Progressive And Efficient Verification For Digital Signatures

Cecilia Boschini[1,2], Dario Fiore[3], and Elena Pagnin[4]

[1] Technion, Haifa, Israel
[2] Reichman University, Herzliya, Israel
cecilia.bo@cs.technion.ac.il
[3] IMDEA Software Institute, Madrid, Spain
dario.fiore@imdea.org
[4] Lund University, Lund, Sweden
elena.pagnin@eit.lth.se

Abstract. Digital signatures are widely deployed to authenticate the source of incoming information, or to certify data integrity. Common signature verification procedures return a decision (accept/reject) only at the very end of the execution. If interrupted prematurely, however, the verification process cannot infer any meaningful information about the validity of the given signature. We notice that this limitation is due to the algorithm design solely, and it is not inherent to signature verification.
In this work, we provide a formal framework to handle interruptions during signature verification. In addition, we propose a generic way to devise alternative verification procedures that progressively build confidence on the final decision. Our transformation builds on a simple but powerful intuition and applies to a wide range of existing schemes considered to be post-quantum secure including the NIST finalist Rainbow.
While the primary motivation of progressive verification is to mitigate unexpected interruptions, we show that verifiers can leverage it in two innovative ways. First, progressive verification can be used to intentionally adjust the soundness of the verification process. Second, progressive verifications output by our transformation can be split into a computationally intensive offline set-up (run once) and an efficient online verification that is progressive.

Keywords: Digital Signatures, Amortized Efficiency, Flexible Verification, Progressive Verification, Post-Quantum Security.

1 Introduction

Digital signatures allow one party (the signer) to use her secret key to authenticate a message in such a way that, at any later point in time, anyone holding the corresponding public key (the verifiers) can check its validity. The typical nature of signature verification procedures is monolithic: the validity of a signature is determined only *after* a sequence of tests is completed. In particular, if the execution is interrupted *in media res* (Latin for "in the midst of things"), no conclusive answer can be drawn from the outcomes of the partial tests. Although this monolithic nature is not a burden in many application scenarios, e.g., validating financial transactions (Bitcoin protocol), installing certified software updates (Android OS), or delivering e-services (e-Health, electronic tax systems), it is a major limitation to the adoption of digital signatures in cyber-physical systems [30] and in secure eager or speculative executions [22], where the speed at which verification is performed plays a crucial role.

Le et al. [21] proposed to address unexpected interruptions using a new cryptographic primitive called *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, or rejects it. While the primary motivation of flexible verifications is to mitigate unexpected interruptions; we observe that the overarching idea of *progressive verification* has further impacts. In particular, progressive verification can be used to customize the soundness of the verification process. For example, a smart device may decide to verify at a 30-bit security level, if the signatures come from specific sources or the battery is below 30%. From the theoretical perspective, progressive verification (as introduced in this work later on) draws interesting connections between classical, information-theoretic and post-quantum security notions.

1.1 Our Contribution

This work sets out to dismantle the monolithic nature of signature verification. by designing *new* verification methods for *existing* signature schemes. Concretely, we investigate two approaches. The first one is to speed-up the verification process for polynomially many signatures by the same signer leveraging a one-time computation on the public key (efficient verification). The second approach is to re-design the verification process so that it allows one to extract sensible information even when the algorithm is executed only partially (progressive verification). In this setting it is of particular interest to investigate the security implications of this new model and what additional features it may bring.

In detail, we introduce formal definitions and security models for both efficient (Section 2) and progressive (Section 3) verification. In terms of realizations, we focus on a specific family of schemes that we call with $\mathbf{Mv}$-style verification (in brief, the verification includes matrix-vector multiplications). For schemes in this class, we propose two compilers, i.e., two information-theoretic transformations that turn monolithic $\mathbf{Mv}$-style verifications into provably-secure efficient (Section 4.1), or progressive (Section 4.2) ones. Our compilers apply to multi-variate polynomials based schemes including the NIST finalist Rainbow[11,12] and LUOV [5]; and lattice-based schemes including GPV [16] (hash & sign), MP [24] (Boyen/BonsaiTree), and GVW [17] (homomorphic). A large part of the security proof is devoted to a detailed analysis of the leakage due to verification queries (that now involve secret randomness). We consider this leakage analysis a result of independent interest as it can be used to estimate leakage in similar information-theoretic approaches to provably secure algorithmic speed-ups or eager executions. Our models for efficient and progressive verification can easily be extended to include signatures with advanced properties including: ring, threshold, homomorphic multi-key, attribute-based and constrained.

1.2 Related Work

The problem of trading security for less computation during a verification has been considered first by Fischlin [14] and Armknecht et al. [1] in the context of message authentication codes (MACs). Le et al. [21] and by Taleb and Vergnaud [28] consider the same question for digital signatures.

Le et al. [21] introduce the notion of flexible signatures and a construction based on the Lamport-Diffie one-time signature [20] with Merkle trees. Taleb and Vergnaud [28] put forth realizations of progressive verification for three specific signature schemes (RSA, ECDSA and GPV). Differently from us, both works demand a modification of the signing or key generation algorithm of the original signature scheme and also a time variable be input to the progressive or flexible verification.

One main difference between our model and those of [14,21,28] is that we aim to capture progressive verification as an independent feature that can enhance existing schemes, rather than a standalone primitive that requires one to change some of the core algorithms of a signature scheme. This is in a way more challenging as it leaves less design freedom when crafting these algorithms. In addition, we define progressive verification as a *stateful* algorithm in contrast to stateless [14,21,28]: 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 [2,9] and verifiable computation [15]; 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. An early instantiation of this technique for speeding up the verification of Rabin-Williams signatures appears in [4]. More recently, Sipasseuth et al. [27] investigate how to speed up lattice-based signature verification while reducing the memory (storage) requirements. The overall idea in [27] 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, [27] focuses on the DRS signature [26], 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 [27], our techniques can be seen as a generalization of what presented in [27].

Notation In what follows, λ denotes the parameter for computational security and $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ a tuple of algorithms identifying a digital signature scheme that satisfies the syntax and the properties of correctness and existential unforgeability as defined in [28].

2 Efficient Verification For Digital Signatures

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. We remark that for secure efficient verification svk should be hidden to the adversary, yet, the knowledge of svk gives no advantage in forging signatures verified in the standard way using just pk. 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. For security, it is fundamental svk remains unknown to the adversary. We remark that generating svk during the offline phase achieves efficient online verification with no impact on the original signing or key generation algorithms, which was a drawback of previous work [21,28].

2.1 Syntax For Efficient Verification

Our definition of efficient verification lets the verifier set the confidence level k at which she wishes to carry out the signature verification. Notably k determines the amount of computation to be performed and thus plays a central role in the security and the efficiency of the new verification.

