On the Efficiency and Flexibility of Signature Verification

Cecilia Boschini[1], Dario Fiore[2], and Elena Pagnin[3]

[1] Università della Svizzera Italiana, Lugano, Switzerland
cecilia.boschini@usi.ch
[2] IMDEA Software Institute, Madrid, Spain
dario.fiore@imdea.org
[3] Lund University, Lund, Sweden
elena.pagnin@eit.lth.se

Abstract. For decades signature verification has been regarded as a unique, monolithic process. Here, we want to look at it with fresh eyes and pose two fundamental questions: *(1) is it possible to extract meaningful information from a partial signature verification?* (flexibility); and *(2) is it possible to speed up the verification process without impacting unforgeability?* (efficiency). We answer both questions in a positive way for specific classes of post-quantum secure schemes.
In detail, we develop formal frameworks for signatures with efficient verification, flexible verification and combinations of the two. Crucially, we regard these as features that may enhance existing constructions. Flexibility is of particular interest as standard verification cannot provide any meaningful information about the validity of a given signature if interrupted *in media res*. We exhibit generic transformations to realize efficient (and) flexible verification for schemes that involve matrix-vector multiplications among the verification checks.
In addition, we present concrete instantiations of efficient (and) flexible verification for Rainbow [ACNS05] (as representative of schemes based on multivariate quadratic equations), MP [EC12] and GVW [STOC15] (as representative of lattice-based constructions). Interestingly, we are able to *efficiently verify* Rainbow signatures using 50% of the original computational cost, and as little as 0.4% for GVW homomorphic signatures, provided a one-time preprocessing and with only negligible impact on security.

Keywords: Digital signatures, amortized efficiency, flexible verification, post quantum signatures.

1 Introduction

Digital technologies gained a fundamental role in our society, affecting the security of crucial systems such as autonomous vehicles, healthcare, payments, e-voting, and access control. Digital signatures are among the cryptographic primitives employed to safeguard such systems from misuse [11,13,26,33]. Concretely, digital signatures allow one party, the signer, to use her secret key to authenticate a message in such a way that anyone holding the corresponding public key, the verifiers, can check its validity at any later point in time. Among many properties, digital signatures serve the purpose of securely establishing the source and integrity of the information. In many applications, the outcome of a signature verification determines what decisions to take, e.g., whether to accept a financial transaction (Bitcoin protocol), install software updates (Android OS), or deliver e-services (e-Health, electronic tax systems).

While the above examples are not time-critical, digital signatures may be employed in real-time cyber-physical systems as well, where the speed at which verification is performed plays a crucial role. This is the case, e.g., for automatic safety measures in connected vehicles or nuclear power plants. In real-time systems, verification speed (due to resource constraints) is not the only problem though. For a variety of reasons, a computation can get arbitrarily interrupted, leaving the verifier with an unsolved answer about the validity of a given signature. This has to do with the fact that standard verification procedures provide a binary outcome (0 or 1, reject or accept), which is established only at the very end of the execution. One may wonder: *is it possible to extract meaningful information from a partial signature verification?* Le et al. [25] proposed to address unexpected interruptions using signatures with flexible verification. In a nutshell, such schemes admit a verification algorithm that increasingly builds confidence on the validity of the signature while it performs more steps. In this way, at the moment of an interrupt, the verifier is left with a value $\alpha \in [0, 1] \cup \perp$ that probabilistically quantifies the validity of the signature. In particular, flexible signatures identify tradeoffs between the amount of computation performed and the integrity of a signature. A different way to approach this problem would be to limit the verification to few, quick steps, as to reduce the chance of interruptions in media res. One may wonder: *is it possible to speed up the verification process without impacting unforgeability?* This is the aim of efficient verification: to leverage special tradeoffs to quickly reach accurate conclusions about the validity of a signature.

1.1 Our contribution

In this work, we address two main challenges: (1) speeding up the verification of digital signatures; and (2) extracting meaningful information from a partial signature verification. We do so by designing *new* verification methods for *existing* signature schemes. Namely, our goal is to keep the signatures (i.e., key generation, sign and verify algorithms) as they are, and to devise techniques for alternative verification methods that are significantly more efficient and/or that retain usefulness even when interrupted *in media res*.

We introduce formal security models for the notions of efficient verification and flexible verification. For both settings, we provide generic compilers and show how these directly apply to a wide range existing signature schemes. Moreover, our compilers can be combined to simultaneously achieve flexible and efficient verification for which we present a formal model, generic realizations, security arguments and implications. Finally, we remark that our models can easily be extended to include signatures with advanced properties including: ring, threshold, homomorphic, attribute-based and constrained.

We focus our realizations on two families of digital signatures: lattice-based and multivariate-polynomials-based. Both are interesting examples thanks to their plausibly post-quantum security. The multivariate-polynomials-based scheme we consider is Rainbow [15,14], one of the NIST candidates for standardization , and LUOV [5]. Our main representatives for lattice-based constructions are GPV [20] (hash & sign), MP [28] (Boyen/ BonsaiTree) and GVW [21] (homomorphic).

It is worth noting that finding faster verification for (families of) existing and very well-studied signature schemes may be a too ambitious goal (e.g., it may require major algorithmic breakthroughs such as finding faster matrix multiplication algorithms). In our paper, we therefore address this problem by introducing new verification models based on a probabilistic secret-coins preprocessing.

Efficient Verification. We build on the well-known offline/online paradigm to define an alternative verification that consists of two phases. An offline probabilistic preprocessing offVer that, given only the public key and a security parameter, outputs a concise (short) verification key svk. An online algorithm onVer that uses such verification key to establish the validity of a signature on a given message. In our paradigm, (i) the preprocessing is one-time, namely its output svk can be reused to verify an unbounded number of signatures, (ii) the running time of onVer is required to be asymptotically smaller than that of the standard verification algorithm, and (iii) the preprocessed verification key svk must be kept secret by the verifier. While the first two properties (i)-(ii) are clearly the ones that provide the desired efficiency boost, one may wonder if property (iii) is really needed. To this end, we observe that devising algorithms $\mathsf{offVer}, \mathsf{onVer}$ with efficient verification (i)-(ii) and a *public* svk for an existing signature scheme would immediately lead to finding a new version of the signature scheme with faster verification. We therefore consider keeping svk secret and trade some usability for more efficiency. We formalize this secrecy requirement in a rigorous security model in which the standard unforgeability experiment is extended by asking the adversary to produce a forgery $(\mu, \boldsymbol{\sigma})$ that verifies under $\mathsf{onVer}(\mathsf{svk}, \cdot, \cdot)$ and by giving it oracle access to the online verification with the same preprocessed svk. This experiment essentially models that learning outcomes of verification do not leak enough information about svk and do not help the adversary in finding a forgery. We stress that *in our model signatures are still publicly verifiable*; svk is only a secret that the verifier generates for itself, in order to speedup its computation.

In terms of realizations, we propose efficient verification algorithms that work for a broad class of digital signatures in which the verification includes a matrix-vector multiplication. This can model the $\mathbf{A} \cdot \boldsymbol{\sigma} = \mathbf{u}$ check of several lattice-based signatures (LBS) where matrix $\mathbf{A}$ may be a public-key constant and $\mathbf{u}$ be message-dependent, as in [16,20,21], or $\mathbf{A}$ may be message-dependent and $\mathbf{u}$ be constant [8,9,28]. Or it can model the check of a system of multivariate quadratic polynomials $\{f_i(\mathbf{s}) = h_i\}_i$, where $\mathbf{s}$ depends on the signature, h_i is message-dependent and $f_i(\cdot)$ is a public-key-dependent polynomial. For example, for the LBS instantiations we consider, assuming a lattice of dimension $n = 128$, our method offers an online verification that is between $97\times$ to $99\times$ faster.

Flexible Verification. We consider flexible verification as an *add-on* property to existing signature schemes. Concretely, we devise flexible verification via an algorithm $\mathsf{flexVer}$ and a confidence function α_{flex} with the following properties. First, $\mathsf{flexVer}$ is stateful, uses private-coins, and consists of at most $N + 1$ inner steps, so that at the end of step i one is either sure about rejection or expects the signature to be valid with confidence $\alpha_{\mathsf{flex}}(i)$. Second, $\alpha_{\mathsf{flex}} : \{0, \ldots, N\} \to [0, 1]$ is a non-decreasing function (that depends both on the signature scheme and on $\mathsf{flexVer}$) satisfying $\alpha_{\mathsf{flex}}(0) = 0$ and $\alpha_{\mathsf{flex}}(N) = (1 - \varepsilon(\lambda))$. In particular, reaching the last flexible verification step provides the same security guarantees as standard verification (except for a negligibe chance of error). Third, we model secure flexible verification by letting the adversary decide at which step to interrupt each execution of $\mathsf{flexVer}$, even in the forgery check. Since in such a case a wrong signature may be marked as 'maybe valid' with non negligible probability (as the adversary may choose to verify it with not enough steps to reach overwhelming confidence), we require the adversary to find a forgery that is not rejected with probability non-negligibly *higher* than the expected confidence level at the chosen interruption step i, i.e., $\alpha_{\mathsf{flex}}(i)$.

In terms of realizations, we again focus on the aforementioned class of signatures with a matrix-vector multiplication and we propose flexible verification algorithms for them. In a nutshell, for signatures that work over $\mathbb{Z}_q$ or $\mathbb{F}_q$, our flexible verification interrupted at step i can achieve confidence $(1 - q^{-i})$. In the case of signature schemes where $q = 2^{poly(\lambda)}$, this confidence level is always overwhelming, even with minimal computation, i.e., $i = 1$. Interestingly, in this case our $\mathsf{flexVer}$ algorithm also satisfies efficient verification, namely, there is a minimum number N so that N executions of $\mathsf{flexVer}$ are faster than N executions of Ver. For signature schemes where $q = \mathsf{poly}(\lambda)$, we show a variant of our compiler that achieves both efficient and flexible verification. This though holds in a weaker security model, in which the confidence function α_{flex} depends, and degrades, with the number of verifications.

1.2 An Overview of Our Techniques

Our Compiler for Efficient Verification. We consider the class of digital signature schemes for which the bulk of computation in the verification procedure consists of a matrix-vector product (what we call '$\mathbf{Mv}$'-style check). Several lattice-based [6,21,16,20,28] and multivariate-polynomials-based signatures [15,5,29] fall in this category. For simplicity, here we give a technical overview for LBS constructions only, for the case $\mathbf{Mv} = \mathbf{u}$, where $\mathbf{M} \in \mathbb{Z}_q^{n\times m}$ is fixed (e.g., the signer's public key), $\mathbf{v}$ is a vector (e.g., a signature) and $\mathbf{u}$ depends on the message [16,20,21]. The key observation to efficient verification is that if a signature $\mathbf{v}$ verifies for a given $\mathbf{M}$ (and $\mathbf{u}$), then it also verifies for any linear combination of the rows of $\mathbf{M}$ (against the same combination to the entries of $\mathbf{u}$). Let $\mathbf{c} \xleftarrow{\$} \mathbb{Z}_q^n$ be a random vector; denote $\mathbf{z} \leftarrow \mathbf{c}\cdot\mathbf{M} \in \mathbb{Z}_q^m$, and $w \leftarrow \mathbf{c}\cdot\mathbf{u} = \sum_{i=1}^n \mathbf{c}[i]\mathbf{u}[i] \in \mathbb{Z}_q$. Then

$$\mathbf{Mv} = \mathbf{u} \mod q \Longrightarrow \mathbf{c}\cdot(\mathbf{M}\boldsymbol{\sigma}) = \mathbf{c}\cdot\mathbf{u} \mod q \Leftrightarrow \mathbf{z}\cdot\mathbf{v} = w \mod q \qquad (1)$$

Our compiler is based on the fact that the first implication in (1) holds left-to-rightwards (always) *and also* right-to-leftwards with *good enough* probability. Moreover, one can adjust this probability by increasing the number of vectors $\mathbf{c}$ and $\mathbf{z}$ to use in the signature verification. Now, we can precompute a set of $k \geq 1$ 'random combinations of rows of $\mathbf{M}$' during an 'offline verification' phase; and re-use the output pairs $(\mathbf{c}_j, \mathbf{z}_j)$, $j \in \{1,\dots,k\}$ to efficiently verify an unbounded number of signatures. Concretely, the efficiency gain in the online verification comes by replacing the check $\mathbf{Mv}$ (that requires nm multiplications modulus q) with k checks of the form $\mathbf{z}_j \cdot \mathbf{v} = w_j$ (that require at most km multiplications). Remarkably, the leftwards implication in (1) holds with all but negligible probability with very small values of k, and even with $k = 1$ for schemes where $q = 2^{\mathsf{poly}(\lambda)}$, e.g., [16,21].

Our Compiler for Flexible Verification. The idea of the compiler presented above can be used to achieve flexible verification in the following way. For a set of freshly sampled vectors $\{\mathbf{c}_j\}_{j=1}^J$, a verification that uses a subset of $k < J$ such vectors is correct with probability $(1 - q^{-k})$; hence this value can be the confidence, i.e., $\alpha_{\mathsf{flex}}(k) = (1 - q^{-k})$.

1.3 Related Work

The problem of trading security for less computation during a signature verification has been considered by Le et al. [25] who introduced the notion of flexible signatures and a construction based on the Lamport-Diffie one-time signature [24] with Merkle trees [27]. A similar idea in the context of message authentication codes (MACs) has been considered by Fischlin [17] who put forth progressive verification for MACs and presented two concrete constructions. One main difference between our model and those of [17,25] is that we aim to capture flexible verification as an independent feature that can enhance existing schemes, rather than a standalone primitive. This is in a way more challenging as it leaves less design freedom when crafting these algorithms. Therefore we decided to define flexible verification as a *stateful* algorithm in contrast to stateless [17,25]: although this makes our model slightly more involved, it is comparably more general and can capture more (existing) schemes.

Our model for efficient verification is close the offline-online paradigm used in homomorphic authentication [2,10] and verifiable computation [19]; where a preprocessing is done with respect to a function f, and its result can be used to verify computation results involving the same f.

A recent work by Sipasseuth et al. [31] investigates how to speed up lattice-based signature verification while reducing the memory (storage) requirements. The overall idea in [31] is similar to ours (and inspired to Freivalds' Algorithm): to replace the inefficient matrix multiplication in the verification with a probabilistic check via an inner product computation. However, [31] focuses on a concrete construction, the DRS signature [30], and investigates the trade-off between pre-computation time for verification and memory storage for this scheme only. Moreover, the work lacks a formal, abstract analysis of the security impact of such a shift in the verification procedure. In contrast, we devise a general framework to model 'more efficient' and 'partial' signature verification. Albeit we developed our approach independently of [31], our techniques can be seen as a generalization of what presented in [31].

Independently from us, Taleb and Vergnaud recently published a paper about speeding up signature verification [32]. The paper analyzes three kinds of signatures, the RSA and ECDSA signatures and the lattice-based signature from [20]. While the first two are complementary to our work, the latter construction is a different take on achieving efficient verification for this class of LBS (they do not have a comparable approach for the flexible case). Their approach exploits a particular type of error correcting codes that do not have words with small Hamming weight. Given a generator matrix $\mathbf{G}$ for the code, their verification algorithm checks that the codeword $\mathbf{G}(\mathbf{A}\boldsymbol{\sigma}-\mathbf{u}) \mod q$ has null components. The sparsity property of the code ensures that if $\mathbf{A}\boldsymbol{\sigma} - \mathbf{u} \neq 0 \mod q$ (i.e., $\boldsymbol{\sigma}$ is not a valid signature), the corresponding codeword has enough nonzero components that a random choice of them would contain a nonzero one with overwhelming probability. The drawback is that such construction is not a compiler, as ours, but requires to modify both the key generation and the verification algorithm, and requires the public key to be longer, thus slowing down the full verification. Moreover, their progressive verification still requires a number of vector multiplications linear in the security parameter, while ours only requires a logarithmic number of linear products.

Remark on terms. We use the term 'flexible signature' to be consistent with previous work [25], however, this wording may be misleading in that there is no flexibility in the signature scheme, but rather the verification procedure is carried out in a progressive way (á la Fischlin [17]) in the sense that the more computation one performs the more confident one becomes of accepting or rejecting a signature (gradually approaching 100% certainty). We believe that an algorithm should be able to read (and understand) its input, and should not have an accessory input that cannot use (the $[[k]]$ of [25]).

