Implementation and Performance Analysis of Homomorphic Signature Schemes

Davide Carnemolla[a,1], **Dario Catalano**[b,1], **Mario Di Raimondo**[c,1], **Federico Savasta**[d,2]

[1]University of Catania, Italy
[2]University of Montpellier, LIRMM, France

Abstract Homomorphic signatures allow to validate computation on signed data. Alice, holding a dataset $\{m_1, \ldots, m_t\}$ uses her secret key sk to sign these data and stores the authenticated dataset on a remote server. The server can later (publicly) compute $m = f(m_1, ..., m_t)$ together with a signature σ certifying that m is indeed the correct output of the computation f. Over the last fifteen years, the problem of realizing homomorphic signatures has been the focus of numerous research works, with constructions now ranging from very efficient ones supporting linear functions to very expressive ones supporting (up to) arbitrary circuits. In this work we tackle the question of assessing the practicality of schemes belonging to this latter class. Specifically, we implement the GVW lattice based scheme for circuits from STOC 2015 (both the standard and the ring variant) and two, recently proposed, pairings based constructions building from functional commitments. Our experiments show that (both) pairings based schemes outperform GVW on all fronts.

Keywords Homomorphic Signatures · Digital Signatures · Cryptographic Engineering · Cryptography

1 Introduction

Digital Signatures are one of the most fundamental primitives of modern cryptography. Their main security property is unforgeability against chosen message attacks: any efficient attacker $\mathcal{A}$ who has access to signatures on messages of his choice should not be able to produce a signature on a new message. Said differently, this means that signatures should be *non-malleable* in the sense that from a given signature on some message m one should not be able to derive signatures on any other $m' \neq m$. Yet, even if this sounds paradoxical at first, there are scenarios where some form of (controlled) malleability for signatures turns out to be useful. This controlled malleability feature is captured, for example, by the notion of homomorphic signatures, originally proposed by Desmedt [1] and Johnson et al. [2], and later formalized by Boneh and Freeman in [3]. Homomorphic Signatures are the focus of this work.

Homomorphic Signatures. Using homomorphic signatures (HS) some user Alice can use her secret key sk to produce signatures $\sigma_1, \ldots, \sigma_t$ on a set of messages $(m_1, \ldots, m_t)$ (often referred to as a *dataset*) like with ordinary signature scheme. The extra feature of HS, however, is that they come equipped with a (publicly) executable algorithm Eval that takes as input the signatures $\sigma_1, \ldots, \sigma_t$ and some function/program $\mathcal{P}$ and outputs a signature $\sigma_{\mathcal{P}}$ for the message $m = \mathcal{P}(m_1, \ldots, m_t)$. Notice that $\sigma_{\mathcal{P}}$ is not a signature on m alone but rather a signature on m *as output of* $\mathcal{P}$. In this sense, homomorphic signatures implement the requirement of controlled malleability mentioned above. More precisely, on input Alice's public key, $\mathcal{P}, m$ and $\sigma_{\mathcal{P}}$, the verification procedure checks whether m is the correct output of $\mathcal{P}$ on messages previously signed by Alice.

What makes this primitive non trivial to realize are the following features. First, homomorphic signatures should be *succinct*, in the sense that their size should be much smaller than the size of the original dataset[1].

Second, when verifying the output of a computation, the process should not require knowledge of the original

[a]e-mail: davide.carnemolla@phd.unict.it
[b]e-mail: dario.catalano@unict.it
[c]e-mail: mario.diraimondo@unict.it
[d]e-mail: federico.savasta@lirmm.fr

[1]Indeed, without this requirement, HS become trivial to realize: one simply sets $\sigma_{\mathcal{P}} = (\mathcal{P}, (\sigma_1, m_1), \ldots, (\sigma_t, m_t))$

dataset, an important property when signing very large datasets that the verifier might not store locally.

Finally, homomorphic signature could be *composable* in the sense that the outputs of previous computations (i.e. signatures obtained by Eval) can be used as inputs for new computations.

Defining security for homomorphic signatures is a much more delicate task[2] as, clearly, HS cannot meet the usual unforgeability notion. Informally, an HS is secure as long as adversaries knowing only the public verification key, can only produce signatures that either come from the legitimate signer or by running Eval on valid signatures. Slightly more in detail, $\mathcal{A}$ is allowed to see signatures for messages belonging for different datasets and wins if she manages to produce a valid signature, message pair (σ, m) such that either (1) m belongs to some new, previously unseen, dataset (this is called type 1 forgery) or (2) for some previously seen dataset $\Delta = \{m_1, \ldots, m_t\}$, she manages to produce a triplet $(\mathcal{P}, \sigma, m)$, where σ verifies correctly but m is not the correct output of $\mathcal{P}$, i.e. $m \neq \mathcal{P}(m_1, \ldots, m_t)$ (this is called a type 2 forgery).

Over the last fifteen years, the problem of realizing homomorphic signatures under various assumptions and settings has been the focus of several research works. It is fair to say that, at least from a theoretical perspective, nowadays we have a pretty clear understanding of the primitive, with known constructions ranging from very efficient ones for the case of programs expressible as linear functions (e.g. [5]) to very expressive ones supporting up to arbitrary circuits ([6, 7]. Yet, especially for this latter class of schemes, their concrete efficiency is far from clear.

Applications of Homomorphic Signatures. Homomorphic Signatures can be a very useful tool in the cloud computing environment.

The first fundamental use is computation on outsourced data. In this case, a user Alice uploads her data to the cloud along with their signature. She can then ask the cloud provider to perform computations on that data and to offer a proof of those computations. By using this primitive, Alice won't have to store the data that was previously uploaded to the cloud in memory in order to verify that the computation is correct. As the only information required to validate the computation is a short signature (the Eval algorithm's output), this will also allow us to avoid wasting network bandwidth.

Another important application is certified data analysis. In this scenario, a data provider (such as a government agency) supplies signed data to organizations so they can perform data analysis. Then, by supplying an output and the methodology (i.e., a function f), these organizations can publish their study based on this data. Organizations that might fabricate (out of self-interest) the results, must offer a proof to make their study publicly verifiable and they can do so without requiring access to the original data by utilizing a homomorphic signature.

[2] See [4] for a detailed discussion about this

1.1 Our contribution

In this paper we implement three different homomorphic signatures to better asses their concrete efficiency and clarify their practical significance. Specifically, we implement the lattice based HS from [6] in its standard lattice and ring lattice variants (GVW15/R-GVW15 from now on)[3] and the pairing based constructions from [8] and [7] (respectively, CFT22 and BCFL23 from now on)[4]. See Table 1 for a brief summary on their main features. These constructions, while concretely very different, share some high level interesting similarities. For instance, they all build from commitments with special properties (GVW15 implicitly builds from homomorphic equivocable commitments, CFT22 and BCFL23 from homomorphic functional commitments). Also, they all admit similar (amortized) efficiency optimizations when verifying the same function on different datasets. These similarities motivated our choices for the implementation which would allow fair comparisons among the three schemes.

Some recent works [9–11] present functional commitments constructions from lattices. We decided to do not take in account them in our comparison for several reasons. Differently from CFT22 and BCFL23, the FC from [9] is not succinct and it does not allow for efficient verification in the sense of CFT22. The FC scheme of [10] is succinct, but it also lacks of an efficient verification procedure. Finally, the protocol of [11] achieves fast verification from a preprocessing phase (also BCFL23 relies in a preprocessing phase). Their fast verification and commitment size are both sublinear in the size of the input. However, we do not compare this last construction with CFT22 and BCFL23 since, differently to the latter twos, it is not an additively homomorphic functional commitment.

All the schemes were implemented from scratch in Rust using state of the art libraries (qFALL [12] for lattices and *blst* [13] for pairings). Precise details on our

[3] We use the GVW15 notation to also to refer to the scheme and its properties, and the R-GVW15 notation when focus on properties specific to the ring setting

[4] We stress here that the construction from [8] only supports low degree polynomials whereas the constructions in [6, 7] support arbitrary circuits

implementations and comparisons are given in Sections 5 and 6, here we simply mention that our experiments show that (both) pairing based schemes outperform /R-GVW15 on essentially all fronts. Table 2 reports a sample from our experiments (more details are given in Section 6) for the variance function computed on a vector of $n = 20$ elements: as later discussed in Subsection 5.2, we were forced to use a very low (almost meaningless) security level for GVW15 in the case of standard lattices in order to complete the experiments. Our results suggest that, if one wants efficient homomorphic signatures beyond linear functions, pairing based constructions are the best, currently available, option.

On the implementation of R-GVW15. For efficiency reasons, we opt to use a computational instantiation of the standard/ring GVW15 public key and for the trapdoor generation, instead of a statistical one. This choice requires the hardness of the LWE problem in addition to the hardness of the SIS problem. For our implementation of R-GVW15, we choose optimistic parameters which guarantee less than 100 bit of security for the RLWE, the ring based variant of the LWE problem, for one of our circuits (the variance one), but they still guarantee 100 bits of security against RSIS, i.e. the ring based variant of the SIS problem. The motivation behind this choice is to show that even if we choose smaller parameters for efficiency, then losing security and partially correctness, even R-GVW15 is still less efficient than CFT22 and BCFL23. In contrast, parameters for 100 bits security of CFT22 and BCFL23 are correct.

	CFT22	BCFL23	GVW15
security assumption	DHE	n-HiKer[5]	SIS
security notion	strong adapt.	strong adapt.	semi adapt.
class of functions	const. deg. poly	circuits	circuits
amortized verification	✓	✓	✓
multi datasets	✓	✓	✓

Table 1: Features

	CFT22	BCFL23	R-GVW15	GVW15
KeyGen	**0.5** s	5.6 m	2.9 s	2.4 h
Sign	0.2 s	**0.1** s	6.4 m	52.5 h
Eval	**0.2** s	17 s	5.6 h	18.7 s
Ver	0.3 s	**0.1** s	11.6 m	6.4 s
signature	39.7 kB	**14.2** kB	2.1 GB	29.9 MB
proof	2.1 kB	**1.7** kB	235 MB	3.9 MB
security level	**100** bits	**100** bits	**100** bits*	30 bits**

* = refer to "On the implementation of GVW15" paragraph in Subsection 1.1.
** = refer to Subsection 5.2

Table 2: Benchmark for variance on $n = 20$ elements

1.2 Other related work

Linearly homomorphic signatures were introduced by Boneh *et al.* in [5] as a tool to prevent pollution attacks in linear network coding. Following this seminal work, many results further explored this notion both in the random oracle (e.g. [14–16]) and in the standard model (e.g. [17–22]). In the symmetric setting, the notion of homomorphic MAC was first considered (for the case of linear functions) in [23] and later extended to more general functionalities in [24–27]. Homomorphic signatures building from commit and prove SNARKs for NP were recently proposed in [28]. Linearly Homomorphic Structure preserving Signatures were introduced in [29] and exploited to construct Simulation Sound Semi Adaptive NIZKs in [30]. Privacy notions for homomorphic signatures were considered in [31].

