Side-Channel Protections for Picnic Signatures

Diego F. Aranha, Sebastian Berndt, Thomas Eisenbarth, Okan Seker, Akira Takahashi, Luca Wilke and Greg Zaverucha

1 Department of Computer Science and DIGIT, Aarhus University, Denmark
{dfaranha,takahashi}@cs.au.dk
2 University of Lübeck, Germany
{s.berndt,thomas.eisenbarth,okan.seker,l.wilke}@uni-luebeck.de
3 Microsoft Research, USA
gregz@microsoft.com

Abstract. We study masking countermeasures for side-channel attacks against signature schemes constructed from the MPC-in-the-head paradigm, specifically when the MPC protocol uses preprocessing. This class of signature schemes includes Picnic, an alternate candidate in the third round of the NIST post-quantum standardization project. The only previously known approach to masking MPC-in-the-head signatures suffers from interoperability issues and increased signature sizes. Further, we present a new attack to demonstrate that known countermeasures are not sufficient when the MPC protocol uses a preprocessing phase, as in Picnic3.
We overcome these challenges by showing how to mask the underlying zero-knowledge proof system due to Katz–Kolesnikov–Wang (CCS 2018) for any masking order, and by formally proving that our approach meets the standard security notions of non-interference for masking countermeasures. As a case study, we apply our masking technique to Picnic. We then implement different masked versions of Picnic signing providing first order protection for the ARM Cortex M4 platform, and quantify the overhead of these different masking approaches. We carefully analyze the side-channel risk of hashing operations, and give optimizations that reduce the CPU cost of protecting hashing in Picnic by a factor of five. The performance penalties of the masking countermeasures ranged from 1.8 to 5.5, depending on the degree of masking applied to hash function invocations.

Keywords: side-channel attacks · masking · MPC-in-the-head · Picnic signatures

1 Introduction
As the possible advent of a quantum computer threatens the security of widely deployed cryptographic schemes, the design of new quantum-resilient alternatives is a pressing task. Motivated by this issue, the US National Institute for Standards and Technology (NIST) is currently holding the Post-Quantum Cryptography (PQC) Standardization Process, in which Round 3 “finalists” and “alternate candidates” have been recently announced. Among them is Picnic, a signature scheme [ZCD+20], which follows Ishai et al.’s MPC-in-the-head (or MPCitH, short for multi-party computation in-the-head) paradigm for constructing zero-knowledge (ZK) proof systems [IKOS07]. One of the attractive features of MPCitH-style signatures is that they can be constructed without number-theoretic hardness assumptions, and rely only on the security of symmetric-key primitives. Concretely, following the standard Fiat–Shamir paradigm [FS87], signatures in the MPCitH paradigm can be proven secure in the random oracle model, as long as the underlying hash function and block cipher are secure. Quoting [Nat20], “NIST also sees
Picnic’s reliance on only assumptions about symmetric primitives as an advantage in case the need arises for an extremely conservative signature standard in the future.

MPC-in-the-head vs probing side-channel analysis. Yet, despite such advantages with regard to the underlying assumptions, implementations of MPCitH-type signatures are not immune to side-channel attacks that threaten unprotected software. Recently, Gellersen et al. [GSE20] and Seker et al. [SBWE20] (which, to the best of our knowledge, are the only previous works that considered side-channel security of MPCitH) observed that a direct implementation of MPCitH signing is susceptible to a probing side-channel attack, which is in practice realized by, e.g., differential power analysis (DPA). Their proposed attack is devastating – it allows a side-channel attacker to successfully recover the secret signing key, after observing 30 signatures. To understand their attack, it is useful to review the underlying “commit-and-open” approach typically used by MPCitH proofs. The attacks target the three-round ZK proof system underlying some Picnic instances, known as ZKB+ [CDG\textsuperscript{+}17] (an optimized variant of ZkBoo [GMO16]).

MPCitH proof systems like ZKB++ follow a typical commit, challenge, response structure (which, in fact, can be seen as a $\Sigma$-protocol [Dam10]), and prove knowledge of a witness $w$ that satisfies a statement for a hard relation. For example, we can prove knowledge of a key (the witness) that relates a plaintext-ciphertext pair (the statement) for a block cipher (the relation). The ZKB++ proof system is built on a three-party MPC protocol $\Pi_f$ for a function $f$ implementing this hard relation (the block cipher in our example). To initialize the protocol, the prover $P$ first additively secret-shares the witness such that $w = w_1 + w_2 + w_3$, and considers each share $w_i$ as a private input to party $P_i$. Then $P$ runs $\Pi_f$ “in the head”, i.e., it simulates a run of the protocol between the three parties, and produces for each $P_i$ a commitment to its view, i.e., the input $w_i$, output, communication, and random tape. These three commitments are sent to the verifier $V$, who replies with a challenge, for $P$ to open two of them. Finally, $V$ checks the openings for consistency and for an accepting output. The (honest verifier) ZK property of the protocol is guaranteed by the 2-privacy of $\Pi_f$, i.e., it is possible to simulate up to 2 parties’ views given their inputs, and the output of the protocol execution.

But once a side-channel adversary probes the unopened party’s view, in particular their share of the witness, the adversary can recover the secret witness. To thwart this devastating probing attack, Seker et al. [SBWE20] proposed the SNI-in-the-head (SNIitH) approach, which naturally retains the privacy of the MPC protocol, but also adds strong non-interference (SNI) [BBD\textsuperscript{+}16] security, which is a strong provable security notion for ISW-style masking countermeasures (as defined by Ishai-Sahai-Wagner [ISW03]). In the SNIitH approach, the number of parties is generalized to $N$ and the underlying MPC protocol is assumed to have $(N - 1)$-privacy, so that the views of up to $N - 1$ parties may be safely revealed. However, instead of opening $N - 1$ parties as a typical MPCitH prover would, an SNIitH prover only reveals $N - t - 1$ parties as a response, where the parameter $t$ serves as the “buffer” for probing security. This way, the prover makes sure that at least one party’s internal state remains completely hidden, even if the side-channel adversary observes up to $t$ variables during the execution of MPC protocol $\Pi_f$. Relying on this idea, Seker et al. generalized ZKB++, and gave a variant of Picnic that is provably secure against probing adversaries. However, as we shall see below, there are still several practical challenges which seem hard to overcome with SNIitH. In this paper, we push forward the study of side-channel resistant MPCitH signing, to tackle these open questions.

Maintaining the verification algorithm and signature size. When applied to Picnic, the SNIitH approach changes the number parties and the number of opened views, and the result is essentially a different parameter set. Accordingly, the verification algorithm has to be modified, and now depends on the masking order. For example, the signer would have to simulate a five-party MPC to achieve first order protection (i.e., $N = 5$, $t = 1$). Hence, the verifier needs to check the consistency of 3-out-of-5 views, instead of 2-out-of-3.
as in the original Picnic/ZKB++ scheme. This is a major drawback in practice, since the introduction of side-channel protection would immediately break interoperability with existing verification algorithms. In order to support signers with varying levels of side-channel protections, verifiers would have to be prepared to accept multiple combinations of parameters \((N, t)\). A more flexible design would allow different signers to set their level of side-channel protection independent of the verification software and costs. Moreover, the SNIitH approach inevitably sacrifices the soundness of underlying ZK proof system, since a malicious prover has a higher cheating probability when the number of views the verifier checks is smaller than \(N − 1\). Therefore to maintain the same security level as in the original proof system, the SNI-in-the-head approach must simulate more MPC instances, leading to larger signature sizes, and slower signing and verification times. In this work we design countermeasures without these drawbacks (as has been done for some other PQ signature schemes [BBE+18, MGTF19, BBE+19, GR19]).

**Masking MPCitH with preprocessing.** To the best of our knowledge, no previous work explored the possibility of masking the newer MPC-in-the-head with preprocessing (MPCitH-PP) paradigm of Katz, Kolesnikov, and Wang (KKW, [KKW18]), which produces much more compact proofs (and shorter signatures in turn). KKW is used in the latest version of Picnic, Picnic3 [KZ20], and in a similar signature scheme BBQ [dDOS19]. The preprocessing phase is independent of the witness, and is used by the parties to establish correlated random values, such as multiplication triples, that they can use during the MPC protocol to improve efficiency. Since it happens before the main, witness-dependent MPC protocol, the preprocessing phase is also called an offline phase, and the main part is called the online phase. We first observe that the preprocessing phase provides additional attack surface not present in MPCitH protocols without preprocessing. In the KKW protocol, the prover first simulates many instances of the MPC protocol consisting of offline and online phases, and commits to each party’s view in both phases. Then for each instance of the MPC protocol, the prover opens either the offline phase or the online phase, depending on the challenge sent from the verifier. To open the offline phase, the prover reveals the views of all \(N\) parties (which is secure, since preprocessing views are independent of the witness). For the online phase, the prover reveals the views of \(N − 1\) parties (which is secure by the \(N − 1\) privacy of the MPC protocol). In the former scenario, since the offline phase contains the correlated random values used by all parties during the online phase, a probing adversary can break the privacy of the underlying MPC protocol by probing values from the corresponding online phase (which is computed by the prover but not revealed).

We note that the attack succeeds for any number of opened views, since MPCitH-PP inherently relies on revealing all views for the offline phase. This indicates that the SNIitH approach cannot mitigate this type of probing attack. Given all this, we are motivated to design an alternative countermeasure addressing the following question:

*Can we mask signature generation in signature schemes constructed with the MPC-in-the-head-with-preprocessing paradigm in a provably secure manner, without modifying the verification algorithm?*

### 1.1 Contributions

**Side-channel attacks against Picnic3 and KKW.** We confirm experimentally that our new probing attack described above indeed applies to Picnic3, by measuring the leakage of the (unmasked) reference code on the Cortex M4. In Section 3 we elaborate on the high-level attack strategy for MPCitH-PP and describe how to mount a practical signing key recovery attack against Picnic3, as well as any three-round KKW-type protocol. Fig. 1 experimentally shows this attack is practical.

**Masking MPC-in-the-head with preprocessing.** In Section 4 we show how to mask the KKW proof system for any masking order. Our first observation is that all building
blocks in both the offline and online computations can be masked with existing SNI-secure gadgets without changing the number of opened parties. The main technical difficulty is that the prover opens a different subset of sensitive information, depending on the challenge. Nevertheless, we are able to present a solution achieving strong non-interference of the basic building blocks of the KKW proof system. Then together with a known generic composition result [BBD+16], we prove that our masked KKW prover algorithm for the commit and response phases satisfies non-interference with public output (NIo) security [BBE+18] for all possible public outputs. The CPU cost of masking follows the common pattern where nonlinear operations have cost $O(T^2)$, and all other operations are linear in $T$, where $T$ denotes the masking order. In addition to giving formal proofs of the (strong) non-interference of our solution, we also use the maskVerif tool [BBD+15a,BBD+16] to verify our conclusions. We stress that our variant outputs a proof identical to that of the unprotected KKW, so it is interoperable with existing verification operations.

Masked implementation of Picnic3. Following our generic masking technique from Section 4, we present an NIo-secure masked implementation of Picnic3 for NIST security level L1 with first-order protection, and we report concrete experimental results on the low-end ARM Cortex-M4 STM32F4 platform (see Sections 5 and 6). The complete specification of NIo-secure Picnic3 is presented in Appendix C.1. The figures were collected from a port of the optimized version of the Picnic3 reference code, with the help of the pqm4 [KRSS] framework. However, the overhead incurred by provably secure masking is expensive – signing time is increased by 5.5x. Using this provably secure implementation as a baseline, we performed a number of optimizations described in Section 5, especially to improve hashing performance. Since a large part of the signing time in Picnic3 is spent hashing (e.g., 70% on the M4), we carefully analyzed which hashing operations must be masked, and found that the majority may remain unmasked. We then show that since all calls have either a non-sensitive input or output, a weaker form of masking is sufficient (under a mild assumption), further reducing the cost. Taken together, these optimizations reduce the cost of protected hashing significantly, and we observed overhead ranging from 1.8x to 2.8x (depending on the type of protections applied to hash function invocations). Since our hash function optimizations do void the provable side-channel-security of the signature scheme implementation as a whole, we experimentally verified the absence of leakage, to support our arguments that security is maintained in practice.

Masked implementation of SHA-3. We also implemented a masked version of SHA-3, optimized with M4 assembly, as none was freely available. As SHA-3 is common to many PQ schemes and the M4 is a common embedded target, we expect this will be of independent interest. The implementation supports a range of options, from slower but provably SNI-secure, to our much faster optimized options. We experimentally verified that there was an absence of leakage in the optimized version.

Our first-order masked Picnic3 and SHA-3 implementations, as well as formal verification scripts, are publicly available under crypto_sign/picnic3l1/masked/ of the Picnic-M4 GitHub repository.

1.2 Related work

Since the seminal work by Ishai et al. [IKOS07] the MPCitH paradigm has been actively studied over the past decade. In particular, two closely related protocols ZKBoo [GMO16] and ZKB+++ [CDG+17] brought MPCitH closer to practice, leading to the submission of Picnic1 to Round 1 of the NIST PQC Standardization Process. Katz, Kolesnikov, and Wang [KKW18] extended the paradigm to MPCitH-PP and corresponding version, Picnic2, was added during Round 2. Kales and Zaaverucha [KZ20] further optimized Picnic2 from various implementation aspects and accordingly proposed Picnic3. Although our

---

masked implementation focuses on Picnic3, which is instantiated with KKW and the LowMC circuit [ABS +15], our generic approach in Section 4 also applies to BBQ (KKW instantiated with the AES circuit) [dDOS19] and Baum and Nof’s variant of KKW (instantiated over an arithmetic circuit for proving SIS instances) [BN20]. A similar offline-online paradigm also appears as a notion called “sigma protocol with helper” [Beu20], used to construct proof of knowledge for systems of quadratic equations, improving a protocol of Stern [Ste94].

The stateful hash-based signature schemes XMSS and LMS are known to be relatively resistant to side-channel attacks, as they basically use pseudorandom keys for each signature. Hence, their resistance against side-channel attacks rests on the resistance of the underlying pseudorandom number generator [EvMY14, KGB +18].

Comparison with other NIST PQC candidates. To the best of our knowledge, most other PQ signatures currently do not have publicly available masked implementations, except for lattice-based Fiat–Shamir with aborts [Lyn09] signatures: GLP [BBE +18], BLISS [BBE +19], Dilithium [MGTF19] (NIST PQC finalist) and qTESLA [GR19] (Round 2 candidate). While these masked signing operations do output a signature compliant with the existing verification algorithm, they rely on an additional non-standard hardness assumption for provable side-channel-security (see [BBE +18, DOTT21] for details). The issue could be circumvented by modifying the “commit” message of the underlying Σ-protocol, but this in turn breaks the interoperability of the output signature. By contrast, signatures directly derived from our generic approach to masking KKW (Section 4) as well as NLo-secure Picnic3 implementation maintain interoperability, and may optionally make additional assumptions for improved performance. Performance-wise, the benchmarks on Cortex M4 given by Gérard and Rossi [GR19, Table 6] show much less overhead than ours: their first-order protected qTESLA-I incurs only 2.1x overhead in signing clock cycles and requires 343 KB of fresh randomness, while provably secure masked Picnic3 is 5.5x slower than unprotected and consumed 158 MB of randomness. However, by trading provable security guarantee as we describe in Section 5, our empirically validated countermeasures achieve a lower overhead overall of 1.8x, requiring 2 MB of randomness. Giving a meaningful performance comparison with masked Dilithium [MGTF19] is hard, as they only provide benchmarks on Intel for the whole signing operation. Although their overhead for first-order protection is about 5.6x, we expect that it can be made faster on the M4 by using the platform’s TRNG.

2 Preliminaries

Notation. We denote the set \{1, \ldots, T\} by [T]. The number of multiplication gates in a circuit \(C\) is denoted by \(|C|\).

Security levels. The parameter sets for the algorithms submitted to the NIST competition must meet one of five security levels. Picnic defines parameters for security levels L1, L3 and L5, corresponding to the security of AES 128, 192 and 256, respectively. For instance, parameters at level L1 aim to provide 128-bit security against classical attacks.

LowMC. The LowMC block cipher is described in detail in Appendix F, and here we briefly review the notation. The block and key size are both \(n\) bits, the number of rounds is denoted \(r\), and the number of S-boxes per round is denoted \(s\). In the Picnic3 parameters, \(n = 3s\), since the three-bit S-box is applied to the full state. There are also constants: \(K_i\) are matrices used to compute the round keys; \(L_i\) are matrices used for the linear layer, and \(R_i\) are vectors used as round constants.

Formal Verification. In order to check our formal security analysis, we use the tool maskVerif developed by Barthe et al. [BBC +19]. The tool provides an automatic and formal security verification of higher-order masking implementations based on the NI and SNI notions. Briefly speaking, it checks every possible attack combination within
the implementation and either provides a security proof for the specified order or gives a list of potential attack targets in the implementation. The bottleneck of the tool is the order of the masking and the complexity of the implementation. It has been used in the literature to provide assurance of masked implementations [BBD+15a, GSDM+19, SEL21]. In this work, we use maskVerif to check SNI security of the basic components (namely, multi-party computation of multiplication in both online and offline phases) of the KKW proof system up to order 4.

Leakage analysis. In this work we follow the test vector leakage assessment (TVLA) method by Goodwill et al. [GJJR11], which is based on Welch’s $t$-test. TVLA implements a pass-fail test to decide if an implementation has exploitable leakage. It detects leakage at a given order and has two different versions: the non-specific and the specific method. The first version is defined as fixed-vs-random (FvR) and it aims to detect all possible sources of first-order leakage. During the trace collection phase, a set of side-channel traces is collected by processing either a fixed input or a random input under the same conditions. If the $t$-test only gives a very small value, this indicates that the run of the algorithm on a fixed input is indistinguishable from a run of the algorithm on a random input. Hence, a small value in the fixed-vs-random scenario implies the absence of sensitive leakage. The second test is defined as random-vs-random (RvR), and employs only traces with a random input using a function of inputs to sort the traces. The main advantage of the RvR method is that it can identify specific sources of exploitable leakage and thus shows the feasibility of an actual attack.