Definition 1 (Efficient Verification). *A signature scheme Σ admits efficient verification if there exist two PPT algorithms* $(\mathsf{offVer}, \mathsf{onVer})$ *with the following syntax:*[5]

$\mathsf{offVer}(\mathsf{pk}, k)$**:** *this is a randomized algorithm that on input a public verification key* pk*, and a positive integer* $k \in \{1, \ldots, \lambda\}$ *(where λ is the security parameter of Σ), 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).*

For convenience we will refer to the signature scheme augmented with the efficient verification algorithms as $\Sigma^E = (\Sigma, \mathsf{offVer}, \mathsf{onVer})$*, and to the integer value k as confidence level.*

To be meaningful, a realization of efficient verification needs to satisfy the properties of correctness, concrete atomized efficiency and security.

Definition 2 (Correctness of Efficient Verification). *A scheme* $\Sigma^E = (\Sigma, \mathsf{offVer}, \mathsf{onVer})$ *realizes efficient verification correctly if the following conditions hold. 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* $k \in \{1, \ldots, \lambda\}$*; it holds that* $\Pr[\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma}) = 1 \mid \mathsf{svk} \leftarrow \mathsf{offVer}(\mathsf{pk}, k)] = 1$ *for any choice of randomness used in* offVer.

Amortized efficiency relies on the fact that running offVer once and reuse its output to run onVer r times is computationally less demanding than running the standard verification Ver r times. To formalize this, we will use the function $\mathsf{cost}(\cdot)$ that given as input an algorithm returns its computational cost (in some desired computational model). In addition, we parameterize concrete amortized efficiency with two intertwined variables: r_0 (number of instances of verification), and e_0 (ratio between the cost of r_0 efficient verifications over r_0 standard verifications). The lower the value of r_0 the sooner Σ^E amortizes the computational cost of offVer. The lower the value of e_0 the more efficient Σ^E is with respect to the standard verification.

Definition 3 (Concrete Amortized Efficiency). *A scheme Σ^E realizes* $(\mathsf{r}_0, \mathsf{e}_0)$*-concrete amortized efficient verification for Σ if given a security parameter λ and a confidence level k; for any key pair* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$*, for any pair* $(\mu, \boldsymbol{\sigma})$ *with* $\mu \in \mathcal{M}$ *and $\boldsymbol{\sigma}$ such that* $\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 1$*; there exist a non-negative integer* r_0*, and a real constant* $0 < \mathsf{e}_0 < 1$ *such that:*

$$\forall\ \mathsf{r} \geq \mathsf{r}_0\ , \quad \frac{\mathsf{cost}\big(\mathsf{offVer}(\mathsf{pk}, k)\big) + \mathsf{r} \cdot \mathsf{cost}\big(\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})\big)}{\mathsf{r} \cdot \mathsf{cost}\big(\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})\big)} < \mathsf{e}_0 \tag{1}$$

[5] Here pk denotes a public verification key output by KeyGen.

2.2 Security Model For Efficient Verification

Intuitively, Σ^E realizes efficient verification in a secure way if onVer accepts a signature that would be rejected by Ver only with negligible probability. In the security game (see Figure 1), the adversary $\mathcal{A}$ has access to the signing oracle $O\mathsf{Sign}$ as well as the efficient verification oracle $O\mathsf{onVer}$. The goal of the adversary is to produce a signature $\boldsymbol{\sigma}^*$ for a message μ^* that was never queried to $O\mathsf{Sign}$ and for which Ver returns 0 (reject) and onVer returns 1 (accept).

$\mathsf{cmvEUF}\ (\lambda,\ \Sigma,\ k)$

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** $(\mu^*, \boldsymbol{\sigma}^*)$

$O\mathsf{Sign}_{\mathsf{sk}}(\mu)$

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

$\mathsf{Exp}^{\mathsf{cmvEUF}}_{\mathcal{A},\Sigma}(\lambda, k)$

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

$O\mathsf{onVer}_{\mathsf{svk}}(\mu, \boldsymbol{\sigma})$

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

Fig. 1: Security model for efficient verification of signatures: existential unforgeability under adaptive chosen message and verification attack (security game, experiment and oracles). $\mathcal{A}$ is a PPT algorithm that can query the oracles in an adaptive and parallel way. L_S is the list of messages queried to the signing oracle.

Definition 4 (Security of Efficient Verification). *A scheme Σ^E realizes a secure efficient verification for Σ if for a given security parameter λ and for any confidence level $k \in \{1, \dots, \lambda\}$, for all PPT adversaries $\mathcal{A}$ the success probability in the* cmvEUF *experiment reported in Figure 1 is negligible, i.e.: $Adv^{\mathsf{cmvEUF}}_{\mathcal{A},\Sigma}(\lambda, k) = \Pr\left[\mathsf{Exp}^{\mathsf{cmvEUF}}_{\mathcal{A},\Sigma}(\lambda, k) = 1\right] \le \varepsilon(\lambda, k)$.*

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 existentially unforgeable. Notably, both the security game and the advantage depend on the confidence level k and assume all algorithms are entirely executed.

3 Progressive Verification For Digital Signatures

The goal of progressive verification is to incrementally increase the confidence on the validity of a signature, for a given message against a public key. Intuitively, the "confidence" should be proportional to the amount of computation invested: the further in the execution we go, the higher the accuracy of the decision, and thus the confidence of the final outcome (accept/reject).

3.1 Signatures With Progressive Verification

Taleb and Vergnaud give a very intuitive definition of progressive verification for digital signatures [28]. They model digital signatures with progressive verification as a 4-tuple of PPT algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver}, \mathsf{ProgVer})$ such that: $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ is a correct digital signature scheme; and $\mathsf{ProgVer}$ takes in input a public verification key pk, a message μ, a signature $\boldsymbol{\sigma}$, and some timing parameter t, and outputs $\alpha \in \{[0,1] \cap \mathbb{R}\} \cup \{\perp\}$, interpreted as an estimate on the accuracy of its decision whether the signature be valid. Moreover, the scheme satisfies the following properties:

Correctness If for some tuple of inputs $\mathsf{ProgVer}(\mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$ outputs $\perp$, then $\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 0$.

Security If for some tuple of inputs $\mathsf{ProgVer}(\mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$ outputs $\alpha \in [0,1]$, then this implies $\Pr[\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 0] \leq 1 - \alpha$ (where the probability is taken over the random coins of $\mathsf{ProgVer}$).

In a nutshell, if $\alpha = \perp$, the progressive verification deems the signature to be invalid (with 100% accuracy). If $\alpha \in [0,1]$, the algorithm considers the signature valid, and α tells how accurate this statement is. Since progressive verification may be interrupted at any arbitrary point t during its execution, in practice α is (the output of) a function $\alpha_{\mathsf{prog}}(t)$ that "converts" the progress in the verification process into a value representing the accuracy of a positive outcome.