[5] The construction is secure under a falsifiable assumption called HintedKernel. We refer the reader to the original paper for the formal definiton.

2 Preliminaries

Notation. In this work we use $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ for the set of natural, integer and real numbers, while $\mathbb{G}$ denotes a group. $\mathbb{Z}_q$ denotes the set of integers mod q if not specified differently. Vectors and matrices are denoted with lowercase and uppercase bold latin letters, respectively, e.g. $\boldsymbol{v}$ and $\mathbf{A}$. λ is the security parameter. With $x \xleftarrow{\$} \mathcal{X}$ we denote the uniform random sampling from a set $\mathcal{X}$, while with $x \leftarrow \mathcal{D}$ a sampling from a distribution or an algorithm $\mathcal{D}$. With $\mathsf{poly}(n)$ we denote the class of polynomial functions in the input n. With $\mathsf{negl}(n)$ we denote the class of negligible functions, i.e. we say that $f(n) \in \mathsf{negl}(n)$ if for all polynomials $p(n)$ we have $f(n) < 1/p(n)$ for sufficiently large values of n. $O(\cdot)$ and $\omega(\cdot)$ are the usual big-O and small-ω asymptotic notations. Remaining notation is introduced in the dedicated sections.

2.1 Functional Commitments

Intuitively, Functional Commitments (FC) [32] allow a sender to commit to a vector $\boldsymbol{x}$ so that she can later open the commitment to $f(\boldsymbol{x})$ for some function f. We present the formal definition of FC as presented in CFT22, which takes into account also the *compactness* property [33].

Definition 1 (Functional Commitments [8]) A functional commitment scheme is a tuple of algorithms $\mathsf{FC} =$

$(\mathsf{Setup}, \mathsf{Com}, \mathsf{Open}, \mathsf{Ver})$ with the following syntax and that satisfies correctness and succinctness (or compactness [33]).

$\mathsf{Setup}(1^\lambda, n, m) \to \mathsf{ck}$ on input the security parameter λ and the vector length n, outputs a commitment key ck, which defines the message space $\mathcal{X}$ and the class of admissible functions $\mathcal{F} \subseteq \{f : \mathcal{X}^n \to \mathcal{X}^m\}$ for some $n, m = \mathsf{poly}(\lambda)$.

$\mathsf{Com}(\mathsf{ck}, \boldsymbol{x}; r) \to (C, \mathsf{aux})$ on input a vector $\boldsymbol{x} \in \mathcal{X}^n$ and (possibly) a randomness r, outputs a commitment C and related auxiliary information aux. r can be omitted and randomly sampled.

$\mathsf{Open}(\mathsf{ck}, \mathsf{aux}, f) \to \pi$ on input an auxiliary information aux and a function $f \in \mathcal{F}$, outputs an opening proof π.

$\mathsf{Ver}(\mathsf{ck}, C, f, \boldsymbol{y}, \pi) \to b \in \{0, 1\}$ on input a commitment C, an opening proof π, a function $f \in \mathcal{F}$ and a value $\boldsymbol{y} \in \mathcal{X}^m$, accepts $(b = 1)$ or rejects $(b = 0)$.

Correctness. FC is correct if for any $n, m \in \mathbb{N}$, all $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, n)$, any $f : \mathcal{X}^n \to \mathcal{X}^m$ in the class $\mathcal{F}$, and any vector $\boldsymbol{x} \in \mathcal{X}^n$, if $(C, \mathsf{aux}) \leftarrow \mathsf{Com}(\mathsf{ck}, \boldsymbol{x})$, then it holds $\mathsf{Ver}(\mathsf{ck}, C, f, f(\boldsymbol{x}), \mathsf{Open}(\mathsf{ck}, \mathsf{aux}, f)) = 1$ with probability 1.

Succinctness/Compactness. A functional commitment FC is *succinct* if there exists a fixed polynomial $p(\lambda) = \mathsf{poly}(\lambda)$ such that for any $n, m = \mathsf{poly}(\lambda)$, any admissible function $f \in \mathcal{F}$ such that $f : \mathcal{X}^n \to \mathcal{X}^m$, honestly generated commitment key $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, n, m)$, vector $\boldsymbol{x} \in \mathcal{X}^n$, commitment $(C, \mathsf{aux}) \in \mathsf{Com}(\mathsf{ck})$ and opening $\pi \leftarrow \mathsf{Open}(\mathsf{ck}, \mathsf{aux}, f)$, it holds that $|C| \leq p(\lambda)$ and $|\pi| \leq p(\lambda) \cdot m$. Furthermore, we say that FC is *compact* if $|\pi| \leq p(\lambda)$.

The considered FCs used in the pairing based construction of Section 3 also satisfy a nice property which allows to additively combine commitments.

Additive-homomorphic FC [8]. Let FC be a functional commitment scheme where $\mathcal{X}$ is a ring. Then FC is *additive homomorphic* if there exist deterministic algorithms $\mathsf{FC.Add}(\mathsf{ck}, C_1, \ldots, C_n) \to C$, $\mathsf{FC.Add}_{\mathsf{aux}}(\mathsf{ck}, \mathsf{aux}_1, \ldots, \mathsf{aux}_n) \to \mathsf{aux}$ and $\mathsf{FC.Add}_r(\mathsf{ck}, r_1, \ldots, r_n) \to r$ such that for any $\boldsymbol{x}_i \in \mathcal{X}$ and $(C_i, \mathsf{aux}_i) \leftarrow \mathsf{Com}(\mathsf{ck}, \boldsymbol{x}_i; r_i)$, if $C \leftarrow \mathsf{FC.Add}(\mathsf{ck}, C_1, \ldots, C_n)$, $\mathsf{aux} \leftarrow \mathsf{FC.Add}_{\mathsf{aux}}(\mathsf{ck}, \mathsf{aux}_1, \ldots, \mathsf{aux}_n)$, and $r \leftarrow \mathsf{FC.Add}_r(\mathsf{ck}, r_1, \ldots, r_n)$, then $(C, \mathsf{aux}) = \mathsf{Com}(\mathsf{ck}, \sum_{i=1}^n \boldsymbol{x}_i; r)$.

Security. The common security notion for FCs is the *evaluation binding*, which consists in the hardness of computing two different opening proofs for two different values which pass the verification step for the same function. As proven in [8], for the security of construction of the FC-based Homomorphic Signature in Section 3, it is enough that the FC satisfies the weaker notion of security below.

Definition 2 (Weak Evaluation Binding [8]) For any PPT adversary $\mathcal{A}$

$$\mathbf{Adv}^{\mathsf{wEvBind}}_{\mathcal{A},\mathsf{FC}}(\lambda) := \Pr\left[\begin{array}{c}(C, \cdot) = \mathsf{Com}(\mathsf{ck}, \boldsymbol{x}; r) \\ \wedge\ y \neq f(\boldsymbol{x}) \wedge \\ \mathsf{Ver}(\mathsf{ck}, C, f, \boldsymbol{y}, \pi) = 1\end{array} : \begin{array}{c}\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, n) \\ (\boldsymbol{x}, r, f, \boldsymbol{y}, \pi) \leftarrow \mathcal{A}(\mathsf{ck})\end{array}\right] = \mathsf{negl}(\lambda)$$

2.2 Lattices

Lattices. A $m-$dimensional lattice $\mathcal{L}$ of rank n is defined as the set of linear combinations of a set of vectors $\mathbf{B} = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n], \boldsymbol{v}_i \in \mathbb{R}^m$, with coefficients in $\mathbb{Z}$, i.e. $\mathcal{L} = \{\mathbf{B}\boldsymbol{z} : \boldsymbol{z} \in \mathbb{Z}^m\}$. $\mathbf{B}$ is called a basis for $\mathcal{L}$.

Usually, we use the term *short basis* to refer to a matrix with norm smaller than a certain bound.

Definition 3 (SIS problem) Given parameters n, m, q, β and a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, the Short Integer Solution problem (SIS) consists in finding a non-zero vector $\boldsymbol{u} \in \mathbb{Z}^m$ such that $\mathbf{A}\boldsymbol{u} = \mathbf{0} \mod q$ and $||\boldsymbol{u}|| \leq \beta$.[6] The SIS problem is said to be hard if the probability of finding such a $\boldsymbol{u}$ is negligible, for all polynomially bounded adversaries.

Ring lattices. For ring lattices, we consider the quotient ring $R = \mathbb{Z}[X]/(\Phi_{2n}(X))$, where $\Phi_n(X)$ denote the n-th cyclotomic polynomial. Ring lattices are defined by vectors $\boldsymbol{a} \in R_q^m$ where $R_q := R/qR$.

Definition 4 (RSIS problem) Given parameters n, m, q, β, and a vector $\boldsymbol{a} \in R_q^m$, the Ring Short Integer Solution problem (RSIS) consists in finding a non-zero vector $\boldsymbol{u} \in R^m$ such that $\boldsymbol{a}^T\boldsymbol{u} = 0 \mod q$ and $||\boldsymbol{u}|| \leq \beta$.[7] The RSIS problem is said to be hard if the probability of finding such a $\boldsymbol{u}$ is negligible, for all polynomially bounded adversaries.

2.3 Homomorphic Signatures

In this work we compare several Homomorphic Signature schemes (HS). Informally, an HS allows a user to

[6] Typically chosen norms are ℓ_2 and ℓ_∞. GVW15 opts for ℓ_∞.