After collecting and sorting the traces, the means ($\mu_0$, $\mu_1$) and standard deviations ($\sigma_0$, $\sigma_1$) for the two sets are calculated. Welch’s $t$-test is computed as $t = (\mu_f - \mu_r)/\sqrt{(\sigma_f^2/n_f) + (\sigma_r^2/n_r)}$, where $n_f$ and $n_r$ denote the number of traces for the sets, respectively. The $t$-test indicates whether the two distributions are indistinguishable for a first-order side-channel analysis. We apply the customary threshold values for long traces suggested by [DZD+18]. We use the value 5.7 for traces of length more than $10^4$, and the value 6.1 for traces of length more than $10^6$. The threshold rejects the null-hypothesis of non-leakage with $> 99.99\%$ probability.

2.1 MPC-in-the-head with preprocessing

MPC-in-the-head. We first describe the basic approach to construct a zero-knowledge proof of knowledge (ZKPoK) system for an arbitrary nondeterministic polynomial-time (NP) language $L$, following Ishai et al. [IKOS07] and its generalization due to Giacomelli et al. [GMO16]. Given $L$, we can define an NP relation $R(x, w)$ which returns 1 if its input consists of a valid pair of statement $x \in L$ and corresponding witness $w$, and outputs 0 otherwise. An MPCitH proof system $(P, V)$ is built upon some $N$-party MPC protocol that jointly computes a function $f$, where $f$ takes $x$ and $w$ as public and private input, respectively, and outputs $f_x(w) = R(x, w)$. For example, for a given encryption algorithm $Enc$ of a block cipher like LowMC [ARS+15], one can define $f_x(w) := Enc(sk, p) \parallel c$, where the statement $x = (p, c)$ is a plaintext-ciphertext pair, and witness $w = sk$ is a private encryption key, respectively. In this case, the prover $P$ proves knowledge of a private key that produces a certain public ciphertext from the corresponding public plaintext.

At a high level, an MPCitH prover $P$ attempts to convince the verifier $V$ that they hold a valid witness $w$, by letting $V$ check that the MPC protocol has been correctly carried out “in $P$’s head” on input $w$. We now consider an MPC protocol $\Pi_C$: for the corresponding arithmetic circuit $C$ defined over a finite field $\mathbb{F}$, where the statement information $x$ (e.g., the plaintext-ciphertext pair) is hard-coded such that $C(\cdot) = f_x(\cdot)$. We assume that the witness is expressed by an $n$-dimensional vector and $C$ takes a set of $n$ input wires denoted...
by \( \text{IN} \). We write \( w = (w)_{w \in \text{IN}} \in \mathbb{F}^n \) for the complete input.\(^2\) To initialize the protocol, the prover \( P \) first additively secret shares each input \( w \) such that \( w = w_1 + \ldots + w_N \) in \( \mathbb{F} \), and considers each share \( w_i \) as a private input to a party \( P_i \). Then \( P \) internally runs \( \Pi_C \) to obtain \( \text{view}_1, \ldots, \text{view}_N \), where \( \text{view}_i \) consists of \( P_i \)'s private input \( w_i \), the random tape of \( P_i \) and all incoming messages that \( P_i \) observes during the execution of \( \Pi_C \). The proof system now proceeds by following the typical “commit–challenge–response” flow. Using a secure commitment scheme, \( P \) sends \( \text{Commit} (\text{view}_i) \) for all \( i \in [N] \) as the first message. Upon receiving distinct challenges \( i_1, \ldots, i_t \in [N] \) from the verifier \( V \), the prover \( P \) sends back the corresponding \( t \) views \( \text{view}_{i_1}, \ldots, \text{view}_{i_t} \) as well as the commitment opening information as a response. Finally, the verifier \( V \) accepts the proof iff the opened views are consistent with each other and they produce 1 as output of the protocol \( \Pi_C \). The (honest verifier) zero knowledge is guaranteed as long as the underlying MPC views are consistent with each other and they produce \( w \) here for the sake of generality.

\( \Pi \) for the output wire \( z \) as a private input to a party \( P_1 \) as a private input to a party \( P_1 \). Then the parties compute random shares for all internal wires, by running the circuit:

- for each addition gate that takes wires \( x \) and \( y \) as input, party \( P_i \) locally computes a new share \( \lambda_i^z = \lambda_i^x + \lambda_i^y \) for the output wire \( z \);
- for each multiplication gate that takes wires \( x \) and \( y \) as input, party \( P_i \) obtains shares of the multiplication triples (sometimes called Beaver triples \([\text{Bea02}]\)) \( (\lambda_i^x, \lambda_i^y, \lambda_i^{xy}) \), such that \( \lambda_i^{xy} = \lambda_i^x \lambda_i^y \).

\( \Pi \) to generate multiplication triples in the MPCitH setting, the parties choose \( \lambda^x \) and \( \lambda^y \) implicitly by reading their shares from their random tapes. Then to obtain shares of \( \lambda^{xy} \), the first \( N - 1 \) parties read random shares from their random tapes. As the prover \( P \) knows all the shares, \( P \) can simply solve for the \( N \)-th party’s share so that the shares reconstruct \( \lambda^z \), as required. We call the sequence of values \( \lambda_i^{xy} \) for all multiplication gates auxiliary information, denoted \( \text{aux} \in \mathbb{F}^{|C|} \). Note that the complete information needed for the first \( N - 1 \) parties can be derived from their respective seeds \( \text{seed}_i \) used to generate \( P_i \)'s tape. The information needed for party \( P_N \) can be derived from \( \text{seed}_N \) and from \( \text{aux} \). Hence, we define each party \( P_i \)'s state information as follows: for all \( i = 1, \ldots, N - 1 \), let \( \text{state}_i \) := \( \text{seed}_i \), and for \( P_N \) we have \( \text{state}_N := \text{seed}_N \| \text{aux} \).

\( \Pi \) Given the preprocessed state information, the online phase \( \Pi^{\text{on}}_C \)

\( \text{proof system is focused on} \) \( \text{KKW} \), e.g., in SPDZ \([\text{DPSZ12}]\), MiniMAC \([\text{DZ13}]\), and TinyOT \([\text{NNOB12}]\). While the original communication in \( \Pi \) phase \( \Pi \) appears in subsequent work \([\text{BN20,Beu20,dDOS19,KZ20}]\). The core idea of MPC in the preprocessing model.

**MPC in the preprocessing model.** Following \([\text{GMO16,CDG+17}]\), Katz, Kolesnikov, and Wang \([\text{KKW18}]\) showed that a particular communication-efficient MPC protocol in the preprocessing model is well suited to MPCitH proofs, and variants of their protocol appear in subsequent work \([\text{BN20,Beu20,dDOS19,KZ20}]\). The core idea of MPC in the preprocessing model is to split the protocol \( \Pi_C \) into an offline phase \( \Pi^{\text{off}}_C \) and an online phase \( \Pi^{\text{on}}_C \). Importantly, the offline phase \( \Pi^{\text{off}}_C \) can be computed independently of the witness. By precomputing correlated randomness in advance during \( \Pi^{\text{off}}_C \), one can reduce communication in \( \Pi^{\text{on}}_C \) drastically. In the traditional MPC setting, this was already used, e.g., in SPDZ \([\text{DPSZ12}]\), MiniMAC \([\text{DZ13}]\), and TinyOT \([\text{NNOB12}]\). While the original KKW proof system is focused on boolean circuits, it also works with arithmetic circuits in a straightforward manner as observed in \([\text{dDOS19,BN20}]\), so we present the latter case here for the sake of generality.

**Offline Phase** The offline phase \( \Pi^{\text{off}}_C \) of KKW works as follows: for each input wire \( w \in \text{IN} \) to the circuit \( C \), and for each output wire \( z \) from all the multiplication gates, each party \( P_i \) locally generates random shares \( \lambda_i^x, \lambda_i^y, \lambda_i^{xy} \in \mathbb{F} \) using its own random tape. Then the parties compute random shares for all internal wires, by running the circuit:

- for each addition gate that takes wires \( x \) and \( y \) as input, party \( P_i \) locally computes a new share \( \lambda_i^z = \lambda_i^x + \lambda_i^y \) for the output wire \( z \);
- for each multiplication gate that takes wires \( x \) and \( y \) as input, party \( P_i \) obtains shares of the multiplication triples (sometimes called Beaver triples \([\text{Bea02}]\)) \( (\lambda_i^x, \lambda_i^y, \lambda_i^{xy}) \), such that \( \lambda_i^{xy} = \lambda_i^x \lambda_i^y \).

**Multiplication Triples** To generate multiplication triples in the MPCitH setting, the parties choose \( \lambda^x \) and \( \lambda^y \) implicitly by reading their shares from their random tapes. Then to obtain shares of \( \lambda^{xy} \), the first \( N - 1 \) parties read random shares from their random tapes. As the prover \( P \) knows all the shares, \( P \) can simply solve for the \( N \)-th party’s share so that the shares reconstruct \( \lambda^z \), as required. We call the sequence of values \( \lambda_i^{xy} \) for all multiplication gates auxiliary information, denoted \( \text{aux} \in \mathbb{F}^{|C|} \). Note that the complete information needed for the first \( N - 1 \) parties can be derived from their respective seeds \( \text{seed}_i \) used to generate \( P_i \)'s tape. The information needed for party \( P_N \) can be derived from \( \text{seed}_N \) and from \( \text{aux} \). Hence, we define each party \( P_i \)'s state information as follows: for all \( i = 1, \ldots, N - 1 \), let \( \text{state}_i := \text{seed}_i \), and for \( P_N \) we have \( \text{state}_N := \text{seed}_N \| \text{aux} \).

**Online Phase** Given the preprocessed state information, the online phase \( \Pi^{\text{on}}_C \)

\( \text{Note that we’re slightly abusing the notation here. Throughout, we use the same notations (typically } w, x, y \text{ and } z \text{ for both wires and wire values, but it should be clear from the context which they indicate.} \)
proceeds by computing the masked witness $\hat{w} = w + \sum_{i \in [N]} \lambda^w_i$ for each input wire. Now, each gate takes (masked) inputs $\hat{x} = x + \sum_{i \in [N]} \lambda^x_i$ and $\hat{y} = y + \sum_{i \in [N]} \lambda^y_i$ and can be computed as follows, where all computations on shares are carried out in $\mathbb{F}$:

- Addition: each $P_i$ locally computes $\hat{x} + \hat{y}$.
- Addition by constant $c$: each $P_i$ locally computes $\hat{x} + c$.
- Multiplication by constant $c$: each $P_i$ locally computes $c \cdot \hat{x}$.
- Multiplication: this computation consumes the triple $((\lambda^x_i)_{i \in [N]}, (\lambda^y_i)_{i \in [N]}, (\lambda^{xy}_i)_{i \in [N]})$. Each party $P_i$ first locally computes $s_i = \lambda^x_i \cdot \hat{x} - \hat{y} \cdot \lambda^y_i - \lambda_i^{xy}$ and broadcasts $s_i$. Then the masked output $\hat{z} = x y + \sum_{i \in [N]} \lambda^z_i$ can be obtained as $\hat{z} = \sum_{i \in [N]} s_i + \hat{x} \hat{y}$ by each party.

Notice that $\Pi^{\text{PA}}$ only broadcasts once for each multiplication gate, thanks to the correlated randomness computed during the offline phase. All other operations are computed locally by the parties.

### 2.2 Picnic

The signature scheme Picnic is an instance of the MPCitH paradigm described above. The function $f$ is the LowMC block cipher, the signer’s secret key is the witness $w$, and the public key is the statement $x = (p, c)$. A signature consists of a proof of knowledge of $w$ such that $f_p(w) = c$. In the block cipher notation, if the secret key is denoted $sk$, then the public key is a plaintext-ciphertext pair $(p, \text{LowMC}_{sk}(p))$ where $p$ is a randomly chosen plaintext block, and the signature proves knowledge of a key relating the plaintext $p$ and the ciphertext $\text{LowMC}_{sk}(p)$. The proof is made non-interactive by the Fiat-Shamir transform, and the message to be signed is bound to the proof by hashing it into the challenge.

The Picnic specification [Pic20] and NIST submission includes parameter sets using both the ZKB++ proof system and the KKW system, as well as specific choices of parameters for LowMC. Since the KKW-based parameters (referred to as Picnic3) are the most efficient in terms of signature size, we choose to focus on those in this paper. In particular our masked implementation is limited to the parameter set Picnic3-L1. Fig. 6 describes the KKW proof system at a high level, and Algorithm 14 describes Picnic3 signing in full detail.

### 2.3 Side-Channel Attacks and Threat Model

Physical attacks are a threat for cryptographic implementations. Attacks such as side-channel analysis (SCA) can be used to extract secret keys by observing physical properties of an implementation, such as timing, power consumption or electromagnetic emanation [Koc96, KJJ99, QS11]. A popular countermeasure to SCA is known as masking [CJRR99a, ISW03]. Masking can protect against a broad class of SCA and probing attacks by splitting secrets into independent shares. A popular model for analyzing masking schemes is the $t$-probing model, where the adversary can probe one (or more) shares of the masked variable [ISW03]. Briefly, a probing adversary may invoke a cryptographic implementation multiple times with chosen inputs. Before each call, the adversary can choose a set of up to $t$ wires of the circuit and observe the values on these wires during the invocation. After $c$ calls, an attacker can then combine the $c \times t$ observations in arbitrary ways to extract information about sensitive variables. This model is closely related to concrete physical attacks: For example, in [KGM+20], eight simultaneous probes are given as an upper limit achievable by modern commercially-available probe stations, and we therefore assume that $t$ is at most sixteen. While the $t$-probing model is a clean theoretical model, Duc et al. [DDF19] showed that security in this model implies security in the more practical SCA-inspired noisy leakage model [PR13]. To protect against multi-probe
attacks, generally higher-order masking is applied, where the number of independent shares is increased. Higher-order SCA is expensive, as the number of measurements needed grows exponentially with the masking order, effectively limiting the attack order or the number of simultaneous probes an adversary may use (which we assume is below sixteen). Kranchenfels et al. [KGM+20] describes a new attack technique, laser logic state imaging, that can potentially have an unlimited number of probes, however this attack is quite new and may not be widely applicable, but in any case must be mitigated with countermeasures below the software level (at the package, device or circuit level). Besides probing and SCA, there are further physical attacks like fault analysis [BDL97] that we do not address in this work.

In this work, we use the noisy leakage model introduced by Chari et al. [CJRR99b] and extended by Prouff and Rivain [PR13]. The model enables an adversary to obtain each intermediate value perturbed with a noisy leakage function. Furthermore, as stated above we use the connection between probing and the noisy leakage model given by Duc et al. [DDF19]. Therefore in our threat model, a probing adversary reflects the capabilities of a real world adversary such as DPA. We assume an adversary who can access the physical device that can run the Picnic3 signature scheme. They can measure side-channel traces, such as power or electromagnetic emanation, of the device while signing chosen messages. Moreover they obtain the signatures as output, and can verify (and thus see the revealed values) or use them arbitrarily in an attack. Observe that according to the noisy leakage model the side-channel trace contains each intermediate value perturbed with a noisy leakage function. Depending on the signature the revealed values vary and the adversary can employ these variables, to recover the secret.

2.4 Security Notions for Masking Countermeasures

For more comprehensive background we refer readers to Appendix G. In the following, we fix a finite field \( F \). As explained above, we are working in the \( t \)-probing model which allows an attacker to obtain the value of \( t \) variables per run of the primitive. The most common technique to mitigate side-channel attacks is by encoding sensitive variables via an additive (or polynomial-based) secret sharing into \( T > t \) parts. We use the additive approach, and say that a vector \((v_j)_{j \in [T]} \in F^T\) is a \( T \)-encoding of \( v := \sum_{j \in [T]} v_j \). For readability, we often write \((v)\) instead of \((v_j)_{j \in [T]}\). For a subset \( I \subseteq [T] \), let \((x)_{I} = (x_i)_{i \in I}\) and furthermore \( T = [T] \setminus I \). Variables are shared both to protect against side-channel attacks and as part of the MPC protocol. To distinguish between these situations, we call a sharing between parties in the MPC protocol a sharing and use the term encoding when the goal is to protect against side-channel attacks. In this work, we aim to prove that our basic building blocks meet the following standard security notions [BBD+16].

**Definition 1** (\( t \)-NI, \( t \)-SNI). Let \( G \) be a gadget with inputs in \( F^T \) and \( t < T \). Suppose that for any set of \( t_1 \) intermediate variables and any subset of \( O \) of output indices with \( t_1 + |O| \leq t \), there exists a subset of indices \( I \) such that the output distribution of the \( t_1 \) intermediate variables and the output variables \( y_{\overline{O}} \) is perfectly simulatable from \( I \). Then

(i) if \( |I| \leq t_1 + |O| \) we say \( G \) is \( t \)-non-interfering (\( t \)-NI), and

(ii) if \( |I| \leq t_1 \) we say \( G \) is \( t \)-strong-non-interfering (\( t \)-SNI).

The above definition of SNI as well as its composability result can be generalized for a gadget with multiple input/output encodings [CS20]. We also say \( G \) is \( t \)-SNI with uniform output-distribution, if the outputs of \( G \) which are not affected by any probes are uniformly distributed (see Appendix G for a definition).
3 Probing Attacks on Picnic3

We describe two probing side-channel attacks against Picnic3 and experimentally confirm them against our M4 port of the optimized Picnic3 implementation. The second attack is deferred to Appendix A since it is a direct consequence of previous work [GSE20]. Probing attacks usually exploit weak leakage of intermediate variables, gathered from several measurements. As described in [SBWE20] and in [GSE20], the values revealed by the prover to allow the verifier to check the consistency of the MPC protocol can be employed by an adversary in a side-channel attack. We assume the same scenario. Furthermore, we assume a leakage model where an implementation leaks weak and noisy information about each intermediate variable, therefore measurements of the MPC-in-the-head simulation have a weak and noisy dependence on secret values due to the revealed values. As mentioned above, we make use of the RvR tests to show the clear presence of leakage.

3.1 Probing the Unopened Online Phase

This attack is novel, as it is specific to MPCitH with preprocessing, and only occurs when 5 protocol rounds are compressed to 3. Hence the attack below works in principle for any direct implementation of signatures derived from three-round KKW-based protocols. We also remark that this attack cannot be mitigated by the SNIitH approach [SBWE20]; in particular, the attack below works independently of the number of unopened parties’ views since it targets an input to the MPC (i.e., \( \hat{w} \)), not a share of the secret. We are thus motivated to design an alternative solution to thwart this attack in the next section.