2 Preliminaries

We denote the set of real values by $\mathbb{R}$, integers by $\mathbb{Z}$, natural numbers by $\mathbb{Z}_{\geq 0}$, and finite fields of integers by $\mathbb{Z}_q$, where q is a (power of a) prime number. We denote vectors by bold, lower-case letters, and matrices by bold, upper-case letters. We use $\mathbf{v}[i]$ to identify the i-th entry of a vector $\mathbf{v}$, and $\mathbf{A}[i,j]$ to identify the entry in the i-th row and j-th column of a matrix $\mathbf{A}$. The norm of a vector is denoted as $\|\mathbf{v}\|$ and unless otherwise specified, it is assumed to be the infinity norm, i.e., $\|\mathbf{v}\| = \max_i\{\mathbf{v}[i]\}$. $\mathbf{A}^T$ denotes the transposed of a matrix. We use $rows(\mathbf{A})$, $cols(\mathbf{A})$, and $rk(\mathbf{A})$ to respectively refer to the number of rows, the number of columns, and the rank of a matrix $\mathbf{A}$; $\mathbf{1}_{1\times n}$ (resp. $\mathbf{0}_{1\times n}$) denotes the row vector of length n that has all entries equal to 1 (resp. 0); while $\mathbf{I}_n$ denotes the n by n identity matrix of dimension n. We omit the explicit dimensions when they are clear from the context. We denote by $L_1|L_2$ the result of appending a list of elements L_2 to L_1. Given two values $a < b$, we denote a continuous interval as $[a,b] \subseteq \mathbb{R}$, and a discrete interval as $\{a,\dots,b\} \subseteq \mathbb{Z}$. A function $\varepsilon : \mathbb{Z}_{\geq 0} \to [0,1]$ is negligible if $\varepsilon(\lambda) < 1/\mathsf{poly}(\lambda)$ for every univariate polynomial $\mathsf{poly} \in \mathbb{R}[\mathsf{X}]$ and a large enough integer $\lambda \in \mathbb{Z}_{\geq 0}$. Throughout the paper,

$\lambda \in \mathbb{Z}_{\geq 0}$ denotes the security parameter of a cryptographic scheme. The abbreviation PPT refers to algorithms that are probabilistic and run in polynomial time.

Definition 1 (Signature Scheme). *A signature scheme Σ a tuple of PPT algorithms $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ defined as follows.*

$\mathsf{KeyGen}(1^\lambda)$**:** *The key generation algorithm takes in input the security parameter and outputs a pair of keys* $(\mathsf{sk}, \mathsf{pk})$.

$\mathsf{Sign}(\mathsf{sk}, \mu)$**:** *The sign algorithm takes in input a secret key* sk *and a message μ; it outputs a signature $\boldsymbol{\sigma}$.*

$\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})$**:** *The verification algorithm takes in input a public key, a message μ and signature $\boldsymbol{\sigma}$. It outputs 0 (reject) or 1 (accept).*

Correctness For a given security parameter λ, for any key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, for any $\mu \in \mathcal{M}$, for any $\boldsymbol{\sigma} \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$, it holds that

$$Prob\big[\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 1\big] = 1.$$

We use the expression 'valid (resp. invalid) signature' to identify signatures that are accepted (resp. rejected) by the verification procedure.

Security The basic security requirement for a digital signature scheme is unforgeability. In a nutshell this notion states that an adversary should not be able to produce a valid signature without knowledge of the secret signing key.

2.1 Lattices

Let $n, m \in \mathbb{Z}_{>0}$, and $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ be a uniformly chosen matrix. We define the lattice that has $\mathbf{A}$ as parity check matrix as

$$\Lambda_q^\perp(\mathbf{A}) = \{\mathbf{z} \in \mathbb{Z}^m \ : \ \mathbf{A}\mathbf{z} = 0 \mod q\} \ .$$

One of the most used lattice-based hardness assumptions is the Short Integer Solution (SIS), defined in the following.

Definition 2 (Short Integer Solution problem). *Given a matrix $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$, the $\mathsf{SIS}_{n,q,m,\beta}$ problem requires to find a vector $\mathbf{s} \in \mathbb{Z}^m$ such that $\mathbf{A}\mathbf{s} = 0 \mod q$ and $\|\mathbf{s}\| \leq \beta$.*
The inhomogeneous *version of the SIS problem (ISIS) requires, given a matrix $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$ and a vector $\mathbf{u} \in \mathbb{Z}_q^n$ to find a vector $\mathbf{s} \in \mathbb{Z}^m$ such that $\mathbf{A}\mathbf{s} = \mathbf{u} \mod q$ and $\|\mathbf{s}\| \leq \beta$.*

3 Efficient Verification for Digital Signatures

In this section, we introduce the concept of *efficient verification* for digital signatures and a suitable formal security model that allows to estimate the impact efficient verification has on the unforgeability of the scheme. Then, we describe a generic compiler to obtain efficient verification for a wide class of signatures, prove its security, show realizations from lattices and multivariate polynomials and finally discuss its concrete efficiency against full-fledged verification. In addition we discuss concrete instantiations and their corresponding concrete efficiency gains. This section ends with a generalization of our results to signatures with properties.

Efficient Verification In a nutshell, the core idea of efficient signature verification is to split the verification process into two steps. The first step is a one-time and signature-independent setup called 'offline verification'. Its purpose is to produce randomness to derive a (short, secret) verification key svk from the signer's public key pk. Note that the offline verification does not change the signature, which remains publicly verifiable; instead it 'randomizes' pk to obtain a concise verification key svk that essentially enables one to verify signatures with (almost) the same precision as the standard verification, but in a more efficient way. The second verification step consists of an 'online verification' procedure. It takes as input svk and can verify an unbounded number of message-signature pairs performing significantly less computation than the standard verification algorithm.

Definition 3 (Efficient Verification). *A signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ admits efficient verification if there exist two PPT algorithms $(\mathsf{offVer}, \mathsf{onVer})$ with the following syntax:*

$\mathsf{offVer}(\mathsf{pk}, k)$**:** *on input a verification key pk, a positive integer $k \in \{1, \ldots, \lambda\}$ (referred to as confidence level), the offline verification algorithm returns a secret verification key svk.*

$\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})$**:** *on input a secret verification key svk, a message μ, and a signature $\boldsymbol{\sigma}$, the efficient online verification algorithm outputs 0 (reject) or 1 (accept).*

The standard properties of an efficient verification scheme are described below.

Correctness. For a given security parameter λ, for any honestly generated key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, for any message $\mu \in \mathcal{M}$, for any signature $\boldsymbol{\sigma}$ such that $\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 1$, and for any confidence level $k \in \{1, \ldots, \lambda\}$; the following condition holds:

$$\Pr[\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma}) = 1 \mid \mathsf{svk} \leftarrow \mathsf{offVer}(\mathsf{pk}, k)] = 1.$$

This guarantees that a valid signature $\boldsymbol{\sigma}$ is always accepted by online verification. We remark that any signature rejected by onVer, would be rejected by Ver as well.

Amortized Efficiency. Informally, the notion of amortized efficiency captures the fact that the global computational cost of doing the preprocessing (running offVer once) and running onVer a given number times is considerably smaller than the cost of running the standard signature verification Ver the same number of times. More formally, let λ be the security parameter and $\mathsf{cost}(\cdot)$ be a function that, given in input an algorithm returns its computational cost (in some desired computational model). The pair $(\mathsf{offVer}, \mathsf{onVer})$ satisfies amortized efficiency if for any key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, for any message-signature pair $(\mu, \boldsymbol{\sigma})$, for any confidence level $k \in \{1, \ldots, \lambda\}$ and $\mathsf{svk} \leftarrow \mathsf{offVer}(\mathsf{pk}, k)$, there exists a 'small' positive real constant $\mathsf{e} \in]0, 1[= [0, 1] \setminus \{0, 1\}$ such that

$$\frac{\mathsf{cost}\big(\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})\big)}{\mathsf{cost}\big(\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})\big)} < \mathsf{e}.$$

Note that in this definition we do not include the cost needed to compute svk (i.e., $\mathsf{cost}(\mathsf{offVer})$). This is justified by the fact that, in our framework, offVer is a polynomial-time *one-time* set up, and its output (svk) can be reused in an unbounded number of executions of onVer. Namely, the computational cost of offVer is amortized over sufficiently many online verifications.

The concrete formula to quantify the amortized efficiency gain compares the computational cost of verifying r signatures with the offline/online approach over running r standard verifications.

Definition 4 (Concrete Amortized Efficiency). *We say that a pair* $(\mathsf{offVer}, \mathsf{onVer})$ *satisfies* $(\mathsf{r}_0, \mathsf{e}_0)$*-concrete amortized efficiency if* r_0 *is the smallest, non-negative integer such that for any* $\mathsf{r} \geq \mathsf{r}_0$ *there exists a small, real constant* $0 < \mathsf{e}_0 < 1$ *for which the following holds true:*

$$\frac{\mathsf{cost}\big(\mathsf{offVer}(\mathsf{pk}, k)\big) + \mathsf{r} \cdot \mathsf{cost}\big(\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})\big)}{\mathsf{r} \cdot \mathsf{cost}\big(\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})\big)} < \mathsf{e}_0. \tag{2}$$

Security Our definition of efficient verification lets the verifier set the confidence level $k \in \{1, .., \lambda\}$ at which she wishes to carry out the signature verification. Notably k also determines the amount of computation to be performed by offVer and onVer and thus, the efficiency gain. Our security notion aims at measuring the potential security loss that comes with performing less checks than the full-fledged verification. Intuitively, we say that an efficient verification is secure if the probability that onVer accepts a signature that would be rejected by Ver is negligible. We cannot expect a more efficient verification to detect more forgeries than the full verification. Since the output of offVer is a possibly secret verification key, in the security game we allow the adversary to interact polynomially many times (in the security parameter λ) with both the sign oracle $O\mathsf{Sign}$ and an additional efficient verification oracle $O\mathsf{onVer}$. As one would expect, the adversary can query the oracles in an adaptive and parallel way. This is formalized by the following definition.

```
cmvEUF ($\lambda$, $\Sigma$, $k$)
1 : $\mathsf{L}_S \leftarrow \varnothing$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
3 : $\mathsf{svk} \leftarrow \mathsf{offVer}(\mathsf{pk}, k)$
4 : $(\mu^*, \boldsymbol{\sigma}^*) \leftarrow \mathcal{A}^{O\mathsf{Sign}, O\mathsf{onVer}}(\mathsf{pk}, k)$
5 : return $(\mu^*, \boldsymbol{\sigma}^*)$
```

```
$\mathsf{Exp}_{\mathcal{A},\Sigma}^{\mathsf{cmvEUF}}(\lambda, k)$
1 : $(\mu^*, \boldsymbol{\sigma}^*) \leftarrow \mathsf{cmvEUF}(\lambda, \Sigma, k)$
2 : if $\mu^* \in \mathsf{L}_S$
3 :     return $0$
4 : if $\mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*) = 1$
5 :     return $0$
6 : $b \leftarrow \mathsf{onVer}(\mathsf{svk}, \mu^*, \boldsymbol{\sigma}^*)$
7 : return $b$
```

```
$O\mathsf{Sign}_{\mathsf{sk}}(\mu)$
1 : $\mathsf{L}_S \leftarrow \mathsf{L}_S \cup \{\mu\}$
2 : $\boldsymbol{\sigma} \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$
3 : return $\boldsymbol{\sigma}$
```

```
$O\mathsf{onVer}_{\mathsf{svk}}(\mu, \boldsymbol{\sigma})$
1 : $b \leftarrow \mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})$
2 : return $b$
```

Fig. 1: Security model for efficient verification of signatures: unforgeability under adaptive chosen message and verification attack (security game, experiment and oracles).

Definition 5 (Security of Efficient Verification). *Let* Σ *be a signature scheme that admits efficient verification. For a given security parameter* λ *and for any confidence level* $k \in \{1, \ldots, \lambda\}$*, the pair of algorithms* $(\mathsf{offVer}, \mathsf{onVer})$ *realizes a secure efficient verification for* Σ *if it is existentially unforgeable under adaptive chosen message and verification attack. In other words, if for all PPT adversaries* $\mathcal{A}$ *the success probability in the* cmvEUF *experiment in Figure 1 is negligible, i.e.:*

$$Adv_{\mathcal{A},\Sigma}^{\mathsf{cmvEUF}}(\lambda, k) = \Pr\Big[\mathsf{Exp}_{\mathcal{A},\Sigma}^{\mathsf{cmvEUF}}(\lambda, k) = 1\Big] \leq \varepsilon = \varepsilon(\lambda, k).$$

Line 5 of the cmvEUF experiment excludes forgeries against the original signature scheme. This is justified by the correctness of efficient verification and by the fact that Σ is supposed to be an unforgeable signature scheme in the usual sense. In other words, the above security experiment checks whether the adversary outputs invalid signatures that are however accepted by the efficient verification mechanism.

Notably, the security definition depends on the confidence level k, and we do not consider a solution to be secure if k is such that $\varepsilon(\lambda, k)$ is actually a non-negligible value. Yet, the latter scenario may occur in cases where, e.g., the execution of onVer is *prematurely interrupted.* A meaningful definition of security in such scenarios requires a new model as we show in Section 4.

3.1 A Generic Compiler for Mv-style Verifications

Until now we presented a framework for efficient verification of digital signatures. In what follows we show how to apply the efficient verification paradigm to a wide class of existing schemes. To this end, we begin with abstracting the verification procedure of a signature scheme into two types of verification checks: a matrix-vector multiplication (referred to as $\mathbf{Mv} = 0$, for appropriate matrix $\mathbf{M}$ and vector $\mathbf{v}$) and other generic checks (collected in the Check subroutine). Notably, our compiler leads to efficient verification whenever the computational complexity of the verification procedure is dominated by the matrix-vector multiplication, i.e. $\mathsf{cost}(\mathsf{Check}) << \mathsf{cost}(\mathbf{Mv}) \sim mn$ field multiplications (for $\mathbf{M} \in \mathbb{Z}_q^{n\times m}$).

We present a generic way to achieve efficient verification (as of Definition 3) for any signature scheme in which the verification procedure includes a matrix-vector ($\mathbf{Mv}$) multiplication. For simplicity, we call these $\mathbf{Mv}$-style check verifications (see Figure 2 for details). Schemes that fall in this category include some seminal lattice-based signatures [8,20,9,28], homomorphic signatures [6,21,16], and multivariate signatures [15,5].

```
Ver(pk, $\mu$, $\boldsymbol{\sigma}$)
    // INITIALIZE ACCEPTANCE BITS
1 :   $b_1 \leftarrow 0, b_2 \leftarrow 0$
    // SPLIT pk INTO MARTIX - AUX. DATA
2 :   parse pk = $(PK, PK.\mathsf{aux})$
    // ADDITIONAL VERIFICATION CHECKS
3 :   $b_1 \leftarrow \mathsf{Check}(PK.\mathsf{aux}, \mu, \boldsymbol{\sigma})$
    // FORMATTING Mv-STYLE CHECK
4 :   $(\mathbf{M}, \mathbf{v}) \leftarrow \mathsf{GetMv}(PK, \mu, \boldsymbol{\sigma})$
    // MATRIX-VECTOR MULT. CHECK
5 :   if $(\mathbf{M} \cdot \mathbf{v} = 0)$
6 :     $b_2 \leftarrow 1$
7 :   return $(b_1 \wedge b_2)$
```

```
Example: Ver(pk, $\mu$, $\boldsymbol{\sigma}$) for GPV08 [20]
1 :  $b_1 \leftarrow 0, b_2 \leftarrow 0$
2 :  parse pk = $(PK, PK.\mathsf{aux})$
       set  $PK \leftarrow \mathbf{A}$
       set  $PK.\mathsf{aux} \leftarrow (\mathcal{H}, \beta)$
3 :  $\mathsf{Check}(PK.\mathsf{aux}, \mu, \boldsymbol{\sigma})$ :
       if  $\|\boldsymbol{\sigma}\| < \beta$  set  $b_1 \leftarrow 1$
4 :  $\mathsf{GetMv}(\mathsf{pk}, \mu, \boldsymbol{\sigma})$ :
       set  $\mathbf{M} \leftarrow [\mathbf{A}| - \mathbf{I}_{rows(\mathbf{A})}]$
       set  $\mathbf{u} \leftarrow \mathcal{H}(\mu) \in \mathbb{Z}_q^{rows(\mathbf{A})\times 1}$
       set  $\mathbf{v} \leftarrow [\boldsymbol{\sigma}^T|\mathbf{u}^T]^T$
5 :  if $(\mathbf{M} \cdot \mathbf{v} = \mathbf{0}_{rows(\mathbf{A})\times 1} \mod q)$
6 :    set  $b_2 \leftarrow 1$
7 :  return $(b_1 \wedge b_2)$
```

Fig. 2: General structure of a verification with an $\mathbf{Mv}$-style check (on the left); an instructive example: the GPV08 signature verification (on the right) [20].