[7] Typically chosen norms are ℓ_2 and ℓ_∞. R-GVW15 opts for ℓ_∞.

sign a dataset $(x_1, \ldots, x_n)$ and obtain a set of signatures $(\sigma_1, \ldots, \sigma_n)$ for which an untrusted server can compute a function $y = f(x_1, \ldots, x_n)$ and a signature σ. If correctly computed, (σ, f, y) can be correctly verified with the user verification key certifying that $y = f(\boldsymbol{x})$ (without the need to know $\boldsymbol{x}$). In this subsection we recall the definition of a Multi-Data Homomorphic Signature scheme and its most basic security notion from [6]. The definition considers more datasets. In what follows Δ denotes the datased identifier and $\mathcal{X}$ the message space.

Definition 5 (Multi-data Homomorphic Signature Scheme [6]) A Multi–data homomorphic signature scheme consists of poly-time algorithms $(\mathsf{PrmsGen}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Eval}, \mathsf{Ver})$ with the following syntax.

$\mathsf{PrmsGen}(1^\lambda, \mathsf{data}) \to \mathsf{prms}$ on input the security parameter λ and some data data specific of the construction, generates public parameters prms.

$\mathsf{KeyGen}(1^\lambda, \mathsf{prms}) \to (\mathsf{pk}, \mathsf{sk})$ on input the security parameter λ and prms, returns a public verification key pk and a secret signing key sk.

$\mathsf{Sign}(\mathsf{sk}, \Delta, (x_1, \ldots, x_N)) \to (\sigma_1, \ldots, \sigma_N)$ on input $\bar{\boldsymbol{x}} \in \mathcal{X}^*$ under a dataset identifier $\Delta \in \{0,1\}^*$, returns a tuple of signature $(\sigma_1, \ldots, \sigma_N)$.

$\mathsf{Eval}(\mathsf{pk}, g, \Delta, ((x_1, \sigma_1), \ldots, (x_\ell, \sigma_\ell))) \to \sigma^*$ takes a set of message/signature pairs under a dataset identifier $\Delta \in \{0,1\}^*$ and homomorphically computes a signature σ^*.[8]

$\mathsf{Ver}(\mathsf{pk}, g, y, \Delta, \sigma^*) \to b \in \{0,1\}$ verifies that $y \in \mathcal{X}$ is indeed the output of g over the data signed with the dataset identifier Δ.

Correctness of Signing. For any $\mathsf{prms} \in \mathsf{PrmsGen}(1^\lambda, \mathsf{data})$, $(\mathsf{pk}, \mathsf{sk}) \in \mathsf{KeyGen}(1^\lambda, \mathsf{prms})$, any $(x_1, \ldots, x_N) \in \mathcal{X}^N$, any $\Delta \in \{0,1\}^*$ and any $(\sigma_1, \ldots, \sigma_N) \in \mathsf{Sign}(\mathsf{sk}, \Delta, (x_1, \ldots, x_N))$, then $\mathsf{Ver}(\mathsf{pk}, \mathsf{id}_i, x_i, \Delta, \sigma_i) = 1$, where id_i is the identity function which sends $(x_1, \ldots, x_N)$ to x_i.

Correctness of Evaluation. For any circuits $h_1, \ldots, h_\ell$ with $h_i : \mathcal{X}^N \to \mathcal{X}$ and any circuit $g : \mathcal{X}^\ell \to \mathcal{X}$, any $(x_1, \ldots, x_\ell) \in \mathcal{X}^\ell$, any $\Delta \in \{0,1\}^*$ and any $(\sigma_1, \ldots, \sigma_\ell)$:

$$\left\{ \begin{array}{c} \{\mathsf{Ver}(\mathsf{pk}, h_i, x_i, \Delta, \sigma_i) = 1\}_{i \in [\ell]} \\ \sigma^* := \mathsf{Eval}(\mathsf{pk}, g, \Delta, (x_1, \sigma_1), \ldots, (x_\ell, \sigma_\ell)) \end{array} \right\}$$
$$\implies \mathsf{Ver}(\mathsf{pk}, (g \circ \bar{h}), g(x_1, \ldots, x_\ell), \Delta, \sigma^*) = 1.$$

An additional property that HS schemes can satisfy is *succinctness*. Informally, an HS is *succinct* if the Eval output size is smaller than $\mathsf{poly}(\lambda) \cdot \log n$, where n is the input size.

Security. For simplicity, we recall here the security definition of an HS given in [6][9] (see [4] for an analysis and a formal description of the security notions for HS schemes).

Security Experiment [6]. Let HS be as in Definition 5. Consider the following experiment with a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$:

1. $\mathcal{C}$ runs $\mathsf{prms} \leftarrow \mathsf{PrmsGen}(1^\lambda, \mathsf{data})$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda, \mathsf{prms})$. pk and prms are then given to $\mathcal{A}$.
2. $\mathcal{A}$ sends an arbitrary number of queries. For each query j
   - $\mathcal{A}$ chooses a fresh and never queried identifier $\Delta_j \in \{0,1\}^*$ and a message vector $(x_{j,1} \ldots x_{j,N_j}) \in \mathcal{X}^*$.
   - $\mathcal{C}$ replies to $\mathcal{A}$ with
   $$(\sigma_{j,1}, \ldots, \sigma_{j,N_j}) \leftarrow \mathsf{Sign}(\mathsf{sk}, \Delta_j, (x_{j,1} \ldots x_{j,N_j}))$$
3. $\mathcal{A}$ chooses a function $g : \mathcal{X}^{N'} \to \mathcal{X}$ and Δ, y', σ'. Let $y := g(x_1, \ldots, x_n)$. $\mathcal{A}$ wins the game if:
   - $\mathsf{Ver}(\mathsf{pk}, g, y', \sigma') = 1$;
   - One of the following types of forgeries happens:
     - Type-1: $\Delta \neq \Delta_j$ for any j or $\Delta = \Delta_j$ for some j and $N' \neq N_j$
     - Type-2: $\Delta = \Delta_j$ for some j such that g is admissible on $x_{j,1}, \ldots, x_{j,N'}$ and $y' \neq g(x_{j,1}, \ldots, x_{j,N'})$

HS is said to be *secure* if for all PPT $\mathcal{A}$, the probability that $\mathcal{A}$ wins is $\mathsf{negl}(\lambda)$ in the security experiment above.

3 Pairing based Homomorphic Signatures

The aim of this section is to introduce the two pairing based schemes of CFT22 and BCFL23 that build upon additive-homomorphic FCs. The main idea is that the signer generates a commitment $C_{\boldsymbol{x}}$ to his dataset and a digital signature $\sigma_{C_{\boldsymbol{x}}}$ on the commitment and gives it to the server. The server can then compute a function f by giving to the verifier the pair $(C_{\boldsymbol{x}}, \sigma_{C_{\boldsymbol{x}}})$ and an opening of $C_{\boldsymbol{x}}$ to the function f. Then the verifier can use the signer public key to verify the server computation. In addition, to dispense the signer from having to commit to the dataset all at once, [8] proposes the use of additive-homomorphic FC and a linearly homomorphic signature (LHS) in the commitment space. Concretely the LHS used in [8] is the structure preserving linearly homomorphic scheme from [34]. The LHS

[8] g must be an admissible function where the notion of admissible function depends on the specific scheme.

[9] In [4] security is defined with respect to (multi)- labeled programs. Since the focus of this paper is on implementations of existing, already proved secure schemes, the security definition is rephrased in a simpler way without relying on labeled programs.

supports FC.Add for evaluations. For what concerns the security, as proven in [8], the HS is strongly-adaptive secure if the underlying LHS is strongly-adaptive secure and the FC satisfies weak evaluation binding. We implement the LHS construction presented in the original paper, which supports commitments to group elements for the pairing based FC under consideration. The LHS implemented is built from any (standard) digital signature scheme. The algorithms of the resulting HS construction are summarized in Figure 1.

Short overview of the implemented FC The first FC scheme (CFT22) we use to instantiate an HS is the one presented in the Section 4 of [8]. This scheme supports a commitment opening to multiple (multivariate) polynomials of bounded constant degree (fixed at setup time) with a single *compact* proof and its security is proven under the standard Diffie-Hellman exponent assumption. On the other hand, the second FC scheme (BCFL23), proposed in [7], supports the evaluation of arbitrary arithmetic circuits with a maximal *width*, the opening proof is *succinct* and its security is proven under a new falsifiable assumption called n-HiKer. Both schemes are additive-homomorphic and have amortized efficient verification.

4 A Lattice based Homomorphic Signature

In this section we recall the Lattice based Homomorphic Signature of GVW15. The authors present two schemes, one with larger public parameters in the standard model and one with shorter public parameters in the random oracle model. The construction of both HS schemes relies upon a primitive introduced in [6] called *Homomorphic Trapdoor Function* (HTDF in brief).The main component of an HTDF is a function $f_{\mathsf{pk},x}$ described by a public key pk and an index x, which can be evaluated on some input u and it gives in output some v. An HTDF can be seen as a homomorphic equivocable commitment scheme, where knowing the secret key it is possible to open any commitment v to u for any index x. More precisely, an HTDF consists of five algorithms: a key generation algorithm KeyGen, the function $f_{\mathsf{pk},x}$, an inversion algorithm (decommit) $\mathsf{Inv}_{sk,x}$, an inner evaluation algorithm Eval^{in} for evaluating an admissible function on a set of inputs and indexes and an outer evaluation algorithm Eval^{out} for evaluating the effect of an admissible function on a set of outputs. For a formal and detailed definition of an HTDF we refer to [6], Section 3.1.
The GVW15 HS is a tuple of algorithms $\mathsf{HS} = (\mathsf{PrmsGen}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver}, \mathsf{Eval})$ where Ver is divided in two subprocedures $(\mathsf{Ver}^*, \mathsf{Process})$. The single-data HS construction from GVW15 built upon an HTDF it is described in Fig. 2. The specific HTDF is introduced in the GVW15 paper (see [6], Section 3.1). In the HS only the depth d of the circuits is fixed in setup, and the protocol can evaluate arbitrary circuits. Their HS allows full composability, meaning that outputs of some evaluations can be used as inputs to a new evaluation which can be verified using the verification key of the legitimate signer.
The implemented version of the scheme, which is built for a single dataset, satisfies a notion of security called *selective security* where an adversary chooses its messages $(x_1, \ldots, x_\ell)$ before knowing the public parameters and the public key, and a possible forgery is only of the second type (see Security experiment in Subsection 2.3). Security of the HS is implied by the security of the HTDF ([6]).