We first note the three-round KKW scheme executes both the offline and online phase of each MPC instance, in contrast to the five-round case. We denote by \( C \) the executions chosen for the online phase, i.e., the executions where the offline phase is not public.

The attack exploits the following: if the \( k \)-th execution of the offline phase is selected to be part of the signature (i.e., if \( k \notin C \)), the preprocessed masks and state of all \( N \) parties is made public for the verifier, therefore the corresponding online phase must remain hidden. Concretely, since the secret witness wire value \( w \) is masked by random bits in step 1c of the prover in Fig. 6, the attacker’s goal is to learn the masked witness wire values \( \hat{w}^{(k)} \) in execution \( k \) for the unopened online executions \( k \notin C \). Since \( \hat{w}^{(k)} = \lambda_1 w^{(k)} + \ldots + \lambda_N w^{(k)} + w \) and \( \lambda_i w^{(k)} \) is made public (for all \( i \)), by probing \( \hat{w}^{(k)} \) the attacker can solve for the secret key bit \( w \). Here, \( \lambda_i w^{(k)} \) denotes the value of \( \lambda_i w \) in execution \( k \).

In order to validate the attack we use our experimental setup (as described in Section 6) and the RvR approach. The experiment shows that there is an exploitable leakage, i.e., an amount of leakage sufficient, despite measurement noise, to allow recovery of intermediate values that depend on the secret key. For this experiment, we reduce the Picnic3 parameters (as in Fig. 6) \( M \) and \( \tau \) to 4 and 2 respectively, in order to collect traces more quickly, however we keep the number of parties \( N \) as 16 and collected traces corresponding to execution of the first MPC instance. During the collection phase, we run the Picnic3 signing function with random messages and a fixed secret key. More specifically, we measure the execution of the first line of Algorithm 18, and collect 22,056 side-channel traces. Note that the root seed is also random due to the choice of a random message to be signed. We first separate the traces belonging to signatures that reveal the first preprocessing phase, since our measurement covers the first MPC instance. The reduced number of MPC instances is only to reduce the number of possible challenges and to increase the number of traces per challenge. Then we classify the remaining traces into two sets according to the revealed values \( \lambda_1 w^{(k)} + \ldots + \lambda_{16} w^{(k)} \). The result of the analysis in Fig. 1 (left side) shows a clear dependence between the unrevealed value \( \hat{w}^{(k)} \) and the observable trace, as the \(|\cdot|\)-value exceeds 5.7 which indicates an exploitable leakage. As seen in the right hand side of the Fig. 1, the leakage becomes clear after 2,725 traces.
Figure 1: A first-order RvR test on the unprotected Picnic3 implementation using revealed values from the offline phase. The traces are classified based on the bit $\lambda_1^w + \cdots + \lambda_1^{16}$. Values above the 5.7 threshold (red line) indicate strong leakage (left). Moreover, the maximum $|t|$-value increases with respect to the number of traces and the leakage becomes clear after 2,725 traces (right).

The code we measure (the first line of Algorithm 18) corresponds to the calculation of roundkey $0$ thus the leakage corresponds to the bits of roundkey $0$ which is equal to $\text{matMul}(\hat{sk}, K_0)$. Solving the equation for the $sk = \hat{sk} - (\lambda_1^sk + \cdots + \lambda_1^{sk})K_0$ where $(\lambda_i^{sk}$ is known for all $i$ and $K_0$ is a constant) leads to the secret value $sk$.

4 Masking Three-Round KKW

In this section we present our masked proof system following the three-round KKW protocol. We first strive for a provably NI-secure algorithm without specifying any particular circuit. In Section 5 we describe more concrete operations tailored to the LowMC circuit of Picnic3, and optimize by partially unmasking several hash computations (and discuss the security implications). The circuit in this presentation is a generic circuit $C$ such that $C((w)_w \in \mathbb{N}) = 1$, where each $w$ is seen as an input wire value to the circuit.

In the description below, we additively secret share some variables in two dimensions: a share held by each party (indexed by $i \in [N]$), further shared $T$ times within each party (indexed by $j \in [T]$). For example, $\lambda_x^i$ denotes a share of of $\lambda_x$ held by the $i$-th party such that $\lambda_x = \sum_{i \in [N]} \lambda_x^i$; the value $\lambda_x^i,j$ for $j \in [T]$ denote shares such that $\lambda_x^i,j = \sum_{j \in [T]} \lambda_x^i,j$. The shares $\lambda_x^i$ are as in the KKW protocol, and the extra $T$-wise encoding of this value is required for SNI security. Note that we only apply the notation $\langle \cdot \rangle$ (see Section 2.4) on the $T$-wise encoding required for the masking countermeasure and never for the sharing of the KKW protocol. The functions requiring $T$-wise masked computation are marked in orange (e.g., $y \leftarrow H(\langle x \rangle)$ indicates that a hash function $H$ is masked). Most of the randomness used in the protocol is from the random tapes of the parties; this randomness is derived from a seed, so that part of it may be efficiently communicated to the verifier (by sending the seed). Some of the masked operations will require additional randomness (e.g., to refresh a secret encoding), and this is sampled from the platform random number generator (RNG), since it is only required by the signer.

4.1 Masked Operations

We present our masked version of the KKW prover in Fig. 2. The function $\text{masked\_offline}$ in Algorithm 1 computes the offline phase $\Pi_{\text{off}}^C$ from Section 2.1 in a straightforward way. The function $\text{masked\_online}$ in Algorithm 2 corresponds to a masked version of $\Pi_{\text{on}}^C$. The SNI-secure multiplier $\text{SMul}( )$ is defined in Algorithm 11. Note that $\text{SMul}( )$ is $t$-SNI with uniform output-distribution, as evident from the proof that it is $t$-SNI [BBD+16, Proposition 2]. For ADD gates, the only change is to work on $T$-encodings $\langle \hat{x} \rangle$, $\langle \hat{y} \rangle$, rather than on $\hat{x}$, $\hat{y}$ directly. Interestingly, the MUL gates can also be computed with a straight-forward
Algorithm Masked KKW Prover

**Inputs** The prover holds a circuit $C$ as a statement and an encoded witness $w = \{w\}_{w \in \mathbb{N}}$ such that $C(w) = 1$. Values $M$, $N$, $\tau$ are parameters of the protocol.

**Commit** For each $k \in [M]$, the prover does:

1. Choose uniform $(\text{seed}^{(k)})$ and use to generate values $(\text{seed}^{(1)}), \ldots, (\text{seed}^{(N)}).$ Also the prover computes $(\text{aux}^{(k)}) \in \mathbb{F}^C$ as in Algorithm 1. For all $i = 1, \ldots, N - 1$, let $(\text{state}^{(k)}_i) = (\text{seed}^{(k)}_i) || (\text{aux}^{(k)}).

2. Commit to the offline phase:

$$\langle \text{com}^{(k)}_i \rangle = H(\langle \text{state}^{(k)}_i \rangle) \text{ and reconstruct } \text{com}^{(k)}_i \text{ for all } i \in [N]$$

and reconstruct $\text{com}^{(k)}_i$.

3. Compute encodings of masked witness $(\hat{w}^{(k)}) = (\lambda_1^k) + \ldots + (\lambda_N^k) + \langle w \rangle$ for each $w \in \mathbb{N}$, where $(\lambda_i^k)$ is the randomness used to mask the witness and is read from the random tape defined by $(\text{state}^{(k)}).

4. Simulate the online phase of the $N$-party protocol as in Algorithm 2 and produce $(\text{msgs}^{(1)}), \ldots, (\text{msgs}^{(N)}).

5. Commit to the online phase:

$$\langle \text{com}_{\text{on}(k)} \rangle = H(\{\langle \hat{w}^{(k)} \rangle\}_{w \in \mathbb{N}}; (\text{msgs}^{(1)}), \ldots, (\text{msgs}^{(N)}))$$

and reconstruct $\text{com}_{\text{on}(k)}$.

6. The prover refreshes the encoding of witness $\langle w \rangle = \text{RefreshM}(\langle w \rangle)$ for each $w \in \mathbb{N}$.

Compute $h_{\text{off}} = H(\langle \text{com}_{\text{off}(1)} \rangle, \ldots, \langle \text{com}_{\text{off}(M)} \rangle)$ and $h_{\text{on}} = H(\langle \text{com}_{\text{on}(1)} \rangle, \ldots, \langle \text{com}_{\text{on}(M)} \rangle)$ and send $h^* = H(h_{\text{off}}, h_{\text{on}})$ to the verifier.

**Challenge** The prover receives the following challenges from the verifier: a uniform $\tau$-sized set $\mathcal{C} \subseteq [M]$ and $P = \{i_k\}_{k \in \mathcal{C}}$ where each $i_k \in [N]$ is uniform.

**Response** For each $k \in [M] \setminus \mathcal{C}$, the prover sends reconstructed $\text{seed}^{(k)}$ and $\text{com}_{\text{on}(k)}$ for all to the verifier. For each $k \in \mathcal{C}$, the prover sends reconstructed values $\text{com}^{(k)}_i, \{\text{state}^{(k)}_i\}_{i \in \mathcal{C}}, \{\hat{w}^{(k)}\}_{w \in \mathbb{N}}$ and $\text{msgs}^{(k)}$ to the verifier.

**Figure 2:** Our masked version of 3-round KKW prover.
adaptation by also encoding the masks $\lambda^x, \lambda^y, \lambda^z$, and $\lambda^{xy}$. Recall that $\hat{x} = x + \lambda^x$, and $\lambda^x$ is shared among the parties, where party $i$ has share $\lambda^x_i$ (resp. $\lambda^y_i, \lambda^z_i$, and $\lambda^{xy}_i$). Each party thus stores their share $\lambda^x_i$ as a $T$-encoding $\langle \lambda^x_i \rangle$ (resp. $\langle \lambda^y_i \rangle, \langle \lambda^z_i \rangle$, and $\langle \lambda^{xy}_i \rangle$). Each party’s broadcast value $s_i$ will now also consist of the $T$-encoding $\langle s_i \rangle$ as follows:

$$
\langle s_i \rangle = \langle \lambda^x_i \rangle - \langle \lambda^{xy}_i \rangle - \text{SMul}(\langle \hat{x} \rangle, \langle \lambda^y_i \rangle) - \text{SMul}(\langle y \rangle, \langle \lambda^z_i \rangle).
$$

### 4.2 Security Analysis

We employ the definition of non-interference by [BBD+16] which guarantees security against $t$ probes for $t < T$ as proposed by Ishai et al. [ISW03]. Recall that a probing adversary may invoke a cryptographic implementation multiple times with chosen inputs and before each call, can fix an arbitrary set of up to $t$ wires of the circuit and observe the values during the invocation. We use a more polished security notion known as $t$-non-interference $(t\text{-NI})$ and $t$-strong non-interference $(t\text{-SNI})$ defined in Appendix G.

In the security analysis we focus on a single MUL operation as extracted from Algorithms 1 and 2, denoted KKW\_MUL, presented as Algorithm 22 and 23, as the ADD operation is linear and thus trivially NI. See Appendix H for the proofs of the lemmas.

**Lemma 1.** Let $G$ be the KKW\_MUL gadget as described in Algorithm 22. Then, $G$ is $t$-SNI for all $t < T$, if $\text{SMul}(\cdot)$ is $t$-SNI with uniform output-distribution.

**Lemma 2.** Let $G$ be the KKW\_MUL gadget as described in Algorithm 23. Then, $G$ is $t$-SNI for all $t < T$, if $\text{SMul}(\cdot)$ is $t$-SNI with uniform output-distribution.

To further support our security analysis, we utilized maskVerif [BBC+19] to confirm that both KKW\_MUL offline and online gadgets are SNI-secure up to the order 4. Although we further provide the verification scripts for the orders 5, 6 and 7, we did not run these, due to higher combinatorial complexity of maskVerif. However, our manual security analysis indeed guarantees SNI security of higher-order masking.

As we have shown that all components of Algorithm 1 and Algorithm 2 are SNI, the composability guaranteed by Lemma 3 as well as its generalization to multi-input/output SNI gadgets [CS20] implies the following theorem by adding suitable refresh gadgets (depending on the topology of circuit $C$ that KKW is instantiated with). Note that SNI security of the refresh gadget (recalled in Algorithm 12) is already proved by Barthe et al. [BBD+16]

**Theorem 1.** Suppose $H(\cdot)$ and RefreshM(·) are $t$-SNI secure gadgets for all $t < T$. The masked version of KKW presented in Fig. 2 is $t$-NIo for all $t < T$ and for public outputs $\{\text{com}_{i\neq i}(k)\}_{k \in C}, \{\text{com}(k)\}_{k \in |M| \cap [N]}, \{\text{com}_\text{off}(k)\}_{k \in |M|}$ and $\{\text{com}_\text{on}(k)\}_{k \in C}$.

The public outputs stated above are not explicitly included in the KKW proof. We thus have to validate the security of proof system in case these values are made public, which is straightforward since they are recomputed as part of verification from the other information in the proof (see the verification step of Fig. 6). Furthermore, note that masking the hash function $H(\cdot)$ is important. There are scenarios where the inputs to $H(\cdot)$ are sensitive, but the outputs are not (such as step 5 in Fig. 2). Leaving the hash function unmasked might therefore leak information about these sensitive inputs.

### 5 Masking Picnic

We start by analyzing the hashing operations in signature generation to determine which ones must be masked, then discuss the options for masking SHA3/SHAKE, and introduce the half-masking technique, then estimate the overhead of masking the hash invocations. Our masked implementation of the Picnic3 signature generation function is a rather direct
adaptation of the masked KKW proof protocol from Section 4. When compared to Section 4, the circuit is LowMC, and operations are done on $N$-bit words packed with a secret share from each of the $N$ parties. Fig. 7 (in Appendix C) gives an overview of the protections for each hashing operation in signature generation. A complete specification of our protected implementation, mirroring the official Picnic specification [Pic20], is given in Appendix C.

5.1 Implementation Security Options

We implemented several versions of masked Picnic3, of which we highlight two: (1) a provably NLo-secure implementation (as a direct consequence of Theorem 1) and (2) a performance-oriented implementation with partially unmasked non-sensitive intermediate values. For the former, we inserted the share refresh gadget (RefreshM) according to the generic composition rule stated in Lemma 3, and we verified with maskVerif that our complete specification of fully masked Picnic3 (see Appendix C.1) is indeed NLo-secure. On the other hand, we do not claim that our second implementation is NLo-secure, as there are some gaps between the analysis of Section 4 and this implementation, that we consciously allowed, in order to improve performance. We now explain these gaps and argue that they do not impact practical security, and in Section 6 we confirm the absence of leakage experimentally.

First, according to Algorithm 4, all intermediate seeds must be $T$-encoded until they are reconstructed at lines 29 and 32 and, as we argue in Section 5.2, reading $t$ bits of a seed reduces security by at most $t$ bits. As we assume $t$ to be small, we accepted this risk to reduce the cost of masking SHAKE and the memory required to store seeds.

By selectively masking the hash function calls (as described in Section 5.2) as opposed to masking all hash function calls, up to $t$ bits of a single-use seed may leak to a side-channel attacker capable of accurately reading $t$ bits from a single trace. (Since the seed is only ever used once, and the signature is randomized, subsequent traces have a fresh seed.) Against such an attack, the security of our L1 implementation decreases from 128 to $128 - t$ bits against side-channel attacks, and we assume that $t \leq 16$ in Section 2.3. As shown in Sections 5.4 and 6 this optimization gives significant performance gains, so we see this as a reasonable trade-off. Recent work by Kannwischer et al. [KPP20] describes single-trace attacks on the unprotected XKCP Keccak implementation. These attacks use a single trace recorded during the computation of $y = SHAKE(sk||x)$ and aim to recover all of a secret key $sk$, or part of $y$. While single-trace attacks could threaten some of the unprotected hash calls in our optimized implementation (e.g., when deriving the per-party or per-MPC instance seeds), the results of [KPP20] do not extend to the M4, and the length constraints on $sk$, $x$, and $y$ in our application. Future work may improve single-trace attacks, and in that case the conclusion of [KPP20] is that lightweight countermeasures will provide effective mitigation.

Half-masking (discussed in Section 5.3) also introduces the assumption that KangarooTwelve [VWA+21] is a secure hash function. This assumption is only for security against the type of $t$-probing side-channel attack we consider; and half-masking can be used by individual implementations without changes to the Picnic specification. We provide benchmarks in Section 6 showing the performance advantage of half-masking: based on this and the fact that security of KangarooTwelve appears to be a relatively mild assumption, we enable half-masking by default in our implementation.

Finally, our provable security analysis assumed an SNI-secure hash implementation. Although one could use the fully SNI-secure masked Keccak as suggested by Barthe et al. [BBD+16], other previous works [BDPA10, Dae17, GSM17] achieved more efficient implementations with smaller amounts of random bits, albeit without a provable security guarantee. We implement three instances of masked Keccak (named IND, DOM, and SNI) with different security levels, which we explain in detail in Section 5.3. In Section 6, we compare the concrete performance and perform practical leakage analysis. From these experiments, we conclude that our implementation of IND – the fastest instance among three – does not leak information.
5.2 Side-Channel Protections for Hashing in Picnic

Hashing Operations for Signing The concrete sign operations are described in Algorithm 3, following the Picnic specification [Pic20, Section 7.1]. When compared to the stylized description of KKW in Fig. 6, here we include more details and list all hashing operations, since we will analyze them with respect to the probing attacks below. Some of the functions related to expanding seeds using a tree construction or creating a Merkle tree of commitments (gen_seed, get_leaves, build_tree, and open_tree) are left to the specification for simplicity.

The hash function calls are denoted by $H$ and we omit the byte used for domain separation present in the specification. The KDF expands an arbitrary length input to an arbitrary length output. Both $H$ and KDF are instantiated with the SHAKE XOF.

Algorithm 3 Description of Picnic signing highlighting hashing operations.

Input: signer’s key pair $sk = (w)_{w \in \mathbb{N}}$, $pk$, message to be signed $Msg$.