Shortcomings First, similarly to [21], also [28] sees signatures with progressive verification as a stand alone primitive. In contrast we view progressive verification as a feature that can augment existing schemes without requiring change to the core algorithms. Second, the definition lacks a precise notion of time complexity and does not model how unexpected interrupts are handled. The model we introduce in the remainder of this section takes care of these aspects. In addition, we generalize progressive verification to be (possibly) stateful, which can capture more signature schemes as well as reuse the same syntax to model both efficient and progressive verification (see the Appendix C).

3.2 Syntax For Progressive Verification

In order to model progressive verification as an add-on algorithm we need to derive from Ver an alternative algorithm $\mathsf{ProgVer}$ (as introduced in Section 3.1), that builds confidence on the final verification outcome in an increasing way. Without loss of generality, this task boils down to identifying a sequence of $T+1$ atomic instructions that we call $\mathsf{ProgStep}$ with the following properties. Each $\mathsf{ProgStep}$ performs a check of some sort on the input it receives. If one step fails, the progressive verification returns $\alpha = \perp$. If none of the initial t steps fails, the progressive verification returns the output of a function $\alpha_{\mathsf{prog}}(t) \in [0,1]$ that measures the probability the input will be accepted by Ver. The fact of increasingly building confidence is reflected by functions α_{prog} that are non-decreasing in t, the number of instructions checked before returning the answer. Figure 1 in [28] provides an intuitive and graphical representation of this statement.

Definition 5 (Stateful Progressive Verification). *Let $T \in \mathbb{Z}_{>0}$ and $\alpha_{\mathsf{prog}} : \{0, \ldots, T\} \to [0,1]$ be an efficiently computable function. A signature scheme Σ admits $(T, \alpha_{\mathsf{prog}})$-progressive verification if there exists a stateful PPT algorithm $\mathsf{ProgVer}$ that takes in input $\mathsf{pk}, \mu, \boldsymbol{\sigma}$ and some interruption parameter $t \in \mathbb{Z}_{>0}$, outputs $\alpha \in \{[0,1] \cap \mathbb{R}\} \cup \{\perp\}$, and satisfies the following syntax:*

$\mathsf{ProgVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$

$\alpha \leftarrow \bot$
if $t < 0$: **return** $\bot$
if $t > T$: **set** $t \leftarrow T$
for $j = 0, \ldots, t$
  $(b, \mathsf{st}) \leftarrow \mathsf{ProgStep}_j(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$
  if $(b = 0)$: **return** $\bot$
  else $(b = 1)$: $\alpha \leftarrow \alpha_{\mathsf{prog}}(j)$
return α

For convenience we will refer to the signature scheme augmented with progressive verification as $\Sigma^P = (\Sigma, \mathsf{ProgVer}, T, \alpha_{\mathsf{prog}})$.

Concretely, ProgVer is made of $T+1$ algorithms $\mathsf{ProgStep}_j$, for $j = 0$ to T, that progressively update the state st. We remark that the formalization into steps is without loss of generality: Ver realizes a trivial progressive verification for $T = 0$ where the only step is Ver itself. Finally, the interruption value t is input to ProgVer only, and it is *not* given to each $\mathsf{ProgStep}_j$. Thus our syntax models the fact that the steps are agnostic of the interruption value and must work without knowing when to stop, which is essential to capture arbitrary interruptions.

Correctness essentially states that signatures accepted by the standard verification should also be accepted by the progressive one, with the highest confidence allowed by the number of steps performed.

Definition 6 (Progressive Verification Correctness). *Let Σ^P be a signature scheme with progressive verification;* ProgVer *satisfies progressive verification correctness if, for any value $t \in \{0, \ldots, T\}$, for any given security parameter λ, for any key pair* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$, *for any admissible state* st *generated by* ProgVer*, for any admissible message, given a signature $\boldsymbol{\sigma}$ such that* $\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma}) = 1$ *it holds that:*
$\Pr[\mathsf{ProgVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t) = \alpha_{\mathsf{prog}}(t)] = 1$.

We follow the approach of [21] and let the progressive verification algorithm output a value α that either rejects the signature ($\alpha = \bot$), or accepts it with certainty α in the real interval $[0, 1]$. We use the same interruption variable t as in [21] to model runtime interruptions of the algorithm execution.[6]

Efficient vs. Progressive Verification At a first glance, efficient verification and progressive verification seem to have the common goal of reducing the computational cost of a signature verification. However the way this objective is achieved in the two models is quite different. In progressive verification, the verifier (and thus each $\mathsf{ProgVer}_i$) is unaware of when the computation will be interrupted, and *its execution is independent of t*. In contrast, in efficient verification the verifier (running offVer) determines the confidence level k prior to any actual verification (running onVer). In the latter, 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 define progressive verification as stateful. This allows us to keep the framework as general as possible. Stateless progressive verification, á la [21,28], can be obtained setting st to $\varnothing$, this also removes the need for analyzing any cross-query leakage due to state reuse.

[6] Our $\alpha_{\mathsf{prog}}(\cdot)$ is essentially the inverse of the function $\mathsf{iExtract}_{\Sigma}(\cdot)$ in [21].

3.3 Security Model For Progressive Verification

Our notion of unforgeability states that signatures rejected by the standard verification should also be rejected by the progressive one, except for an inaccuracy factor due to interruptions. More formally, Ver and $\mathsf{ProgVer}$ should have the same behavior (accept/reject) with discrepancies happening with probability negligibly close to $\alpha_{\mathsf{prog}}(t)$.

Our security game has three main differences compared to [21]:

State in order to take into account that $\mathsf{ProgVer}$ maintains a possibly non-trivial state we allow the adversary $\mathcal{A}$ to interact with the progressive verification oracle $O\mathsf{ProgVer}$ during the query phase, as well as the signing oracle $O\mathsf{Sign}$, in a concurrent manner.

Interruption queries to $O\mathsf{ProgVer}$ have the form $(\mu, \boldsymbol{\sigma}, t')$, where t' is the desired interruption value submitted by $\mathcal{A}$ (and chosen adaptively).

Output instead of 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 $t^* \in \{0, \ldots, T\}$ reports the interruption position used in the final progressive verification. Including t^* in the output of the experiment allows us to measure security in terms of *how close* the probability of $\mathcal{A}$ wining the experiment is from the expected accuracy value $1 - \alpha_{\mathsf{prog}}(t^*)$.