Our generic compiler for efficient verification is detailed in Figure 3 with a sketch of instantiation for the LBS scheme GPV08 [20] as a running example (more details on all the proposed instantiations can be found in Section 3.2). The key observation is that for any pair of vectors $\boldsymbol{\sigma}$ and $\mathbf{u}$, and for any matrix $\mathbf{A}$ (of opportune dimensions) if $\mathbf{A} \cdot \boldsymbol{\sigma} = \mathbf{u}$ then for any random vector $\mathbf{c}$ (of opportune dimension) it holds that $\mathbf{c} \cdot (\mathbf{A} \cdot \boldsymbol{\sigma}) = \mathbf{c} \cdot \mathbf{u}$. By collecting variables on the left hand side we get: $\mathbf{c} \cdot [\mathbf{A}| - \mathbf{1}] \cdot (\boldsymbol{\sigma}, \mathbf{u}) = 0$. Thus one can precompute the vector $\mathbf{z} \leftarrow \mathbf{c} \cdot [\mathbf{A}| - \mathbf{1}]$ and perform efficient 'online' verification by checking whether $\mathbf{z} \cdot \mathbf{u} = 0$, where $\mathbf{u} \leftarrow (\boldsymbol{\sigma}, \mathbf{u})$ (note that the matrix-vector multiplication is now replaced by a vector-vector multiplication, aka inner-product). Correctness and efficiency are immediate. Soundness comes from the fact that if $\mathbf{z} \cdot \mathbf{u} = 0$ then with all but negligible probability also the original system of linear equations $\mathbf{A} \cdot \boldsymbol{\sigma} = \mathbf{u}$ is satisfied, as proven in Theorem 1.

```
offVer(pk, $k$)
  // PARSE PUBLIC KEY (FOR EFFICIENCY)
  1 :  parse $\mathsf{pk} = (PK, PK.\mathsf{aux})$
  // e.g., in GPV08 $PK = \mathbf{A}$, $PK.\mathsf{aux} = \mathcal{H}$,
  // GENERATE PUBLIC MATRIX OF CORRECT DIMENSIONS
  2 :  $\mathbf{M} \leftarrow \mathsf{GetM}(PK)$  // e.g., in GPV08 $\mathbf{M} = (\mathbf{A}| - \mathbf{1}_{n\times n})$
  // CHECK PARAMETER CONSISTENCY
  3 :  if $(k > rows(\mathbf{M}) \vee k < 1)$  return $\perp$
  // GENERATE RANDOMIZED KEY
  4 :  $\mathbf{Z} \leftarrow \mathsf{GetZ}(\mathbf{M}, k)$
         i :  $\mathbf{z}_0 \leftarrow \mathbf{0}_{1\times cols(\mathbf{M})}$  // for good indexing purpose
        ii :  for $j = 1, \ldots, k$
       iii :      $\mathbf{c} \xleftarrow{\$} \mathbb{Z}_q^{1\times rows(\mathbf{M})}$
        iv :      $\mathbf{z} \leftarrow \mathbf{cM} \in \mathbb{Z}_q^{1\times cols(\mathbf{M})}$
         v :      if $\mathbf{z} \in \langle \mathbf{z}_0, \ldots, \mathbf{z}_{j-1} \rangle_q$  go to line iii.
        vi :      $\mathbf{z}_j \leftarrow \mathbf{z}$  // store new linearly independent vector
       vii :  set $\mathbf{Z} \leftarrow [\mathbf{z}_1^T | \ldots | \mathbf{z}_k^T]^T \in \mathbb{Z}_q^{k\times cols(\mathbf{M})}$
  5 :  return $\mathsf{svk} \leftarrow (k, \mathbf{Z}, PK.\mathsf{aux})$
```

(a) The offline verification algorithm.

```
onVer(svk, $\mu$, $\boldsymbol{\sigma}$)
  // LIGHTWEIGHT VERIFICATION CHECKS
  1 :  if $\mathsf{Check}(PK.\mathsf{aux}, \mu, \boldsymbol{\sigma}) = 0$
  2 :      return 0
  // FORMATTING FOR EFFICIENT VER.
  3 :  $(\mathbf{Z}', \mathbf{v}) \leftarrow \mathsf{GetZV}(\mathsf{svk}, \mu, \boldsymbol{\sigma})$
  4 :  parse $\mathbf{Z}' = [\mathbf{z}_1'^T | \ldots | \mathbf{z}_k'^T]^T \in \mathbb{Z}_q^{k\times cols(\mathbf{Z}')}$
  5 :  parse $\mathbf{v} = [\mathbf{v}_1^T | \ldots | \mathbf{v}_k^T]^T \in \mathbb{Z}_q^{k\times cols(\mathbf{Z}')}$
  // LINE-BY-LINE INNER PRODUCTS
  6 :  for $j = 1, \ldots, k$
  7 :      if $\mathbf{z}_j' \cdot \mathbf{v}_j \neq 0 \mod q$
  8 :          return 0
  9 :  return 1
```

(b) The online verification algorithm.

Fig. 3: Our compiler for efficient verification of signatures with **Mv**-style verification. The four scheme-dependent subroutines are: **parse** pk and GetZ (in offVer); Check and GetZV (in onVer). The complexity of the onVer is linear in k, the chosen confidence level.

Security Analysis of Our Compiler. Next, we prove the security of our generic compiler. We discuss the tradeoffs between $q(\lambda)$ and k in the next section.

Theorem 1. *Let Σ be a signature scheme with the structure of Figure 2. Then our compiler for Σ (depicted in Figure 3) generates a secure efficient verification realization of Σ–i.e., (offVer, onVer) are existentially unforgeable under adaptive chosen message and verification attacks– with $\varepsilon = \frac{q_V+1}{q^k}$, where $k \in \{1, \ldots, rk(\mathbf{M})\}$ denotes the chosen confidence level and $q_V = \mathsf{poly}(\lambda)$ is a bound on the total number of verification queries.*

Proof. Let us note that the winning condition of the experiment requires $\mathcal{A}$ to produce a message-signature pair $(\mu^*, \boldsymbol{\sigma}^*)$ such that μ^* has not been queried to the signing oracle during the game, the signature is invalid under standard verification, i.e., $\mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*) = 0$, but it is accepted by the online verification, i.e., $\mathsf{onVer}(\mathsf{svk}, \mu^*, \boldsymbol{\sigma}^*) = 1$. The goal of the proof is to bound the probability that this event occurs.

First, let us define $\mathbf{G}_0$ to be the cmvEUF experiment of Figure 1. Next, we define a series of q_V hybrid games where, for every $i = 1$ to q_V, $\mathbf{G}_i$ is the same as $\mathbf{G}_{i-1}$ except that the i-th query to onVer is answered by using the standard verification algorithm Ver. Clearly, for every i, the view of the adversary in $\mathbf{G}_i$ is the same as in $\mathbf{G}_{i-1}$ unless, at the i-th verification query $(\mu_i, \boldsymbol{\sigma}_i)$, this is answered with a reject in $\mathbf{G}_i$ (i.e., $\mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 0$) whereas it would have been accepted in $\mathbf{G}_{i-1}$ (i.e., $\mathsf{onVer}(\mathsf{svk}, \mu_i, \boldsymbol{\sigma}_i) = 1$). Let us define this event as

$$\mathsf{bad}_i = \{\mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 0 \ \wedge \ \mathsf{onVer}(\mathsf{svk}, \mu_i, \boldsymbol{\sigma}_i) = 1\} \ . \tag{3}$$

Then, it is clear that, $\forall i \in \{1, \ldots, q_V\}$,

$$|\Pr[\mathbf{G}_{i-1} = 1] - \Pr[\mathbf{G}_i = 1]| \leq \Pr[\mathsf{bad}_i].$$

Moreover, observe that the adversary wins in game $\mathbf{G}_{q_V}$ if the same event occurs for the message-signature pair $(\mu^*, \boldsymbol{\sigma}^*)$ (that we also denote below with $(\mu_{q_V+1}, \boldsymbol{\sigma}_{q_V+1})$):

$$\mathsf{bad}_{q_V+1} = \{\mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*) = 0 \ \wedge \ \mathsf{onVer}(\mathsf{svk}, \mu^*, \boldsymbol{\sigma}^*) = 1\}.$$

So, by a union bound it follows that

$$Adv^{CMVA}_{\mathcal{A},\Sigma}(\lambda, k) = \Pr[\mathbf{G}_0 = 1] \leq \sum_{i=1}^{q_V+1} \Pr[\mathsf{bad}_i] \tag{4}$$

In the following claim we show that for every $i \in [q_V + 1]$, $\Pr[\mathsf{bad}_i] \leq 1/q^k$ over the choice of the $\mathbf{c}_j$'s, which concludes the proof of the theorem.

Claim. $\Pr[\mathsf{bad}_i] \leq 1/q^k$, where q is the modulus (or the size of the field) and k is the chosen confidence level to run the efficient verification.

Let $\mathbf{M}$ and $\mathbf{v}$ be the matrix and vector returned by $\mathsf{GetMv}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i)$ during the standard verification. Since this is a bad forgery the online verification returns 1. This means that (1) the signature $\boldsymbol{\sigma}_i$ must satisfy all of the 'consistency checks' determined by $\mathsf{Check}(\mu_i, \boldsymbol{\sigma}_i)$ (otherwise it would be rejected also by the online verification); and (2) $\mathbf{Mv} \neq 0$ but $(\mathbf{c}_j\mathbf{M})\mathbf{v} = 0$ for all $j \in \{1, \ldots, k\}$. We can split $\Pr[\mathsf{bad}_i]$ as:

$$\Pr[\mathsf{bad}_i] = \Pr[\mathsf{Check}(PK.\mathsf{aux}, \mu_i, \boldsymbol{\sigma}_i) = 1] \cdot \Pr[\mathbf{c}_j(\mathbf{Mv}) = 0 \text{ for all } j \in \{1, \ldots, k\}].$$

We remark that $\mathbf{z}'_j = \mathbf{c}_j\mathbf{M}$ is (part of) the output of GetZV in the online verification. Assuming the worst case scenario where $\mathcal{A}$ can trivially generate a pair $(\mu_i, \boldsymbol{\sigma}_i)$ that passes the consistency checks in Check, we bound only the second factor in the equation above. We argue that:

$$\Pr[\mathbf{c}_j(\mathbf{Mv}) = 0 \text{ for all } j \in \{1, \ldots, k\}] = \frac{1}{q^k}.$$

Let $\mathbf{w} = (\mathbf{Mv}) \neq \vec{0}_{rows(\mathbf{M}) \times 1}$; thus there exists at least one entry $\ell \in [rows(\mathbf{M})]$ such that $\mathbf{w}[\ell] \neq 0$. The online verification accepts the adversary's output if and only if $\mathbf{c}_j \cdot \mathbf{w} = 0$ for all j, i.e., if and only if, for all k (linearly independent constraints) it holds that:

$$\mathbf{c}_j[\ell] = \mathbf{w}[\ell]^{-1} \cdot \sum_{i=1, i\neq\ell}^{n} \mathbf{c}_j[i]\mathbf{w}[i]$$

At this point we observe that all online verification queries before this i-th one were answered *without* using the $\mathbf{c}_j$ vectors. Therefore, for the sake of bounding $\Pr[\mathsf{bad}_i]$ we can assume that all the vectors $\mathbf{c}_j$ are randomly sampled at this stage, namely they are freshly random, and unknown to $\mathcal{A}$. So, for every j, the equality above holds with probability $1/q$. Since we have k independent such relations (i.e., the $\mathbf{c}_j$ and the $\mathbf{z}'_j$ are linearly independent), it follows that: $\Pr[\mathsf{bad}_i] \leq \prod_{j=1}^{k} \frac{1}{q} = \frac{1}{q^k}$. Such probability is negligible for appropriate choices of $q = q(\lambda)$ and $k \in \mathbb{Z}_{>0}$. Concretely, if $q = 2^{\mathsf{poly}(\lambda)}$ as in [21,16] our compiler defines a secure generalized efficient verification already for any $k = 1$. □

3.2 Concrete Instantiations of Our Compiler

Any instantiation of our compiler is completely determined by the four subroutines **parse** pk, GetM, Check, and GetZV.

From Lattices We present concrete instantiations of our compiler for two categories of LBS: 'hash & sign' with representative the GPV08 signature [20]; and 'Boyen/BonsaiTree' style with representative MP12 [28].

Efficient Verification for GPV08 [20]. The **parse** pk procedure splits the public key into $PK = \mathbf{A} \in \mathbb{Z}_q^{n\times m}$ (the matrix identifying the signer's public key), and the auxiliary public information $PK.\mathsf{aux} = \mathcal{H}$, i.e., a description of a full-domain hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_q^n$. The Check procedure is exactly as in the original verification (enforcing the norm bound on the signature). The GetM algorithm takes in input the public matrix $PK = \mathbf{A}$, and appends to it the identity matrix to obtain $\mathbf{M} = [\mathbf{A}| - \mathbf{I}_n]$. The GetZV routine returns the matrix $\mathbf{Z}'$ and the vector $\mathbf{v}$. The matrix $\mathbf{Z}'$ is made up of the same 'randomized key' vectors produced by GetZ during the offline verification, i.e., $\mathbf{z}'_j = \mathbf{z}_j \leftarrow \mathbf{c}_j\mathbf{M} = [\mathbf{c}_j\mathbf{A}| - \mathbf{c}_j]$, i.e., $\mathbf{v} = [\boldsymbol{\sigma}|\mathcal{H}(\mu) \cdot \mathbf{1}_{1\times n}]$. Thus the core verification check (line 7 in onVer) is actually ensuring that $\mathbf{z}'_j\mathbf{v}_j = 0$, i.e., $\mathbf{c}_j \cdot \mathbf{A} \cdot \boldsymbol{\sigma} = \mathbf{c}_j\mathcal{H}(\mu)$ which is the probabilistic check of the original verification equality.

Efficient Verification for MP12 [28]. The **parse** pk procedure assigns $PK \leftarrow \mathbf{A} = [\tilde{\mathbf{A}}|\mathbf{A}_0 \mid \ldots \mid \mathbf{A}_\ell] \in \mathbb{Z}_q^{n\times(\bar{m}+n\lceil\log q\rceil\ell)}$ (the matrix identifying the signer's public key), where $\bar{m} = O(n\lceil\log q\rceil)$, and ℓ denotes the number of bits in the message, i.e., $\mu \in \{0,1\}^\ell$. The auxiliary public information is $PK.\mathsf{aux} = \mathbf{u}$. The Check procedure is exactly as in the original verification (enforcing the norm bound on the signature). The GetM algorithm takes in input the public matrix $PK = \mathbf{A}$, and appends to it the identity matrix to obtain $\mathbf{M} = [\mathbf{A}| - \mathbf{I}_{n\times n}]$. The GetZV routine returns the matrix $\mathbf{Z}'$ and the vector $\mathbf{v}$. The matrix $\mathbf{Z}'$ is made up of vectors of the form $\mathbf{z}'_j = [\tilde{\mathbf{z}}_j \mid \mathbf{z}_j^0 + \sum_{i=1}^{\ell} \mu[i]\mathbf{z}_j^i|\mathbf{c}_j]$ that identify a message-dependent lattice (called $\mathbf{A}_\mu$ in [28]). The vector $\mathbf{v}$ is the concatenation of the signature with the auxiliary vector, i.e., $\mathbf{v} = [\boldsymbol{\sigma}|\mathbf{u}]$. Note that $\mathbf{u}$ is the same for all messages; thus, one could further optimize the online verification by computing (once and for all) the k inner products $\mathbf{z}_j[\bar{m} + n\lceil\log q\rceil + 1] = \mathbf{c}_j \cdot \mathbf{u}$ during the offline phase. To conclude we notice that the online verification ensures that $\mathbf{z}'_j\mathbf{v}_j = 0$, i.e., $\mathbf{c}_j \cdot \mathbf{A}_\mu \cdot \boldsymbol{\sigma} = \mathbf{c}_j \cdot \mathbf{u}$ which is the probabilistic check of the original verification equality.

From Multivariate Equations For signatures schemes based on multivariate equations we take Rainbow [15,14] as representative example as this is one of the NIST candidates for standardization. For completeness, we also show how to apply our compiler to the LUOV scheme [5].