1. // Derive root seed:
   Sample random $R \in \{0, 1\}^{|\kappa|}$, $(\text{seed}^*, \text{salt}) \leftarrow \text{KDF}(sk||Msg||pk||\kappa||R)$
2. $i\text{Seed}_\text{tree} \leftarrow \text{gen}_\text{seed}(\text{seed}^*, M, 0)$ // Tree of initial seeds
3. // Initial seed for each MPC instance:
   $(i\text{Seed}^{(1)}, \ldots, i\text{Seed}^{(M)}) \leftarrow \text{get_leaves}(i\text{Seed}_\text{tree})$
4. for each $k \in [M]$
   $\text{seed}_\text{tree}^{(k)} \leftarrow \text{gen}_\text{seed}(\text{seed}^{(k)}, M, 1)$ // Seeds for MPC instance $k$
5. $(\text{seed}_1, \ldots, \text{seed}_N) \leftarrow \text{get_leaves}(\text{seed}_\text{tree}^{(k)})$ // $N$ per-party seeds
6. For each $i \in [N]$: $\text{tapes}_i \leftarrow \text{KDF}(\text{seed}_i, salt, k, i)$
7. $(\text{aux}_i^{(k)}, \text{tapes}_N^{(k)}) \leftarrow \text{offline}(\text{tapes}_1^{(k)}, \ldots, \text{tapes}_N^{(k)})$
8. For each $i \in [N - 1]$: $\text{com}_i \leftarrow H(\text{seed}_i, salt, k, i)$
9. $\text{com}_N \leftarrow H(\text{seed}_N, \text{aux}_i^{(k)}, salt, k, N)$
10. $\text{com}_{off}^{(k)} \leftarrow H(\text{com}_1^{(k)}, \ldots, \text{com}_N^{(k)})$
11. For each input wire $w$: $\hat{w}^{(k)} \leftarrow w \oplus \sum_{i \in [N]} \lambda_i^w$
12. $(\text{msgs}_1^{(k)}, \ldots, \text{msgs}_N^{(k)}) \leftarrow \text{online}((\hat{w}^{(k)})_{w \in \mathbb{N}}, \text{tapes}_1^{(k)}, \ldots, \text{tapes}_N^{(k)}, pk)$
13. $\text{com}_{on}^{(k)} \leftarrow H((\hat{w}^{(k)})_{w \in \mathbb{N}}, (\text{msgs}_1^{(k)}, \ldots, \text{msgs}_N^{(k)}))$
14. $\text{com}_{on-tree} \leftarrow \text{build_tree}(\text{com}_{on}^{(1)}, \ldots, \text{com}_{on}^{(M)})$
15. $h \leftarrow H(\text{com}_{off}^{(1)}, \ldots, \text{com}_{off}^{(M)}, \text{com}_{on-tree.root}, salt, pk, Msg)$
16. Parse $h$ as $(C, P)$ where $C \subseteq [M]$ and $P = \{i_k\}_{k \in C}, i_k \in [N]$
17. $\text{com}_{on-info} \leftarrow \text{open_tree}(\text{com}_{on-tree}, M, C)$
18. $\text{iSeed}_\text{info} \leftarrow \text{reveal_seed}(i\text{Seed}_\text{tree}, M, C)$
19. For each $k \in C$: $\text{seed}_\text{info}^{(k)} \leftarrow \text{reveal_seed}(\text{seed}_\text{tree}^{(k)}, N, i_k)$
20. $Z \leftarrow (\text{com}_{on-info}, \text{iSeed}_\text{info}, (\text{seed}_\text{info}^{(k)}), \text{aux}_i^{(k)}, ((\hat{w}^{(k)})_{w \in \mathbb{N}}, \text{com}_i^{(k)}, \text{msgs}_i^{(k)}))_{k \in C}$.

Output: $(h, salt, Z)$ as a signature

We now consider which hashing operations must be protected against side-channel attacks, and to what degree. The Picnic specification supports randomized signatures (and recommends this option, following [AOTZ20]) by appending a random value to the KDF input when deriving the root seed. We assume this option is used throughout, as otherwise the cost of side-channel protections would be significantly higher, since all hash function calls would require masking (as opposed to only 35% shown below), and all random seeds would need to be $T$-encoded. First, we note that all inputs to the challenge computation are public, so this hash does not need to be masked. We now analyze the other hash function calls in order. Fig. 7 (in Appendix C) gives an overview of the operations. When referring to individual hashing operations, we refer to the steps in Algorithm 4.
We implement multiple methods to protect the SHA3 family of function against DPA attacks. As a first option, one could choose the root seed at random, and avoid the KDF altogether. However, deriving the root seed from the secret key and random data hedges against failures in the RNG, see the analysis for Picnic in [AOTZ20]. If sk is stored T-encoded, then we can hash all of the shares in place of sk, and append a random value. Our implementation masks this hash function call since it is relatively cheap in the context of a signature, and it makes testing easier because our implementation can produce signatures that match known test vectors.

**Deriving the root seed.** For step 1, the SHAKE XOF is used as a KDF to derive a root seed for signature generation. The input sk must be protected from side-channel attacks. As a first option, one could choose the root seed at random, and avoid the KDF altogether. However, deriving the root seed from the secret key and random data hedges against failures in the RNG, see the analysis for Picnic in [AOTZ20]. If sk is stored T-encoded, then we can hash all of the shares in place of sk, and append a random value. Our implementation masks this hash function call since it is relatively cheap in the context of a signature, and it makes testing easier because our implementation can produce signatures that match known test vectors.

**Deriving other seeds.** When generating the seeds in steps 2, 3, 5, and 6, protecting against the limited type of leakage we consider in this work is not necessary, since seeds are unique per-signature and are always hashed before use. Suppose an attacker A can read t bits of a leaf or intermediate seed s. With overwhelming probability each seed is only ever used in one signature, so traces from multiple signing operations will not give more information about s.

There are three possible uses of s to consider. When s is a seed from a leaf of the tree, case 1 is that s is hidden and the attacker has a commitment to it (computed in steps 9 and 10), and case 2 is when s is used to seed KDF (in step 7), and A has some of the output bits. In case 3, s is a hidden intermediate seed, the attacker has one of the two child seeds, derived by hashing s.

We can model all three cases as the attacker having $C = H(s)$ along with t bits of s, where H is a secure hash function. In practice H is the SHAKE XOF, which the existing analysis of Picnic already assumes is a random oracle. Then if A makes q queries to H, they recover the missing $\kappa - t$ bits of s with probability not more than $q/2^{\kappa-t}$. This considers only a single seed and digest, which we can do since each input to H is unique, by construction (the Picnic spec uses a domain separation value, random salt, and counters to prevent multi-target attacks [DN19]). In practice $\kappa \geq 128$ and t will be 16 or less (see Section 2.3), therefore the security of our implementation is still at least 112 bits.

**Computing random tapes.** In step 7 we expand the per-party seeds to random tapes. The inputs do not need to be protected (as discussed in the previous paragraph), but all output bits must be protected, since some of the random tape bits will correspond to shares of the unopened party and must be kept secret as shown in in Section 3. We mask these calls, so the output is T-encoded (which increases the amount of memory required to store the tapes by a factor of T).

**Computing commitments.** In step 9, a commitment of the form $H(\text{seed}||salt)$ is computed. Here the private input is a seed, which is not sensitive to leakage of up to t bits, as discussed above, and the output is public. Therefore, step 9 does not require masking.

In step 10, the last party’s commitment has the additional input aux, which is sensitive to leakage of individual bits. We must mask this call, but since the output is public, we can use the half-masking technique of Section 5.3.

In step 11, we hash only public values, and no masking is required. In step 14, all inputs are sensitive to leaking individual bits (e.g., $\hat{w}$ is sensitive due to the attack described in Section 3.1). Because the output is public, the half-masking technique is applicable.

### 5.3 Masking SHAKE

We implement multiple methods to protect the SHA3 family of function against DPA attacks. In all of them, the Keccak-f state array A is secret shared into two arrays a, b, such that $A = a + b$. In the basic method proposed in [BDPA10], the linear operations are performed on the individual state arrays, then for the non-linear step (denoted $\chi$) $A_i \leftarrow A_i + (A_{i+1}+1)A_{i+2}$ the shares $(a, b)$ are updated as $a_i \leftarrow a_i + (a_{i+1}+1)a_{i+2} + a_{i+1}b_{i+2}$ and $b_i \leftarrow b_i + (b_{i+1}+1)b_{i+2} + b_{i+1}a_{i+2}$, evaluated left-to-right. The cost of the linear
operations are doubled, addition of constants have the same cost, and the cost of \(\chi\) is doubled, plus two additional AND and XOR operations, so the computational cost of the masked round function is roughly doubled. One must also consider the cost of generating random values to create the secret shares. This method (herein called IND) only achieves independence from the native variables, and the same approach can be generalized to three or more shares. In Domain-Oriented Masking (DOM) [GSM17], the AND operations between shares \(a\) and \(b\) are further protected to satisfy SNI-security with a random mask \(Z\) as \((a_{i+1}b_{i+2} + Z)\) and \((b_{i+1}a_{i+2} + Z)\), respectively. However, this is still not sufficient for the masked Keccak as a whole to be SNI secure: due to the \(\theta\)-layer, which applies a linear transform to the state array \(A\), both inputs to the AND gadget in \(\chi\) depend on the same previous state bit. This is a typical pattern of insecure composition observed in [BBD+15b, §2.3]. Therefore, the third method (denoted by SNI) achieves SNI-security by additionally refreshing shares of the state array \(A\) for every invocation of \(\chi\), as already suggested by Barthe et al. [BBD+16, §8.2].

**Half-Masked SHAKE.** When expanding a seed to a random tape, we have shown that security is maintained when leaking a small part of the seed (\(t\) bits of fewer), so the input is not sensitive to this bounded leakage, but the output is sensitive. Conversely, when creating a commitment, the individual input bits may be sensitive but the output is public. An established assumption (for SHAKE128) is that security is preserved using only half the number of rounds, and there is a proposal called KangarooTwelve (K12) [VWA+21], that uses 12 instead of 24 rounds. Therefore, for short inputs and outputs, one can view SHAKE as two calls to K12, and mask only one of the calls. In the case of sensitive inputs, we mask the first 12 rounds: an attacker who learns the state at the 13th round is effectively given a K12 digest of the input, which sufficiently hides the input under the assumption that K12 is a secure hash function. Similarly, when only the output bits are sensitive, we mask the last 12 rounds, any state bits observed by the adversary in round 11 does not leak useful information about the output assuming that K12 is secure.

### 5.4 Estimated Overhead of Hash Function Masking in Picnic

Here we provide a rough estimate of the overhead introduced by masking the SHAKE calls in Picnic3, which will have a high impact on the cost of signing since hashing is a large portion of the signing time (e.g., at L1 it is about 57% of the signing time on x64 [KZ20], and for our ARM M4 implementation it is about 71%).

- For seed tree hashing, we have about \(M + \log_2 M\) hashes to compute the round seeds, and \(MN + M \log_2 N\) hashes for the per-party seeds. None of these must be masked.
- For random tape expansion, we have \(MN\) hashes, all of which must be masked.
- For commitments, we have \(NM + 2M + \log_2 M\) hashes and must mask \(2M\) of these. The total number of hashes is thus \(3MN + 3M + 2\log_2 M + \log_2 N\) and \(MN + 2M\) of these must be masked. At L1, all hash operations involve one call to Keccak-f, so all calls have approximately the same cost. Again at L1, \(M = 250, N = 16\) so we find that about 35% of hashing must be masked. Since all masked hash operations have either non-sensitive input or output they need only be half-masked (as explained below).

Now suppose we focus on first-order protection (the case \(T = 2\)), and assume that masked SHA-3 is about 2.73 times slower than unmasked SHA-3, and that a half-masked SHA-3 is about 1.95 times slower (these are the ratios from our implementation described in Section 6). Then we expect a 1.61x increase in time spent hashing in masked Picnic3, and 1.35x increase when half-masking is used.

### 6 Implementation and Experimental Evaluation

In this section we benchmark our implementation and discuss performance, then describe our experiments to ensure that our implementation is side-channel resistant in practice.
6.1 Implementation and Benchmarks

We implemented our masked version of Picnic signing and benchmarked it on the ARM Cortex M4, using the pqm4 [KRSS] suite and the STMicro developer board STM32F407G-DISC1. This board has one MB of flash memory and 192KB of RAM and comes equipped with a true random number generator implemented as a hardware peripheral. The microcontroller clock frequency ranges from 24 to 168MHz, so following standard practice our benchmarks were executed at the lowest frequency to avoid the impact of memory wait states [FA17, HL19].

Our implementation is derived from the Picnic optimized implementation, which is primarily optimized for x64 platforms, and is not well-optimized for the M4. As such, our implementation results aim to bound the overhead of masking countermeasures. We also focus only on first-order protection, i.e., the case $T = 2$, and implement only the L1 parameter set picnic3-L1. We decided for first-order protection to show the general effectiveness of our countermeasure. Also, first-order protection already significantly raises the bar for attacks, as higher-order attacks require a much more sophisticated adversary and the attacks become very expensive due to the long traces. As most of our countermeasures are general, we expect them to apply equally to more optimized M4 implementations of Picnic, and (with some effort) to implementations of other MPCitH-based proof systems.

Since our masked implementation produces picnic3-L1 signatures compatible with the specified version [Pic20], we do not repeat signature or key sizes in our benchmarks: public keys are 34 bytes, secret keys are 17 bytes, signatures are 12.4KB. All other parameters such as number of MPC parties, MPC instances, digest lengths, etc. are as specified in [Pic20]. For reference, the verification time in our implementation is 204M cycles.

Masked Keccak. We implemented three different flavors of masked Keccak described in the last section: IND, DOM, and SNI. The implementation was built on top of the in-place 32-bit ARMv7-M assembly code found in the official Keccak code package 3 (XKCP) to operate over a double-sized state storing the two shares. We implement the same Keccak API used in Picnic by replicating functions over each share of the state, and modifying the round function to implement the non-linear operations. Because of the larger state, additional pressure was put on the registers and several intermediate variables had to be spilled onto the stack. This caused some additional performance overhead beyond the raw cost of masking. In order to prevent leakage, we took additional care to rotate registers between rounds to prevent them from loading shares of the same variable [BDPA10].

Benchmarks. In Table 1 we give cycle counts for our masked implementation with various options for how the hash function calls are masked. The masking cost for the non-hashing operations is 158M cycles, which represents an overhead of 1.5x over baseline. Since this is effectively doing the MPC simulation with 2-encoded values, we might expect a factor two slowdown rather than 1.5, however, this is explained by the fact that many of the operations to implement LowMC are ANDs and XORs with public constants, which are more efficient than operations on 2-encoded values. Then the cost when masking the hashing naively (by masking all operations), is given for the three Keccak masking options (SNI, DOM and IND) and we see that the overhead is 1201M, 827M and 395M cycles respectively. By using our analysis of Section 5.2, and selectively masking only sensitive hash function calls, the overhead for IND drops to 153M cycles, and all the way down to 86M cycles when we additionally use the half-masking optimization. In this most performant case, we have roughly 2/3 of the overhead accruing to the Picnic and MPC operations, and 1/3 to the hashing.

Stack usage was essentially constant for all configurations we benchmarked, since the total amount of memory required is dominated by storing the signature, the commitment and seed trees and not by the storage space for intermediate values that we must $t$-encode.

3https://github.com/gvanas/KeccakCodePackage
Table 1: Benchmarks in millions of Cortex-M4 cycles showing the masking overhead for types of side-channel protections. The options for Picnic are “No” masking and \( T = 2 \) masking, as described in Section 5. For SHAKE, the “None” option indicates no masking is used for hash computations, “All-” prefix means every call is masked in one of three possible ways: SNI-secure [BBD+16], Domain-Oriented Masking (DOM) [GSM17] or using independent values (IND) [BDPA10]. “Selective” means that only sensitive calls are masked with independent values (as described in Section 5.2), and “Selective Half” means that in addition to Selectively masking and IND, we use half-masked SHAKE. The Hashing column provides the fraction of the signing time spent computing SHAKE.

<table>
<thead>
<tr>
<th>Picnic Masking</th>
<th>SHAKE Masking</th>
<th>Signing cycles</th>
<th>Hashing</th>
<th>Masking Overhead</th>
<th>Stack Code</th>
<th>Random bytes (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>None</td>
<td>302</td>
<td>70%</td>
<td>1.00</td>
<td>32,460</td>
<td>81,397</td>
</tr>
<tr>
<td>Yes</td>
<td>None</td>
<td>460</td>
<td>50%</td>
<td>1.52</td>
<td>32,516</td>
<td>103,114</td>
</tr>
<tr>
<td>Yes</td>
<td>All-SNI</td>
<td>1661</td>
<td>86%</td>
<td>5.50</td>
<td>32,740</td>
<td>137,872</td>
</tr>
<tr>
<td>Yes</td>
<td>All-DOM</td>
<td>1287</td>
<td>81%</td>
<td>4.26</td>
<td>32,740</td>
<td>130,432</td>
</tr>
<tr>
<td>Yes</td>
<td>All-IND</td>
<td>855</td>
<td>72%</td>
<td>2.83</td>
<td>32,740</td>
<td>122,256</td>
</tr>
<tr>
<td>Yes</td>
<td>Selective</td>
<td>613</td>
<td>62%</td>
<td>2.03</td>
<td>32,740</td>
<td>120,372</td>
</tr>
<tr>
<td>Yes</td>
<td>Sel. Half</td>
<td>546</td>
<td>57%</td>
<td>1.81</td>
<td>32,740</td>
<td>120,372</td>
</tr>
</tbody>
</table>

Code size increases by 1.5x in the most performant masked implementation (selective half-masked), and by 1.7x in the fully SNI-secure version. Finally, the randomness requirements range from the baseline of \( \approx 2\text{MB} \) when Keccak is masked with the IND method, to the much higher 80MB for the DOM method and 158MB for the SNI method, as these methods require additional refreshing of nonlinear operations within Keccak. Using the selective halfmasking option reduces the randomness requirements of the DOM and SNI options significantly, since the number of hash function calls is decreased and some calls are only half-masked. By our estimate in Section 5.4 this would reduce randomness usage by about 65% for the DOM and SNI options. In terms of the \( \approx 2\text{MB} \) of randomness used for the non-hashing operations, these are partly due to the refresh operations within the LowMC implementation (Line 11), they are required for NI security and since they did not have a significant impact on run time, we did not investigate the option of removing them. The other significant randomness consumer in the masked LowMC implementation is the masked AND operation (Algorithm 11). Here, future work could experiment with an implementation that masks ANDs as in the IND method for Keccak, with the aim of reducing randomness and improving run time.