$\mathsf{progEUF}(\Sigma^P, \lambda)$

$\mathsf{L}_S \leftarrow \varnothing$
$\mathsf{st} \leftarrow \varnothing$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(\mu^*, \boldsymbol{\sigma}^*, t') \leftarrow \mathcal{A}^{O\mathsf{Sign}, O\mathsf{ProgVer}}(\mathsf{pk}, \lambda)$
return $(\mu^*, \boldsymbol{\sigma}^*, t')$

$\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma^P}(\lambda)$

$(\mu^*, \boldsymbol{\sigma}^*, t') \leftarrow \mathsf{progEUF}(\Sigma, \lambda)$
$\beta \leftarrow \mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*)$
$t^* \leftarrow O\mathsf{Int}(t')$
$\alpha \leftarrow \mathsf{ProgVer}(\mathsf{st}, \mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*, t^*)$
if $\mu^* \in \mathsf{L}_S \vee \alpha = \perp \vee \beta = 1$
  return $(0, t^*)$
return $(1, t^*)$

$O\mathsf{Sign}_{\mathsf{sk}}(\mu)$

$\mathsf{L}_S \leftarrow \mathsf{L}_S \cup \{\mu\}$
$\boldsymbol{\sigma} \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$
return $\boldsymbol{\sigma}$

$O\mathsf{ProgVer}_{\mathsf{st},\mathsf{pk}}(\mu, \boldsymbol{\sigma}, t')$

$t \leftarrow O\mathsf{Int}(t')$
$\alpha \leftarrow \mathsf{ProgVer}(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t)$
return α

Fig. 2: Security model for progressive verification of signatures: existential unforgeability under adaptive chosen message and progressive verification attack (security game, experiment and oracles). $\mathcal{A}$ can query the oracles adaptively, in parallel and polynomially many times in λ. L_S is the list of messages queried to the signing oracle.

Definition 7 (Security of Progressive Verification ($\mathsf{progEUF}$)). *Let Σ be a signature scheme that admits a progressive verification realization Σ^P. Σ^P realizes a secure progressive verification for Σ if for any given security parameter λ, for all PPT adversaries $\mathcal{A}$ the success probability in the* $\mathsf{progEUF}$ *experiment in Figure 2 is negligible, i.e.,:*

$$Adv^{\mathsf{progEUF}}_{\mathcal{A},\Sigma^P}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma^P}(\lambda) = (1, t^*)\right] - (1 - \alpha_{\mathsf{prog}}(t^*)) = \varepsilon \leq \varepsilon(\lambda).$$

Intuitively, Definition 7 states that an adversary has only negligible probability to make $\mathsf{ProgVer}$ 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 $\alpha_{\mathsf{prog}}(t) = 1 - \mathsf{bad}(t)$, we get $Adv^{\mathsf{progEUF}}_{\mathcal{A},\Sigma^P}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma^P}(\lambda) = (1, t^*)\right] - \mathsf{bad}(t^*) \leq \varepsilon(\lambda)$.

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 $O\mathsf{ProgVer}$ during the game, we will see that by running $O\mathsf{ProgVer}$ on 'too few' steps, $\mathcal{A}$ may learn information about the internal state st.

Modelling Interruptions. In [21], unexpected interruptions are modeled via an interruption oracle $\mathsf{iOracle}(\lambda)$ that returns a value $t \in \{0, \dots, T\}$ used by the progressive verification. However, it is not clear whether $\mathcal{A}$ may control $\mathsf{iOracle}$ or not. We overcome these ambiguities by letting $\mathcal{A}$ output t' with every progressive verification query. 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. It is possible to relax and generalize our model by setting a different 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 progressive 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 Constructions

In this section, we present generic transformations (compilers) that augment a signature scheme Σ with either efficient (Section 4.1) or progressive verification (Section 4.2).
Our technique works for a specific class of signature schemes that we call *with* $\mathbf{Mv}$*-style verification*. In such schemes, Ver can be seen as the combination of two types of verification checks: a matrix-vector multiplication (referred to as $\mathbf{Mv} = \mathbf{0}$, for appropriate matrix $\mathbf{M}$ and vector $\mathbf{v}$) and other generic checks (collected in the Check subroutine), see Figure 3 for details and an explanatory example. Among the schemes with $\mathbf{Mv}$-style verification we highlight some of the seminal lattice-based signatures [7,8,16,24], homomorphic signatures [6,13,17], and multivariate signatures [5,12].

$\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})$

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

Example: $\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})$ for GPV08 [16]

$b_1 \leftarrow 0, b_2 \leftarrow 0$
parse $\mathsf{pk} = (PK, PK.\mathsf{aux})$
 set $PK \leftarrow \mathbf{A}$
 set $PK.\mathsf{aux} \leftarrow (\mathcal{H}, \beta)$
$\mathsf{Check}(PK.\mathsf{aux}, \mu, \boldsymbol{\sigma})$:
 if $\|\boldsymbol{\sigma}\| < \beta$ **set** $b_1 \leftarrow 1$
$\mathsf{GetMv}(\mathsf{pk}, \mu, \boldsymbol{\sigma})$:
 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 $\mathbf{v} \leftarrow [\boldsymbol{\sigma}^T | \mathbf{u}^T]^T$
if $(\mathbf{M} \cdot \mathbf{v} = \mathbf{0}_{rows(\mathbf{A}) \times 1} \mod q)$
  set $b_2 \leftarrow 1$
return $(b_1 \wedge b_2)$

Fig. 3: General structure of a signature with $\mathbf{Mv}$-style verification (on the left); an instructive example: the GPV08 [16] signature verification (on the right).

4.1 A Compiler For Efficient MV-Style Verifications

We present a generic way to realize efficient verification for signatures with $\mathbf{Mv}$-style verification, whenever the computational complexity of Ver is dominated by the matrix-vector multiplication, i.e., $\mathsf{cost}(\mathsf{Check}) << \mathsf{cost}(\mathbf{Mv}) \sim mn$ field multiplications (for $\mathbf{M} \in \mathbb{Z}_q^{n \times m}$).

Our compiler for efficient verification is detailed in Figure 4 with a sketch of instantiation for the lattice-based scheme GPV08 [16] as a running example. Further details on this scheme as well as instantiations and details on the concrete efficiency estimates for MP12 [24], Rainbow [12] and LUOV [5], are deferred to Appendix B. Table 1 summarizes the efficiency results. We obtain secure efficient online verification using as little as 0.4% (resp. 50%) of the computational cost of the standard verification for lattice-based signatures on exponentially large fields (resp. for Rainbow).

We start by a quick recap of the notation we use when describing our compilers.