5 Implementations

As our main contribution in this work, we implemented the three constructions of Sections 3 and 4 in order to compare them in terms of execution time and bandwidth consumption. Additionally, we carried on a feasibility study for some commonly used statistical functions to understand their applicability in real-world scenarios. Our implementations were written in Rust programming language, along with several additional libraries we will cite later[10].

5.1 Pairing based schemes

As previously mentioned, CFT22 and BCFL23 are based on pairings. For the implementation of both schemes, we used the open-source *blstrs* cryptographic library provided by Protocol Labs [35]. This is a Rust binding for the well-known *blst* library [13], written in C and Assembly, which provides an efficient implementation of the Barreto-Lynn-Scott pairing-friendly elliptic curve [36] with embedding degree 12 and a 381 bits modulus (BLS12-381). The BLS12-381 curve was instantiated from the BLS12 curve family by appropriately choosing the instantiation parameter to minimise its Hamming weight, which is useful for achieving the best performance on the Miller's Weil pairing algorithm (also known as Miller's Loop), while maximizing the size of the field modulus q. This allows fast 32/64-bit arithmetic. BLS12-381 was originally designed to provide a 128 bit security level, however it is known that the extended tower number field sieve (exTNFS) [37]

[10] The source code is available upon request of the reviewers.

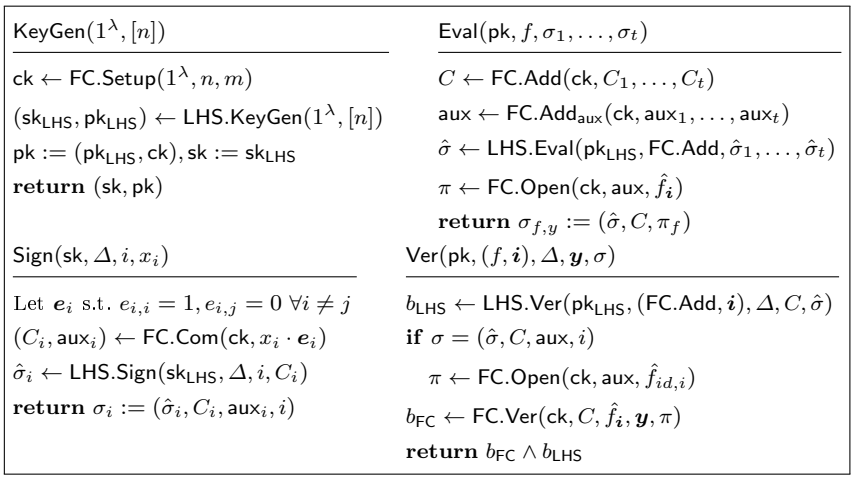


Fig. 1: HS from additive FC and LHS for FC.Add.

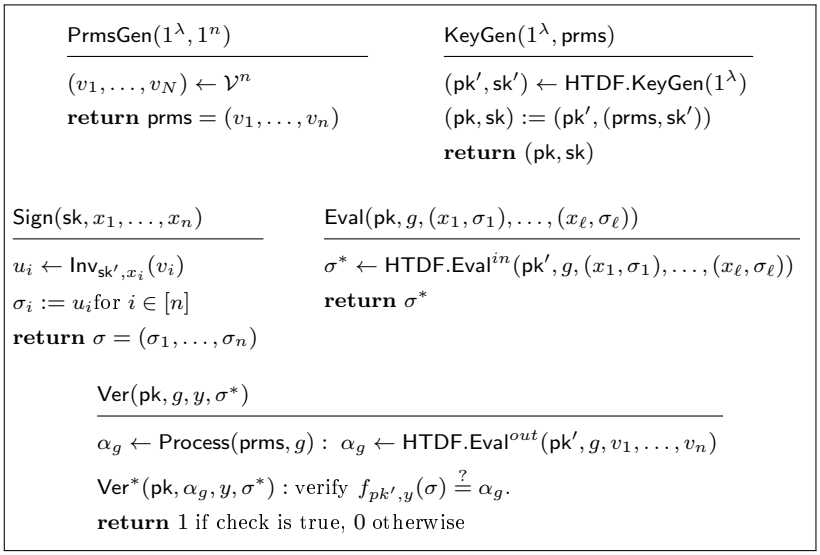


Fig. 2: The GVW15 HS from HTDF.

reduces its security to approximately 100 bits [38]. For each of the aforementioned schemes a library was developed using a modular approach. Two different modules have been created: one for the FC scheme and one for the LHS construction. These modules were then called up in another module specifically created for the HS construction. For both schemes we implemented the LHS proposed in [8] using BLS [39] as standard digital signature, namely the implementation of the *blst* [13] library.

5.2 Lattice based scheme

We first implemented the lattice based scheme (GVW15) over standard lattices using the qFALL library [12], which provides mathematical primitives for lattice based cryptography and the implementation of the trapdoor functions proposed in [40]. qFALL uses the optimized widespread FLINT library [41] as backend for the arithmetic on standard rings. First of all, we tried to find proper parameters, using the Lattice Estimator project [42], to instantiate the scheme in order to provide a security of approximately 100 bits. After the early experiments we realised that with such parameters, the execution time of the benchmarks would be totally unacceptable, mainly due to the KeyGen and Sign algorithms. Specifically, in addition to compute a parity matrix $\mathbf{A}$ and an associated trapdoor td, the KeyGen algorithm also generates a short base of the lattice of interest using td and precomputes its Gram-Schmidt Orthogonalisation. This is necessary to optimise the execution of the SamPre algorithm during the

signing phase. Unfortunately, it is the Gram-Schmidt Orthogonalisation that is the most time-consuming operation in our KeyGen implementation. As previously outlined, the Sign algorithm simply invokes the Inv algorithm, whose execution time is exclusively dependent on the parameters selected for the SIS instance. In fact, this algorithm solves a certain number of systems of linear equations depending on the chosen SIS parameters. Using the qFALL library, our implementation of this algorithm employs the Gaussian elimination method. Despite this optimization, the Sign algorithm remains very time-consuming due to the size of the involved matrices: more precisely if n and q are the parameters for the SIS instance, then we have to solve $n\lceil \log_2 q \rceil$ systems of linear equations modulus q. For these reasons, the only complete benchmarks that we could obtain for the standard-lattice implementation used lower non-secure parameters, namely $n = 16$, $q = 2601023$ and Gaussian parameter $s = 24$. According to the Lattice Estimator, this would give a security of about 30 bits. Nevertheless, the scheme already performs poorly with these parameters: for an input vector of length 20, KeyGen takes about 8700 seconds and signing takes more than two days. These numbers made clear that this implementation could not provide a meaningful comparison at a security level close to the pairing based schemes.

We then implemented a ring-lattice version of GVW15. This version uses the same qFALL/FLINT stack, extended with the ring sampler and the ring arithmetic optimizations described below. In all the ring-lattice benchmarks we set the ring dimension to $N = 1024$. For the weighted sum we used $q = 17592186044417$ and Gaussian parameter $s = 200$; for the variance, whose quadratic circuit produces larger noise, we used $q = 241785163922925834941235 3$ and $s = 600$. The security of these instances was estimated, as for the standard-lattice case, with the Lattice Estimator project [42]. The ring-lattice instantiation is still much slower than the pairing based schemes, but it is substantially more informative than the standard-lattice benchmark: it keeps KeyGen in the order of a few seconds and signing in the order of minutes, while using lattice parameters much closer to the security level of the other constructions. For this reason, in Section 6 we report the data for the ring-lattice version, denoted R-GVW15, rather than the first standard-lattice implementation. The scheme was implemented in a generic way, enabling the evaluation of both boolean and arithmetic circuits as outlined in the original article. However, our benchmarks were carried out on arithmetic circuits to reduce execution time and bandwidth consumption. Indeed, using boolean circuits would have posed challenges in expressing the chosen functions and produced more signatures (one for each bit of the vector elements), leading to wasted bandwidth and CPU cycles.

5.3 Optimizations

Before discussing the experimental results, we describe the main optimizations applied to our implementations. We distinguish between optimizations that reduce the amount of algebraic work performed by a single thread and optimizations obtained by parallel execution. This distinction is important for the methodology of our benchmarks: the multi-threaded implementation of Section 5.4 is always compared against the optimized sequential implementation, and not against a naive baseline. In this way, the speed-up due to parallelism is not artificially inflated by improvements that are in fact purely algorithmic.

Pairing based schemes. For both CFT22 and BCFL23, a first relevant optimization concerns the verification of pairing equations. Several verification checks require equations of the form $e(a, b) = e(c, d)$, or equations with one additional pairing term on the right hand side. Instead of computing each pairing independently, we rewrite these checks as a single multi-Miller loop followed by one final exponentiation. Since the final exponentiation is one of the most expensive components of a pairing computation on BLS12-381, this optimization reduces the verification time without changing the equations being verified.

For CFT22, we also optimized the generation of the commitment key and the routines for commitment, opening and evaluation. In KeyGen, the powers of the secret scalar are computed by a linear multiplicative scan, rather than by invoking a separate exponentiation for each index. In the algorithms that compute linear combinations of group elements, the implementation accumulates directly into pre-allocated group elements initialized to the identity, avoiding temporary vectors which would be immediately consumed by a subsequent summation. Finally, in verification, the group elements that appear repeatedly in pairing equations are converted to affine form only once and then reused. These changes mainly reduce redundant scalar multiplications, projective-to-affine conversions and memory allocations.

For BCFL23, the most important optimization comes from exploiting sparsity. In the opening procedure, both the linear and the quadratic part of the proof are computed by iterating only over the non-zero coefficients of the function. This is particularly relevant

for the quadratic part, where the dense implementation would traverse all pairs of input coordinates, even when the corresponding coefficient is zero. The same idea is used in function evaluation: zero coefficients are skipped and the Kronecker products of the input vectors are precomputed once instead of being recomputed for every output component. Moreover, during key generation, several scalar products that occur in nested loops are precomputed and reused, and the field inversions needed for the relevant scalar vectors are performed with batched Montgomery inversion. This replaces many independent inversions with one inversion and a linear number of multiplications, which is substantially cheaper in the underlying field.