6.2 Experimental Leakage Analysis

To ensure our masked implementations of Keccak and Picnic are practically side-channel resistant, we performed measurements of the implementation to confirm the absence of leakage. Our measurement setup comprises the STMicro developer board STM32F407G-DISC1 also used for the performance benchmarks, operated at 168MHz. We measure EM emanations using a Langer LF-U2.5 near field probe connected to a Langer PA 303 preamplifier [EmP]. The EM probe is placed over the C29 blocking cap at a distance of approx. 1 mm. Measurements are recorded using a Tektronix MSO 6. For the Keccak implementation, we sampled at 3.125 GS/s with a 12 bit resolution and 200Mhz bandwidth. For the Picnic measurements, which are 2 orders of magnitude longer, we reduced the sampling rate to 625 MS/s in order to obtain feasible measurement time and storage sizes. Note that this still over-samples the board (168MHz) by a rate of 3.7 which is well above the minimal oversampling threshold of 2 from he Nyquist–Shannon sampling theorem.

Masked Keccak Leakage Evaluation. For Keccak we evaluate the IND method and follow the FvR approach to detect all possible sources of first-order leakage. During the
trace collection phase, a set of side-channel traces is collected by processing either a fixed input or a random input under the same conditions. The fixed or random choice for the input is made at random. After that, we calculate the means and standard deviations of the two side-channel trace sets separately. The $t$-test indicates whether the two distributions are indistinguishable for a first-order SCA. We apply the customary threshold values for long traces 5.7 as suggested by [DZD+18].

To show the sensitivity of the measurement setup for first-order leakage, we apply the test once to the correctly masked implementation and once to the same implementation with fixed masks. When masks are not chosen at random, the test must detect the resulting first-order leakage. The left hand side of the Fig. 3 shows the evaluation results of the masked Keccak implementation with fixed masks based on 2,000 measurement traces. As expected, the leakage test indicates strong leakage, with $|t|$ clearly above 5.7. When masks are chosen uniformly at random, the $t$-value remains below 5.7 as shown in the right hand side of the Fig. 3, even if the number of measurement traces is increased to 1,000,000. We thus conclude that the masked Keccak implementation is secure and provides the expected resistance to first-order attacks.

Masked Picnic Leakage Evaluation. In order to analyze the leakage for the whole first-order masked Picnic3 implementation, we follow a similar methodology as for Keccak and employ the FvR approach. We collect the traces starting at the beginning of signature
generation until the end of the first MPC instance, i.e., including a single preprocessing phase and simulation of an online phase (Line 1 to Line 22 in Algorithm 4). Note that after this point, everything is public, and any leakage gives no additional information beyond what is made public in the signature. To analyze our signature implementation, we choose the FvR key scenario, under randomized messages, as proposed in [TG16] for asymmetric cryptosystems. In addition we needed to add artificial wait cycles before accessing the board’s hardware TRNG to ensure that we will never need varying amounts of time for it to become ready, as this would destroy the constant time property required for the TVLA. Note that this change is only necessary for the test setup and not required in the production code.

The sets of side-channel traces are collected by signing a random message using either a fixed key or a random key. As shown in Fig. 4, the $t$-value remains below 6.1 using 100,000 traces which indicates the absence of leakage. Moreover, the maximum $|t|$-value is indeed bounded and has a stable pattern. By contrast, an exploitable leakage as shown in Fig. 1 or Fig. 5 exceeds the threshold value within as few as 2,725 traces and has a clear increasing pattern. Thus we can conclude that the first-order masked Picnic3 implementation provides the expected SCA resistance.

7 Conclusion and Future Work

In this paper we study the side-channel security of MPCitH proof protocols and related signature schemes. We found and demonstrated a new probing attack on the KKW proof protocol (as implemented by Picnic). We then show that masking the signing operations is a practical countermeasure for side-channel attacks, and prove our masked KKW and Picnic3 meet the standard security notion (NIo), with a mix of both manual proofs and formal verification with the maskVerif tool.

We implemented multiple masked versions of the Picnic3 signature scheme for the ARM Cortex M4 as a case study, and found that the cost of masking (in terms of runtime) is high when we simply apply SNI-secure masking to all hashing operations. After careful analysis of the hashing operations, we found that the masking overhead can be made quite reasonable (as low as 1.8x) under modest assumptions that we verified with practical leakage analysis of our implementation. We recommend this version with selective half-masking as the best security-efficiency trade-off among our implementations, and have configured it as the default in our software package. With hardware support for side-channel protected hashing, our work shows that the overhead of masking the non-hashing parts of Picnic signing is about 1.5x, and our NI analysis applies here.

Our flexible masked SHA-3 implementation is the first publicly available one, and will be useful to other projects as SHA-3 becomes more common. We also expect our

---

4The FvR-message scenario with a fixed key is not a good fit for randomized signature schemes like Picnic3 where the entire internal state is randomized even for fixed messages. If the randomness is fixed (by fixing $R$ in Line 1 of Algorithm 4), the signature scheme becomes artificially deterministic. Then several public parts of the signature generation process, including the signature itself, will be picked up by the TVLA test. Similarly, in the deterministic case the root seed is fixed for fixed messages (and random for random ones), also creating leakage which does not exist for randomized signatures. We still performed the analysis and verified that leakage only occurs in such expected places.
half-masking optimization to find application in other implementations, as most hash operations have a non-sensitive input or output.

Performance improvements (while maintaining resistance to side-channel attacks) are an obvious direction for future work, both on the M4 and other embedded platforms. Reducing the amount of randomness consumed by our mitigations is also an interesting way to improve performance, together with generalizing to higher order protection efficiently.

Finally, an implementation that combines SCA resistance and resistance to fault attacks (perhaps leveraging the fault-resistance results for Picnic in [AOTZ20]) would also make interesting follow-up work.

Acknowledgments

We thank Daniel Kales and Sebastian Ramacher for help porting the optimized Picnic implementation to the M4. We thank the anonymous reviewers of TCHES for their detailed comments. This work has been supported by: the German Federal Ministry of Education and Research (BMBF) under the projects PQC4MED (FKZ 16KIS1045) and Jupiter (FKZ 16ME0234); the Concordium Blockchain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Unions’s Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC).

References


[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl,


A Probing the Unopened Party

The second attack uses the revealed values for the online phase i.e. $\hat{w}^{(k)}$ and $\lambda_i^w$ for $i \neq i_k$. This attack is a straightforward variant of the one by Gellersen et al. [GSE20] and Seker et al. [SBWE20], but adapted to work with Picnic3. In contrast to the attack described above, we now target an MPC execution whose online phase is selected to be part of the signature (i.e., if $k \in \mathcal{C}$). In that case there is a single party $P_{i_k}$ whose internal state must remain hidden for the privacy of the MPC protocol to hold. By design, the values $\hat{w}^{(k)}$ and $\lambda_i^w$ for $i \neq i_k$ are revealed during the verification. Thus the measurements have a weak and noisy dependence to the value $\lambda_i^w$ which can be exploitable due to the revealed values.

We validate the attack using the same experimental setup and parameters as in Section 3.1. During the collection phase, we again process the Picnic3 with random messages with a fix secret key and measure the execution of the preprocessing phase (Algorithm 15), where the following is computed:

$$\text{roundkey}_0 = \lambda_{i_k} + \sum_{i \neq i_k} \lambda_i \quad \text{and} \quad \lambda_{sk} = \text{matMul} (\text{roundkey}_0, K_0^{-1}). \quad (1)$$

Since $\hat{s}k = sk + \lambda_{sk}$, we have the following equation for $\lambda_{i_k}$,

$$\hat{s}k = sk + (\lambda_{i_k} + \sum_{i \neq i_k} \lambda_i) K_0^{-1}, \quad \text{and} \quad \lambda_{i_k} = (\hat{s}k - sk) K_0 - \sum_{i \neq i_k} \lambda_i. \quad (2)$$

Finally, we substitute the secret value $\lambda_{i_k}$ into Eq. (1),

$$\text{roundkey}_0 = (\hat{s}k - sk) K_0 - \sum_{i \neq i_k} \lambda_i + \sum_{i \neq i_k} \lambda_i = (sk + \hat{s}k) K_0. \quad (3)$$

From Eq. (3) we observe that $\text{roundkey}_0$ can be probed (over multiple traces) since $sk$ is a constant value. Then $\lambda_{i_k}$ can be calculated as $\text{roundkey}_0 - \sum_{i \neq i_k} \lambda_i$, and used to obtain the secret (as described above). The result of the analysis in Fig. 5 shows a clear dependence between the unrevealed value $\text{roundkey}_0$ and the observable trace, as the $|t|$-value clearly exceeds 5.7 which shows an exploitable leakage.

B Complete Description of the KKW Proof System

In Fig. 6 we present a three-round KKW proof system. We remark that commitments (i.e., generation of $\text{com}^{(k)}_{\text{on}}$ and $\text{com}^{(k)}_{\text{off}}$) are de-randomized and replaced by a hash function as
suggested in [KKW18, §3] (which loses HVZK but is still sufficient for provable security of signature), and the protocol is mildly generalized to work with arithmetic circuits according to [DiOS19, BN20].

**Protocol.** Below we present a basic framework for three-round MPCitH-PP proof systems. Here we describe the protocol for one MPC instance (with non-negligible soundness error) and in Fig. 6 we include a complete description of the KKW proof system that uses many instances in parallel (to achieve negligible soundness error). As the offline protocol proceeds independently of the secret witness, an MPCitH-PP prover can safely open the states of all \( N \) parties for the verification of the preprocessing phase (i.e., triple generation).

**(Commit)** The prover \( P \) first samples a random seed for each \( P_i \) and executes \( \Pi_C^{\text{off}} \) to obtain the states of all \( N \) parties after the offline phase. Then using these states and the masked witness \((\bar{w})_{u \in \mathbb{N}}\) as input, \( P \) executes \( \Pi_C^{\text{off}} \) to obtain all broadcast messages observed during the online phase. Finally, \( P \) sends commitments to the states and broadcast messages to the verifier \( V \).

**(Challenge)** \( V \) asks \( P \) to open either the offline or the online phase. For the latter case, \( V \) also randomly picks a party index \( i^* \), whose view is to remain hidden.

**(Response)** To open the offline phase, \( P \) sends all random seeds used during \( \Pi_C^{\text{off}} \). To open the online phase, \( P \) sends broadcast messages coming from the party \( P_i^* \) during \( \Pi_C^{\text{off}} \), as well as all the state information of the remaining \( N - 1 \) parties.

**(Verification)** To check the offline phase, \( V \) simply uses random seeds to execute \( \Pi_C^{\text{off}} \) as \( P \) would, to obtain the resulting states of all \( N \) parties. Then \( V \) checks that these states form a correct opening to the commitment of the offline phase. To check the online phase, \( V \) simulates \( \Pi_C^{\text{off}} \) with the broadcast messages from \( P_i^* \) and the states of the remaining \( N - 1 \) parties as input, so as to obtain the broadcast messages of the other \( N - 1 \) parties. Then, \( V \) checks that these broadcast messages form a correct opening to the commitments of the online phase.

## C Our Protected Picnic3 Implementation

In this section we give a detailed description of our masked Picnic3 implementation. Algorithm 4 has the top-level signature generation function, that calls the other algorithms in this section. Fig. 7 gives an overview of the optimized hashing operations mentioned in Section 5.2, indicating which optimizations are applied to each one.

**Notation.** \( T \) is the number of shares used by our implementation, and masked values will be \( T \)-encoded. For LowMC, \( n \) is the blocksize, and the precomputed constants \( K_i, L_i \) and \( R_i \) are as defined in Appendix F. The parameter \( N \) is the number of MPC parties, and \( M \) is the number of MPC instances.

**Data Types and Helper functions.**

- \( T \)-encoding: an additive secret sharing in \( GF(2) \). For a bit \( b \), the \( T \)-encoding is a vector of \( T \) values \( b_1, \ldots, b_T \) such that \( b = \sum_{i=1}^{T} b_i \). For a bitstring \( s \), the \( T \)-encoding is \( T \) bitstrings over \( GF(2)^{|s|} \) that XOR to \( s \). As in other parts of the paper we use \( \langle b \rangle \) to indicate that \( b \) is \( T \)-encoded.

- \( \oplus_T \): XOR operation of \( T \)-encoded values. For two \( T \)-encoded inputs \( \langle a \rangle \) and \( \langle b \rangle \), we define \( \langle c \rangle = \langle a \rangle \oplus_T \langle b \rangle \) as \( c_i = a_i \oplus b_i \) for \( i = 1, \ldots, T \); for one \( T \)-encoded input XOR’d by a non-encoded constant \( b \), we define \( \langle c \rangle = \langle a \rangle \oplus_T b \) as \( c_i = a_i \oplus b \) and \( c_i = a_i \) for \( i = 2, \ldots, T \).

- \( \text{SMul} \): AND operation of two \( T \)-encoded values. We use Algorithm 11. For example, with 2-encoded inputs \( \langle a \rangle \) and \( \langle b \rangle \), this algorithm outputs \( \langle c \rangle = \)
Protocol KKW

Inputs Both prover and verifier receive circuit $C$ as a statement. The prover also holds a witness $w = \{w_k\}_{k \in \mathbb{N}}$ such that $C(w) = 1$. Values $M$, $N$, $\tau$ are parameters of the protocol.

Commit 1. For each $k \in [M]$, the prover:
(a) Choose a uniform $\text{seed}^{(k)}$ and use to generate values $\{\text{seed}_{i}^{(k)}\}_{i \in [N]}$. Also the prover computes $\text{aux}^{(k)} \in \mathbb{F}^{[C]}$ by running the offline phase of MPC $\Pi_{\mathbb{F}}^{\text{gen}}$ on input $\{\text{seed}^{(k)}_{i}\}_{i \in [N]}$.
For all $i = 1, \ldots, N - 1$, let $\text{state}^{(k)}_{i} = \text{seed}^{(k)}_{i}$ and let $\text{state}^{(k)}_{N} = \text{seed}^{(k)}_{N}||\text{aux}^{(k)}$.
(b) Commit to the offline phase:
\[
\text{com}_{i}^{(k)} = H(\text{state}_{i}^{(k)}) \text{ for all } i \in [N]
\]
\[
\text{com}^{\text{off}}_{i} = H(\text{com}_{i}^{(k)}, \ldots, \text{com}_{N}^{(k)}).
\]
(c) Compute the masked witness $\hat{w}^{(k)} = \sum_{i \in \mathbb{N}} w + w$ for each $w \in \mathbb{N}$, where each $\lambda_{w}^{i}$ corresponds to party $P_i$’s random share to mask the witness, and is read out from $\text{state}^{(k)}_{i}$.
(d) Simulate the online phase of the $N$-party protocol by running the offline phase of MPC $\Pi_{\mathbb{F}}^{\text{on}}$ on input $(\hat{w}^{(k)})_{w \in \mathbb{N}}$ and $\{\text{state}^{(k)}_{i}\}_{i \in [N]}$, to produce $\{\text{msgs}_{i}^{(k)}\}_{i \in [N]}$.
(e) Commit to the online phase:
\[
\text{com}^{\text{on}}_{i}^{(k)} = H(\{\hat{w}^{(k)}\}_{w \in \mathbb{N}}, \text{msgs}_{1}^{(k)}, \ldots, \text{msgs}_{N}^{(k)}).
\]
2. Compute $h_{\text{eff}} = H(\text{com}^{\text{off}}_{1}, \ldots, \text{com}^{\text{off}}_{M})$ and $h_{\text{on}} = H(\text{com}^{\text{on}}_{1}, \ldots, \text{com}^{\text{on}}_{M})$ and send $h^{\ast} = H(h_{\text{eff}}, h_{\text{on}})$ to the verifier.

Challenge The prover receives the following challenges from the verifier: a uniform $r$-sized set $\mathcal{C} \subset [M]$ and $\mathcal{P} = \{i_{k}\}_{k \in \mathcal{C}}$ where each $i_{k} \in [N]$ is uniform.

Response For each $k \in [M] \setminus \mathcal{C}$, the prover sends $\text{seed}^{(k)}$ and $\text{com}^{\text{off}}_{i}^{(k)}$ for all to the verifier. For each $k \in \mathcal{C}$, the prover sends $\text{com}^{\text{on}}_{i}^{(k)}$, $\{\text{state}^{(k)}_{i}\}_{i \not\in \mathcal{C}}$, $\{\hat{w}^{(k)}\}_{w \in \mathbb{N}}$ and $\text{msgs}_{i}^{(k)}$ to the verifier.

Verification The verifier accepts if and only if all the following checks succeed:
1. Check the offline phase:
(a) For every $k \in \mathcal{C}$ and $i \neq i_{k}$, the verifier uses $\text{state}^{(k)}_{i}$ to compute $\text{com}^{\text{off}}_{i}$ as the prover would. Then produce $\text{com}^{\text{off}}_{i} = H(\text{com}_{i}^{(k)}, \ldots, \text{com}_{N}^{(k)})$ using the received value $\text{com}_{i}^{(k)}$.
(b) For every $k \in [M] \setminus \mathcal{C}$ the verifier uses $\text{seed}^{(k)}$ to compute $\text{com}^{\text{off}}_{i}^{(k)}$ as the prover would.
(c) Finally, the verifier computes $h_{\text{eff}} = H(\text{com}^{\text{off}}_{1}, \ldots, \text{com}^{\text{off}}_{M})$.
2. Check the online phase:
(a) For $k \in \mathcal{C}$ the verifier simulates the online phase using $\{\text{state}^{(k)}_{i}\}_{i \not\in \mathcal{C}}$, masked witness $(\hat{w}^{(k)})_{w \in \mathbb{N}}$ and $\text{msgs}_{i}^{(k)}$ to compute $\{\text{msgs}_{i}^{(k)}\}_{i \not\in \mathcal{C}}$. Then compute $\text{com}^{\text{on}}_{i}^{(k)}$ as if the prover would do.
(b) The verifier computes $h_{\text{on}} = H(\text{com}^{\text{on}}_{1}, \ldots, \text{com}^{\text{on}}_{M})$ using the received $\text{com}^{\text{on}}_{i}^{(k)}$ for $k \in [M] \setminus \mathcal{C}$.
3. The verifier checks that $H(h_{\text{eff}}, h_{\text{on}}) = h^{\ast}$.