Notation. We denote the set of real values by $\mathbb{R}$, integers by $\mathbb{Z}$, natural numbers by $\mathbb{N}$, 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 the span (linear space) generated by a set of vectors $\mathbf{z}_1, \ldots, \mathbf{z}_i$ in the discrete vector space $\mathbb{Z}_q^m$ as $\langle \mathbf{z}_1, \ldots, \mathbf{z}_i \rangle_q = \{\mathbf{z} \in \mathbb{Z}_q^m \ : \ \mathbf{z} = \sum_{j=1}^{i} a_j \mathbf{z}_j \bmod q, \exists a_1, \ldots, a_i \in \mathbb{Z}_q\}$. 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 $\{a, \ldots, b\} \subseteq \mathbb{Z}$.

$\mathsf{offVer}(\mathsf{pk}, k)$

// 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 : $\quad \mathbf{c} \xleftarrow{\$} \mathbb{Z}_q^{1\times rows(\mathbf{M})}$
  iv : $\quad \mathbf{z} \leftarrow \mathbf{cM} \in \mathbb{Z}_q^{1\times cols(\mathbf{M})}$
  v : $\quad$ **if** $\mathbf{z} \in \langle \mathbf{z}_0, \ldots, \mathbf{z}_{j-1} \rangle_q$ **go to line** iii.
  vi : $\quad \mathbf{z}_j \leftarrow \mathbf{z}$ // 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** $\mathsf{svk} \leftarrow \big(k, \mathbf{Z}, PK.\mathsf{aux}\big)$

(a) The offline verification algorithm.

$\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})$

// LIGHTWEIGHT VERIFICATION CHECKS
1 : **if** $\mathsf{Check}(PK.\mathsf{aux}, \mu, \boldsymbol{\sigma}) = 0$
2 : $\quad$ **return** 0
// FORMATTING FOR EFFICIENT VERIF.
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 : $\quad$ **if** $\mathbf{z}_j' \cdot \mathbf{v}_j \neq 0 \mod q$
8 : $\quad\quad$ **return** 0
9 : **return** 1

(b) The online verification algorithm.

Fig. 4: 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 computational complexity of onVer is linear in k, the chosen confidence level.

Table 1: A summary of the concrete efficiency achieved by various instatiations of our compiler for efficient verification. In the table, k_0 denotes the minimum accuracy level that ralizes efficient verification with 128 bits of security, i.e., for which $\Pr[\mathsf{Bad}] \leq 2^{-128}$ is negligible (cf. proof of Theorem 1, with $q_V = 2^{30}$); r_0 is the smallest positive integer for which $\frac{\mathsf{cost}(\mathsf{offVer}(\mathsf{pk}, k_0)) + \mathsf{r} \cdot \mathsf{cost}(\mathsf{onVer})}{\mathsf{r} \cdot \mathsf{cost}(\mathsf{Ver})} < 1$, and e_0 is a (tight) upperbound on this ratio.

Ring or Field Size (representative schemes)	**Min. Accuracy Level for 128-bit security**	**Concrete Amortized Efficiency (see Definition 3)**	**Online Efficiency** $\frac{\mathsf{cost}(\mathsf{onVer})}{\mathsf{cost}(\mathsf{Ver})} = \frac{k_0}{n}$
exponential: $q = 2^{128}$ (FMNP [13]; GVW [17])	$k_0 = 1$	$(\mathsf{r}_0 = 2, \mathsf{e}_0 = 0.51)$	$\frac{1}{256} < 0.4\%$
large poly.: $q = 2^{30}$ (Boyen [7]; GPV [16]; MP [24])	$k_0 = 5$	$(\mathsf{r}_0 = 6, \mathsf{e}_0 = 0.86)$	$\frac{5}{256} < 2\%$
small poly.: $q = 16$ (Rainbow [12] $\mathbb{F}_{2^4}$-$(32, 32, 32)$)	$k_0 = 32$	$(\mathsf{r}_0 = 65, \mathsf{e}_0 = 0.99)$	$\frac{32}{64} = 50\%$

Overview of our technique Our transformation takes as input Σ, a signature scheme with $\mathbf{Mv}$-style verification; and it returns $\Sigma^E = (\Sigma, \mathsf{offVer}, \mathsf{onVer})$ that securely instantiates efficient verification for Σ. The heart of our compiler leverages the fact that for any pair of vectors $\boldsymbol{\sigma}$ and $\mathbf{u}$ (often derived from the message μ), 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 $\mathbf{c} \cdot (\mathbf{A} \cdot \boldsymbol{\sigma}) = \mathbf{c} \cdot \mathbf{u}$. Collecting variables on the left hand yields $(\mathbf{c} \cdot [\mathbf{A}| - \mathbf{I}_n]) \cdot \begin{bmatrix} \boldsymbol{\sigma} \\ \mathbf{u} \end{bmatrix} = 0$. Thus one can precompute the vector $\mathbf{z} \leftarrow \mathbf{c} \cdot [\mathbf{A}| - \mathbf{I}_n]$ and run the efficient online verification check $\mathbf{z} \cdot \mathbf{v} \stackrel{?}{=} 0$, where $\mathbf{v} \leftarrow (\boldsymbol{\sigma}, \mathbf{u})$. In a nutshell the idea is to replace the matrix-vector multiplication with a vector-vector multiplication in a sound way. Correctness and efficiency are immediate. Soundness essentially comes from the fact that if $\mathbf{z} \cdot \mathbf{v} = 0$, then with all but negligible probability the original system of linear equations $\mathbf{A} \cdot \boldsymbol{\sigma} = \mathbf{u}$ is satisfied too, as proven in Theorem 1.

Security Analysis Despite the construction being intuitive, analysing the leakage due to verification queries that reuse the same svk is not trivial and is one main technical contribution of this result.

Theorem 1. *Let Σ be an existentially unforgeable signature scheme with* $\mathbf{Mv}$*-style verification (as in Figure 3). The scheme $\Sigma^E = (\Sigma, \mathsf{offVer}, \mathsf{onVer})$ obtained via our compiler depicted in Figure 4 is existentially unforgeable under adaptive chosen message and efficient verification attacks. Concretely, the advantage is $Adv^{CMVA}_{\mathcal{A},\Sigma}(\lambda, k) \leq \frac{q_V + 1}{q^k - q_V}$ where $k \in \{1, \ldots, rk(\mathbf{M})\}$ denotes the chosen confidence level that grows up to the rank of the matrix $\mathbf{M}$, $q_V = \mathsf{poly}(\lambda) << q^k$ is a bound on the total number of verification queries and q is the modulo of the algebraic structure on which Σ is built.*

Remark For simplicity, Theorem 1 considers only existential unforgeability. The statement and the proof actually adapt with ease to other security models such as strong and selective unforgeability.

Proof. In the cmvEUF security experiment (Figure 1), the winning conditions require $\mathcal{A}$ to produce a message-signature pair $(\mu^*, \boldsymbol{\sigma}^*)$ such that μ^* has not been queried to the signing oracle during the game (existential unforgeability); $\boldsymbol{\sigma}^*$ is invalid under the standard verification, i.e., $\mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*) = 0$; and the pair 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 this event occurs.