Lattice based scheme. The optimizations for GVW15 are more invasive, because the bottleneck is mostly in the underlying linear algebra and in the trapdoor preimage sampling. In the integer instantiation, the signing routine was reorganized so that key-independent quantities, such as the gadget matrix, the dimensions and the modulus, are computed once and reused across the whole input vector. The preimage sampler is then invoked column by column, avoiding repeated setup work. Verification was also modified to avoid materializing the full verification map: the implementation computes the product between the public key and the signature directly and adds the gadget contribution only in the positions where it is structurally non-zero.

For R-GVW15, a direct implementation of the Micciancio-Peikert sampler [40] would require constructing the full embedded lattice basis and computing an exact Gram-Schmidt Orthogonalisation. This approach becomes quickly infeasible. We therefore implemented a perturbation-based sampler for the ring setting, exploiting the structure of the gadget trapdoor so that the expensive part of the sampling is reduced to a much smaller Gaussian sampling problem. The numerical core of this sampler, including the Cholesky factorization and the computation of the conditional mean, is carried out in double precision, while the integer and modular operations that define the final preimage identity remain exact. Furthermore, the covariance factorization depends only on the key, and is therefore precomputed during key generation and stored in the signing key instead of being rebuilt for every signature.

Finally, for the ring operations we added a negacyclic number-theoretic transform when the modulus is an NTT-friendly prime, namely when $q \equiv 1 \pmod{2n}$ and n is a power of two. The transform is enabled only for the ring multiplications where both operands are full-size modulo q, such as the multiplication gates used in the variance evaluation and the public-key multiplications in verification. We deliberately do not use it for the multiply-by-constant gate, since in that case the operands have small coefficients and the overhead of the transform is larger than the gain. If the modulus does not satisfy the NTT condition, the implementation falls back to the standard multiplication routine, producing the same result.

5.4 Multi-thread implementation

The sequential optimizations described above are orthogonal to parallelism. For this reason, the parallel implementation is exposed as an optional Cargo feature and the default build remains sequential. The same public interfaces are used in both configurations, so the benchmark harness can compare the sequential and parallel builds without changing the tested algorithms.

In the pairing based schemes, parallelism is mainly used where the computation naturally decomposes into independent group operations. During key generation, independent parts of the functional commitment key and of the linearly homomorphic signature key are generated concurrently. The vectors, matrices and tensors of group elements are also populated in parallel, since each entry depends only on its index and on immutable scalar values. Commitment, opening and evaluation use parallel reductions over group elements, with the group identity as neutral element. In verification, independent checks, such as the linearly homomorphic signature verification and the functional commitment verification, are executed concurrently. The only shared mutable state is the dataset history maintained by the homomorphic signature layer; in the parallel build it is protected by a read-write lock, while the sequential build uses a lighter interior-mutability structure.

For R-GVW15, the parallel implementation acts both at the level of the homomorphic signature algorithms and inside the underlying mathematical routines. Signing is organized as a batch pipeline: the coset representatives for the messages are computed in parallel, the batch is given to the preimage sampler, and the resulting signature matrices are recomposed in parallel. Parameter generation overlaps the sampling of independent public matrices with the generation of the secret key. Evaluation distributes independent terms across threads and then combines them with the additive homomorphic operations.

In our local versions of the mathematical libraries used by the lattice implementation, we added row-blocked parallel modular matrix multiplication, parallel row-wise uniform sampling and parallel batched gadget inversion and preimage sampling. These routines

are guarded by size thresholds, so that small inputs are still handled sequentially and do not pay unnecessary scheduling overhead. Moreover, the global threading initializer fixes the external thread pool and disables the internal parallelism of the underlying linear algebra library. This avoids oversubscription and makes the parallel benchmarks more stable.

5.5 Benchmarks

First of all, we implemented a benchmark unit to compare the schemes in terms of execution time. We used the Divan library [43] which provides a good approximation of the strategy proposed in [44] for benchmarking even in noisy environments. We also considered the use of the Criterion library: indeed it is considered the *de facto* standard for Rust benchmarks. A wider flexibility in the test management but also a better accuracy guaranteed by the native use of the Time Stamp Counter (TSC), determined our final choice. For each benchmark, we collected as many samples as possible for at least 60 seconds. This guaranteed a proper number of runs on fine-grain operations but also a dynamic experimental environment on heterogeneous operations. In experiments where we collected more than one sample, the value in Section 6 represents the median of the collected samples. Our benchmarks keep track of the bandwidth usage among the parties in the implemented schemes. Due to the complexity of the involved data structures, we decided to use as metric the size of the serialization obtained by means of the Concise Binary Object Representation (CBOR)[45] provided by the Serde serialization library. Benchmark units have been implemented for two representative functions in statistics: the weighted sum and the variance. The first one gives us an understanding of the behaviour of constructions in the linear case, the second one in the quadratic case[11]. For the optimized sequential implementation we varied the input size, while for the multi-threaded implementation we fixed the input size to $n = 20$ and measured the same operations with 2, 4 and 8 worker threads.

Amortized verification. We also implemented a dedicated benchmark unit for the amortized verification procedure supported by the three optimized schemes considered in the experiments, namely CFT22, BCFL23 and R-GVW15. This benchmark models the scenario in which the verifier checks the same function on several independently signed datasets. In this setting, verification can be split into a one-time preprocessing phase, depending only on the public verification material and on the function, and a faster online verification phase that depends on the evaluated output and on the corresponding homomorphic signature. For the pairing based schemes the precomputation extracts the function-dependent combinations of the commitment key that are recomputed by the functional-commitment verifier in the baseline path, while leaving the linearly homomorphic signature checks unchanged. For R-GVW15 the preprocessing corresponds to the computation of the processed public parameters α_g, which are then reused by the final verification equation. We measured, for input size $n = 20$, the cost of the preprocessing phase and the cost of verifying batches of 1, 2, 4, 8, 16 and 32 outputs of the same function. The benchmark was run for both the weighted sum and the variance, using the optimized single-threaded implementation.

[11] We implemented the variance using a pair of grade 2 polynomials omitting the problematic division operation for the public and function-related parameter n.

6 Experimental results

In this section we report the benchmark results we collected in our experiments. In all the executions we evaluated the chosen statistic functions on the same randomly-generated vector of 8-bit unsigned integers. All our benchmarks were executed on a server equipped with an Intel(R) Xeon(R) Gold 6238R CPU running at 2.20 GHz and 72 GiB of RAM. Figures 3 and 4 show the results of the optimized sequential implementation for, respectively, the weighted sum and variance functions. Figures 5 and 6 report the corresponding multi-threaded benchmarks for an input vector of size $n = 20$. Since the collected data are spread over a wide range, all graphs use a logarithmic scale. They give an idea of the running times of the main operations and of the sizes of the generated signatures and proofs.

In the sequential benchmarks, the size of the dataset ranges from 5 to 40 elements. The results for R-GVW15 must be interpreted with the caveat that they were obtained with very small and insecure parameters. For the weighted sum in Figure 3, CFT22 has by far the fastest key generation among the pairing-based schemes: on average, it is about 2800 times faster than BCFL23, and the gap increases with the input size. CFT22 is also faster than BCFL23 in evaluation and verification by average factors of about 27 and 1.5, respectively. Conversely, BCFL23 is about 3.4 times faster in the signing phase, produces signatures about 2.8 times smaller, and produces proofs about 1.3 times smaller.

For the variance function in Figure 4 the picture is similar, with the notable exception of veri-

fication. CFT22 key generation remains much faster than BCFL23, by an average factor of about 1800, and CFT22 is also faster in evaluation, by an average factor of about 111. Verification, instead, favours BCFL23 as soon as the input is not trivially small: for $n = 5$ the two schemes are comparable, with CFT22 about 1.1 times faster, while for $n \geq 10$ BCFL23 is faster, by about 2.3 times on average and up to about 2.8 times for $n = 40$. This is a consequence of the sparse verifier described in Section 5: the function-dependent part of the BCFL23 verification only involves the non-zero coefficients of the variance polynomial. BCFL23 keeps the advantage in signing, by a factor of about 3.4, and in bandwidth consumption: its signatures are again about 2.8 times smaller, while its proofs are about 1.2 times smaller. R-GVW15 is several orders of magnitude larger in signature and proof size and is also much slower in the signing, evaluation and verification phases. In particular, with respect to CFT22, R-GVW15 is slower by average factors of about 400 in signing, 1900 in evaluation and 300 in verification for the weighted sum, and by about 2100, 91000 and 2300, respectively, for the variance.

Table 3 reports the amortized verification benchmarks for a fixed input size $n = 20$. These experiments quantify the benefit of reusing the same verification preprocessing when the same function is checked on several different datasets. The baseline for each batch size is the optimized non-amortized verification repeated independently, whereas the amortized variant pays the preprocessing cost once and then runs the online verification procedure on each output. Therefore, the total amortized cost for a batch is the sum of the *Precomp.* column and the corresponding online verification time.

The break-even point is already reached, up to measurement noise, after one verified output in all the tested cases. This is expected from the structure of the optimized verifiers: the one-time preprocessing extracts almost exactly the function-dependent part that the baseline recomputes at every verification. The effect is modest but consistent for the weighted sum in the pairing-based schemes: for batches of 32 outputs, the total amortized verification time is about 1.3 times smaller than the baseline for both CFT22 and BCFL23. The gain is much larger for the variance. For CFT22 the batch of 32 outputs goes from 9.181 s in the baseline to about 819 ms including preprocessing, while for BCFL23 it goes from 4.000 s to about 1.148 s. R-GVW15 benefits the most from this optimization: for the weighted sum, the batch of 32 outputs decreases from 3.191 minutes to about 25.5 seconds including preprocessing, and for the variance from 6.229 hours to about 16.0 minutes. This confirms that the processed public parameters α_g account for the dominant part of repeated verification in the ring-lattice construction, especially for the quadratic function.