Figure 6: 3-round KKW proof system for an arithmetic circuit $C$ defined over $\mathbb{F}$. 
Figure 7: Summary of our masking protections and hashing optimizations from Section 5.2. The figure is fully expanded for one of the $M$ MPC instances, and shows the signer’s operations for the commit phase of the protocol (i.e., before the challenge is computed). Hash functions in green are half-masked hash with sensitive inputs, hash functions in purple are half-masked with sensitive outputs, functions in orange are NI/SNI-secure gadgets, and the white functions are unprotected. The secret key (witness) is denoted $sk$ and $Msg$ is the message to be signed. In the figure, we omit hashing of $(com_1, \ldots, com_N)$ into $com_{\text{off}}$, since the inputs are public values that can be reconstructed from the signature, and therefore the hash computation is unmasked regardless of our optimization.
SMul((a), (b)) as
\[ c_1 = a_1 b_1 + r \]
\[ c_2 = a_2 b_2 + a_1 b_2 + r + a_2 b_1 \]

where \( r \) is a fresh random bit. Note that \( c = c_1 + c_2 = (a_1 + a_0)(b_1 + b_0) = ab \).

- Additional functions: Two additional helper functions from the literature are described in Appendix D. These are for refreshing the randomness of a \( T \)-encoded value and decoding (or unmasking) a \( T \)-encoded value.

\begin{itemize}
  \item \texttt{matMul}_T: This is a generalization of the \texttt{matMul} matrix multiplication function in [Pic20, §6.4.4], modified to work on \( T \)-encoded input vectors. The matrix remains unshared. The input is a \( T \)-encoded vector \( v \) of length \( n \), an \( n \times n \) matrix \( M \), and the output is the length-\( n \) vector \( vM \). If \( T = 2 \), we have \( v = (v_1, v_2) \), the output is \( (v_1 M, v_2 M) = (\text{matMul}(v_1, M), \text{matMul}(v_2, M)) \)
  \item \texttt{tapes}: an object representing the \( N \) random tapes, one per party. In case we need to be explicit about individual per-party tapes, it is parsed as \( \text{tapes} = \text{tape}_1 || \ldots || \text{tape}_N \). Each tape is expanded from a seed masked version of SHAKE that produces \( T \)-encoded outputs, and we store the \( T \)-encoded outputs. We also store a \( T \)-encoded representation of the aux tape, the \( N \)-th party’s share.
  \item \texttt{tapes\_to\_word}(\texttt{tapes}, \texttt{offset}): Read one bit from each of the \( N \) tapes at the index \( \text{offset} \), and output an \( N \)-bit word. When the tapes are \( T \)-encoded, the output word is also \( T \)-encoded.
  \item \texttt{tapes\_to\_parity}(\texttt{tapes}, \texttt{offset}): Reads \( n \) bits from each tape at the index \( \text{offset} \), the strings \( s_1, \ldots, s_N \), returns a \( T \)-encoding of \( \bigoplus_{i=1}^{N} s_i \). Our implementation computes and stores a \( T \)-encoding of the parity tape (i.e., the XOR of all \( N \) tapes) and uses this to implement the \texttt{tapes\_to\_parity} function.
\end{itemize}

C.1 Specification of Fully Masked Picnic3

Algorithm 4 specifies the masked Picnic3 signing operations without the half-masked hashing optimizations that we described in Section 5.2. The notation is as defined elsewhere in the paper, and in the appendix on LowMC (Appendix F). All functions marked in \textcolor{orange}{orange} modify \( T \)-encoded values during the computation. The \texttt{Unmask} function takes a \( T \)-encoded value and returns the non-encoded value, by summing the shares after refresh (see Algorithm 13).

We verified that Algorithm 4 is indeed NI secure with \texttt{maskVerif} for up to second order (implying it is also NIo for any public outputs).

C.2 Simulation of the Offline Phase

Algorithms 5 to 7 describe the protected preprocessing phase. The description is very similar to the preprocessing phase in the specification [Pic20, Section 7.4], however, the data types and helper functions are different. Primarily, all variables are \( T \)-encoded. Algorithm 6 describes our masked version of the LowMC S-box used for preprocessing (called by Algorithm 5), which in turn calls Algorithm 7 the AND operation for the preprocessing phase. Also, our presentation assumes that Algorithm 5 is used for signature generation only, since verification can use an unprotected implementation.

C.3 Simulation of the Online Phase

We now describe how the online phase of the MPC simulation is masked. Algorithm 8 is the MPC simulation for the online phase, implementing the LowMC circuit. For each AND gate, each party \( i \) broadcasts a bit and these are output to \texttt{msgs}_i, these are also
$T$-encoded. In Algorithm 9 we describe the S-box implementation used in Algorithm 8. Finally we have Algorithm 10 that describes the online simulation of an individual AND gate. The broadcast values (written to $\text{msgs}_i$) and output bit are also $T$-encoded. Recall that $\text{SMul}$ is implemented with the ISW multiplier (Algorithm 11).

Note that we need to refresh $T$-encoded $\text{st}$ before each invocation of $\text{masked}\_\text{sbox}\_\text{online}$ (line 6 of Algorithm 8). On one hand, since every round of LowMC involves a linear transformation of $\text{st}$ (line 1 and 7), every bit of $\text{st}$ depends on all $n$ bits of the previous $\text{st}$, which corresponds to a problematic composition pattern mentioned in [BBD + 15b, Diagram 1]. On the other hand, all the other gadgets are in fact characterized as an affine gadget, which can be security composed in an arbitrary fashion. Hence, inserting $\text{RefreshM}$ as we do is necessary and sufficient for the entire construction to be provably NIo secure.

Storage of secret keys. We assume that the Picnic secret key (a bitstring of length $n$), is stored in a $T$-encoded representation. Picnic key pair generation may be modified to generate $T$-encoded secret keys, or an implementation may use regular key generation in a trusted environment (e.g., during device manufacture), and then encode the secret key. As this is not important for performance, our implementation takes the regular key and $T$-encodes it at the beginning of signing. Then the input to MPC simulation is the $T$-encoded value $\langle \lambda^k \rangle = (\lambda^k) \oplus_T \langle sk \rangle$, where $\langle \lambda^k \rangle$ is the $T$-encoded random masks output by preprocessing.
Algorithm 4 masked_Sign

Input: Msg, pk and (sk) for each input wire w to the circuit.

Output: (h, salt, Z)

1: Sample random $R \in \{0, 1\}^n$
2: $(\text{seed}^N), (\text{salt}), \left< \text{sk}, \text{Msg}, pk, \lambda, R \right>$ and salt $\leftarrow$ Unmask((salt))
3: $(\text{Seed_tree}) \leftarrow \text{gen_seed}((\text{seed}^N), \text{salt}, M, 0)$
4: $(\text{Seed}^1), \ldots, (\text{Seed}^M)) \leftarrow \text{get_leaves}((\text{Seed_tree}))$
5: for each $k \in [M]:$
6: $(\text{seed_tree}(k)) \leftarrow \text{gen_seed}((\text{Seed}(k)), \text{salt}, N, k)$
7: $(\text{seed}^N_k), \ldots, (\text{seed}^M_k)) \leftarrow \text{get_leaves}((\text{seed_tree}(k)))$
8: for each $i \in [N]:$
9: $(\text{tape}^{(k)}_i) \leftarrow \text{KDF}((\text{seed}^{(k)}_i), \text{salt}, k, i)$
10: $(\lambda^{(k)} \leftarrow \text{masked_offline}((\text{tape}^{(k)}_i), \ldots, (\text{tape}^{(k)}_N), pk)$ (see Algorithm 5)
11: $(\text{aux}^{(k)}_i) \leftarrow \text{get_aux}((\text{tape}^{(k)}_i))$
12: for each $i \in [N]:$
13: if $i \neq N$ then
14: $(\text{com}^{(k)}_i) \leftarrow H(\langle \text{seed}^{(k)}_i, \text{salt}, k, i \rangle)$
15: else
16: $(\text{com}^{(k)}_i) \leftarrow H(\langle \text{seed}^{(k)}_i, \text{aux}^{(k)}_i, \text{salt}, k, i \rangle)$
17: $(\text{com}^{(k)}_i) \leftarrow \text{Unmask}((\text{com}^{(k)}_i))$
18: $(\text{com_off}^{(k)}_i) \leftarrow H(\text{com}^{(k)}_1, \ldots, \text{com}^{(k)}_N)$
19: $(\text{sk}) \leftarrow \text{RefreshM}(\langle \text{sk} \rangle)$
20: $(\lambda^{(k)} \leftarrow \lambda^{(k)} \oplus \lambda^{(k)}$ (see Algorithm 8)
21: $(\langle \text{msg}^{(k)}_1, \ldots, \text{msg}^{(k)}_N \rangle) \leftarrow \text{masked_online}(\langle \lambda^{(k)}_i, (\text{tape}^{(k)}_i), \ldots, (\text{tape}^{(k)}_N), pk)$ (see Algorithm 8)
22: $(\text{com_on}^{(k)}_i) \leftarrow H(\langle \lambda^{(k)}_i, \langle \text{msg}^{(k)}_1, \ldots, \text{msg}^{(k)}_N \rangle)$
23: $(\text{com_on}^{(k)}_i) \leftarrow \text{Unmask}((\text{com_on}^{(k)}_i))$
24: $(\text{com_on_tree} \leftarrow \text{build_tree}((\text{com_on}^{(k)}_1, \ldots, \text{com_on}^{(k)}_N)$
25: $h \leftarrow H(\text{com_off}^{(k)}, \ldots, \text{com_off}^{(N)}, \text{com_on_tree}, \text{salt}, pk, \text{Msg})$
26: Parse $h$ as $[C, P]$ where $C \subseteq [M]$ and $P = \{i_k \}_{k \in C}, i_k \in [N]$
27: $(\text{com_on_info} \leftarrow \text{open_tree}(\text{com_on_tree}, M, C)$
28: $(\text{Seed_info} \leftarrow \text{reveal_seed}(\text{Seed_tree}, M, C)$
29: $(\text{Seed_info} \leftarrow \text{Unmask}((\text{Seed_info}))$
30: for each $k \in C:$
31: $(\text{seed_info}^{(k)} \leftarrow \text{reveal_seed}((\text{seed_tree}(k))), N, i_k)$
32: $(\text{seed_info}^{(k)} \leftarrow \text{Unmask}((\text{seed_info}^{(k)})), \langle \text{sk}^{(k)} \rangle \leftarrow \text{Unmask}((\text{sk}^{(k)})); \text{msg}^{(k)}_i \leftarrow \text{Unmask}((\text{msg}^{(k)}_i))$
33: if $i_k = N$ then $(\text{aux}^{(k)}_i) \leftarrow \text{Unmask}((\text{aux}^{(k)}))$
34: output $(h, \text{salt}, Z)$ as a signature
Algorithm 5 masked_offline (corresponds to compute_aux in Section 7.4 of [Pic20])

Input: The tapes (tapes). The signer’s public key \( pk = (c, p) \).
Output: The \( n \)-bit key mask \( \lambda^k \). The (tapes) is updated inside masked_sboxaux.

1. \( \text{roundkey}_i^{(i)} \leftarrow \text{tapes_to_parity}_T(n, (\text{tapes}), 0) \)
2. \( \lambda^k \leftarrow \text{matMul}_T((\text{roundkey}_i^{(i)}), K_i^{-1}) \)
3. \( \text{st}(\text{in}) \leftarrow (0^n) \)
4. for each LowMC round \( i \) from \( r \) down to 1
5. \( \text{roundkey}_i^{(i)} \leftarrow \text{matMul}_T(\lambda^k, K_i) \)
6. \( \text{st}(\text{out}) \leftarrow \text{st}(\text{in}) \oplus_T (\text{roundkey}_i^{(i)}) \)
7. \( \text{st}(\text{out}) \leftarrow \text{matMul}_T((\text{st}(\text{out})), L_i^{-1}) \)
8. if \( i = 1 \) then
9. \( \text{st}(\text{in}) \leftarrow (\text{roundkey}_0^{(i)}) \)
10. else
11. \( \text{st}(\text{in}) \leftarrow \text{tapes_to_parity}_T(n, (\text{tapes}), 2n(i - 1)) \)
12. \( \text{offset} \leftarrow 2n(i - 1) + n \)
13. masked_sboxaux((\text{st}(\text{in}), (\text{st}(\text{out})), (\text{tapes}), \text{offset}) \) (see Algorithm 6)
14. return \( \lambda^k \)

Algorithm 6 masked_sboxaux (corresponds to aux_sbox in Section 7.4.1 of [Pic20])

Input: The input and output states (\text{st}(\text{in}) and (\text{st}(\text{out})). The tapes (\text{tapes}). The tape offset \text{offset}.
Output: \text{tapes} is updated with the auxiliary bits.

1. for each \( i \) from 0 to \( 3n \), in steps of 3
2. \( \lambda^k(i) \leftarrow (\text{st}(\text{in})(i+2)) \) / \( T \) bits of \text{st}(\text{in}) at position \( i+2 \), i.e., \( \lambda^k(i) = (\text{st}(\text{in})(i+2), \ldots, \text{st}(\text{in})(i+2)) \)
3. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
4. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
5. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
6. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
7. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
8. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
9. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
10. \( \lambda^k \leftarrow (\text{st}(\text{in})(i+2)) \)
11. masked_ANDaux((\lambda^k(i), (\lambda^k(i), (\lambda^k(i), (\text{tapes}, \text{offset} + i + 2))
12. masked_ANDaux((\lambda^k(i), (\lambda^k(i), (\lambda^k(i), (\text{tapes}, \text{offset} + i + 2))
13. masked_ANDaux((\lambda^k(i), (\lambda^k(i), (\lambda^k(i), (\text{tapes}, \text{offset} + i + 2))

Algorithm 7 masked_ANDaux (corresponds to aux_AND in [Pic20, §7.4.2])

Input: The input mask bits \( \lambda^x \) and \( \lambda^y \). The fresh output mask bit \( \lambda^z \). The taps (\text{tapes}). The tape offset \text{offset}.
Output: The function updates (\text{tape}_n).

1. \( \text{and_helper} \leftarrow \text{tapes_to_parity}_T(1, (\text{tape}_i)[i]|w|,...|(\text{tape}_{n-1}), \text{offset}) \)
2. \( \lambda^y \leftarrow \text{SMul}(\lambda^x), (\lambda^y) \)
3. \( \text{aux_bits} \leftarrow \lambda^y \oplus_T \text{and_helper} \oplus_T (\lambda^y) \)
4. \( \text{tape}_n(\text{offset}) \leftarrow (\text{aux_bits}) \) / Ensuring \( \lambda^y \oplus \lambda^z = \text{tape}_n(\text{offset}) \oplus_T \ldots \oplus_T \text{tape}_n(\text{offset}) \)
Algorithm 8 masked_online (corresponds to \texttt{mpc	extunderscore simulate} in Section 7.5 of [Pic20])

\textbf{Input:} The masked input \((\hat{s}, \hat{\phi})\). The tapes \(\text{\texttt{tapes}}\). The signer’s public key \(pk = (c, p)\).

\textbf{Output:} The broadcast messages \(\langle \text{msg}_1, \ldots, \text{msg}_N \rangle\)

1. \(\text{\texttt{roundkey}}(\hat{\phi}) \leftarrow \text{matMul}_T((\hat{s}, K_0))\)
2. \(\langle \hat{s} \rangle \leftarrow \text{\texttt{roundkey}}(\hat{\phi}) \oplus \tau p\)
3. \(\langle \hat{s} \rangle \leftarrow \text{RefreshM}(\langle \hat{s} \rangle)\)
4. Initialize empty arrays \(\langle \text{msg}_1, \ldots, \text{msg}_N \rangle\)
5. For each LowMC round \(i\) from 1 to \(r\)
   6. \(\text{\texttt{masked	extunderscore sboxonline}}(\langle \hat{s} \rangle, \langle \text{\texttt{tapes}} \rangle, \langle \text{\texttt{msg}_1} \rangle, \ldots, \langle \text{\texttt{msg}_N} \rangle, 2n(i - 1))\) (see Algorithm 9)
   7. \(\langle \hat{s} \rangle \leftarrow \text{matMul}_T((\hat{s}, L_i))\)
   8. \(\langle \hat{s} \rangle \leftarrow \langle \hat{s} \rangle \oplus \tau T\)
   9. \(\text{\texttt{roundkey}}(\hat{\phi}) \leftarrow \text{matMul}_T((\hat{s}, K_i))\)
10. \(\langle \hat{s} \rangle \leftarrow \langle \hat{s} \rangle \oplus \tau T \left(\text{\texttt{roundkey}}(\hat{\phi})\right)\)
11. \(\langle \hat{s} \rangle \leftarrow \text{RefreshM}(\langle \hat{s} \rangle)\)
12. Compare \(st = \text{Unmask}(\langle \hat{s} \rangle)\) and \(c\) component of \(pk\). If they differ, fail.
13. \textbf{return} \(\langle \text{msg}_1, \ldots, \text{msg}_N \rangle\)

Algorithm 9 masked_sboxonline (corresponds to \texttt{mpc	extunderscore sbox3} in Section 7.5.1 of [Pic20])

\textbf{Input:} A \(T\)-encoding of \(n\)-bit LowMC state \(\langle \hat{s} \rangle\). The tapes \(\text{\texttt{tapes}}\). The broadcast message holder \(\langle \text{msg}_1, \ldots, \text{msg}_N \rangle\). The tape offset \(\text{offset}\).

