, 2009.
40. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology—Asiacrypt 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
41. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems – CHES 2010*, LNCS, pages 413–427. Springer, 2010.
42. M. Rosulek and L. Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. LNCS, pages 94–124. Springer, 2021.
43. O. P. Team. OpenSSL Coding Style. https://www.openssl.org/policies/codingstyle.html, 2015.
44. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.
45. Y. Yu, F.-X. Standaert, O. Pereira, and M. Yung. Practical leakage-resilient pseudorandom generators. In *ACM Conf. on Computer and Communications Security (CCS) 2010*, pages 141–151. ACM Press, 2010.
46. S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology—Eurocrypt 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, 2015.
47. R. Zhang, S. Qiu, and Y. Zhou. Further improving efficiency of higher order masking schemes by decreasing randomness complexity. *IEEE Transactions on Information Forensics and Security*, 12(11):2590–2598, 2017.