Let Win be the event $\{\mathsf{Exp}^{\mathsf{cmvEUF}}_{\mathcal{A},\Sigma}(\lambda, k) = 1\}$. Let $i = 1$ to q_V be the index of the queries $(\mu_i, \boldsymbol{\sigma}_i)$ submitted by $\mathcal{A}$ to the $O\mathsf{onVer}$ oracle. Define the family of events bad_i (for $i = 1$ to $q_V + 1$) as:

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

where bad_{q_V+1} corresponds to $\mathcal{A}$ returning a valid forgery $(\mu^*, \boldsymbol{\sigma}^*) := (\mu_{q_V+1}, \boldsymbol{\sigma}_{q_V+1})$ at the end of the experiment. We can rewrite the winning condition of the security experiment as $\mathsf{Win} = \{\mathsf{bad}_{q_V+1} \ \wedge \ \mu^* \notin \mathsf{L}_S\}$ (recall that L_S is the list of messages queried to the signing oracle in the game execution). Consider the event Bad defined as "there exists at least one query index i in the game execution for which bad_i occurs". Formally:

$$\mathsf{Bad} := \{\exists\ i \in \{1, \ldots, q_V\} : \mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 0 \ \wedge \ \mathsf{onVer}(\mathsf{svk}, \mu_i, \boldsymbol{\sigma}_i) = 1\} \ .$$

$$\begin{aligned} Adv^{CMVA}_{\mathcal{A},\Sigma}(\lambda, k) &= \Pr[\mathsf{Win} \wedge \mathsf{Bad}] + \Pr[\mathsf{Win} \wedge \neg\mathsf{Bad}] \\ &\leq \Pr[\mathsf{Bad}] + \Pr[\mathsf{Win} \mid \neg\mathsf{Bad}] \end{aligned}$$

where the inequality comes from applying the definition of conditional probability and upperbounding $\Pr[\mathsf{Win} \mid \mathsf{Bad}]$ and $\Pr[\neg\mathsf{Bad}]$ by 1.

We notice that $\Pr[\mathsf{Win} \mid \neg\mathsf{Bad}]$ is essentially the probability that the event bad_i occurs only for $i = q_V + 1$ and never before, i.e.,

$$\Pr[\mathsf{Win} \mid \neg\mathsf{Bad}] \leq \Pr\left[\mathsf{bad}_{q_V+1} \mid \bigwedge_{i=1}^{q_V} \neg\mathsf{bad}_i\right]$$

In order to bound $\Pr[\mathsf{Bad}]$, we define events Bad^*_i (for $i = 1$ to q_V) as "bad_i occurs *for the first time* at query i", namely $\mathsf{Bad}^*_i = \mathsf{bad}_i \wedge \left(\bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right)$. Then we have

$$\Pr[\mathsf{Bad}] = \Pr\left[\bigvee_{i=1}^{q_V} \mathsf{Bad}^*_i\right] = \sum_{i=1}^{q_V} \Pr[\mathsf{Bad}^*_i] \leq \sum_{i=1}^{q_V} \Pr\left[\mathsf{bad}_i | \bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right]$$

where the second equality holds because the events Bad^*_i are all disjoint, and the inequality follows from applying the definition of conditional probability and upperbounding $\Pr\left[\bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right]$ by 1, for all i. Thus:

$$Adv^{CMVA}_{\mathcal{A},\Sigma}(\lambda, k) \leq \sum_{i=1}^{q_V+1} \Pr\left[\mathsf{bad}_i \mid \bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right] . \tag{2}$$

Lemma 1. *For every $i = 1$ to $q_V + 1$, it holds that*

$$\Pr\left[\mathsf{bad}_i = 1 \mid \bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right] \leq \tfrac{1}{q^k - (i-1)}.$$

The proof of Lemma 1 is deferred momentarily to let us complete the reasoning that proves the theorem. Using the inequality provided by Lemma 1, it is easy to see that $\sum_{i=1}^{q_V+1} \Pr\left[\mathsf{bad}_i = 1 \mid \bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right]$ $\leq \sum_{i=1}^{q_V+1} \frac{1}{q^k-(i-1)}$. Indeed, $\frac{1}{q^k-(i-1)} \leq \frac{1}{q^k - q_V}$ for all integers i in $[1, q_V + 1]$ and for all $q_V, q, k \in \mathbb{N}$ satisfying $q_V < q^k$. Thus $\sum_{i=1}^{q_V+1} \frac{1}{q^k-(i-1)} \leq \frac{q_V+1}{q^k-q_V}$, which proves the bound on the advantage. □

Proof of Lemma 1. The goal of this proof is to give a generic structure for estimating the leakage of infromation due to reuse of svk (i.e., probabilities in Equation (2)); the concrete values of these probabilities are calculated in the Lemma 2 in the Appendix A.

To upperbound $\Pr\left[\mathsf{bad}_i = 1 \mid \bigwedge_{j=1}^{i-1} \neg\mathsf{bad}_j\right]$ we need to analyze the information leakage due to verification queries. First of all, by correctness $\mathsf{onVer}(\mathsf{svk}, \mu_i, \boldsymbol{\sigma}_i) = 0 \Rightarrow \mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 0$ and $\mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 1 \Rightarrow \mathsf{onVer}(\mathsf{svk}, \mu_i, \boldsymbol{\sigma}_i) = 1$ for every possible svk generated by offVer from pk. Leakage about svk happens in two cases: when an event bad_i occurs ($O\mathsf{onVer}$ accepts where the standard verification would reject); and when $O\mathsf{onVer}$ rejects a query (here $\mathcal{A}$ may learn that some combination of rows of pk must appear in svk). Equation (2) gives us a way to bound the adversary's advantage (and thus, the magnitude of this leakage) in terms of the events bad_i and $\neg\mathsf{bad}_i$.