Efficient Verification for Rainbow [14]. In the description below we consider the standard Rainbow verification. A similar approach can be used to speed up the verification also in the "cyclic" and the "compressed" Rainbow variants as in those cases the verification includes an additional initial phase to reconstruct the full public key. We recall that in this scheme the public key contains a system of m multivariate quadratic polynomials in n variables. For convenience, let $N = n(n+1)/2$ and $\mathbb{F} = \mathbb{F}_{2^r}$. Using a Macaulay matrix representation we can visualize this system as a wide matrix composed of a quadratic term $\mathbf{Q}$ (actually a $m \times N$ submatrix), a linear term $\mathbf{L}$ ($m \times n$ submatrix) and a constant term $\mathbf{C}$ (a $m \times 1$ vector). The **parse** pk procedure extracts from the public key PK this matrix $\mathsf{pk} = [\mathbf{Q}|\mathbf{L}|\mathbf{C}] \in \mathbb{F}^{m\times(N+n+1)}$ and a description of a full-domain hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{F}^m$ as the auxiliary public information $PK.\mathsf{aux} = \mathcal{H}$. The Check procedure is trivial and always returns 1. This is because the whole verification can be written as a matrix-vector multiplication. The GetM algorithm extracts from PK the matrix representing the system of quadratic multivariate equations $[\mathbf{Q}|\mathbf{L}|\mathbf{C}]$. Finally, it appends to this the identity matrix, so $\mathbf{M} \leftarrow [\mathbf{Q}|\mathbf{L}|\mathbf{C}| - \mathbf{I}_m]$. We remark that $\mathbf{M}$ can be seen as a matrix of blocks, where any block has the same height (m = number of rows), but different length (number of columns). The GetZV routine reads the matrix $\mathbf{Z}' = \mathbf{Z}$ made up of the rows $\mathbf{z}'_j = \mathbf{z}_j \leftarrow \mathbf{c}_j\mathbf{M} = [\mathbf{c}_j \cdot \mathbf{Q}|\mathbf{c}_j \cdot \mathbf{L}|\mathbf{c}_j \cdot \mathbf{C}| - \mathbf{c}_j] \in \mathbb{F}^{1\times N+n+1}$. In addition, this algorithm parses the signature as $\boldsymbol{\sigma} = (\mathbf{s}, \mathsf{salt})$, computes the (salted) hash of the message $\mathbf{d}$ as $\mathbf{h} \leftarrow \mathcal{H}(\mathcal{H}(\mathbf{d})|\mathsf{salt})$ and outputs the vector $\mathbf{v} = [\tilde{\mathbf{s}}|\mathbf{s}|1|\mathbf{h}]$, where $\mathbf{s}$ is part of the signature and $\tilde{\mathbf{s}}$ is the 'quadratic vector' obtained by computing all products of pairs of elements in $\mathbf{s}$ (with monomials ordered lexicographically), i.e., $\tilde{\mathbf{s}} \leftarrow [\mathbf{s}[1]^2, \mathbf{s}[1]\mathbf{s}[2], \ldots, \mathbf{s}[n-1]\mathbf{s}[n], \mathbf{s}[n]^2]$. Clearly $\mathbf{z}'_j \cdot \mathbf{v} = 0$ if and only if $\mathbf{c}_j \cdot (\mathbf{Q}\tilde{\mathbf{s}} + \mathbf{L}\mathbf{s} + \mathbf{C}) = \mathbf{c}_j \cdot \mathbf{h}$, which is a probabilistic check of the original system of verification equations in Rainbow.

Efficient Verification for LUOV [5]. The **parse** pk procedure splits the public key into $PK = (\mathsf{public.seed}, \mathbf{Q}_2)$ (the concise information needed to retrieve the full signer's public key), and the auxiliary public information $PK.\mathsf{aux} = \mathcal{H}$, i.e., a description of a full-domain hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{F}^m$, where $m = rows(\mathbf{Q}_2)$ and $\mathbb{F} = \mathbb{F}_{2^r}$. The Check procedure is trivial and always returns 1. This is because the whole LUOV verification can be written as a matrix-vector multiplication The GetM algorithm takes in input $PK = (\mathsf{public.seed}, \mathbf{Q}_2)$ and derives the full public key as done in the original verification: it runs $[\mathbf{C}||\mathbf{L}||\mathbf{Q}_1] \leftarrow \mathcal{G}(\mathsf{public.seed})$ to get the constant constant (vector), the linear (matrix) and the first quadratic (matrix) parts of the verification equation; and then it reconstructs the full quadratic term as $\mathbf{Q} \leftarrow [\mathbf{Q}_1||\mathbf{Q}_2]$. Finally it appends to the public key the identity matrix $\mathbf{M} \leftarrow (\mathbf{C}, \mathbf{L}, \mathbf{Q}, -\mathbf{I}_{rows(\mathbf{Q})})$, we remark that $\mathbf{M}$ can be seen as a matrix of blocks, where any block has the same height (number of rows), but different lenghth (number of columns). The GetZV routine reads the matrix $\mathbf{Z}' = \mathbf{Z}$ made up of the rows $\mathbf{z}'_j = \mathbf{z}_j \leftarrow \mathbf{c}_j\mathbf{M} = (\mathbf{c}_j \cdot \mathbf{C}, \mathbf{c}_j \cdot \mathbf{L}, \mathbf{c}_j \cdot \mathbf{Q}, -\mathbf{c}_j)$. It also outputs the vector $\mathbf{v} = [1|\mathbf{s}|\tilde{\mathbf{s}}|\mathbf{h}]$, where $\mathbf{s}$ is part of the signature $\boldsymbol{\sigma} = (\mathbf{s}, \mathsf{salt})$, $\tilde{\mathbf{s}}$ is the 'quadratic vector' obtained by computing all products of pairs of elements in $\mathbf{s}$, i.e., $\tilde{\mathbf{s}} \leftarrow [\mathbf{s}[1]^2, \mathbf{s}[1]\mathbf{s}[2], \ldots, \mathbf{s}[n-1]\mathbf{s}[n], \mathbf{s}[n]^2]$, finally $\mathbf{h}$ is the hash of the message and the salt, i.e., $\mathbf{h} \leftarrow \mathcal{H}(\mu||0\mathrm{x}0||\mathsf{salt})$. Clearly $\mathbf{z}'_j \cdot \mathbf{v} = 0$ if and only if $\mathbf{c}_j \cdot (\mathbf{C} + \mathbf{L}\mathbf{s} + \mathbf{Q}\tilde{\mathbf{s}}) = \mathbf{c}_j \cdot \mathbf{h}$, which is a probabilistic check of the original verification equation in LUOV.

3.3 Concrete Efficiency Estimates for Our Compiler

In what follows, we evaluate the efficiency gains provided by our compiler using the $(\mathsf{r}_0, \mathsf{e}_0)$-concrete efficiency notion of Equation (2). In brief, a realization of our compiler provides $(\mathsf{r}_0, \mathsf{e}_0)$-concrete amortized efficiency if r_0 is the smallest, non-negative integer for which it holds that $\mathsf{e}_0 < 1$, where e_0 is an upperbound on the ratio of running r_0 offline/online verifications versus r_0 standard signature verifications. For convenience, we estimate only the cost of the most expensive 'steps' in the verification, namely the ones involving several field element multiplications (matrix-vector products), and disregard the cost of adding elements, generating random values, reading algorithm inputs or evaluating hash functions. Moreover, we do not consider ad-hoc optimizations of matrix multiplication due to probabilistic checks using, e.g., Freivalds' Algorithm or its variant [31]. We remark that in this work, we develop a more general and full-fledged approach that is not limited to the specific technique in [31]. Table 1 collects the common notation, while Table 2 displays a summary of our findings, that we motivate below.

Table 1: Parameters involved in the efficiency estimate of our compiler.

q	Modulus of the lattice or size of the field
n	Number of rows in the public key
$m \in \Omega(n \log q)$	Number of columns in the public key
β	Bound on the noise / size of signatures
$\boldsymbol{\sigma}$ or $\mathbf{U}$	Vector or matrix signatures
k	Number of steps in the online verification (confidence level)
r	Number of signatures verified (repetitions of onVer)
$\mathsf{cost}(alg)$	Number of field multiplications needed to compute alg

The computational complexity of Ver in $\mathbf{Mv}$-style verification, e.g., [8,20,28,5,14] is dominated by a matrix-vector multiplication. Let $n = rows(\mathbf{M})$ and $m = cols(\mathbf{M})$, with $m \geq n$. Thus the cost of computing $\mathbf{M} \cdot \mathbf{v}$ is, in the worst case, nm filed multiplications. Our offline verification algorithm executes k vector-matrix multiplications (one for each $\mathbf{z}'_j$ in $\mathbf{Z}'$), resulting in knm multiplications in the worst case. The computational complexity of our online verification is dominated by the k vector-vector (inner) products $\mathbf{z}_i \cdot \mathbf{v}$, resulting in km multiplications in the worst case. Thus, the compiler presented in Section 3.1 outputs an efficient verification for signature with $\mathbf{Mv}$-style verification that has the following concrete amortized efficiency:

$$\frac{\mathsf{cost}(\mathsf{offVer}) + \mathsf{r} \cdot \mathsf{cost}(\mathsf{onVer})}{\mathsf{r} \cdot \mathsf{cost}(\mathsf{Ver})} = \frac{knm + \mathsf{r}km}{\mathsf{r}nm} = \frac{k}{\mathsf{r}} + \frac{k}{n}. \tag{5}$$

Clearly the first addend in Equation (5) comes from amortizing the cost of offVer (over verifying r signatures), while the second term is the fix trade-off between the computational costs of onVer and Ver (at each and every verification). Table 2 collects the main results described in the reminder of the section. In detail, k_0 determines the lenght of the svk, the computational complexity and the unforgeability of the efficient verification; r_0 states how many signatures one should verify in order to have a concrete efficiency gain of e_0 (lower values of e_0 correspond to better efficiency gains). The last column displays how 'cheaper' the online verification is with respect to the fullfledged verification (ignoring the one-time cost of running the offline verification). Again, lower values in this column correspond to better efficiency; for instance, a ratio of 0.4% means that the computational cost of Ver is 99.6× higher than the one of onVer (in other words, onVer is expected to be about 99× faster).

For convenience, in our analysis we categorize signatures according to the size of their underlying algebraic structure (q exponential or polynomial in the security parameter).

The modulus q is exponential in λ: To the best of our knowledge, the only LBS constructions that fall in this category are the homomorphic signatures by Gorbunov et al. [21] and by Fiore et al. [16]. In this case, using our compiler (with some caveat, as we show in the next section) yields that the advantage in the cmvEUF experiment (in Equation (4)) is negligible in the security parameter for $k = 1$. However, in [21,16] the complexity of Ver is dominated by the matrix-matrix multiplication $\mathbf{AU}$ where $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ is the fixed public key, and $\mathbf{U} \in \mathbb{Z}_q^{m\times m}$ is the signature[4].We have computed parameters for this scheme according to the methodology by Albrecht et al. [1]. Setting $\lambda = 128$, $q = 2^\lambda$ and $n = 256$ yields that reduction algorithms (in particular, the optimized BKZ algorithm) would have runtime 2^{128} and would solve at most $\mathsf{SIS}_{256,2^{128},65536,2^{80}}$, while the security of the scheme relies on a SIS instance with norm bound $\beta = 2^{49d}$, where d is the depth of the circuit. We can now use this set of parameters to determine the concrete amortized efficiency (as of Definition 4) reached by our compiler for the LBS in [16,21]. Let $k = 1$ and $n = 256$ in Equation (3.3), we want to extract the minimum r_0 for which $1/\mathsf{r} + 1/256$ is smaller than 1. It is easy to see that $\mathsf{r}_0 = 2$ suffices and we get $\mathsf{e}_0 = 0.504 < 1/2 + 1/256$. In other words, the cost of setting up the online verification (running offVer) *plus* performing $\mathsf{r} = 2$ online verifications is about half of the cost of $\mathsf{r} = 2$ standard verifications, while preserving the security level. Moreover, for this set of parameters $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k}{n} = \frac{1}{256} < 0.004$, i.e., our online verification requires about 0.4% of the computational cost of running the standard verification algorithm; alternatively, we can read this results as our onVer is 99× faster than Ver.

The modulus q is polynomial in λ: This is the most common setting given the 'small' size of q. In this category fall the schemes by Gentry et al. [20], Boyen [8] and its improved version by Micciancio and Peikert [28], Boneh Franklin linearly homomorphic signature [6], Rainbow [15] and LUOV [5]. For the LBS constructions, in order to guarantee a negligible advantage in the cmvEUF experiment (see Equation (4)) we need to set an appropriate value of $k > 1$. We argue that 'appropriate' values of k are still 'small' in comparison to n and lead to a 'good' amortized efficiency even for 'few' verifications. We recall that for these constructions Ver computes a product $\mathbf{A}\boldsymbol{\sigma}$ where $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and the signature is just a vector $\boldsymbol{\sigma} \in \mathbb{Z}_q^{m\times 1}$. To guarantee the security of our efficient verification, the value k should be set so that q^{-k} be negligible. In other words, for the advantage in Equation (4) to be negligible it must hold that $q^{-k} = 2^{-\lambda}$. Hence, to estimate k, one need to first fix the value of λ, compute the corresponding q that can guarantee such level of security, and then extract k from the relation above.

Computing parameters for lattice-based schemes is not straightforward, as so far there is no unique way to derive the parameters from a given λ. However, a good measure of the security of a set of parameters can be extracted computing a value δ that was introduced by Gama and Nguyen [18]. Concretely, δ provides an indication of how reduction algorithms would perform against the hardness assumption underlying the lattice-based construction. Generally, the 'smaller' the δ, the 'more secure' the scheme.

For Boyen's signature [7] and its variant by Micciancio and Peikert [28], we use the parameters provided in Figure 2 in [28]. Since in [28] they set $\delta = 1.007$, to ensure a fair comparison, we use the same value to compute the parameters Gentry et al.'s signature [20]. As a result, we observed that for this δ all of the schemes require about the same modulus $q \sim 2^{30}$ (for $n \sim 256$). For this set of parameters, our efficient verification provides 80 (resp. 250) bits of security with just only $k = 4$ (resp. $k = 9$). Thus our compiler achieves a $(5, 0.81)$-concrete amortized efficiency (resp. $(10, 0.94)$),

[4] In [16] the dimension m additionally depends on the number $t \geq 1$ of distinct identities (users) involved in labeled program. For simplicity, in what follows we consider $t = 1$.

and a concrete tradeoff between onVer and Ver of $k/n = 1.5$ (resp. 0.0352), i.e., onVer is about $98\times$ faster than Ver. The take away is that for any $\mathsf{r} > k$ there is a concrete efficiency gain.

We remark that $\delta = 1.007$ was 'enough' back in 2012, but it is now obsolete. As of now, this value of δ probably guarantees less than $\lambda = 60$ bits of security. This does not make our analysis meaningless. Indeed, the value of δ decreases for larger q and n (intuitively, this is because it is harder to find small vectors in larger lattices). In particular, the decrease in the value of δ is more dramatic for larger q than for larger n. Hence, a value of δ that would guarantee 128 bits of (post-quantum) security today implies a larger q than what we consider in this work, thus a k smaller than, e.g., $128/30 \sim 4.3 < 5$.

For **Rainbow**, we follow the latest guidelines provided during the second round of the NIST competition. We recall that the Rainbow signature scheme is based on the unbalanced oil and vinegar approach, and it is fully determined by the field on which it operates and a sequence of integer numbers indicating the amount of vinegar variables and oil variables per each layer. One of the current settings utilizes $\mathbb{F} = \mathbb{F}_{2^4}$ and a two-layer oil variable setting with $(v_1, o_1, o_2) = (32, 32, 32)$, which lead to $m = 96$ and $n = 64$ (for consistency in this paper we set n to be the number of rows of a matrix and m to denote the number of columns, classically the variables are swapped for multivariate signatures). A suitable k to achieve NIST security category level II is $k = 32$, since $q^{-k} = 2^{-4 \cdot 32} = 2^{-128}$. In regards to the concrete efficiency estimates, plugging in $k = 32$ and $n = 64$ in Equation (3.3) shows that the minimum number of repetitions r_0 to achieve amortized efficiency (i.e., for which we have $k/\mathsf{r} + k/n < 1$) is $\mathsf{r}_0 = 65$, the corresponding amortization factor is $\mathsf{e}_0 = 0.9923 = 32/65 + 32/64$. For this set of parameters we have $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k}{n} = \frac{32}{64} = 0.5$, in other words, our compiler produces an online verification that is $2\times$ *faster* than the standard verification. For $\mathbb{F} = \mathbb{F}_{2^8}$ and $(v_1, o_1, o_2) = (68, 36, 36)$, we have $m = 140$ and a suitable k in this case would be $k = 16$, since $q^{-k} = 2^{-8 \cdot 16} = 2^{-128}$. For the highest NIST security category, [14] suggests to use $\mathbb{F} = \mathbb{F}_{2^8}$ and $(v_1, o_1, o_2) = (92, 48, 48)$. As a result we have $m = 188$, $n = 96$ and again $k = 16$ but a better amortize efficiency factor $\mathsf{e}_0 = 0,9666$ already for $\mathsf{r}_0 = 20$. We remark that for this set of parameters $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} < 0.166$, i.e., our compiler produces an online verification procedure that is $6\times$ *faster* than the standard verification. The verification is dominated by the matrix-vector multiplication, which requires nm field multiplications in the worst case. Our offline verification algorithm executes k vector-matrix multiplications (one for each $(\mathbf{c}_j, \mathbf{z}_j)$), resulting in knm multiplications. The online verification is dominated by the k vector-vector (inner) products $\mathbf{z}_i \cdot \sigma$, resulting in kn multiplications. Thus, for [8,20,28], the concrete amortized efficiency can be computed as $\frac{\mathsf{cost(offVer)} + \mathsf{rcost(onVer)}}{\mathsf{rcost(Ver)}} = \frac{knm + \mathsf{r}km}{\mathsf{r}nm} = \frac{k}{\mathsf{r}} + \frac{k}{n}$. The first addend in Equation (3.3) comes from amortizing the cost of offVer (over r repetitions), while the second term is the fixed trade-off between the computational costs of onVer and Ver (at every verification).