\textbf{Output:} \(\langle \text{msg}_i \rangle\) is updated with the broadcast messages of party \(i\)

1. For each \(i\) from 0 to \(3s\), in steps of 3
   2. \(\langle \hat{s} \rangle \leftarrow \langle \hat{s} \rangle[i + 2]\)
   3. \(\langle \hat{s} \rangle \leftarrow \langle \hat{s} \rangle[i + 1]\)
   4. \(\langle \hat{s} \rangle \leftarrow \langle \hat{s} \rangle[i]\)
   5. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
   6. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
   7. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
   8. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
   9. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
10. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
11. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
12. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
13. \(\langle \text{\texttt{tapes}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)

Algorithm 10 masked_ANDonline (corresponds to \texttt{mpc	extunderscore and3} in Section 7.5.2 of [Pic20])

\textbf{Input:} A \(T\)-encoding of two masked input bits \((\hat{x}, \hat{y})\). The masking bits \((\langle \text{\texttt{lambda}}_1 \rangle, \ldots, \langle \text{\texttt{lambda}}_N \rangle)\) and \((\langle \text{\texttt{lambda}}'_1 \rangle, \ldots, \langle \text{\texttt{lambda}}'_N \rangle)\). The tapes \(\text{\texttt{tapes}}\). The message holder \(\langle \text{msg}_1, \ldots, \text{msg}_N \rangle\). The tape offset \(\text{offset}\).

\textbf{Output:} Masked AND output \((\langle \hat{x}, \hat{y} \rangle)\). The function updates \(\langle \text{msg}_i \rangle\).

// and_helper contains party \(i\)'s share of \(\langle \text{\texttt{lambda}}_i \rangle \oplus \tau T\)

1. \(\langle \text{\texttt{and	extunderscore helper}} \rangle, \ldots, \langle \text{\texttt{and	extunderscore helper}} \rangle \leftarrow \text{\texttt{tapes}} \oplus \tau T\)
2. For each \(i\) in \(\langle N \rangle\)
   3. \(\langle a_i \rangle \leftarrow \text{SMul}(\langle \hat{x} \rangle, \langle \text{\texttt{lambda}}_i \rangle)\)
   4. \(\langle b_i \rangle \leftarrow \text{SMul}(\langle \hat{y} \rangle, \langle \text{\texttt{lambda}}'_i \rangle)\)
   5. \(\langle s_i \rangle \leftarrow \langle a_i \rangle \oplus \tau T \langle b_i \rangle \oplus \tau T \langle \text{\texttt{and	extunderscore helper}} \rangle\)
   6. Append \(\langle s_i \rangle\) to \(\langle \text{msg}_i \rangle\)
7. \(\langle c \rangle \leftarrow \text{SMul}(\langle \hat{x} \rangle, \langle \hat{y} \rangle)\)
8. \(\langle s \rangle \leftarrow \sum_{i \in \langle N \rangle} \langle s_i \rangle\)
9. \(\langle \hat{x}, \hat{y} \rangle \leftarrow \langle c \rangle \oplus \tau T \langle s \rangle\)
10. \textbf{return} \(\langle \hat{x}, \hat{y} \rangle\)
D Additional Gadgets

Algorithm 11 SMul [ISW03, Theorem 1]

Require: The encodings \((x_1, \ldots, x_T)\) and \((y_1, \ldots, y_T)\).
Output: The encoding of \(xy\) as \((z_1, \ldots, z_T)\).

1: for \(1 \leq i \leq T\)
2: \(z_i \leftarrow x_i y_i\)
3: for \(1 \leq i \leq T\)
4:     for \(i < j \leq T\)
5:         \(r_{i,j} \leftarrow \text{rand}() /\!/ \text{Not from the random tapes}\)
6:     \(z_i \leftarrow z_i + r_{i,j} /\!/ \text{Denoted by } z_{i,j}\)
7:     \(r_{j,i} \leftarrow (x_i y_j - r_{i,j}) + x_j y_i\)
8:     \(z_j \leftarrow z_j + r_{j,i} /\!/ \text{Denoted by } z_{j,i}\)
9: return \((z_1, \ldots, z_T)\)

Algorithm 12 RefreshM [BBD\textsuperscript{+}16]

Require: The encoding \((x_1, \ldots, x_T)\).
Output: The encoding \((y_1, \ldots, y_T)\) such that \(y_1 + \ldots + y_T = x_1 + \ldots + x_T\)

1: for \(1 \leq i \leq T\)
2: \(y_i \leftarrow x_i\)
3: for \(1 \leq i \leq T\)
4:     for \(i < j \leq T\)
5:         \(r_{i,j} \leftarrow \text{rand}() /\!/ \text{Not from the random tapes}\)
6:     \(y_i \leftarrow y_i + r_{i,j}\)
7: \(y_j \leftarrow y_j - r_{i,j}\)
8: return \((y_1, \ldots, y_T)\)

Algorithm 13 Unmask [BBE\textsuperscript{+}19]

Require: The encoding \((x_1, \ldots, x_T)\).
Output: The shared value \(x\) such that \(x = x_1 + \ldots + x_T\)

1: \((x'_1, \ldots, x'_T) \leftarrow \text{RefreshM}(x_1, \ldots, x_T)\)
2: \(x \leftarrow x'_1\)
3: for \(2 \leq i \leq T\)
4: \(x \leftarrow x + x_i\)
5: return \(x\)

E Specification of Unprotected Picnic3

For completeness, we include the full specifications of Picnic3 signing adapted from [Pic20]. The notation is as defined elsewhere in the paper, and in the appendix on LowMC (Appendix F).
Algorithm 14 Sign

Input: $Msg, pk$ and $sk$ for each input wire $w$ to the circuit.

Output: $(h, salt, Z)$

1. Sample random $R \in \{0, 1\}^{2w} // \text{derive root and initial seeds}$
2. $(seed^*, salt) \leftarrow \text{KDF}(sk, Msg, pk, \lambda, R)$
3. $iSeed\_\text{tree} \leftarrow \text{gen\_seed}(seed^*, salt, M, 0)$
4. $(iSeed^{(1)}, \ldots, iSeed^{(M)}) \leftarrow \text{get\_leaves}(iSeed\_\text{tree})$
5. for each $k \in [M]$
6. $seed\_\text{tree}^{(k)} \leftarrow \text{gen\_seed}(iSeed^{(k)}, salt, N, k) // \text{Derive random tapes from the initial seed}$
7. $(seed^{(k)}, \ldots, seed^{(k)}_{N}) \leftarrow \text{get\_leaves}(seed\_\text{tree}^{(k)})$
8. for each $i \in [N]$
9. $tape^{(k)}_{i} \leftarrow \text{KDF}(seed^{(k)}, salt, k, i)$
10. $\lambda^{(k)} \leftarrow \text{offline}(tape^{(k)}_{1}||\ldots||tape^{(k)}_{N}, pk)$ (see Algorithm 15)
11. $aux^{(k)} \leftarrow \text{get\_aux}(tape^{(k)}_{N})$
12. for each $i \in [N]$
13. if $i \neq N$ then
14. $com^{(i)}_{(k)} \leftarrow H(seed^{(k)}, salt, k, i)$
15. else
16. $com^{(i)}_{(k)} \leftarrow H(seed^{(k)}, aux^{(k)}, salt, k, i)$
17. $com\_\text{off}^{(k)} \leftarrow \text{H}(com^{(1)}_{(k)}, \ldots, com^{(N)}_{(k)}) // \text{Commit to preprocessing phase}$
18. $sk^{(k)} \leftarrow sk \oplus \lambda^{(k)} // \text{Mask input bits}$
19. $msgs_{i}^{(k)} \rightarrow \text{online}(sk^{(k)}, tape^{(k)}_{1}||\ldots||tape^{(k)}_{N}, pk)$ (see Algorithm 18)
20. $com\_\text{on}^{(k)} \leftarrow H(sk^{(k)}, aux^{(k)}_{(k)}, \ldots, msgs^{(k)}_{N}) // \text{Commit to MPC online phase}$
21. $com\_\text{off}^{(k)} \leftarrow \text{build\_tree}(com\_\text{on}^{(1)}, \ldots, com\_\text{on}^{(M)})$
22. $h \leftarrow \text{H}(com\_\text{off}^{(1)}, \ldots, com\_\text{off}^{(M)}, com\_\text{off}^{(\text{root}), com\_\text{on\_\text{tree}}, salt, pk, Msg})$
23. Parse $h$ as $(C, P)$ where $C \subseteq [M]$ and $P = \{i_{k} \}_{k \in C}$, $i_{k} \in [N]$
24. $com\_\text{on\_\text{info}} \leftarrow \text{open\_\text{tree}}(com\_\text{on\_\text{tree}}, M, C) // \text{Include only com\_\text{on}^{(k)} for } k \not\in C$
25. $iSeed\_\text{info} \leftarrow \text{reveal\_\text{seed}}(iSeed\_\text{tree}, M, C) // \text{Include only iSeed^{(k)} for } k \not\in C$
26. for each $k \in C : // \text{Reveal online phases selected by challenge}$
27. $seed\_\text{info}^{(k)} \leftarrow \text{reveal\_\text{seed}}(seed\_\text{tree}^{(k)}, N, i_{k}) // \text{Include only seed^{(k)} for } i \neq i_{k}$
28. if $i_{k} = N$ then $aux^{(k)} \leftarrow \bot$
29. let $Z = (com\_\text{on\_\text{info}}, iSeed\_\text{info}, (seed\_\text{info}^{(k)}, aux^{(k)}, sk^{(k)}, com^{(k)}_{i}, msgs^{(k)}_{i})_{k \in C}$
30. output $(h, salt, Z)$ as a signature
Algorithm 15 offline

Input: The tapes (tapes). The signer’s public key \( pk = (c, p) \).
Output: The \( n \)-bit key mask \( \lambda^{|x|} \). The tapes is updated inside \texttt{sbox}_{aux}.

1. roundkey_\(i\) \(\leftarrow\) tapes_to_parity(\(n\), tapes, 0)
2. \( \lambda^a \leftarrow \) matMul(roundkey_\(i\), \(K_1^{-1}\))
3. st_in \(\leftarrow\) 0^n
4. for each LowMC round \( i \) from \( r \) down to 1
   5. roundkey_\(i\) \(\leftarrow\) matMul(\(\lambda^a\), \(K_1\))
   6. st_out \(\leftarrow\) st_in \(\oplus\) roundkey_\(i\)
   7. st_out \(\leftarrow\) matMul(st_out, \(L_i^{-1}\))
   8. if \( i = 1 \) then
      9. st_in \(\leftarrow\) roundkey_\(0\)
   10. else
      11. st_in \(\leftarrow\) tapes_to_parity(\(n\), tapes, 2n(\(i - 1\)) + n)
      12. offset \(\leftarrow\) 2n(\(i - 1\)) + n
13. \texttt{sbox}_{aux}(st_in, st_out, tapes, offset) (see Algorithm 16)
14. return \( \lambda^a \)

Algorithm 16 \texttt{sbox}_{aux}

Input: The input and output states st_in and st_out. The tapes tapes. The tape offset offset.
Output: tapes is updated with the auxiliary bits.

1. for each \( i \) from 0 to \( 3n \), in steps of 3
   2. \( \lambda^a \leftarrow \) st_in[\(i\)] + 2
   3. \( \lambda^b \leftarrow \) st_in[\(i\)]
   4. \( \lambda^c \leftarrow \) st_in[\(i\)]
   5. \( \lambda^d \leftarrow \) st_out[\(i\)] + 2
   6. \( \lambda^e \leftarrow \) st_out[\(i\)] + 1
   7. \( \lambda^f \leftarrow \) st_out[\(i\)]
   8. \( \lambda^{aux} \leftarrow \lambda^d \oplus \lambda^b \)
   9. \( \lambda^{aux} \leftarrow \lambda^c \oplus \lambda^e \oplus \lambda^b \)
10. \( \lambda^{aux} \leftarrow \lambda^f \oplus \lambda^d \oplus \lambda^b \oplus \lambda^c \)
11. \( \texttt{AND}_{aux}(\lambda^a, \lambda^a, \lambda^{aux}, tape_1, \ldots, tape_{n'}, offset + i + 1) \)
12. \( \texttt{AND}_{aux}(\lambda^a, \lambda^a, \lambda^{aux}, tape_1, \ldots, tape_{n'}, offset + i + 1) \)
13. \( \texttt{AND}_{aux}(\lambda^a, \lambda^a, \lambda^{aux}, tape_1, \ldots, tape_{n'}, offset + i + 1) \) (see Algorithm 17)

Algorithm 17 \texttt{AND}_{aux}

Input: The input mask bits \( \lambda^a \) and \( \lambda^b \). The fresh output mask bit \( \lambda^c \). The tapes tapes. The tape offset offset.
Output: The function updates tape_\(n\).

1. \( \texttt{and\_helper} \leftarrow \) tapes_to_parity(1, tape_\(1\), \ldots, tape_\(N\), offset)
2. \( \lambda' \leftarrow \lambda^a \land \lambda^b \)
3. aux_bit \(\leftarrow\) \(\lambda'^a \oplus\) \texttt{and\_helper} \(\oplus\) \(\lambda^c\)
4. tape_\(N\)(offset) \(\leftarrow\) aux_bit // Ensuring \( \lambda'^a \oplus \lambda' = \) tape_\(1\)(offset) \(\oplus\) \ldots \(\oplus\) tape_\(N\)(offset)
AND gates (low multiplicative complexity). In this work we assume the LowMC instances

\begin{algorithm}
\caption{Algorithm 18 \texttt{online}}
\textbf{Input}: The masked input $\hat{sk}$. The tapes $\text{tapes}$. The signer’s public key $pk = (c, p)$.
\textbf{Output}: The broadcast messages $\text{msgs}_1, \ldots, \text{msgs}_N$
1: roundkey$_0 \leftarrow \text{matMul}(\hat{sk}, K_0)$
2: $st \leftarrow \text{roundkey}_0 \oplus p$
3: Initialize empty arrays $\text{msgs}_1, \ldots, \text{msgs}_N$
4: for each LowMC round $i$ from 1 to $r$
5: \hspace{1em} $\text{sbox}_\text{online}(st, \text{tapes}, \text{msgs}_1, \ldots, \text{msgs}_N, 2n(i - 1))$ (see Algorithm 19)
6: \hspace{1em} $st \leftarrow \text{matMul}(st, L_i)$
7: \hspace{1em} $st \leftarrow st \oplus R_i$
8: \hspace{1em} roundkey$_i \leftarrow \text{matMul}(\hat{sk}, K_i)$
9: \hspace{1em} $st \leftarrow st \oplus \text{roundkey}_i$
10: Compare $st$ and $c$ component of $pk$. If they differ, fail.
11: \textbf{return} $\text{msgs}_1, \ldots, \text{msgs}_N$
\end{algorithm}

\begin{algorithm}
\caption{Algorithm 19 $\text{sbox}_\text{online}$}
\textbf{Input}: The masked state $st$. The tapes $\text{tapes}$. The message holder $\text{msgs}_1, \ldots, \text{msgs}_N$. The tape offset $\text{offset}$.
\textbf{Output}: The function updates $\text{msgs}$.
1: \textbf{for} each $i$ from 0 to $3s$, in steps of 3
2: \hspace{1em} $\hat{a} \leftarrow st[i+2]$
3: \hspace{1em} $\hat{b} \leftarrow st[i+1]$
4: \hspace{1em} $\hat{c} \leftarrow st[i]$
5: \hspace{1em} $(\lambda_1^x, \ldots, \lambda_N^x) \leftarrow \text{tapes}_\text{to}_\text{word}(\text{tapes}, \text{offset} + i + 1)$
6: \hspace{1em} $(\lambda_1^y, \ldots, \lambda_N^y) \leftarrow \text{tapes}_\text{to}_\text{word}(\text{tapes}, \text{offset} + i + 1)$
7: \hspace{1em} $(\lambda_1^z, \ldots, \lambda_N^z) \leftarrow \text{tapes}_\text{to}_\text{word}(\text{tapes}, \text{offset} + i)$
8: \hspace{1em} $bc \leftarrow \text{AND}_\text{online}(b, c, (\lambda_1^y, \lambda_1^z, \text{tapes}, \text{msgs}_1)_\in [N], \text{offset} + n + i + 2)$
9: \hspace{1em} $\widehat{\alpha} \leftarrow \text{AND}_\text{online}(\hat{c}, \hat{a}, (\lambda_1^y, \lambda_1^z, \text{tapes}, \text{msgs}_1)_\in [N], \text{offset} + n + i + 1)$
10: \hspace{1em} $\widehat{\alpha}bc \leftarrow \text{AND}_\text{online}(\hat{a}, b, (\lambda_1^y, \lambda_1^z, \text{tapes}, \text{msgs}_1)_\in [N], \text{offset} + n + i)$
11: \hspace{1em} $st[i+2] \leftarrow \hat{a} \oplus bc$
12: \hspace{1em} $st[i+1] \leftarrow \hat{a} \oplus b \oplus \widehat{\alpha}$
13: \hspace{1em} $st[i] \leftarrow \hat{a} \oplus b \oplus \widehat{\alpha} \oplus \hat{b}$
\end{algorithm}

\begin{algorithm}
\caption{Algorithm 20 \texttt{AND}\_online}$
\textbf{Input}: The masked input bits $\hat{x}$ and $\hat{y}$. The masking bits words $(\lambda_1^x, \ldots, \lambda_N^x)$ and $(\lambda_1^y, \ldots, \lambda_N^y)$. The tapes $\text{tapes}$. The message holder $\text{msgs}_1, \ldots, \text{msgs}_N$. The tape offset $\text{offset}$.
\textbf{Output}: Masked AND output $\widehat{xy}$. The function updates $\text{msgs}$.
\hspace{1em} // and\_helper contains party $i$’s share of $\lambda^x \oplus \lambda^y$
1: \hspace{1em} $(\text{and\_helper}_1, \ldots, \text{and\_helper}_N) \leftarrow \text{tapes}_\text{to}_\text{word}(\text{tapes}, \text{offset})$
2: \hspace{1em} for each $i \in [N]$
3: \hspace{2em} $s_i \leftarrow (\hat{x} \land \lambda_i^y) \oplus (\hat{y} \land \lambda_i^x) \oplus \text{and\_helper}_i$
4: \hspace{1em} Append $s_i$ to $\text{msgs}_i$
5: \hspace{1em} $\widehat{xy} \leftarrow \text{parity}(s_1, \ldots, s_N) \oplus (\hat{x} \land \hat{y})$
6: \textbf{return} $\widehat{xy}$
\end{algorithm}

\section{LowMC}
LowMC [ARS$^+$15] is a parameterizable block cipher designed to have a small number of AND gates (low multiplicative complexity). In this work we assume the LowMC instances are those from the Picnic3 design; however, our analysis and countermeasures generalize easily to other choices of LowMC parameters.