The parallel benchmarks in Figures 5 and 6 show that parallelism is useful mainly on the operations with enough internal work to amortize scheduling overhead. For the weighted sum in Figure 5, moving from 2 to 8 threads gives only a small key-generation improvement for CFT22 and R-GVW15, by factors of about 1.2 and 1.0, respectively, while BCFL23 improves by a factor of about 3.4. Signing improves by factors of about 3.4 for CFT22 and BCFL23 and 3.9 for R-GVW15. Verification improves by factors of about 2.6 and 2.5 for CFT22 and R-GVW15, respectively, whereas BCFL23 gains only about 12%: after the sparsity optimization its verification is dominated by a small number of pairings, which leaves little work to distribute among the threads. The effect on evaluation is more scheme-dependent: CFT22 becomes about 16% slower, while BCFL23 and R-GVW15 improve by factors of about 1.5 and 2.7, respectively. For the variance in Figure 6, signing improves by about 3.5 for all schemes. Verification improves by factors of about 4.0 and 3.4 for CFT22 and R-GVW15, respectively, while BCFL23 verification is essentially insensitive to the number of threads, for the same reason as in the weighted sum. In evaluation, BCFL23 and R-GVW15 obtain an almost fourfold speed-up and CFT22 improves only marginally.

Declarations

Conflict of interests The authors have no relevant financial or nonfinancial interests to disclose.

Ethical approval The authors declare full compliance with ethical standards. This article does not contain any studies involving humans or animals performed by any of the authors.

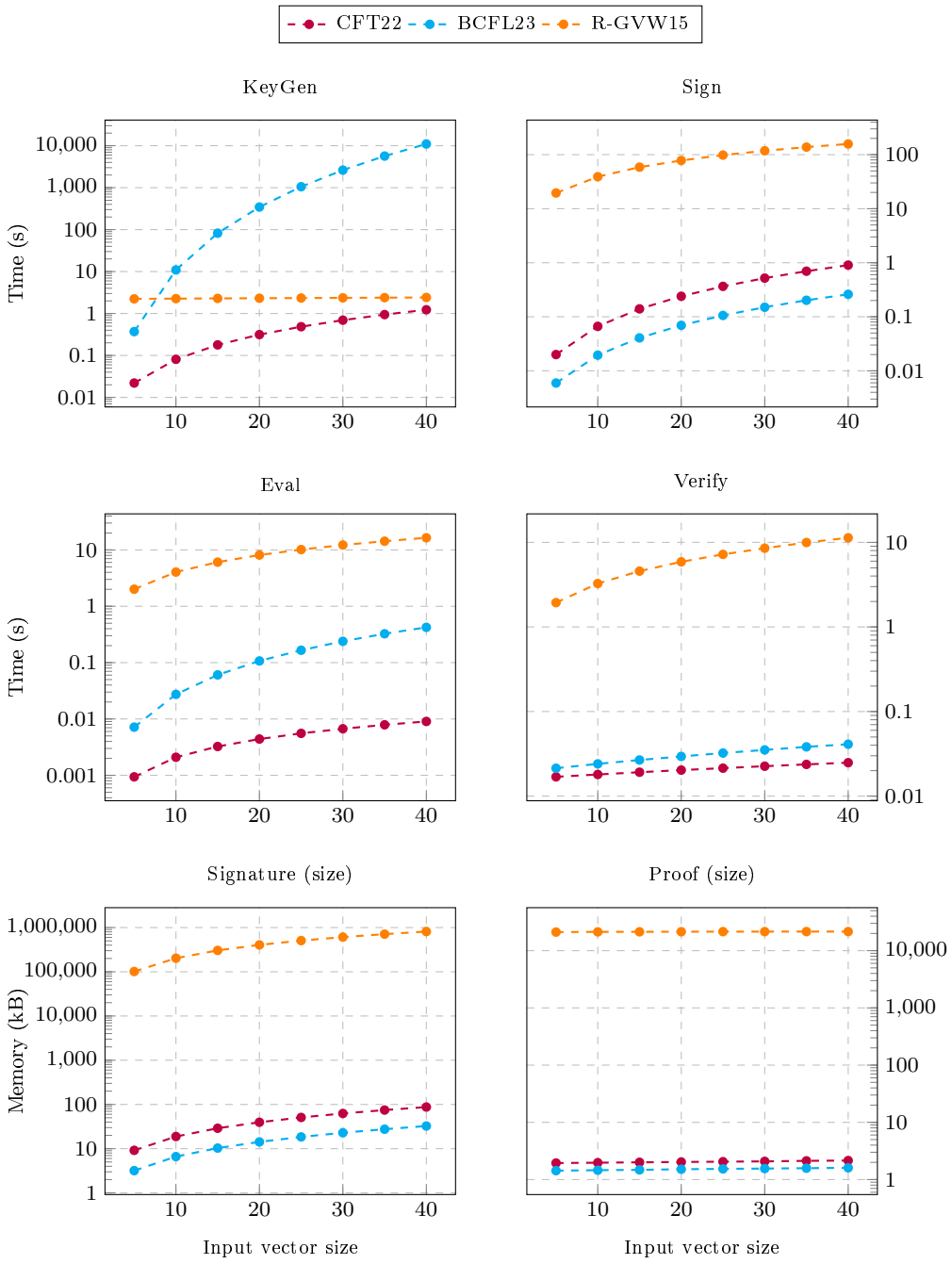


Fig. 3: Optimized sequential benchmarks for the weighted sum function. The input vector size ranges from $n = 5$ to $n = 40$. The first four panels report the running time of KeyGen, Sign, Eval and Verify in seconds; the last two panels report the sizes of the generated signatures and proofs in kB. All vertical axes use a logarithmic scale to make the comparison between CFT22, BCFL23 and R-GVW15 readable across the full range of measurements.

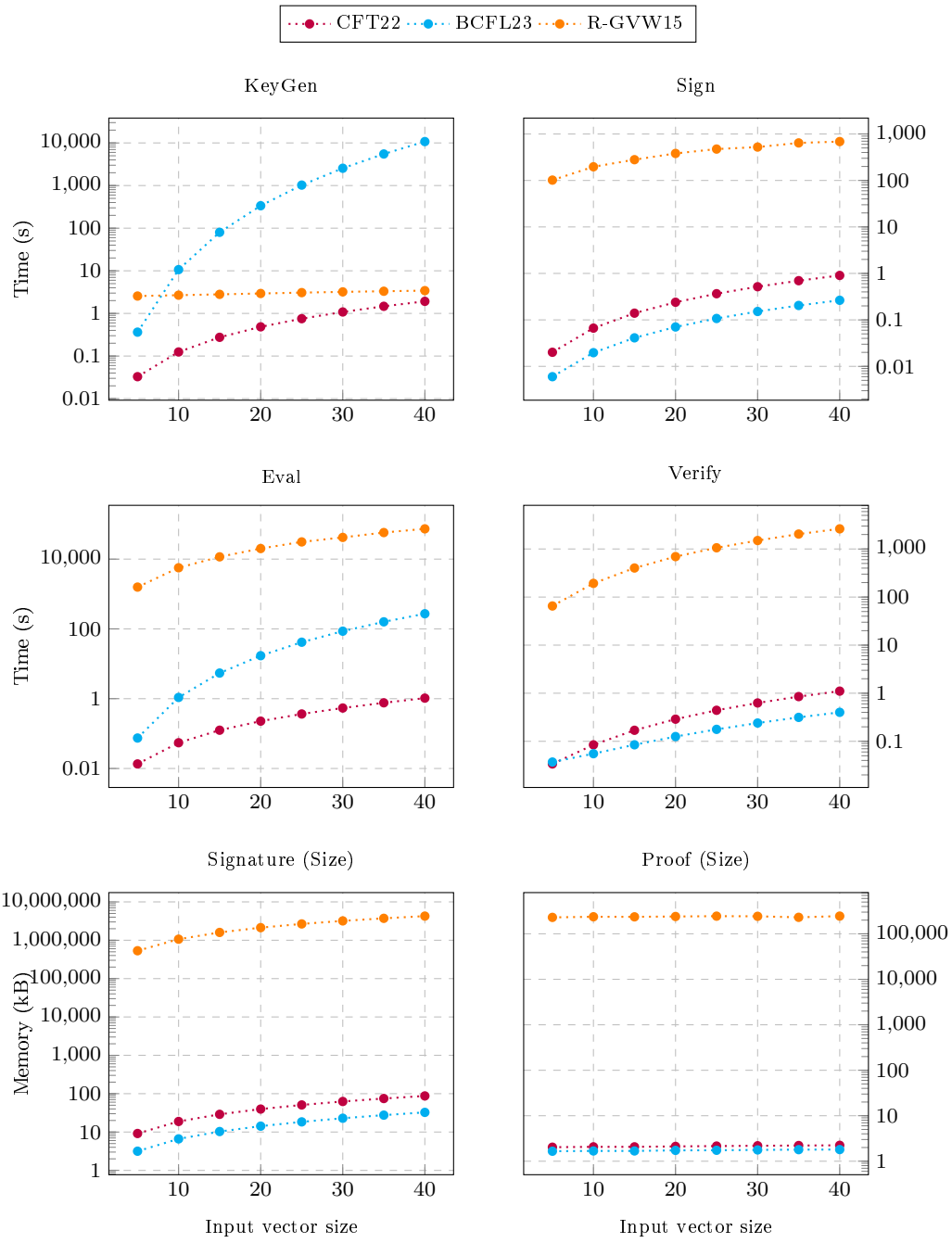


Fig. 4: Optimized sequential benchmarks for the variance function. The input vector size ranges from $n = 5$ to $n = 40$. The first four panels report the running time of KeyGen, Sign, Eval and Verify in seconds; the last two panels report the sizes of the generated signatures and proofs in kB. All vertical axes use a logarithmic scale to make the comparison between CFT22, BCFL23 and R-GVW15 readable across the full range of measurements.