3.4 Generalization to Signatures with Properties

Our model can be easily generalized to the case of signatures with properties (e.g., homomorphic signatures, group signatures, threshold signatures etc.). Such a generalization is particularly important in the case of LBS. Indeed, lattice-based hardness assumptions allow to build a wide range of LBS with properties that often have a verification algorithm very similar to standard **Mv**-style verification (cf. Figure 2).

Signatures with properties require more complex security definitions than standard signatures. Typically, they include (some specific flavor of) unforgeability. In Figure 4 we present a very general description of the unforgeability experiment for signatures with properties.

Table 2: A summary of the concrete efficiency achieved by various instatiations of our compiler. In the table, k_0 denotes the minimum accuracy level that securely ralizes efficient verification, i.e., for which $\Pr[\mathsf{bad}_1] = \frac{1}{q^{k_0}}$ is negligible (cf. proof of Theorem 1); r_0 is the smallest positive integer for which $\mathsf{e}_0 < 1$ where e_0 is a (tight) upperbound on the ratio $\mathsf{e}_0 > \frac{\mathsf{cost}(\mathsf{offVer}_{k_0}) + \mathsf{r} \cdot \mathsf{cost}(\mathsf{onVer})}{\mathsf{r} \cdot \mathsf{cost}(\mathsf{Ver})}$.

Signature Scheme	**Ring-Field size**	**Min. Accuracy Level**	**Concrete Amortized Efficiency**	**Online Efficiency** $\frac{\mathsf{cost}(\mathsf{onVer})}{\mathsf{cost}(\mathsf{Ver})}$
FMNP16 [16] GVW15 [21]	$q = 2^{\mathsf{poly}(\lambda)}$	$k_0 = 1$	$(\mathsf{r}_0 = 2, \mathsf{e}_0 = 0.5)$	$< 0.4\%$
Boyen10 [8] GPV08 [20] MP12 [28]	$q = \mathsf{poly}(\lambda)$	$k_0 = 3$	$(\mathsf{r}_0 = 4, \mathsf{e}_0 = 0.77)$	$< 1.1\%$
Rainbow[15] with $\mathbb{F}_{2^4} - (32, 32, 32)$	$q = \mathsf{poly}(\lambda)$	$k_0 = 15$	$(\mathsf{r}_0 = 18, \mathsf{e}_0 = 0.99)$	$< 2.4\%$

Essentially, all of them require:

1. A setup phase, where a probabilistic routine (denoted Setup in Figure 4) generates a set of secret values sval and other auxiliary values pval (that includes the verification key pk). The latter are given as input to the adversary, while the former are used by the oracles.
2. A challenge phase, where the adversary is given access to some, possibly stateful, oracles (usually, at least an oracle that returns signatures by honest users), and has to output a message and a forged signature on it. We model this by defining two oracles: $OK(\cdot; \mathsf{sval}, \mathsf{st}_K)$: Returns signing/secret keys (of users or other entities that $\mathcal{A}$ may corrupt). $O(\cdot; \mathsf{st})$: Encompasses all the other possible oracles (signing, opening for group signatures, etc.).
3. A check phase, where the experiment checks whether the signature output by $\mathcal{A}$ is valid and if $\mathcal{A}$ won the experiment. The former requires an execution of the verification algorithm. The latter includes a variety of other checks that essentially are supposed to ensure that the signature was not trivially produced by the adversary (e.g., the signature has not been output by the signing oracle). We model this second check with the $\mathsf{WinCond}$ predicate. Clearly, the specification of $\mathsf{WinCond}$ depends on each primitive.

Adapting the syntax and security experiment of efficient verification to signatures with properties and to the generic security experiment described above is rather straightforward. Similarly, our

```
$\mathsf{Exp}^{\mathsf{generic-unf}}_{\mathcal{A},\Sigma}(\lambda)$
1 : $(\mathsf{pval}, \mathsf{sval}) \leftarrow \mathsf{Setup}(1^\lambda)$
2 : $\mathsf{st}_K \leftarrow \emptyset,\ \mathsf{st} \leftarrow \emptyset$
3 : $(\mu^*, \boldsymbol{\sigma}^*, \mathsf{aux}^*) \leftarrow \mathcal{A}^{OK(\cdot;\mathsf{sval},\mathsf{st}_K),\ O(\cdot;\mathsf{st})}(\mathsf{pval})$
4 : if $\mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*, \mathsf{aux}^*) = 1 \ \wedge\ 1 \leftarrow \mathsf{WinCond}(\mu^*, \boldsymbol{\sigma}^*, \mathsf{aux}^*, \mathsf{pval}, \mathsf{st}_k, \mathsf{st})$
5 :     return 1
6 : else return 0.
```

Fig. 4: Generic description of the unforgeability under adaptive chosen message attacks experiment for signatures with properties.

compiler of Section 3.1 can be easily adapted to signatures with properties that have a similar $\mathbf{Mv}$-style verification. In the following we analyze the impact of our compiler on the efficiency of some schemes whose verification is structured as in Figure 2: the constrained LBS in [34], the (indexed) attribute-based LBS in [23], the homomorphic LBS in [16], the threshold LBS in [4], and the multivariate-based ring signature RingRainbow [29].

Constrained Signatures (CS). CS allow a signer to sign a message only if either the message or the key satisfies certain preset constraints. The verification algorithm of the lattice-based instantiation of CS by Tsabary [34] is syntactically equivalent to the 'standard' verification: it includes a '$\mathbf{Mv}$'-style check (where the matrix has n rows) and a norm check. Hence, our compiler applies directly to this scheme. Unforgeability requires that $n \geq \lambda$ and $q \leq 2^{\lambda}$, so for an average value $q \sim 2^{32}$, we can set $k = 8 \ll \lambda$ so that $1/q^k = 1/2^{256}$. Remark that larger values of q (that could be required to have higher security guarantees) imply smaller values of k. Therefore, for this less conservative choice of parameters the efficiency gain is $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k}{n} = \frac{8}{256} < 0.032$, i.e., the online verification requires about 3.2% of the computational cost of running the standard verification algorithm.

Indexed Attribute-based Signatures (iABS) and Homomorphic Signatures (HS). iABS allow a signer to generate a valid signature on a message only if the signer holds a set of attributes that satisfy some policy (represented by a circuit C). HS allow a signer to sign messages μ_i so that it is possible to publicly derive a valid signature for a message μ that corresponds to the output of a computation on the original messages, i.e., $\mu = \mathsf{C}(\mu_1, \ldots, \mu_r)$. According to the type of homomorphism supported by the scheme, the circuit C can encode only linear functions, polynomial functions, or any function of bounded multiplicative degree. In both iABS in [23] and HS in [16] the signature verification is composed by three steps:

1. Computation of the public matrix $\mathbf{M}$ from the circuit C (either the policy, or the homomorphic computation specified by the labelled program);
2. An '$\mathbf{Mv}$'-style check;
3. A norm check on the signature.

The first step is critical because the public matrix $\mathbf{M}$ can be generated through a non-linear transformation, i.e., it might include multiplications of the public matrix by itself (or by a gadget matrix). This would not allow to compute the first step online from the $\mathbf{z}_i$'s, but the verifier would have to use $\mathbf{M}$ and the $\mathbf{c}_i$'s instead, defying the purpose of our compiler. Hence, our compiler can be applied to these signatures in an efficient way only if either (1) C involves solely linear operations on the public matrix, or (2) C is fixed, or (3) C is known before running verification. In these cases, we achieve efficient verification by letting offVer take as (additional) input C and compute $\mathbf{M}$ using the algorithm $\mathsf{PubEval}$ from [22]. The vectors $(\mathbf{Z}', \mathbf{v})$ used in the verification might (as in [16]) or might not (as in [23]) depend on the message. In the latter case the subroutine GetZV in onVer simply returns the input.
The impact of the compiler on the efficiency of HS was already analyzed in Section 3.3. Regarding the iABS, the suggested value of the modulus q is such that $q \geq n^8$. The standard requirement $n \geq 2$ already implies that $1/q^k \leq 1/(2^8)^k = 1/256^k$. However, to guarantee the hardness of lattice-based problems usually n needs to be at least $n = 128$. In this case $q \geq 2^{56}$, hence $k = 5$ already guarantees the unforgeability of this iABS. As $n = O(d \log d)$ (where d is the depth of C) and the efficiency gain can be bounded as follows: $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} \leq \frac{k}{O(d \log d)} = \frac{5}{O(d \log d)}$. From this

inequality is clear that already for a circuit of depth 4 the online verification only requires 62.5% of the computation required by standard verification; the impact of our compiler increases for larger size of the circuit.

Threshold Signatures (TS). TS allow h out of ℓ parties to produce a signature on a message. Unforgeability is guaranteed for up to t colluding parties. Tsabary [34] introduced a compiler that allows to distribute the signature generation step of the GPV08 signature, and convert it into a TS. The idea is to share the signing trapdoor among the parties using a h-out-of-ℓ secret sharing scheme. Signing requires at least h parties to come together to generate a signature satisfying a $\mathbf{Mv}$-type equation (where $\mathbf{M}$ is the public verification key). Verification is composed by the standard $\mathbf{Mv}$ equation and norm checks. Therefore, the thresholdizing compiler is composable with our compiler for efficient verification. As neither of them change the parameters of the underlying GPV08 scheme, the efficiency gain is the same (cf. Section 3.3).

RingRainbow [29]. RingRainbow is a ring signature scheme – i.e., a signature that allows a user to sign a message anonymously on behalf of a group – based on multivariate equations. This scheme is a hash-and-sign type of signature built as a modification of Rainbow. Verification requires to check whether the signature satisfies a multivariate quadratic system, and can be converted in a $\mathbf{Mv}$-style verification with the same technique used for Rainbow (cf. Section 3.3). Therefore, our compiler can be applied to RingRainbow as well. To evaluate the efficiency gain due to our compiler, we consider the efficient version of RingRainbow, (whose parameters can be found in Table 2 in [29]). For $\lambda = 128$ and a group of 5 users the authors set $\mathbb{F} = \mathbb{F}_{2^8}$ and $(v_1, o_1, o_2) = (36, 21, 22)$, which yield $m = 5 * (v_1 + o_1 + o_2) = 395$ and $n = 43$. Theorem 1 requires at least $1/q^k = 1/2^{256}$ for 128 bits of post-quantum security, hence $k = 32$. Plugging this in Equation (3.3) yields that the minimum number of repetitions r_0 to achieve amortized efficiency is $\mathsf{r}_0 = 126$, that results in an amortization factor of $\mathsf{e}_0 = 32/126 + 32/43 = 0.744$. In this case, our compiler produces an online verification that is $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k}{n} = \frac{32}{43} < 0.75$, in other words, our compiler produces an online verification that requires only 75% of the computation required by the standard verification.

4 Modeling Stateful and Flexible Verification

We introduce our framework for *flexible* verification of digital signatures. The basic idea is to design a flexible verification algorithm, $\mathsf{flexVer}$, that instead of returning a binary answer (accept/reject) as done by Ver, outputs a value α expressing the confidence on the validity of the signature. The confidence α is proportional to the amount of computation invested in the verification: the more verification steps performed the higher the confidence. In this section we show how to realize flexible verification disregarding performance impacts. In the next section we address the question to securely achieve efficient and flexible verification simultaneously.

Differently from Le et al. [25], who define flexible signatures as a standalone primitive, we prefer to treat flexible verification as an add-on feature to enhance existing schemes. Compared to [25] our model is more general, yet slightly more involved. In particular, we let $\mathsf{flexVer}$ be a *stateful* algorithm; the state st is maintained by the verifier and should not be disclosed to the adversary. Albeit st can be seen as a secret, updatable, compact verification key, we remark that it is always possible to verify signatures using the signer's public key and the standard verification procedure. Moreover, any verifier can generate a st from the signer's pk, but st does not allow the verifier to forge signatures in the name of the signer.

4.1 Syntax

The core idea behind flexible verification is to derive from the original verification procedure of a signature scheme (Ver) an alternative algorithm $\mathsf{flexVer}$ with certain properties. Concretely, this task boils down to identifying a sequence of $K_{\mathsf{flex}} + 1$ atomic instructions (that we informally call 'steps') that ensure that, if interrupted at any point, $\mathsf{flexVer}$ returns a meaningful value about the correctness of the signature. In more detail, if one of the steps fails, $\mathsf{flexVer}$ returns $\alpha = \perp$ and rejects the signature. On the other hand, if none of the initial t steps fails, it is possible to derive a real value $\alpha_{\mathsf{flex}}(t) \in [0, 1]$ stating the probability that the given signature is valid. Intuitively, we want that the more steps $\mathsf{flexVer}$ executes successfully, the higher the value of $\alpha_{\mathsf{flex}}(t)$.

Definition 6 (Flexible Verification). *Let $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ be a signature scheme. We say that Σ admits flexible verification if there exists a non-negative integer $K_{\mathsf{flex}} \in \mathbb{Z}_{\geq 0}$, an efficiently computable confidence function $\alpha_{\mathsf{flex}} : \{0, \ldots, K_{\mathsf{flex}}\} \to [0, 1]$, a set of admissible states*[5] *$\mathcal{S}$ (that depends on K_{flex} and a public key pk), and an algorithm $\mathsf{flexVer}$ consisting of $K_{\mathsf{flex}}+1$ steps $\mathsf{flexVer}_0$, ..., $\mathsf{flexVer}_{K_{\mathsf{flex}}}$, such that $\mathsf{flexVer}$ has the following syntax:*

$\mathsf{flexVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$**:** *This a randomized algorithm takes in input a state $\mathsf{st} \in \mathcal{S}$, a public key pk (generated by KeyGen), a message μ, a signature $\boldsymbol{\sigma}$ and an interruption position $t \in \mathbb{Z}_{\geq 0}$. It outputs a value $\alpha \in [0, 1] \cup \perp$. Concretely, $\mathsf{flexVer}$ works as shown below: it is made of $K_{\mathsf{flex}} + 1$ algorithms $\mathsf{flexVer}_j(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$ that update the state and return a bit b. If $b = 0$ the flexible verification outputs $\alpha = \perp$ ($\boldsymbol{\sigma}$ is rejected); otherwise, $b = 1$ and the flexible verification upgrades the confidence level to $\alpha \leftarrow \alpha_{\mathsf{flex}}(j)$ ($\boldsymbol{\sigma}$ is valid at step j).*

```
flexVer(st, pk, $\mu$, $\boldsymbol{\sigma}$, $t$)
1: $\alpha \leftarrow \perp$
2: if $t < 0$ :   return $\perp$
3: if $t > K_{\mathsf{flex}}$ :   set   $t \leftarrow K_{\mathsf{flex}}$
4: for $j = 0, \ldots, t$
5:    $(b, \mathsf{st}) \leftarrow \mathsf{flexVer}_j(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$
6:    if $(b = 0)$ :   return $\perp$
7:    else $(b = 1)$ :   $\alpha \leftarrow \alpha_{\mathsf{flex}}(j)$
8: return $\alpha$
```

*Additionally $\mathsf{flexVer}$ satisfies **flexible correctness**: For a given security parameter λ, for any key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, for any admissible state $\mathsf{st} \in \mathcal{S}$, for any $\mu \in \mathcal{M}$ and signature $\boldsymbol{\sigma}$ such that $\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 1$ and for any value $t \in \{0, \ldots, K_{\mathsf{flex}}\}$, it holds:*

$$\Pr[\mathsf{flexVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t) = \alpha_{\mathsf{flex}}(t)] = 1.$$

Flexible correctness as defined above essentially states that on valid signatures, i.e., signatures that would be accepted by the standard verification, $\mathsf{flexVer}$ accepts with confidence $\alpha_{\mathsf{flex}}(t)$. We follow the approach of [25] and let the flexible verification algorithm output a value α that either rejects the signature ($\alpha = \perp$), or accepts it with certainty α in the real interval $[0, 1]$. We use the same interruption variable t as in [25] to model runtime interruptions of the algorithm execution.[6] We

[5] The set $\mathcal{S}$ includes the initial state $\mathsf{st} = \emptyset$ and any possible state output by some $\mathsf{flexVer}_i$.

[6] Our $\alpha_{\mathsf{flex}}(\cdot)$ is essentially the inverse of the function $\mathsf{iExtract}_{\Sigma}(\cdot)$ in [25].

point out that while t is input to $\mathsf{flexVer}$, it is *not* given to each $\mathsf{flexVer}_j$; this syntax models the fact that the algorithm must work without knowing where it will stop, which is essential to capture arbitrary interruptions. In the case of an interruption, $t < K_{\mathsf{flex}}$, the algorithm would return a *premature* decision α on the validity of the signature: if $\alpha = \perp$ the signature is invalid (and this is certain, there are no false negatives); if $\alpha \in [0, 1]$ this means that according to the t steps performed, the signature seems valid.