Consider the i-th query $(\mu_i, \boldsymbol{\sigma}_i)$ to $O\mathsf{onVer}$. If the oracle returns 0, the adversary learns that $\mathbf{C}\cdot(\mathbf{M}_i\cdot\mathbf{v}_i) \neq \mathbf{0} \mod q$. In other words, there is *at least one* row of $\mathbf{C} \in \mathcal{C} := \{\mathbf{C} \in \mathbb{Z}_q^{k\times n} \ : \ rk(\mathbf{C}) = k\}$, say $\mathbf{c}_j$, that is *not* in the hyperplane orthogonal to $\mathbf{w}_i := \mathbf{M}_i \cdot \mathbf{v}_i$, i.e., $\mathbf{c}_j \cdot \mathbf{w}_i \neq 0 \mod q$. Note that $\mathcal{A}$ knows $\mathbf{w}_i$ since $(\mathbf{M}_i, \mathbf{v}_i)$ can be computed from the pk, μ_i and $\boldsymbol{\sigma}_i$. Let us introduce the sets $\mathcal{H}_i \subseteq \mathcal{C}$ of full-rank matrices $\mathbf{C} \in \mathcal{C}$ whose rows are *all orthogonal* to $\mathbf{w}_i$, formally:

$$\mathcal{H}_i := \left\{ \mathbf{C} \in \mathcal{C} \ : \ \mathbf{C} = \begin{bmatrix} \mathbf{c}_1 \\ \dots \\ \mathbf{c}_k \end{bmatrix} \wedge \ \mathbf{c}_j \cdot \mathbf{w}_i = 0 \mod q \ \forall\ j = 1, \dots, k \right\} .$$

We assume $\mathcal{A}$ be able to pick the vectors $\mathbf{w}_i \in \mathbb{Z}_q^n \setminus \{0\}$ of her choosing (e.g., by generating suitable pairs $(\mu_i, \boldsymbol{\sigma}_i)$). This assumption is generous as it gives the adversary a large amount of power and freedom in the game. The restriction $\mathbf{w}_1 \neq \mathbf{0}$ is technical, as otherwise $\mathsf{Ver}(\mathsf{pk}, \mu_1, \boldsymbol{\sigma}_1) = 0$, which is a necessary condition for $O\mathsf{onVer}$ leaking information about svk.

At the first verification query $(\mu_1, \boldsymbol{\sigma}_1)$, $\mathcal{A}$ has no information about $\mathbf{C}$ beyond the fact that it was uniformly sampled from the set $\mathcal{C} := \{\mathbf{C} \in \mathbb{Z}_q^{k\times n} \ : \ rk(\mathbf{C}) = k\}$. Therefore, for any choice of $\mathbf{w}_1 \neq \mathbf{0}$, if the event bad_1 occurs, then $\mathsf{bad}_1 = \{\mathbf{C}\cdot\mathbf{w}_1 = \mathbf{0} \mod q \ \wedge \ \mathbf{C} \xleftarrow{\$} \mathcal{C}\}$, thus $\Pr[\mathsf{bad}_1] = \Pr\left[\mathbf{C}\cdot\mathbf{w}_1 = \mathbf{0} \mod q \ \wedge \ \mathbf{C} \xleftarrow{\$} \mathcal{C}\right] = \frac{|\mathcal{H}_1|}{|\mathcal{C}|}$. The first (rejected) verification query leaks the fact that $\mathbf{C} \in \mathcal{C} \setminus \mathcal{H}_1$.

For the second verification query, without loss of generality let $\mathbf{w}_2$ be linearly independent from $\mathbf{w}_1$, i.e., $\mathbf{w}_2 \notin \langle \mathbf{w}_1 \rangle_q$. In this case, we have

$$\begin{aligned}
\Pr[\mathsf{bad}_2 \mid \neg\mathsf{bad}_1] &= \Pr[\mathbf{C}\cdot\mathbf{w}_2 = \mathbf{0} \mod q \mid \mathbf{C} \xleftarrow{\$} \mathcal{C} \ \wedge \ \mathbf{C} \in (\mathcal{C} \setminus \mathcal{H}_1)] \\
&= \frac{\Pr\left[\mathbf{C}\cdot\mathbf{w}_2 = \mathbf{0} \mod q \ \wedge \ \mathbf{C} \xleftarrow{\$} \mathcal{C} \ \wedge \ \mathbf{C} \in (\mathcal{C} \setminus \mathcal{H}_1)\right]}{\Pr\left[\mathbf{C} \xleftarrow{\$} \mathcal{C} \ \wedge \ \mathbf{C} \in (\mathcal{C} \setminus \mathcal{H}_1)\right]} \\
&\leq \frac{\Pr\left[\mathbf{C}\cdot\mathbf{w}_2 = \mathbf{0} \mod q \ \wedge \ \mathbf{C} \xleftarrow{\$} \mathcal{C}\right]}{\Pr\left[\mathbf{C} \xleftarrow{\$} \mathcal{C} \ \wedge \ \mathbf{C} \in (\mathcal{C} \setminus \mathcal{H}_1)\right]} \\
&= \frac{\frac{|\mathcal{H}_2|}{|\mathcal{C}|}}{\frac{|\mathcal{C}\setminus\mathcal{H}_1|}{|\mathcal{C}|}} = \frac{|\mathcal{H}_1|}{|\mathcal{C} \setminus \mathcal{H}_1|}
\end{aligned}$$

where the inequality follows from the fact that, given three events E_1, E_2, E_3, it always holds that $\Pr[E_1 \ \wedge \ E_2 \ \wedge \ E_3] \leq \min\{\Pr[E_1 \ \wedge \ E_2],\ \Pr[E_1 \ \wedge \ E_3],\ \Pr[E_2 \ \wedge \ E_3]\}$; and the last equality follows since the hyperplanes $\mathcal{H}_1$ and $\mathcal{H}_2$ have the same dimension.

The same reasoning applies to the generic i-th verification query, where, w.l.o.g., $\mathcal{A}$ chooses $\mathbf{w}_i$ outside the space generated by the previous $\mathbf{w}_j$'s, i.e., $\mathbf{w}_i \notin \langle \mathbf{w}_1, \ldots, \mathbf{w}_{i-1} \rangle_q$. At such query, $\mathcal{A}$ knows that $\mathbf{C} \in \mathcal{C} \setminus \left(\bigcup_{j=1}^{i-1} \mathcal{H}_j \right)$. Analogously as before we get that

$$\begin{aligned} \Pr\left[\mathsf{bad}_i = 1 \mid \bigwedge_{j=1}^{i-1} \neg \mathsf{bad}_j \right] &\leq \frac{\Pr\left[\mathbf{C} \cdot \mathbf{w}_i = \mathbf{0} \mod q \ \wedge \ \mathbf{C} \xleftarrow{\$} \mathcal{C} \right]}{\Pr\left[\mathbf{C} \in \mathcal{C} \setminus \left(\bigcup_{j=1}^{i-1} \mathcal{H}_j \right) \ \wedge \ \mathbf{C} \xleftarrow{\$} \mathcal{C} \right]} \\ &= \frac{|\mathcal{H}_1|}{\left| \mathcal{C} \setminus \left(\bigcup_{j=1}^{i-1} \mathcal{H}_j \right) \right|} \ . \end{aligned} \tag{3}$$

Lemma 2 concludes the proof showing that

$$\left| \mathcal{C} \setminus \left(\bigcup_{j=1}^{i-1} \mathcal{H}_j \right) \right| \geq |\mathcal{H}_1| \cdot \left(\tfrac{q^n - q}{q^{n-k} - 1} - (i-1) \right) \quad \forall\, i = 2, \ldots, q_V + 1 \ .$$