Function	Scheme	Mode	Precomp.	1	2	4	8	16	32
Weighted sum	CFT22	Baseline	–	20.36 ms	40.71 ms	81.48 ms	163.2 ms	326.2 ms	653.0 ms
		Amortized	4.437 ms	15.90 ms	31.84 ms	63.76 ms	127.5 ms	255.4 ms	511.1 ms
	BCFL23	Baseline	–	29.51 ms	59.04 ms	118.3 ms	236.7 ms	473.4 ms	946.6 ms
		Amortized	7.693 ms	21.74 ms	43.50 ms	87.05 ms	174.5 ms	348.7 ms	696.5 ms
	R-GVW15	Baseline	–	5.987 s	11.97 s	23.95 s	47.89 s	1.597 m	3.191 m
		Amortized	5.343 s	630.6 ms	1.260 s	2.522 s	5.041 s	10.08 s	20.15 s
Variance	CFT22	Baseline	–	286.9 ms	573.5 ms	1.147 s	2.294 s	4.589 s	9.181 s
		Amortized	270.1 ms	17.16 ms	34.31 ms	68.63 ms	137.2 ms	274.4 ms	548.7 ms
	BCFL23	Baseline	–	125.0 ms	249.8 ms	499.7 ms	1.000 s	2.000 s	4.000 s
		Amortized	91.87 ms	33.00 ms	66.00 ms	132.0 ms	264.0 ms	528.0 ms	1.056 s
	R-GVW15	Baseline	–	11.72 m	23.57 m	47.57 m	1.586 h	3.112 h	6.229 h
		Amortized	11.44 m	8.506 s	17.01 s	34.06 s	1.136 m	2.281 m	4.596 m

Table 3: Amortized verification benchmarks for the weighted sum and the variance, with input size $n = 20$. *Precomp.* is the one-time preprocessing cost for the fixed function. The remaining columns report the total online verification time for batches of repeated verifications of the same function over different datasets. The amortized total cost for a batch is obtained by adding the corresponding preprocessing time once. All values are medians over the collected samples.

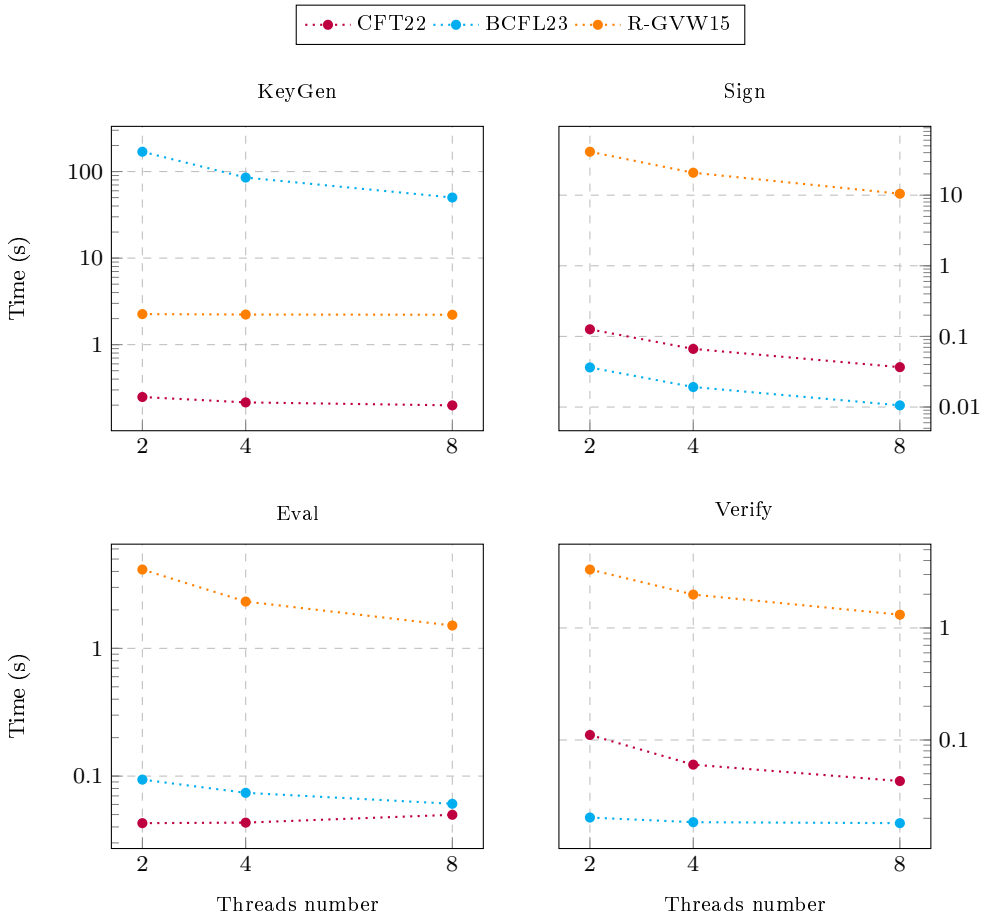


Fig. 5: Multi-thread benchmarks for the weighted sum function with fixed input size $n = 20$. The horizontal axis shows the number of worker threads, while the vertical axis reports the running time in seconds on a logarithmic scale. The four panels compare the parallel behaviour of KeyGen, Sign, Eval and Verify for CFT22, BCFL23 and R-GVW15.

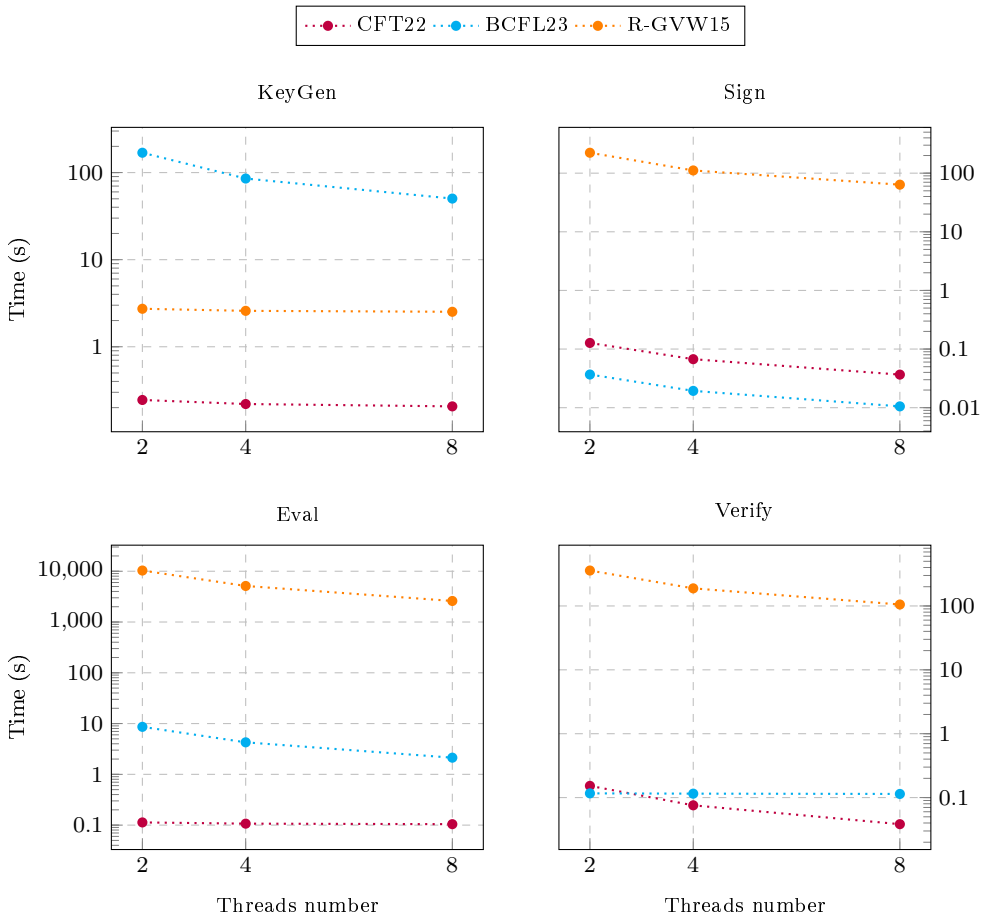


Fig. 6: Multi-thread benchmarks for the variance function with fixed input size $n = 20$. The horizontal axis shows the number of worker threads, while the vertical axis reports the running time in seconds on a logarithmic scale. The four panels compare the parallel behaviour of KeyGen, Sign, Eval and Verify for CFT22, BCFL23 and R-GVW15.