Defining the confidence function $\alpha_{\mathsf{flex}}(\cdot)$ is fundamental to showing a construction of flexible verification for a given signature scheme. Any signature scheme Σ admits trivial flexible verification and confidence function. One way to see it is to consider Ver as a sequence of K_Σ instructions (e.g., computation or verification steps); set $K_{\mathsf{flex}} = K_\Sigma$ and $\mathsf{flexVer}_j$ be the j-th step of Ver. So the confidence function can be defined as

$$\alpha_{\mathsf{flex}}(t) = \begin{cases} 0 \text{ for } t \in \{0, \dots, K_\Sigma - 2\} \\ 1 \text{ for } t = K_\Sigma - 1 \end{cases}$$

A trivial confidence function entails un-interesting flexible verification: on correct signatures, $\mathsf{flexVer}$

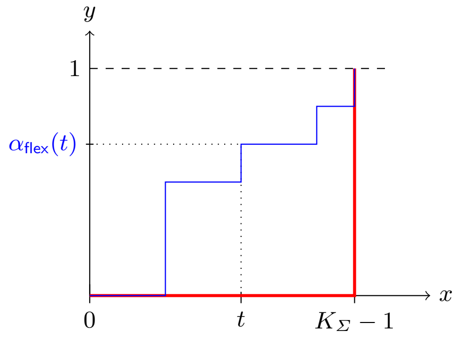

Fig. 5: Examples of confidence functions for flexible verification: the trivial case (thick, red); a non-trivial case (thin, blue).

returns 0 if interrupted at any step $t < K_\Sigma - 1$. While all signature schemes admit a trivial confidence function, not all support a non-trivial one, (e.g., a step function as shown in Figure 5). We remark that the confidence function $\alpha_{\mathsf{flex}}(\cdot)$ depends both on the scheme Σ and on how each $\mathsf{flexVer}_i$ is defined. In Section 4.3 we show non-trivial realizations for post-quantum secure signature schemes based either on multivariate quadratic equations or on lattices. It is easy to see that, without loss of generality, $\alpha_{\mathsf{flex}}(\cdot)$ has a non-decreasing trend, $\alpha_{\mathsf{flex}}(0) = 0$ and $\alpha_{\mathsf{flex}}(K_{\mathsf{flex}}) \leq 1$. Looking ahead, we will define $\alpha_{\mathsf{flex}}(t)$ to be $1 - \mathsf{bad}(t)$, where $\mathsf{bad}(t)$ represents the probability of accepting a forgery after t verification steps. While $\mathsf{bad}(t)$ decreases, $\alpha_{\mathsf{flex}}(t)$ increases and so does the confidence in the correctness of the signature.

Flexible Verification vs. Offline/Online Signature Verification. While both flexible verification and efficient verification may be seen as possible ways of reducing the computational cost of a signature verification, the way this objective is achieved in the two models is quite different. One can think that these are two sides of the same coin. In flexible verification, the verification algorithm (and thus the verifier) is unaware of when the computation will be interrupted, and *its execution is independent of* t. On the other hand, the offline verification (and thus the verifier) sets precisely what is the confidence level k –which may be seen as an interruption value for onVer– that otherwise would check all of the verification checks imposed by Ver. The fundamental difference is that in the latter setting, the (online) verification is aware of the confidence level k (seen as interruption value), and *adapts its execution to k.*

Stateful vs. Stateless Verification We choose to define flexible signatures as stateful algorithms to keep the framework as general as possible. The same model could capture stateless flexible verification by considering the special case in which the state st is always $\varnothing$. In the latter case, there is no need for a verification oracle in the security experiment (Figure 6). In this sense, our approach is more general than the one by Le et al. [25] as we can potentially capture more schemes.

4.2 Security Model

Our security model revisits the notion of existential unforgeability under chosen message attack for flexible signatures proposed by Le et al. [25] with three twists.

First, our security game needs to take into account that $\mathsf{flexVer}$ maintains a possibly non trivial state. We handle this by allowing the adversary to interact with the flexible verification oracle $O\mathsf{flexVer}$ during the query phase (similarly to what we did in the efficient verification model of Section 3) and with the signing oracle $O\mathsf{Sign}$, in a concurrent manner.

Second, queries to $O\mathsf{flexVer}$ have the form $(\mu, \boldsymbol{\sigma}, t')$, where t' is the desired interruption value submitted by $\mathcal{A}$ (and chosen adaptively). To make the model generic and reduce the assumptions on $\mathcal{A}$, we include an interruption oracle $O\mathsf{Int}$ that takes as input t' and returns the actual value t to be used in $\mathsf{flexVer}$. We discuss in detail the role of $O\mathsf{Int}$ in a dedicated paragraph after stating our security notion. For the purpose of this work, we consider the strongest security model in which the interruption oracle returns the adversary's value, i.e., $t \leftarrow O\mathsf{Int}(t')$ with $t = t'$. This resembles side-channel attack settings, where $\mathcal{A}$ may try to freeze the execution of the verification.

Finally, a minor change: instead of outputting a single bit, our experiment returns a pair (b, t^*). The bit $b \in \{0, 1\}$ flags the absence or the potential presence of a forgery, while $t^* \in \{0, \dots, K_{\mathsf{flex}}\}$ reports the interruption position used in the final flexible verification. Including t^* in the output of the experiment allows us to measure security in terms of *how far* is the probability of $\mathcal{A}$ 'winning' the experiment, from the expected value $\alpha_{\mathsf{flex}}(t^*)$. Concretely, we consider a flexible verification to be *secure* if, for any PPT adversary, the probability that the security experiment returns $(1, t^*)$ is only negligibly higher than the expected probability of non detecting a forgery after t^* verification steps. For the purpose of this work, t^* is the interruption parameter chosen by $\mathcal{A}$ for its forgery.

Our security game and experiment for existential unforgeability under adaptive chosen message *with flexible verification* attack ($\mathsf{flexEUF}$) are reported in Figure 6.

Definition 7 (Security of Flexible Signatures ($\mathsf{flexEUF}$)). *Let Σ be a signature scheme that admits a non-trivial realization of flexible verification* $\mathsf{flexVer}$ *with corresponding confidence function* α_{flex}*. For a given security parameter λ,* $(\mathsf{flexVer}, \alpha_{\mathsf{flex}})$ *realize a secure flexible verification for Σ if for all PPT adversaries $\mathcal{A}$ the success probability in the* $\mathsf{flexEUF}$ *experiment in Figure 6 is negligible, i.e.,:*

$$Adv_{\mathcal{A},\Sigma}^{\mathsf{flexEUF}}(\lambda) = \Pr\Big[\mathsf{Exp}_{\mathcal{A},\Sigma}^{\mathsf{flexEUF}}(\lambda) = (1, t^*)\Big] - (1 - \alpha_{\mathsf{flex}}(t^*)) = \varepsilon \leq \varepsilon(\lambda).$$

Intuitively, Definition 7 states that an adversary has only negligible probability to make the flexible verification output a confidence value α^* higher than the expected one. Let $\mathsf{bad}(t)$ denote the probability of accepting a forgery after t verification steps. Then by setting $\alpha_{\mathsf{flex}}(t) = 1 - \mathsf{bad}(t)$, we get $Adv_{\mathcal{A},\Sigma}^{\mathsf{flexEUF}}(\lambda) = \Pr\big[\mathsf{Exp}_{\mathcal{A},\Sigma}^{\mathsf{flexEUF}}(\lambda) = (1, t^*)\big] - \mathsf{bad}(t^*) \leq \varepsilon(\lambda)$.

```
flexEUF ($\Sigma$, $\lambda$)
1 :  $\mathsf{L}_S \leftarrow \varnothing$
2 :  $\mathsf{st} \leftarrow \varnothing$
3 :  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
4 :  $(\mu^*, \boldsymbol{\sigma}^*, t') \leftarrow \mathcal{A}^{O\mathsf{Sign}, O\mathsf{flexVer}}(\mathsf{pk}, \lambda)$
5 :  return $(\mu^*, \boldsymbol{\sigma}^*, t')$

Exp $^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda)$
1 :  $(\mu^*, \boldsymbol{\sigma}^*, t') \leftarrow \mathsf{flexEUF}(\Sigma, \lambda)$
2 :  $\beta \leftarrow \mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*)$
3 :  $t^* \leftarrow O\mathsf{Int}(t')$
4 :  $\alpha \leftarrow \mathsf{flexVer}(\mathsf{st}, \mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*, t^*)$
5 :  if $\mu^* \in \mathsf{L}_S \vee \alpha = \perp \vee \beta = 1$
6 :      return $(0, t^*)$
7 :  return $(1, t^*)$

$O\mathsf{Sign}_{\mathsf{sk}}(\mu)$
1 :  $\mathsf{L}_S \leftarrow \mathsf{L}_S \cup \{\mu\}$
2 :  $\boldsymbol{\sigma} \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$
3 :  return $\boldsymbol{\sigma}$

$O\mathsf{flexVer}_{\mathsf{st},\mathsf{pk}}(\mu, \boldsymbol{\sigma}, t')$
1 :  $t \leftarrow O\mathsf{Int}(t')$
2 :  $\alpha \leftarrow \mathsf{flexVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$
3 :  return $\alpha$
```

Fig. 6: Security game, experiment and oracles for stateful and flexible signature verification: unforgeability under adaptive chosen message with flexible verification attack.

In this work, we prove security in the strongest model where $t' = t$, i.e., $\mathcal{A}$ has the power to choose when to stop the verification. Since we put no restriction on the values t queried by $\mathcal{A}$ to $O\mathsf{flexVer}$ during the game, we will see that by running $O\mathsf{flexVer}$ on 'too few' steps, $\mathcal{A}$ may learn information about the internal state st.

Modelling Interruptions. In [25], unexpected interruptions are modeled via an interruption oracle $\mathsf{iOracle}(\lambda)$ that returns a value $t \in \{0, \dots, K_{\mathsf{flex}}\}$ used by the flexible verification. However, it is not clear whether $\mathcal{A}$ may control $\mathsf{iOracle}$. We overcome these ambiguities by letting $\mathcal{A}$ output t' at every flexible verification query. It is possible to relax and generalize our model by introducing an interruption oracle $O\mathsf{Int}$, programmed at the beginning of the game. At each verification query, $O\mathsf{Int}$ takes as input the adversary's suggestion for an interruption position t' and outputs the value t to be used by the flexible verification. In case $t = t'$, we are modelling side channel attacks, but we can also let t be independent of t'. A realistic definition of $O\mathsf{Int}$ is outside the scope of this work.

4.3 A compiler for flexible '$\mathbf{Mv}$'-style Verifications

We exhibit a compiler to obtain flexible verification for the class of post quantum signature schemes with '$\mathbf{Mv}$'-style verification introduced in Section 3.1. Specifically, our compiler for flexible verification builds on our previous compiler for efficient verification.

Two Insecure Solutions. Let us warm up with an instructive example. For schemes with '$\mathbf{Mv}$'-style verification there is a natural way to divide the verification algorithm into atomic steps: the initial step, $\mathsf{flexVer}_0$, includes the consistency checks (e.g., the norm bound on the signature); while for $i = 1$ to $K_{\mathsf{flex}} = rows(\mathbf{M})$, $\mathsf{flexVer}_i$ performs the check $\mathbf{M}[i, *] \cdot (\boldsymbol{\sigma}, \mathbf{u}) = 0 \bmod q$, where $\mathbf{M}[i, *]$ denotes the i-th row of the matrix $\mathbf{M}$. Notably this solution does not require a secret state. Checking only $t < K_{\mathsf{flex}}$ constraints, however, may not be a secure choice. Indeed, an adversary that knows $\mathbf{M}$ and $\mathbf{u}$ can efficiently produce a signature that verifies the first $t << K_{\mathsf{flex}}$ linear constraints and trivially break unforgeability ($\mathsf{flexEUF}$). A naive way to bypass this issue is to randomize the order in which the steps are performed (assuming $\mathsf{flexVer}_0$ is always run). In this case, setting $t^* = 1$, the adversary can produce a valid forgery for say the i^*-th row of $\mathbf{M}$ and win the $\mathsf{flexEUF}$ game with probability $1/K_{\mathsf{flex}}$ (which corresponds to the probability that $\mathsf{flexVer}$ executes only $\mathsf{flexVer}_0$ and $\mathsf{flexVer}_{i^*}$). Notably, $1/K_{\mathsf{flex}}$ is not negligible, so neither of the two presented approaches is secure.

```
flexVer_0(st, pk, μ, σ)                         flexVer_i(st, pk, μ, σ)

1 : svk ← offVer(pk, $K_{flex}$)                 1 : $b \leftarrow 0$
2 : parse svk = ($K_{flex}$, $\mathbf{Z}$, $PK$.aux)   2 : parse st = ($\mathbf{Z}'$, $\mathbf{v}$)
3 : $b$ ← Check($PK$.aux, μ, σ)                  3 : if $\mathbf{Z}'[i,*] \cdot \mathbf{v}[*,i] = 0 \mod q$
4 : st ← GetZV(svk, μ, σ)                         4 :     return ($b \leftarrow 1$, st)
5 : return ($b$, st)                              5 : return ($b \leftarrow 0$, st)
```

$$\alpha_{\mathsf{flex}} : \{0, \ldots, K_{\Sigma}\} \to [0,1], \qquad \alpha_{\mathsf{flex}}(t) = (1 - \tfrac{1}{q^t})$$

Fig. 7: Generic compiler for flexible verification of $\mathbf{Mv}$-style signatures. The algorithms offVer, Check and GetZV are defined in Section 3.1, Figure 3. Here $K_{\mathsf{flex}} = rows(\mathbf{M})$.

Our Compiler. We define the algorithm flexVer by describing its atomic steps $\mathsf{flexVer}_i$. Analogously to Section 3.1, we present a generic compiler for signatures with $\mathbf{Mv}$-style verification in Figure 7.

To formally define flexible verification we need to specify the value K_{flex} (total verification steps), the set $\mathcal{S}$ (valid states), and the confidence function α_{flex}. The value K_{flex} sets the upper bound on the number of linear constraints the verifier wants to check, hence $K_{\mathsf{flex}} = rows(\mathbf{M})$, where $\mathbf{M}$ is the matrix employed in the matrix-vector multiplication check in the signature verification. The set $\mathcal{S}$ includes $\emptyset$ and any possible state output by some $\mathsf{flexVer}_i$, specifically $\mathcal{S} = \{0,1\} \times \mathbb{Z}_q^{rows(\mathbf{Z}') \times cols(\mathbf{Z}')} \times \mathbb{Z}_q^{rows(\mathbf{v}) \times cols(\mathbf{v})} \times \{0,1\}^{\lambda} \cup \emptyset$.

We extract the confidence level from the probability of a flex forgery (as motivated by the proof of security given in Theorem 1). Alike efficient verification, the probability that an adversary creates a flex forgery with interruption step t is $1/q^t$. If the size of the underlying algebraic structure is $q = 2^{\mathsf{poly}(\lambda)}$ this probability is negligible even for $t = 1$, thus for signatures with large q efficient verification and flexible verification coincide, trivially. The interesting case is $q = \mathsf{poly}(\lambda)$, as the adversary could create a flex forgery with non negligible probability. We remark that in this section we are not targeting efficiency, and our instantiations of flexible verification refresh the svk produced by offVer at every verification query. This way, $\mathcal{A}$ cannot exploit the information possibly leaked by a flex forgery in future forgery attempts.

Theorem 2. *Let Σ be a signature scheme with $\mathbf{Mv}$-style verification (as of Fig. 2). Then our compiler for flexible verification (in Figure 7) returns* (flexVer, α_{flex}) *that realize secure flexible verfication for Σ.*

Proof. Recall that an adversary $\mathcal{A}$ wins the security experiment in Definition 7 if it outputs a pair message-signature $(\mu^*, \boldsymbol{\sigma}^*)$ and an interruption t' such that (1) $(\mu^*, \boldsymbol{\sigma}^*)$ is rejected by Ver, but accepted by the flexible verification flexVer when it is interrupted at step $t^* \leftarrow O\mathsf{Int}(t')$, and (2) the flexible verification algorithm outputs a too high confidence level $\alpha_{\mathsf{flex}}(t^*)$. Following Definition 7 we can realize secure flexible verification by setting $\alpha_{\mathsf{flex}}(t) = 1 - \Pr\left[\mathsf{Exp}^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1,t)\right] + \varepsilon(\lambda) \;\; \forall\, t$. The core part of the proof is to estimate this probability.