Let $n$ be the block size and key size, $s$ be the number of S-boxes per round, and $r$ the number of rounds. For each LowMC instance, the spec defines random and independent

- round constants $R_i \in \mathbb{F}_2^n$,
- linear layer matrices $L_i \in \mathbb{F}_2^{n \times n}$ (of full rank), and
• key matrices $K_i \in \mathbb{F}_2^{n \times n}$ for the computation of round keys.

There are $r$ round and linear layer constants $R_i$, $L_i$, and $r+1$ key matrices $K_i$. The matrices are invertible, and the Picnic3 implementation uses the inverses $K_0^{-1}$ and $L_r^{-1}$.

LowMC keys are sampled uniformly at random from $\mathbb{F}_2^n$.

LowMC encryption starts by adding the first round key to the plaintext, which is followed by $r$ rounds. Each round key is generated by multiplying the key with the key matrix $K_i$. A single round of LowMC is composed of an S-box layer, a linear layer, addition with constants, and addition of the round key as shown in Algorithm 21. The S-box layer applies the same 3-bit S-box on the first $3 \cdot s$ bits of the state. The S-box is defined as $S(a, b, c) = (a \oplus bc, a \oplus b \oplus ac, a \oplus b \oplus c \oplus ab)$. The other layers only consist of $\mathbb{F}_2$-vector space arithmetic, all local operations in our MPC setting.

Algorithm 21 LowMC encryption. Parameters $K_i$, $L_i$ and $R_i$ are as described in the text.

Require: plaintext $p \in \mathbb{F}_2^n$ and key $k \in \mathbb{F}_2^n$

\[
\begin{align*}
&\text{for } i \in [1, r], \\
&\text{st} \leftarrow K_0 \cdot k \oplus p, \\
&\text{st} \leftarrow SBOXLAYER(st), \\
&\text{st} \leftarrow L_i \cdot st // \text{LINEARLAYER}, \\
&\text{st} \leftarrow R_i \oplus st // \text{CONSTANTADDITION}, \\
&\text{return } st
\end{align*}
\]

G Additional Background on Security Notions for Masking Countermeasures

In the following, we fix some finite field $\langle \mathbb{F}, 0, 1, +, \cdot, \rangle$. As explained above, we are working in the $t$-probing model which allows an attacker to obtain the value of $t$ variables per run of the primitive. The most common technique to mitigate side-channel attacks is by encoding sensitive variables via an additive (or polynomial-based) secret sharing into $T > t$ parts. We say that a vector $(v_j)_{j \in [T]} \in \mathbb{F}^T$ is a $T$-encoding of $v := \sum_{j \in [T]} v_j$. For readability, we often write $(v)$ instead of $(v_j)_{j \in [T]}$. For a subset $I \subseteq [T]$, let $(v)_I = (v_i)_{i \in I}$ and furthermore $T = [T] \setminus I$. Variables are shared both to protect against side-channel attacks and as part of the MPC protocol. To distinguish between these situations, we call a sharing between parties in the MPC protocol a sharing and an encoding when the goal is to protect against side-channel attacks.

Without loss of generality, we only give the security definitions for circuits that receive a single encoded input $(x)$ and produce a single encoded output $(y)$. In the following, we use the terms circuit and gadget interchangeably. Consider a (possibly randomized) gadget $G$, which on input $x$ produces a value $y$ according to some probability distribution $G_x$. To ensure that the computation of $G$ does not leak any information, we modify it into a gadget $G'$ that takes $(x)$ as input and outputs $(y)$ with $Pr[G'_x = y] = Pr[G_x = y]$. Informally, we want to argue that the $t$ probes made by an attacker do not reveal any information about the sensitive input $x$. Assume that the attacker probes the values $(v^{(1)})_{I_1}, (v^{(2)})_{I_2}, \ldots, (v^{(k)})_{I_k}$ with $\sum_{j=1}^k |I_j| \leq t$. If $(x)$ is a sufficiently random encoding (e.g. uniformly sampled) of $x$, then $(x)_I$ does not reveal any information about $x$ for all $I \subseteq [T]$. Now, if there exists a set $I \subseteq [T]$ such that all $(v^{(j)})_{I_j}$ can be simulated from $(x)_I$, this implies that the $(v^{(j)})_{I_j}$ do not contain any information about $x$.

\[\text{\footnotesize (While ISW-style masking countermeasures are known to work in a ring setting, we focus on a field case since previous KKW-style protocols [KKW18, dDOS19, BN20, KZ20] are defined for circuits over a field.)}\]
To formalize this intuition, we consider a distribution ensemble \( \{D(x)\}_{x \in \mathbb{F}^t} \). This ensemble is a probability distribution on \( v \in \mathbb{F}^t \), capturing the probed variables of the attacker. We say that \( \{D(x)\}_{x \in \mathbb{F}^t} \) is perfectly simulatable from the indices \( I \subseteq [T] \), if there is a probabilistic algorithm \( S \) that, on input \( \langle x \rangle_I \), has output distribution exactly \( D(x) \), i.e., \( \Pr[S(\langle x \rangle_I) = v] = \Pr[D(x) = v] \) for every \( \langle x \rangle \).

**Example 1.** The simplest example for this notion of simulatability concerns projections. For example, if the ensemble \( \{D(x)\}_{x \in \mathbb{F}^2} \) is defined on the value \( y = x_1 \), it can easily be simulated from \( I = \{ 1 \} \), as the knowledge of \( x_1 \) is sufficient to simulate \( y \).

**Example 2.** Consider the following distribution \( \{D(x)\}_{x \in \mathbb{F}^2} \) on the triple of values \( (y_1, y_2, t) \) with \( t = x_1 + r, \ y_1 = t \cdot x_2 = (x_1 + r) \cdot x_2 \), and \( y_2 = x_2 \). Here, \( r \) is uniformly sampled from \( \mathbb{F} \), independent from \( \langle x \rangle \). This distribution ensemble is perfectly simulatable from \( I = \{ 2 \} \): The value of \( t \) can simply be simulated by sampling a random element from \( \mathbb{F} \) without any knowledge on \( x_1 \). Now, both \( y_2 \) and \( y_1 \) can be simulated by using the knowledge about \( x_2 \) and the already simulated value \( t \).

### G.1 Non-interference, Strong Non-interference, and Public Outputs

The most basic security notion for a masking countermeasure is the \( t \)-privacy of a gadget \( G \) [ISW03]. Informally, this means that the information provided by \( t \) probes of outputs or intermediate variables can also be obtained by probing \( t \) input variables, as long as the inputs are an encoding of \( x \). While the idea behind the notion is relatively simple, it is unfortunately not composable as the output of a \( t \)-private gadget is not necessarily a truly uniform encoding. The composition of two \( t \)-private gadgets is thus not necessarily \( t \)-private. In order to remove the requirement that the inputs have a certain distribution, the notion of non-interference was introduced [BBD+15a].

Non-interference gets rid of the dependency of the uniformly encoded inputs, but a more subtle issue still prevents a composability result. To give an intuitive overview of this problem, consider a gadget \( G \) with two sensitive inputs \( \langle x \rangle \) and \( \langle x' \rangle \) and sensitive output \( \langle y \rangle \). Non-interference now implies that for any \( I_y \subseteq [T] \), the values \( \langle y \rangle_{I_y} \) can be simulated from \( \langle x \rangle_{I_x} \) and from \( \langle x' \rangle_{I_{x'}} \), for two sets \( I_x, I_{x'} \) of cardinality at most \( |I_y| \). Now, if \( G \) is used in another circuit, it might be the case that \( x \) and \( x' \) are correlated (or even identical). Then \( \langle x \rangle_{I_x \cup I_{x'}} \) might reveal information about \( x \). See e.g., [BBD+16] for a more detailed explanation. Hence, an even stronger notion, called strong non-interference was introduced [BBD+16], that guarantees a clear separation between input variables and output variables.

**Definition 2 \((t\text{-NI}, t\text{-SNI})\).** Let \( G \) be a gadget with inputs in \( \mathbb{F}^T \) and \( t < T \). Suppose that for any set of \( t_1 \) intermediate variables and any subset of \( O \) of output indices with \( t_1 + |O| \leq t \), there exists a subset of indices \( I \) such that the output distribution of the \( t_1 \) intermediate variables and the output variables \( y_O \) is perfectly simulatable from \( I \). Then

- (i) if \(|I| \leq t_1 + |O| \) we say \( G \) is \( t \text{-non-interfering (t-NI)} \), and
- (ii) if \(|I| \leq t_1 \) we say \( G \) is \( t \text{-strong-non-interfering (t-SNI)} \).

Note that linear operations which can be performed share-wise (such as addition or multiplication by a constant) are trivially \( t \text{-NI} \), as each computation on share \( i \) can be simulated from the input share \( i \).

Note that in SNI the size of the input set \( I \) depends only on the intermediate variables. For a given set \( \text{Int} \) of \( t_1 \) intermediate variables and a subset \( O \) of output indices with \( t_1 + |O| \leq t \), we say that the output variables \( O' \subseteq O \) that are simulatable without any knowledge about \( I \) are (\( \text{Int}, O, t \)-input-ignorant). Hence, if no intermediate variable was probed, the output of the circuit is independent of its input (from observing at most \( t \) positions) and thus all subsets of at most \( t \) outputs are input-ignorant. We will occasionally
talk about the concrete distribution of these input-ignorant variables. If all (Int, O, t)-inputignorant output variables of a t-SNI gadget $G$ are distributed according to a distribution $D$, we say that $G$ is t-strong-non-interfering (t-SNI) with output-distribution $D$.

Finally, the SNI notion guarantees that the composition of two t-SNI gadgets is t-SNI again. In Appendix D we recall two t-SNI-secure gadgets $\text{SMul}$ and $\text{RefreshM}$ that we use as building blocks of our masked KKW proof system. For the sake of completeness, we repeat the corresponding proposition from [BBD⁺16].

**Lemma 3** (Proposition 4 [BBD⁺16]). Let $C$ be a circuit built from gadgets $G_1, \ldots, G_r$ such that all $G_i$ are t-NI, all of the inputs and outputs are sensitive. To also handle public, non-sensitive values, variables $\lambda$ are refreshed through $\text{RefreshM}$ before output. Then $C$ is t-NI. Moreover, $C$ is t-SNI if it is t-NI and all encodings corresponding to the outputs of $C$ are refreshed through $\text{RefreshM}$ before output.

The commonly used term probing-security can either mean privacy [BBC⁺19] or non-interference [CGPZ16]. Classically, the non-interference notions only deal with gadgets where all of the inputs and outputs are sensitive. To also handle public, non-sensitive values, the notion of NI with public output (t-NIo) was proposed in [BBE⁺18]. As mentioned in [BBE⁺18, Lemma 1], if a gadget $G$ is t-NI secure it is also t-NIo secure for any public outputs. Clearly, the same claim also holds for t-SNI and t-SNio. While the KKW-protocol also contains public variables, we are able to show the stronger guarantee of t-(S)NI.

## H Omitted Proofs

In this section we give formal security proofs for the SNI security of Algorithm 22 and 23.

### Algorithm 22 Masked KKW·MUL (offline)

| Input: | The masks of $x$: $\langle \lambda_x^i \rangle$ for $i \in [N]$. |
|        | The masks of $y$: $\langle \lambda_y^i \rangle$ for $i \in [N]$. |
|        | The auxiliary shares of $xy$: $\langle \lambda_{xy}^i \rangle$ for $i \in [N-1]$. |
| Output: | The final auxiliary share of $xy$: $\langle \lambda_{xy}^N \rangle$. |

1. compute $\langle \lambda_x^i \rangle = \sum_{i\in[N]} \langle \lambda_x^i \rangle$ 
2. compute $\langle \lambda_y^i \rangle = \sum_{i\in[N]} \langle \lambda_y^i \rangle$ 
3. compute $\langle \lambda_{xy}^i \rangle \leftarrow \text{SMul}(\langle \lambda_x^i \rangle, \langle \lambda_y^i \rangle)$ 
4. compute $\langle \lambda_{xy}^N \rangle = (\langle \lambda_{xy}^i \rangle - \sum_{i\in[N-1]} \langle \lambda_{xy}^i \rangle) + \langle \lambda_{xy}^N \rangle$ 
5. return $\langle \lambda_{xy}^N \rangle$.

### Algorithm 23 Masked KKW·MUL (online)

| Input: | The input shares of $x$: $\langle \xi \rangle$ |
|        | The masks of $x$: $\langle \lambda_x^i \rangle$ for $i \in [N]$. |
|        | The input shares of $y$: $\langle \eta \rangle$. |
|        | The auxiliary shares of $xy$: $\langle \lambda_{xy}^i \rangle$ for $i \in [N]$, |
|        | The output masks of $\lambda_{xy}^i$ for $i \in [N]$. |
| Output: | The output shares of $\hat{z}$: for $i \in [N]$. |

1. for $i \in [N]$ / Players |
2. \( (a_i) \leftarrow \text{SMul}(\langle \xi \rangle, \langle \lambda_x^i \rangle) \) 
3. \( (b_i) \leftarrow \text{SMul}(\langle \eta \rangle, \langle \lambda_y^i \rangle) \) 
4. \( (s_i) \leftarrow (\langle \lambda_x^i \rangle - \langle \lambda_{xy}^i \rangle - a_i) - b_i \) 
5. \( \langle \hat{z} \rangle \leftarrow \text{SMul}(\langle \xi \rangle, \langle \eta \rangle) \) 
6. \( \langle \hat{z} \rangle \leftarrow \sum_{i\in[N]} \langle s_i \rangle \) 
7. \( \langle \hat{z} \rangle \leftarrow \langle \hat{z} \rangle + \langle s_i \rangle \) 
8. return $\langle \hat{z} \rangle$ and $\langle (s_i) \rangle_{i\in[N]}$.

### H.1 Proof of Lemma 1

**Proof.** We thus need to show that for any set of $O < T$ intermediate variables and any subset $O \subset [\hat{z}, \ldots, \hat{z}]$ of output shares such that $t + |O| < T$, for each input variable $v$, there is an input set $I_v$ with $|I_v| \leq t$ such that the $t$ intermediate variables and the output variables $\lambda_{xy}^{\langle s \rangle}$ can be perfectly simulated from these input sets.

Both the computation of $\langle \lambda_x^i \rangle$ and $\langle \lambda_y^i \rangle$ are straightforward and can be simply simulated, as they are linear operations. Whenever one of the terms involved in the computation of $\langle \lambda_{xy}^i \rangle \leftrightarrow \text{SMul}(\langle \lambda_x^i \rangle, \langle \lambda_y^i \rangle)$ is probed, we add the corresponding values from the proof of the SNI-security (found e.g. in [BBD⁺16, Proposition 2]) of $\text{SMul}$ to the input sets $I_v$. The result $\lambda_{xy}^{\langle s \rangle}$ can be simulated without any input as $\text{SMul}(\langle \hat{z} \rangle)$ is SNI. To simulate the output later on, we add all $\lambda_{xy}^{\langle s \rangle}$ to the inputs $I_v$.

Finally, the computation of $\langle \lambda_{xy}^N \rangle$ is again linear.
For the output, suppose that $\lambda_{N,j}^{xy}$ was probed. There are two cases to consider: If $\lambda_{N,j}^{xy}$ was probed, the inputs $I_v$ already contains all $\lambda_{i,j}^{xy}$ and we can thus simulate $\lambda_{N,j}^{xy}$ perfectly. If $\lambda_{N,j}^{xy}$ was not probed, $\lambda_{N,j}^{xy}$ looks like a uniformly random element from $F$ that is not used anywhere else, as it is produced by a $t$-SNI gadget with uniform output-distribution. We can thus uniformly sample a random element $r \in F$ and replace $\lambda_{N,j}^{xy}$ by $r$. This implies strong non-interference.

\section*{H.2 Proof of Lemma 2}

\textbf{Proof.} We thus need to show that for any set of $t < T$ intermediate variables and any subset $O$ of output shares such that $t + |O| < T$, for each input variable $v$, there is an input set $I_v$ with $|I_v| \leq t$ such that the $t$ intermediate variables and the output variables indexed by $O$ can be perfectly simulated from these input sets. To show this, we go through all variables of the algorithm and explain for all input variables $v$ which indices are added to $I_v$.

Whenever one of the terms involved in the $\text{SMul}(\ )$-computation for a term $a_{i,j}, b_{i,j},$ or $c_j$ is probed, we add the corresponding values from the proof of the strong non-interference of $\text{SMul}(\ )$ to the input sets $I_v$. Note that no inputs need to be added to $I_v$ if $a_{i,j}, b_{i,j},$ or $c_j$ were probed, as they are the result of a $t$-SNI gadget.

Whenever $s_{i,j}$ or a sub-term of $s_{i,j}$ is probed, we add the variables corresponding to $a_{i,j}, b_{i,j}, \lambda_{z,i,j},$ and $\lambda_{xy,i,j}$ to the input sets $I_v$. This clearly allows us to simulate all $s_{i\star,j}$ and all sub-terms perfectly.

Whenever a sum $\sum_{i'=1}^{i'} s_{i,j}$ (including $s_j$ itself) is probed, we distinguish two cases. If $s_{1,j}, s_{2,j}, \ldots, s_{i',j}$ were all probed, we can simply simulate the complete sum. Otherwise, there is a term $s_{i\star,j}$ with $i\star \in \{1, \ldots, i'\}$ such that $s_{i\star,j}$ was not probed. As $s_{i\star,j}$ is the only place where $a_{i,j}$ is used, we make use of the fact that $a_{i,j}$ is constructed by an $t$-SNI gadget with uniform output-distribution. In other words, this means that $a_{i,j}$ looks like a uniformly random element from $F$ that is not used anywhere else. We can thus uniformly sample a random element $r \in F$ and replace the complete sum $\sum_{i=1}^{i'} s_{i,j}$ by $r$. Note that in the previous argument, we did not add anything to $I_v$.

Finally, whenever $\hat{z}_j$ is probed, we simply simulate $s_j$ and $c_j$. As $c_j$ is the result of a $t$-SNI gadget, we can simulate it without needing to add anything to the input sets $I_v$. As shown in the discussion about $s_j$, we can also simulate it without needing to add anything to the input sets $I_v$. This implies strong non-interference.

\qed