References

1. Y. Desmedt, in *Proceedings on the 1992-1993 Workshop on New Security Paradigms* (Association for Computing Machinery, New York, NY, USA, 1993), NSPW '92-93, p. 160–166. DOI 10.1145/283751.283834
2. R. Johnson, D. Molnar, D.X. Song, D.A. Wagner, in *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*, *Lecture Notes in Computer Science*, vol. 2271 (Springer, 2002), *Lecture Notes in Computer Science*, vol. 2271, pp. 244–262
3. D. Boneh, D.M. Freeman, in *EUROCRYPT 2011*, *LNCS*, vol. 6632, ed. by K.G. Paterson (Springer, Heidelberg, 2011), *LNCS*, vol. 6632, pp. 149–168. DOI 10.1007/978-3-642-20465-4_10
4. D. Catalano, D. Fiore, L. Nizzardo, in *ACNS 18*, *LNCS*, vol. 10892, ed. by B. Preneel, F. Vercauteren (Springer, Heidelberg, 2018), *LNCS*, vol. 10892, pp. 183–201. DOI 10.1007/978-3-319-93387-0_10
5. D. Boneh, D. Freeman, J. Katz, B. Waters, in *PKC 2009*, *LNCS*, vol. 5443, ed. by S. Jarecki, G. Tsudik (Springer, Heidelberg, 2009), *LNCS*, vol. 5443, pp. 68–87. DOI 10.1007/978-3-642-00468-1_5
6. S. Gorbunov, V. Vaikuntanathan, D. Wichs, in *47th ACM STOC*, ed. by R.A. Servedio, R. Rubinfeld (ACM Press, 2015), pp. 469–477. DOI 10.1145/2746539.2746576
7. D. Balbás, D. Catalano, D. Fiore, R.W.F. Lai, in *Theory of Cryptography - 21st International Conference, TCC 2023, Taipei, Taiwan, November 29 - December 2, 2023, Proceedings, Part III* (2023), pp. 363–393. DOI 10.1007/978-3-031-48621-0_13. URL https://doi.org/10.1007/978-3-031-48621-0_13
8. D. Catalano, D. Fiore, I. Tucker, in *ASIACRYPT 2022, Part IV*, *LNCS*, vol. 13794, ed. by S. Agrawal, D. Lin (Springer, Heidelberg, 2022), *LNCS*, vol. 13794, pp. 159–188. DOI 10.1007/978-3-031-22972-5_6
9. L. de Castro, C. Peikert, in *EUROCRYPT 2023, Part III*, *LNCS*, vol. 14006, ed. by C. Hazay, M. Stam (Springer, Heidelberg, 2023), *LNCS*, vol. 14006, pp. 287–320. DOI 10.1007/978-3-031-30620-4_10
10. H. Wee, D.J. Wu, in *EUROCRYPT 2023, Part III*, *LNCS*, vol. 14006, ed. by C. Hazay, M. Stam (Springer, Heidelberg, 2023), *LNCS*, vol. 14006, pp. 385–416. DOI 10.1007/978-3-031-30620-4_13
11. H. Wee, D.J. Wu, in *ASIACRYPT 2023, Part V*, *LNCS*, vol. 14442, ed. by J. Guo, R. Steinfeld (Springer, Heidelberg, 2023), *LNCS*, vol. 14442, pp. 201–235. DOI 10.1007/978-981-99-8733-7_7
12. M. Beckmann, P. Milewski, L. Porzenheim, M.L. Schmidt, J.N. Siemer. qFALL – rapid prototyping of lattice-based cryptography. Cryptology ePrint Archive, Paper 2026/069 (2026). URL https://eprint.iacr.org/2026/069
13. Supranational. blst. Online: https://github.com/supranational/blst (2020)
14. R. Gennaro, J. Katz, H. Krawczyk, T. Rabin, in *PKC 2010*, *LNCS*, vol. 6056, ed. by P.Q. Nguyen, D. Pointcheval (Springer, Heidelberg, 2010), *LNCS*, vol. 6056, pp. 142–160. DOI 10.1007/978-3-642-13013-7_9
15. D. Boneh, D.M. Freeman, in *PKC 2011*, *LNCS*, vol. 6571, ed. by D. Catalano, N. Fazio, R. Gennaro, A. Nicolosi (Springer, Heidelberg, 2011), *LNCS*, vol. 6571, pp. 1–16. DOI 10.1007/978-3-642-19379-8_1
16. D. Catalano, D. Fiore, R. Gennaro, K. Vamvourellis, in *TCC 2013*, *LNCS*, vol. 7785, ed. by A. Sahai (Springer, Heidelberg, 2013), *LNCS*, vol. 7785, pp. 680–699. DOI 10.1007/978-3-642-36594-2_38
17. N. Attrapadung, B. Libert, in *PKC 2011*, *LNCS*, vol. 6571, ed. by D. Catalano, N. Fazio, R. Gennaro, A. Nicolosi (Springer, Heidelberg, 2011), *LNCS*, vol. 6571, pp. 17–34. DOI 10.1007/978-3-642-19379-8_2
18. D. Catalano, D. Fiore, B. Warinschi, in *EUROCRYPT 2011*, *LNCS*, vol. 6632, ed. by K.G. Paterson (Springer, Heidelberg, 2011), *LNCS*, vol. 6632, pp. 207–223. DOI 10.1007/978-3-642-20465-4_13
19. D.M. Freeman, in *PKC 2012*, *LNCS*, vol. 7293, ed. by M. Fischlin, J. Buchmann, M. Manulis (Springer, Heidelberg, 2012), *LNCS*, vol. 7293, pp. 697–714. DOI 10.1007/978-3-642-30057-8_41
20. D. Catalano, D. Fiore, B. Warinschi, in *PKC 2012*, *LNCS*, vol. 7293, ed. by M. Fischlin, J. Buchmann, M. Manulis (Springer, Heidelberg, 2012), *LNCS*, vol. 7293, pp. 680–696. DOI 10.1007/978-3-642-30057-8_40
21. N. Attrapadung, B. Libert, T. Peters, in *PKC 2013*, *LNCS*, vol. 7778, ed. by K. Kurosawa, G. Hanaoka (Springer, Heidelberg, 2013), *LNCS*, vol. 7778, pp. 386–404. DOI 10.1007/978-3-642-36362-7_24
22. D. Catalano, D. Fiore, L. Nizzardo, in *CRYPTO 2015, Part II*, *LNCS*, vol. 9216, ed. by R. Gennaro, M.J.B. Robshaw (Springer, Heidelberg, 2015), *LNCS*, vol. 9216, pp. 254–274. DOI 10.1007/978-3-662-48000-7_13

23. S. Agrawal, D. Boneh, in *ACNS 09*, *LNCS*, vol. 5536, ed. by M. Abdalla, D. Pointcheval, P.A. Fouque, D. Vergnaud (Springer, Heidelberg, 2009), *LNCS*, vol. 5536, pp. 292–305. DOI 10.1007/978-3-642-01957-9_18
24. R. Gennaro, D. Wichs, in *ASIACRYPT 2013, Part II*, *LNCS*, vol. 8270, ed. by K. Sako, P. Sarkar (Springer, Heidelberg, 2013), *LNCS*, vol. 8270, pp. 301–320. DOI 10.1007/978-3-642-42045-0_16
25. D. Catalano, D. Fiore, in *EUROCRYPT 2013*, *LNCS*, vol. 7881, ed. by T. Johansson, P.Q. Nguyen (Springer, Heidelberg, 2013), *LNCS*, vol. 7881, pp. 336–352. DOI 10.1007/978-3-642-38348-9_21
26. M. Backes, D. Fiore, R.M. Reischuk, in *ACM CCS 2013*, ed. by A.R. Sadeghi, V.D. Gligor, M. Yung (ACM Press, 2013), pp. 863–874. DOI 10.1145/2508859.2516681
27. D. Catalano, D. Fiore, R. Gennaro, L. Nizzardo, in *PKC 2014*, *LNCS*, vol. 8383, ed. by H. Krawczyk (Springer, Heidelberg, 2014), *LNCS*, vol. 8383, pp. 538–555. DOI 10.1007/978-3-642-54631-0_31
28. D. Fiore, I. Tucker, in *ACM CCS 2022*, ed. by H. Yin, A. Stavrou, C. Cremers, E. Shi (ACM Press, 2022), pp. 1067–1080. DOI 10.1145/3548606.3560630
29. B. Libert, T. Peters, M. Joye, M. Yung, in *CRYPTO 2013, Part II*, *LNCS*, vol. 8043, ed. by R. Canetti, J.A. Garay (Springer, Heidelberg, 2013), *LNCS*, vol. 8043, pp. 289–307. DOI 10.1007/978-3-642-40084-1_17
30. B. Libert, T. Peters, M. Joye, M. Yung, in *EUROCRYPT 2014*, *LNCS*, vol. 8441, ed. by P.Q. Nguyen, E. Oswald (Springer, Heidelberg, 2014), *LNCS*, vol. 8441, pp. 514–532. DOI 10.1007/978-3-642-55220-5_29
31. N. Attrapadung, B. Libert, T. Peters, in *ASIACRYPT 2012*, *LNCS*, vol. 7658, ed. by X. Wang, K. Sako (Springer, Heidelberg, 2012), *LNCS*, vol. 7658, pp. 367–385. DOI 10.1007/978-3-642-34961-4_23
32. B. Libert, S.C. Ramanna, M. Yung, in *ICALP 2016*, *LIPIcs*, vol. 55, ed. by I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, D. Sangiorgi (Schloss Dagstuhl, 2016), *LIPIcs*, vol. 55, pp. 30:1–30:14. DOI 10.4230/LIPIcs.ICALP.2016.30
33. R.W.F. Lai, G. Malavolta, in *CRYPTO 2019, Part I*, *LNCS*, vol. 11692, ed. by A. Boldyreva, D. Micciancio (Springer, Heidelberg, 2019), *LNCS*, vol. 11692, pp. 530–560. DOI 10.1007/978-3-030-26948-7_19
34. D. Catalano, A. Marcedone, O. Puglisi, in *ASIACRYPT 2014, Part II*, *LNCS*, vol. 8874, ed. by P. Sarkar, T. Iwata (Springer, Heidelberg, 2014), *LNCS*, vol. 8874, pp. 193–212. DOI 10.1007/978-3-662-45608-8_11
35. Protocol Labs. blstrs. Online: https://github.com/filecoin-project/blstrs (2020)
36. P.S.L.M. Barreto, B. Lynn, M. Scott, in *SCN 02*, *LNCS*, vol. 2576, ed. by S. Cimato, C. Galdi, G. Persiano (Springer, Heidelberg, 2003), *LNCS*, vol. 2576, pp. 257–267. DOI 10.1007/3-540-36413-7_19
37. T. Kim, R. Barbulescu, in *CRYPTO 2016, Part I*, *LNCS*, vol. 9814, ed. by M. Robshaw, J. Katz (Springer, Heidelberg, 2016), *LNCS*, vol. 9814, pp. 543–571. DOI 10.1007/978-3-662-53018-4_20
38. R. Barbulescu, S. Duquesne, Journal of Cryptology **32**(4), 1298 (2019). DOI 10.1007/s00145-018-9280-5
39. D. Boneh, B. Lynn, H. Shacham, Journal of Cryptology **17**(4), 297 (2004). DOI 10.1007/s00145-004-0314-9
40. D. Micciancio, C. Peikert, in *EUROCRYPT 2012*, *LNCS*, vol. 7237, ed. by D. Pointcheval, T. Johansson (Springer, Heidelberg, 2012), *LNCS*, vol. 7237, pp. 700–718. DOI 10.1007/978-3-642-29011-4_41
41. T.F. team, *FLINT: Fast Library for Number Theory*. https://flintlib.org
42. M.R. Albrecht, R. Player, S. Scott, Journal of Mathematical Cryptology **9**(3), 169–203 (2015). DOI 10.1515/jmc-2015-0016. URL http://dx.doi.org/10.1515/jmc-2015-0016
43. N. Vazquez. Divan. Online: https://github.com/nvzqz/divan
44. J. Chen, J. Revels. Robust benchmarking in noisy environments (2016)
45. C. Bormann, P.E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949 (2020). DOI 10.17487/RFC8949. URL https://www.rfc-editor.org/info/rfc8949