Recall that our compiler for efficient $\mathbf{Mv}$-style verification (in Figure 7) runs offVer at every verification query (step 1 in $\mathsf{flexVer}_0$). This means that every verification query is answered using a freshly generated svk. In particular, the final verification (line 4 in the $\mathsf{Exp}^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda)$ in Figure 8) checks $\mathcal{A}$'s output using independent randomness from the previous queries. So, whatever information the adversary may collect from previous queries is useless to win the experiment. As a consequence, the probability that the adversary wins the game equals the probability that the adversary outputs a valid forgery *without querying* $\mathcal{O}$flexVer. The latter is precisely the probability of a bad forgery we investigated in Theorem 1, hence $\Pr\left[\mathsf{Exp}^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1, t^*)\right] = \frac{1}{t^*}$ and:

$$Adv^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1, t^*)\right] - (1 - \alpha_{\mathsf{flex}}(t^*))$$
$$= \frac{1}{q^{t^*}} - \left(1 - \left(1 - \frac{1}{q^{t^*}}\right)\right) = 0 \ .$$

□

4.4 Flexible and Efficient Verification

Given the two notions of efficient verification (Section 3) and flexible verification (Section 4), the reader might be wondering now whether it would be possible to build a compiler for flexible *and* efficient verification of **Mv**-style signatures. Our approach is to start from our compiler for flexible verification (depicted in Fig. 7), and skip line 1 of $\mathsf{flexVer}_0$ (where it runs offVer to generate svk) in all verifications except for the first one. This means that svk is created at the first verification and reused in all subsequent runs of flexVer. While this obviously improves the (amortized) efficiency of the compiler, it also impacts security.

We now verify the unforgeability of our compiler for q exponential. From Theorem 1 we already know that in this case verification only checks the linear equation for one $\mathbf{c}$. Hence adding flexibility to the model does not impact it, as the adversary can only interrupt verification either before or after the linear equation has been checked.

Theorem 3 (informal). *For signatures with* **Mv***-style verification relying on algebraic structures of size* $q = 2^{\mathsf{poly}(\lambda)}$ *our compiler outlined for efficient flexible verification is unforgeable according to Definition 7 and achieves* $(2, 1/2 + 1/rows(\mathbf{M}))$*-concrete amortized efficiency as per Definition 4.*

Proof. (sketch) The proof is based on the same argument used in the proof of Theorem 1. Using the same notation as in Theorem 1, take the same sequence of q_V hybrid games and replace any appearance of onVer by flefVer (essentially each game delays the query on which the first flexible verification is run). Since q is exponential in the security parameter, the interruption values t' can be all set to 1. We can bound the adversary's success probability by the occurrence of bad events.

$$\Pr\left[\mathsf{Exp}^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1, t^*)\right] \leq \sum_{i=1}^{q_V} \Pr[\mathsf{bad}_i] + \Pr[\mathbf{G}_{q_V} = 1] \leq \frac{q_V}{q} + \frac{1}{q} \ .$$

We used $\Pr[\mathbf{G}_{q_V} = 1] \leq \frac{1}{q}$ since in that game the vectors svk is freshly generated. To conclude, $q = 2^{\mathsf{poly}(\lambda)}$ and $q_V \leq \mathsf{poly}(\lambda)$ it holds that,

$$Adv^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{flexEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1, t^*)\right] - (1 - \alpha_{\mathsf{flex}}(t^*))$$
$$\leq \frac{q_V}{q} + \frac{1}{q^{t^*}} - \left(1 - \left(1 - \frac{1}{q^{t^*}}\right)\right) = \frac{q_V}{q} = \varepsilon(\lambda) \ .$$

In this setting the concrete efficiency of our compiler is achieved already with $\mathsf{r}_0 = 2$ repetitions, and the corresponding amortized efficiency value is $\mathsf{e}_0 = \frac{\mathsf{cost}(\mathsf{offVer}(\mathsf{pk},1))}{\mathsf{r}_0 \cdot \mathsf{cost}(\mathsf{Ver}(\mathsf{pk},\mu,\boldsymbol{\sigma}))} + \frac{\mathsf{r}_0 \cdot \mathsf{cost}(\mathsf{onVer}(\mathsf{svk},1,\mu,\boldsymbol{\sigma}))}{\mathsf{r}_0 \cdot \mathsf{cost}(\mathsf{Ver}(\mathsf{pk},\mu,\boldsymbol{\sigma}))} = \frac{1}{2} + \frac{1}{rows(\mathbf{M})}$. □

In case $q = \mathsf{poly}(\lambda)$, the freshness argument on svk can no longer be used to bound the probability of finding a forgery: some non-negligible amount of information about the secret combinators $\mathbf{c}_i$ in svk is leaked at every flex-forgery submitted during verification queries. To formally handle these cases, in the next Section we introduce the notion of r-bounded randomness reuse.

5 Modeling Efficient & Flexible Signature Verification with r-Bounded Randomness Reuse

We introduce the concept of *fl*exible and *ef*ficient *s*ignature verification (flefs) with r-bounded randomness reuse (or r-flefs in short). Similarly to Definition 6 (flexible signatures), this *sustainable* variant is defined for a given value k, that determines the maximum desired confidence level achievable by the verification. In addition to k, we need a second parameter, r, that determines the maximum number of times the verification randomness can be reused. For correctness and security, both k and r are input to the confidence function, which now is named α_{flef}.

Definition 8 (r-flefs). *A signature scheme Σ admits a (k, r)-efficient and flexible verification realization if there exist two positive integers k and r; an efficiently computable confidence function $\alpha_{\mathsf{flef}} : \{0, \ldots, k\} \times \{0, \ldots, \mathsf{r}\} \to [0, 1]$; a set of admissible sequences of states $\mathcal{S} = \{st^{(1)}, st^{(2)}, \ldots\}$ (each sequence $st^{(j)}$ contains $\mathsf{r}+1$ states, i.e., $st^{(j)} = (\mathsf{st}_i)_{i=0}^{\mathsf{r}}$, $\mathsf{st}_0 = \varnothing$) and a flexible verification algorithm $\mathsf{flefVer}$ consisting of $k+1$ steps $\mathsf{flefVer}_0, \ldots, \mathsf{flefVer}_k$ satisfying the properties of correctness, efficiency and security stated below.*

r-*Reuse k-Flexible Correctness* For a given security parameter λ, for any key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, for any one sequence of admissible states $st \leftarrow \mathcal{S}$, $st = (\mathsf{st}_i)_{i=0}^{\mathsf{r}}$, for any choice of r message-signature pairs $(\mu_i, \boldsymbol{\sigma}_i)$ with $i \in \{1, \ldots, \mathsf{r}\}$, $\mu_i \in \mathcal{M}$ and $\boldsymbol{\sigma}_i$ such that $\mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 1$ and for any sequence of interruption values $t_i \in \{1, \ldots, k\}$, it holds that:

$$\Pr[\mathsf{flefVer}(\mathsf{st}_i, \mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i, t_i) = \alpha_{\mathsf{flef}}(t_i, i)] = 1 \quad \forall\, i \in \{1, \ldots, \mathsf{r}\}.$$

r_0-*Concrete Amortized Efficiency* For a given security parameter λ, for any key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, for any one sequence of admissible states $st \leftarrow \mathcal{S}$, there exists a minimal positive integer number $\mathsf{r}_0 \leq \mathsf{r}$ for which it holds that:

$$\sum_{i=0}^{\mathsf{r}_0-1} \mathsf{cost}\big(\mathsf{flefVer}(\mathsf{st}_i, \mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i, k)\big) < \mathsf{r}_0 \cdot \mathsf{cost}\big(\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})\big) \tag{6}$$

r-*bounded flexible security* Figure 8 collects a description of our security game and experiment for existential unforgeability under adaptive chosen message attack for signatures with *flexible and efficient verification* (r-$\mathsf{flefEUF}$).

Definition 9 (r-Bounded Flexible Security (r-$\mathsf{flefEUF}$)). *Let Σ be a signature scheme that admits a non-trivial realization of (k, r)-efficient and flexible verification* $\mathsf{flefVer}$. *Then for a given security parameter λ the derived verification is existential unforgeable under adaptive chosen message attack with flexible and efficient verification* (r-$\mathsf{flefEUF}$) *if for all efficient PPT adversaries $\mathcal{A}$ the success probability in the* r-$\mathsf{flefEUF}$ *experiment is:*

$$\Pr\left[\begin{array}{c}\mathsf{Exp}_{\mathcal{A},\Sigma,\mathsf{r}}^{\mathsf{r}\text{-}\mathsf{flefEUF}}(\lambda, k, \mathsf{r}) = (\mathsf{ctr}^*, t^*) \\ \wedge\ (\mathsf{ctr}^*, t^*) \neq (0, 0)\end{array}\right] \leq (1 - \alpha_{\mathsf{flef}}(t^*, \mathsf{ctr}^*)) + \varepsilon(\lambda).$$

```
r-flefEUF($\Sigma, \lambda, k$)
1 : $\mathsf{ctr} \leftarrow 0, \mathsf{st}_0 \leftarrow \varnothing, \mathsf{L}_S \leftarrow \varnothing$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
3 : $(\mu^*, \boldsymbol{\sigma}^*, t^*) \leftarrow \mathcal{A}^{O\mathsf{Sign}, \mathsf{flefVer}}(\mathsf{pk}, \lambda)$
4 : return $(\mathsf{ctr}, \mu^*, \boldsymbol{\sigma}^*, t^*)$

OflefVer_k($\mathsf{st}_{\mathsf{ctr}}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t'$)
1 : if $(\mathsf{ctr} \geq \mathsf{r})$  return $\perp$
2 : $t \leftarrow O\mathsf{Int}(t')$
3 : $\alpha \leftarrow \mathsf{flefVer}(\mathsf{st}_{\mathsf{ctr}}, \mu, \boldsymbol{\sigma}, t)$
4 : if $(\alpha \neq \perp)$
5 :    $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
6 : return $\alpha$
```

```
Exp$^{\mathsf{r\text{-}flefEUF}}_{\mathcal{A}, \Sigma, \mathsf{r}}(\lambda, k)$
1 : $(\mathsf{ctr}, \mu^*, \boldsymbol{\sigma}^*, t') \leftarrow \mathsf{r\text{-}flefEUF}(\Sigma, \lambda, k)$
2 : $\beta \leftarrow \mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*)$
3 : $t^* \leftarrow O\mathsf{Int}(t')$
4 : $\alpha \leftarrow \mathsf{flefVer}_k(\mathsf{st}_{\mathsf{ctr}}, \mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*, t^*)$
5 : if $(\mu^* \in \mathsf{L}_S \vee \alpha = \perp \vee \beta = 1)$
6 :    return $(0, 0)$
7 : return $(\mathsf{ctr}, t^*)$

OSign_sk($\mu$)
1 : $\mathsf{L}_S \leftarrow \mathsf{L}_S \cup \{\mu\}$
2 : $\boldsymbol{\sigma} \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$
3 : return $\boldsymbol{\sigma}$
```

Fig. 8: Security model for signatures with stateful, (k, r)-efficient and flexible verification: security game, experiment and oracles.

5.1 A Generic Compiler

First of all we notice that our compiler for efficient verification described in Section 3.1 is trivially $\infty-$flefEUF (i.e., existentially unforgeable against an unbounded polynomial number of verification queries), by defining $\mathsf{flefVer} = (\mathsf{flefVer}_0, \mathsf{flefVer}_1)$ as shown in Figure 9. Recall that by assumption $\mathsf{st}_0 = \varnothing$ and that for q exponential in the security parameter Theorem 1 shows that we can set $k = 1$ and have unforgeability.

We now present a compiler for signatures with $\mathbf{Mv}$-style verification and $q = \mathsf{poly}(\lambda)$ that realizes efficient bounded flexible verification. Our r-flefs compiler builds on top of the two compilers presented in Section 3.1 and 4.3. Intuitively, the problem with flexible verification is that if interrupted after $t < k$ steps the process may erroneously accept an invalid signature with a non-negligible probability of $1/q^t$. Moreover, from one such forgery the adversary learns that the employed $\mathbf{z}_i$'s satisfy a specific linear system. In particular, using the terminology introduced in the proof of Theorem 1, given one 'bad' forgery, the adversary has higher chance to derive a new 'bad' forgery. In Section 4.3 we mitigate this leakage of information between queries by refreshing the vectors in svk after every verification that outputs $\alpha > 0$. This, however, has a clear negative impact on the efficiency of our solutions. Here we want to prioritize efficiency at the cost of accuracy, and investigate how the confidence function degrades when the same set of vectors $\mathbf{z}_i$ is used to perform r flex-verifications.

Our r-flefs compiler works essentially as the efficient verification compiler in Figure 7, except that the offVer algorithm (that generates a fresh randomness for the secret verification key svk) is run only *once every* r *accepting verifications*, i.e., verifications that return an outcome $\alpha \neq \perp$. To further optimize the scheme, we replace the GetZV algorithm by k algorithms GetZV_i each of which is run by the corresponding $\mathsf{flefVer}_i$. The behavior of GetZV_i depends on the signature scheme and in what follows we define it for each of the three major classes we identified in this paper.

GPV08 [20]: the GetZV_i routine returns $\mathbf{z}'_i = \mathbf{z}_i = \mathbf{c}_i\mathbf{M}$, and $\mathbf{v}_i = [\boldsymbol{\sigma}|\mathcal{H}(\mu)]$.
MP12 [28]: the GetZV_i routine outputs $\mathbf{z}'_i = [\tilde{\mathbf{z}}_i \mid \mathbf{z}^0_i + \sum_{j=1}^{\ell} \mu[j]\mathbf{z}^j_i|\mathbf{c}_i]$ and $\mathbf{v}_i = [\boldsymbol{\sigma}|\mathbf{u}]$.

```
flefVer(st, pk, $\mu$, $\boldsymbol{\sigma}$, $t$)        $k = 1$
1 :  $\beta \leftarrow 0$
2 :  if $t < 0$
3 :    return $\bot$
4 :  if $t > 1$
5 :     set $t \leftarrow 1$
6 :  for $i = 0, t$
7 :    $(\mathsf{st}, \beta) \leftarrow \mathsf{flefVer}_i(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$
8 :    if $\beta = 0$   return $\alpha = \bot$
9 :  return $\alpha = (1 - 1/q^t)$
```

```
$\alpha_{\mathsf{flef}} : \{0,1\} \times \mathbb{Z}_{>0} \to [0,1]$
$\alpha_{\mathsf{flef}}(t, \mathsf{ctr}) = (1 - \frac{1}{q^t} - t \cdot \frac{\mathsf{ctr}}{q})$
```

```
flefVer$_0$(st, pk, $\mu$, $\boldsymbol{\sigma}$)
1 :  parse st = (svk, ctr)
2 :  if st = $\varnothing$
3 :    svk $\leftarrow$ offVer(pk, 1)
4 :    st $\leftarrow$ (svk, 1)
5 :  else st $\leftarrow$ (svk, ctr + 1)
6 :  return (st, 1)
```

```
flefVer$_1$(st, $\mu$, $\boldsymbol{\sigma}$)
1 :  parse st = (svk, ctr)
2 :  $\beta \leftarrow$ onVer(st, $\mu$, $\boldsymbol{\sigma}$)
3 :  return (st, $\beta$)
```

Fig. 9: The trivial two-step $\mathsf{flefVer}$ solution based on our compiler for efficient verification (Figure 3).

Rainbow [14]: the GetZV_i routine outputs $\mathbf{z}'_i = \mathbf{z}_i = \mathbf{c}_i\mathbf{M}$, and $\mathbf{v}_i = [\tilde{\mathbf{s}}|\mathbf{s}|1|\mathbf{h}]$.

Finally, the confidence function $\alpha_{\mathsf{flef}}(\cdot,\cdot)$ is defined as:

$$\alpha_{\mathsf{flef}}(t, \mathsf{ctr}) = \begin{cases} \left(1 - \frac{1}{q^t} - \frac{\mathsf{ctr}}{q}\right) & \text{if } t \text{ ¿0} \\ 0 & \text{if } t = 0 \end{cases} \tag{7}$$

$\mathsf{r_0}$-concrete amortized efficiency. The cost of $\mathsf{flefVer}_i$ varies depending whether $i = 0$ or $i > 0$. When $\mathsf{flefVer}$ is run the first time (or with an empty state), the step $\mathsf{flefVer}_0$ generates the state. This includes computing knm multiplications, in the worst case. After that, every step $\mathsf{flefVer}_i$ computes only $(n + m)$ multiplications (the first term represents the cost of running GetZV_i). Therefore,

$$\mathsf{cost}\big(\mathsf{flefVer}(\mathsf{st}_0, \mu_0, \boldsymbol{\sigma}_0, k)\big) = knm + k(n + m)\ .$$

However, this is true only for the first execution of $\mathsf{flefVer}$, as when executing the verification $1 < \mathsf{r_0} < \mathsf{r}$ times, the algorithm $\mathsf{flefVer}_0$ does not refresh the multipliers. Hence, for $i > 0$

$$\mathsf{cost}\big(\mathsf{flefVer}(\mathsf{st}_i, \mu_i, \boldsymbol{\sigma}_i, k)\big) = k(n + m)\ .$$

This yields $\sum_{i=0}^{\mathsf{r_0}-1} \mathsf{cost}\big(\mathsf{flefVer}(\mathsf{st}_i, \mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i, k)\big) = knm + \mathsf{r_0}k(n + m)$. The cost of a verification is dominated by $\mathsf{cost}\big(\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})\big) = nm$ multiplications, in the worst case. Therefore, Equation (6) yields

$$knm + \mathsf{r_0}k(n + m) < \mathsf{r_0}nm \quad \Rightarrow \quad \mathsf{r_0} > \frac{knm}{nm - k(n + m)}\ .$$

From the above formula we can derive a lower bound on values of r that yield efficiency (recall that by definition $\mathsf{r_0} \leq \mathsf{r}$). A concrete security approach should lead to a meaningful upper bound on the value r that can be safely used in realistic applications. Intuitively, lower values of r yield higher accuracy (and unfogeability), higher ones guarantee better amortized efficiency.

```
$\mathsf{flefVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$
$\beta \leftarrow 0$
if $t < 0$
    return $\perp$
if $t > k$
    set $t \leftarrow k$
parse $\mathsf{st} = (\mathsf{svk}, \mathsf{ctr})$
if $\mathsf{st} \neq \varnothing \wedge \mathsf{ctr} > \mathsf{r}$
    set  $\mathsf{st} \leftarrow \varnothing$
for $i = 0, \ldots, t$
    $(\mathsf{st}, \beta) \leftarrow \mathsf{flefVer}_i(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$
    if $\beta = 0$
        return $\alpha = \perp$
$\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$  // update  st
return $\alpha = (1 - 1/q^t - \mathsf{ctr}/q)$
```

```
$\mathsf{flefVer}_0(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$
parse $\mathsf{st} = (\mathsf{svk}, \mathsf{ctr})$
if $\mathsf{st} = \varnothing$
    $\mathsf{svk} \leftarrow \mathsf{offVer}(\mathsf{pk}, k)$
    $\mathsf{st} \leftarrow (\mathsf{svk}, 0)$  // ctr ← 0
return $(\mathsf{st}, 1)$
```

```
$\mathsf{flefVer}_1(\mathsf{st}, \mu, \boldsymbol{\sigma})$
parse $\mathsf{st} = (\mathsf{svk}, \mathsf{ctr})$
$(\mathbf{z}', \mathbf{v}) \leftarrow \mathsf{GetZV}_i(\mathsf{svk}, \mu, \boldsymbol{\sigma})$
if $\mathbf{z}' \cdot \mathbf{v} = 0 \mod q$
    return $(\mathsf{st}, 1)$
return $(\mathsf{st}, 0)$
```

$$\alpha_{\mathsf{flef}} : \{0, \ldots, k\} \times \{0, \ldots, \mathsf{r}\} \to [0, 1] , \qquad \alpha_{\mathsf{flef}}(t, \mathsf{ctr}) = (1 - \tfrac{1}{q^t} - \tfrac{\mathsf{ctr}}{q}), \text{ if } t > 0$$

Fig. 10: Generic compiler to obtain efficient and flexible verification of signature schemes with **Mv**-style verification and q polynomial in the security parameter.

5.2 On the Concrete Security of r-flefs

The previous section seems to imply that flexibility and efficiency are in general two mutually exclusive properties of a provably secure signature scheme. This is true when considering asymptotic security definitions for lattice-based parameters that are (only) polynomial in λ. However, this does not imply that efficient flexible signatures are always insecure in practice. On the contrary, in this section we show that by limiting the amount of reuse of a secret verification key we get the best of both worlds. Concretely, we show that it is possible to simultaneously be both efficient and (arguably) accurate even for those signatures with **Mv**-style verification that rely on algebraic structures of size q polynomial in the security parameter λ. The idea is to limit the amount of verification queries the adversary is allowed to make during the security game. This simulates the fact that, after r flef verifications that return a value $\alpha > 0$, the verifier re-sets svk, i.e., the secret randomness used for the efficient verification is refreshed. We keep track of this quantity via a counter ctr, as done in the compiler presented before.

In the following we formally prove the security of our compiler and conclude with remarks on actual values.

Theorem 4. *Let Σ be a signature with* **Mv***-style verification. Then our compiler for efficient and flexible verification provided in Figure 10 is* r*-bounded flexible secure for* $\mathsf{r} = k$*.*

Proof. (sketch) The proof proceeds quite similarly to that of Theorem 2 for the case of $q \approx 2^\lambda$. Let $\mathbf{G}_0$ be the r-flefEUF experiment in Figure 8, and define a series of q_V hybrid games where, for every $i = 1$ to q_V, $\mathbf{G}_i$ is the same as $\mathbf{G}_{i-1}$ except that the i-th query to flefVer is answered using

Table 3: Concrete accuracy and efficiency values for realistic parameter choices for latticse based signatures with polynomila modulus: size of $q = 2^{30}$, $n = 256$, $k = 5$, and the worst case scenario where $t = 1$. Values are obtained by setting $\alpha_{\mathsf{flef}}(1, \mathsf{r}) = 1 - (\mathsf{r} + 1)/q$ (cf. Equation (7)) to the stated value, and estracting the corresponding r from the formula. Then, r is plugged in Equation (3.3) to derive the corresponding e.

$\min \alpha_{\mathsf{flef}}$	$1 - 10^{-2}$	$1 - 10^{-4}$	$1 - 10^{-6}$	$1 - 10^{-8}$
r	$\approx 2^{23}$	$\approx 2^{16}$	$1072 \approx 2^{10}$	$9 \approx 2^{3}$
e	$\approx 10^{-6}$	$\approx 10^{-4}$	$0.0047 \approx 10^{-2}$	$1.94596 \approx 10^{-0.2}$

the standard verification algorithm Ver. Let bad_i be the event that the adversary distinguishes $\mathbf{G}_i$ from $\mathbf{G}_{i-1}$ as in Equation (3). Then we have the following

$$\begin{aligned} \Pr\left[\mathsf{Exp}^{\mathsf{r\text{-}flefEUF}}_{\mathcal{A},\Sigma}(\lambda) = (j^*, t^*), j^* > 0\right] &= \Pr[\mathbf{G}_0 = 1] \\ &\leq \sum_{i=1}^{j^*} \Pr[\mathsf{bad}_i] + \Pr[\mathbf{G}_{j^*} = 1] \\ &\leq \sum_{i=1}^{j^*} \frac{1}{q^{t_i}} + \frac{1}{q^{t^*}} \leq \frac{j^*}{q} + \frac{1}{q^{t^*}} \end{aligned}$$

where t_i the interruption value used in the i-th verification, and t^* is the interruption value used in the verification of the forgery. Note that above we used $\Pr[\mathbf{G}_{j^*} = 1] \leq \frac{1}{q^{t^*}}$ since in that game the vectors $\{\mathbf{c}\}$ have never been used until the forgery's verification, hence they can be considered fresh. Also, $\Pr[\mathbf{G}_i = 1] = 0$ for all $i > j^*$ since j^* is the index of the last verification query. Therefore, assuming $t > 0$ (otherwise, by construction $\alpha_{\mathsf{flef}}(0) = 0$) we have:

$$\begin{aligned} Adv^{\mathsf{r\text{-}flefEUF}}_{\mathcal{A},\Sigma}(\lambda, k) &= \Pr\left[\mathsf{Exp}^{\mathsf{r\text{-}flefEUF}}_{\mathcal{A},\Sigma}(\lambda) = (j^*, t^*), j^* > 0\right] - (1 - \alpha_{\mathsf{flef}}(t^*, j^*)) \\ &\leq \frac{j^*}{q} + \frac{1}{q^{t^*}} - \left(1 - \left(1 - \frac{1}{q^{t^*}} - \frac{j^*}{q}\right)\right) = 0 \end{aligned}$$

□

Parameters for concrete security depend on the minimum accuracy that the verifier decides to tolerate, i.e., the minimum value of α_{flef} allowed. Such a value comes up in the worst-case scenario, in which the adversary only allows for $t = 1$ verification steps. From the description of α_{flef} given in Equation (7), the bound r on the number of "bad forgeries" (i.e., signatures whose verification with $\mathsf{flexVer}$ returns $\alpha > 0$ while with Ver returns 0) our compiler can tolerate. Hence, r is the maximum number of times the randomness svk can be reused while having a confidence level of at least α_{flef}. In Table 3 we report some values of r and α_{flef} (1) for different parameters of the main LBS considered thus far. We also provide the lower bound e on the amortized efficiency reached by our compiler, if run on r flef verifications. The third column of Table 3 states the following information: if we run our compiler on a lattice-based signature scheme with $\mathbf{Mv}$-style verification and modulus $q = 2^{30}$ of polynomial size, $n = 256$ with maximum confidence level $k = 5$, we get:

- A $\mathsf{r\text{-}flefs}$ scheme with minimal accuracy 0.0001%; in other words $\mathsf{flefVer}$ may accept an invalid signature once every one million verifications, i.e., our compiler is correct 99.9999% of the times.

– For this accuracy level, we can 'safely' run at most $\mathsf{r} = 1072$ flexible and efficient verifications using the same svk.
– For this accuracy level, running $\mathsf{flefVer}$ this number of repetitions leads to an efficiency ratio $\mathsf{e} = 0.0047$, i.e., our flexible and efficient verification requires 200x less computation than running the standard signature verification about a 1000 times.

Overall this implies the verification yields at least 30 bits of security. This number might seem low when compared to asymptotic security. However, let us compare it with another real-world case, and consider a credit card with a 4 digit PIN code. To mitigate the chance for an attack to be successful the bank tolerates up to 3 unsuccessful trials. Thus, if the PIN is chosen at random, a credit card is secured at a level equal to $3/10000 > 2^{-12}$ bits of security.

Moreover, even for high accuracy levels (where we expect$\mathsf{flefVer}$ to mistakenly not recognize a forgery among one million signatures), we get astonishing efficiency gain, and can reuse the randomness a surprisingly large number of times.

6 Conclusions and Future Work

We presented a study on how to achieve efficiency and flexibility in a signature verification. In addition to putting forth these notions and formal models for them, we also presented two compilers that allow one to realize efficient (resp. flexible) verification for a wide class of existing lattice-based signatures. While our constructions show the feasibility of the desired properties, they also raise some natural follow up questions. For instance, is it possible to realize a compiler for LBS with $q \sim \mathsf{poly}(\lambda)$ that simultaneously provides efficient *and* flexible verification? We address this question in a positive way in Section 5, albeit in weaker security model. A solution with full fledged flexible security remains an interesting open problem. Another question is, is it possible to generalize our approach to other classes of digital signatures, e.g., code-based or LBS obtained through the Fiat-Shamir heuristic or from ideal lattices? Finally, it would be worth to explore more applications of flexible and efficient verification. On top of the already mentioned applications to real-time systems, another possible venue is parallel and distributed verification of digital signatures. Consider a public bulletin board that stores authenticated (signed) data. For security reasons, one may be tempted to use post quantum signature schemes such as LBS. However, the large sizes of the public keys and signatures and the slow speed of the verification are notorious bottlenecks to deploy them in such scenarios. Using our approach, a pool of parties –acting as verifiers– can be made in charge of running each a single verification check (i.e., $\mathsf{flexVer}$ includes only $\mathsf{flexVer}_0$ and $\mathsf{flexVer}_1$). In terms of security, although a single verifier may be wrong with non-negligible probability $1/q$, the probability that k honest verifiers are all wrong becomes negligible already for $k = 5$. Finally, we think that it would be interesting to explore the study of efficient and flexible verification also for more cryptographic primitives, such as commitments and zero-knowledge proofs.

Acknowledgments. This work was partly funded by: ELLIIT, the Swedish Foundation for Strategic Research grant RIT17-0035, SNSF, Swiss National Science Foundation project number 182452. Part of this work was made while C.B. was at IBM Research - Zurich (CH) and visiting the University of Aarhus (DK). This work has also received funding in part from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program under project PICOCRYPT (grant agreement No. 101001283), by the Spanish Government under projects SCUM (ref. RTI2018-102043-B-I00), CRYPTOEPIC (ref. EUR2019-103816), and SECURITAS (ref.

RED2018-102321-T), and by the Madrid Regional Government under project BLOQUES (ref. S2018/TCS-4339).

References

1. M. R. Albrecht, B. R. Curtis, A. Deo, A. Davidson, R. Player, E. W. Postlethwaite, F. Virdia, and T. Wunderer. Estimate all the lwe, ntru schemes! In *Security and Cryptography for Networks SCN*, LNCS, 2018.
2. M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS.*, pages 863–874. ACM, 2013.
3. M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *J. Cryptology*, 16(3):185–215, 2003.
4. R. Bendlin, S. Krehbiel, and C. Peikert. How to share a lattice trapdoor: Threshold protocols for signatures and (H)IBE. In *Applied Cryptography and Network Security ACNS*, LNCS, 2013.
5. W. Beullens, A. Szepieniec, F. Vercauteren, and B. Preneel. Luov: Signature scheme proposal for nist pqc project. 2019.
6. D. Boneh and D. M. Freeman. Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2011.
7. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.
8. X. Boyen. Lattice mixing and vanishing trapdoors: A framework for fully secure short signatures and more. In *International Workshop on Public Key Cryptography (PKC)*, pages 499–517. Springer, 2010.
9. D. Cash, D. Hofheinz, E. Kiltz, and C. Peikert. Bonsai trees, or how to delegate a lattice basis. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 523–552. Springer, 2010.
10. D. Catalano, D. Fiore, and B. Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In *Advances in Cryptology – CRYPTO*, 2014.
11. D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology - CRYPTO*, pages 199–203. Springer, 1983.
12. J. Coron and A. Gini. A polynomial-time algorithm for solving the hidden subset sum problem. In D. Micciancio and T. Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 3–31. Springer, 2020.
13. R. Del Pino, V. Lyubashevsky, G. Neven, and G. Seiler. Practical quantum-safe voting from lattices. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1565–1581, 2017.
14. J. Ding, M.-S. Chen, A. Petzoldt, D. Schmidt, and B.-Y. Yang. Rainbow. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`. Accessed: 2020-09-21.
15. J. Ding and D. Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *Applied Cryptography and Network Security, ACNS. Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.
16. D. Fiore, A. Mitrokotsa, L. Nizzardo, and E. Pagnin. Multi-key homomorphic authenticators. In *International Conference on the Theory and Application of Cryptology and Information Security - ASIACRYPT*, 2016.
17. M. Fischlin. Progressive verification: The case of message authentication. In *International Conference on Cryptology in India*, pages 416–429. Springer, 2003.
18. N. Gama and P. Q. Nguyen. Predicting lattice reduction. In *Advances in Cryptology - EUROCRYPT. Proceedings*, pages 31–51, 2008.
19. R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
20. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *ACM Symposium on Theory of Computing. Proceedings*, pages 197–206, 2008.
21. S. Gorbunov, V. Vaikuntanathan, and D. Wichs. Leveled fully homomorphic signatures from standard lattices. In *ACM symposium on Theory of computing. Proceedings*, pages 469–477. ACM, 2015.
22. S. Gorbunov and D. Vinayagamurthy. Riding on asymmetry: Efficient ABE for branching programs. In T. Iwata and J. H. Cheon, editors, *Advances in Cryptology - ASIACRYPT. Proceedings*, Lecture Notes in Computer Science, 2015.

23. S. Katsumata and S. Yamada. Group signatures without NIZK: from lattices in the standard model. In *Advances in Cryptology - EUROCRYPT*, 2019.
24. L. Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International, 1979.
25. D. V. Le, M. Kelkar, and A. Kate. Flexible signatures: Making authentication suitable for real-time environments. In *European Symposium on Research in Computer Security, (ESORICS)*, pages 173–193. Springer, 2019.
26. L. P. Malasinghe, N. Ramzan, and K. Dahal. Remote patient monitoring: a comprehensive study. *Journal of Ambient Intelligence and Humanized Computing*, 10(1):57–76, 2019.
27. R. C. Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology - ASIACRYPT*, pages 218–238. Springer, 1989.
28. D. Micciancio and C. Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *EUROCRYPT*, 2012.
29. M. S. E. Mohamed and A. Petzoldt. RingRainbow - An efficient multivariate ring signature scheme. In *Progress in Cryptology - AFRICACRYPT. Proceedings*, volume 10239 of *Lecture Notes in Computer Science*, pages 3–20, 2017.
30. T. Plantard, A. Sipasseuth, C. Dumondelle, and W. Susilo. Drs: diagonal dominant reduction for lattice-based signature. In *PQC Standardization Conference*, 2018.
31. A. Sipasseuth, T. Plantard, and W. Susilo. Using freivalds' algorithm to accelerate lattice-based signature verifications. In *International Conference on Information Security Practice and Experience*, pages 401–412. Springer, 2019.
32. A. R. Taleb and D. Vergnaud. Speeding-up verification of digital signatures. *Journal of Computer and System Sciences*, 2020.
33. S. Tayeb, M. Pirouz, G. Esguerra, K. Ghobadi, J. Huang, R. Hill, D. Lawson, S. Li, T. Zhan, J. Zhan, et al. Securing the positioning signals of autonomous vehicles. In *2017 IEEE International Conference on Big Data*, 2017.
34. R. Tsabary. An equivalence between attribute-based signatures and homomorphic signatures, and new constructions for both. In *Theory of Cryptography TCC*, 2017.