Substituting this value into Equation (3) returns:

$$\Pr[\mathsf{bad}_i = 1 \mid \bigwedge_{j=1}^{i-1} \neg \mathsf{bad}_j] \leq \frac{1}{q^k \cdot \frac{1 - q^{1-n}}{1 - q^{k-n}} - (i-1)} \leq \frac{1}{q^k - (i-1)}$$

where the last bound follows from the chain:

$$q^{n-1} > q^{n-k} \ \Leftrightarrow\ \frac{1}{q^{n-1}} < \frac{1}{q^{n-k}} \ \Leftrightarrow\ 1 - \frac{1}{q^{n-1}} > 1 - \frac{1}{q^{n-k}} \ \Leftrightarrow\ \frac{1 - \frac{1}{q^{n-1}}}{1 - \frac{1}{q^{n-k}}} > 1 \ ,$$

as $1 < k < n$ and $q > 1$. $\square$

4.2 A Compiler For Progressive MV-Style Verification

Our compiler for progressive verification builds on the result presented in Section 4.1. Given a signature scheme Σ with $\mathbf{Mv}$-style verification, we define the T steps of a progressive verification Σ^P for Σ as shown in Figure 5.

$\mathsf{ProgStep}_0(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$

- $\mathsf{svk} \leftarrow \mathsf{offVer}(\mathsf{pk}, T)$
- **parse** $\mathsf{svk} = (T, \mathbf{Z}, PK.\mathsf{aux})$
- $b \leftarrow \mathsf{Check}(PK.\mathsf{aux}, \mu, \boldsymbol{\sigma})$
- $\mathsf{st} \leftarrow \mathsf{GetZV}(\mathsf{svk}, \mu, \boldsymbol{\sigma})$
- **return** (b, st)

$\mathsf{ProgStep}_j(\mathsf{st}, \mathsf{pk}, \mu, \boldsymbol{\sigma})$

- $b \leftarrow 0$
- **parse** $\mathsf{st} = (\mathbf{Z}', \mathbf{v})$
- **if** $\mathbf{Z}'[i, *] \cdot \mathbf{v}[*, i] = 0 \mod q$
  - **return** $(b \leftarrow 1, \mathsf{st})$
- **return** $(b \leftarrow 0, \mathsf{st})$

$\alpha_{\mathsf{prog}} : \{0, \ldots, T\} \to [0, 1], \qquad \alpha_{\mathsf{prog}}(t) = (1 - \frac{1}{q^t})$

Fig. 5: Our compiler for progressive verification of signatures with $\mathbf{Mv}$-style verification. The algorithms offVer, Check and GetZV are precisely as defined in Section 4.1, Figure 4, and $T = rows(\mathbf{M})$. The notation $\mathbf{Z}'[i, *]$ describes the i-th row of the matrix $\mathbf{Z}'$, similarly $\mathbf{v}[*, i]$ describes the i-th column of $\mathbf{v}$ (which is usually a vector $\mathbf{v}$, but may be a matrix in some constructions).

The value T sets the upper bound on the number of linear constraints the verifier wants to check, hence $T = rows(\mathbf{M})$, where $\mathbf{M}$ is the matrix employed in the original signature verification of Σ. The set of admissible states $\mathcal{S}$ includes $\emptyset$ and any possible state output by some $\mathsf{ProgVer}_i$, specifically $\mathcal{S} = \{0,1\} \times \mathbb{Z}_q^{rows(\mathbf{Z}') \times cols(\mathbf{Z}')} \times \mathbb{Z}_q^{rows(\mathbf{v}) \times cols(\mathbf{v})} \times \{0,1\}^\lambda \cup \varnothing$. We extract the confidence level from the probability of a progressive forgery (as motivated by the proof of security given in Theorem 1). It is easy to see that the probability that an adversary creates a progressive forgery for an interruption step t is at most $\frac{q^{n-t}-1}{q^n-1}$, this follows from the same reasoning as in the proof of Theorem 1 for efficient verification. Concretely, the bound is derived from the proof of Lemma 2, where we only consider $\Pr[\mathsf{bad}_1]$ as svk is refreshed with every new efficient verification query, and so there is no useful cross-query leakage, and we replace the confidence level k of the efficient verification with the interruption parameter t. If the size of the underlying algebraic structure is $q = 2^{\mathsf{poly}(\lambda)}$ this probability is negligible already for $t = 1$. In other words, for signatures with $\mathbf{Mv}$-style verification defined on *exponentially large* algebraic structures *efficient verification and progressive verification coincide*, trivially. The interesting case is $q = \mathsf{poly}(\lambda)$, as the adversary could create a progressive forgery with non-negligible probability. We remark that in this section we are not targeting efficiency, and our instantiations of progressive verification refresh the svk produced by offVer at every verification query. This way, $\mathcal{A}$ cannot exploit the information possibly leaked by a progressive forgery in future forgery attempts.

Theorem 2. *Let Σ be an existentially unforgeable signature scheme with $\mathbf{Mv}$-style verification (as of Fig. 3). Then the scheme Σ^P obtained via our compiler (in Figure 5) is a secure realization of progressive verification for Σ.*

Proof. Recall that an adversary $\mathcal{A}$ wins the security experiment in Definition 7 if it outputs a message-signature pair $(\mu^*, \boldsymbol{\sigma}^*)$ and an interruption t' such that: (1) $(\mu^*, \boldsymbol{\sigma}^*)$ is rejected by Ver, but accepted $\mathsf{ProgVer}$ when it is interrupted at step $t^* \leftarrow O\mathsf{Int}(t')$; and (2) the progressive verification algorithm outputs a too high confidence level $\alpha_{\mathsf{prog}}(t^*)$. Following Definition 7, we can realize secure progressive verification by setting $\alpha_{\mathsf{prog}}(t) = 1 - \Pr\left[\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1,t)\right] + \varepsilon(\lambda)$ for all $t = 0, \ldots, T$. The core part of the proof is to estimate this probability.

Recall that our compiler for efficient $\mathbf{Mv}$-style verification (in Figure 5) runs offVer at every verification query (line 1 in $\mathsf{ProgVer}_0$). This means that every verification query is answered using a freshly generated svk. In particular, the final verification (line 4 in the $\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma^P}(\lambda)$ in Figure 2) checks $\mathcal{A}$'s output using independent randomness from the previous queries. So, whatever information the adversary may have collected 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* $O\mathsf{ProgVer}$. The latter is precisely the probability of the event bad_1 defined in the proof of Theorem 1, where now we consider the matrix $\mathbf{C}$ to have t^* rows instead of k. Hence from Lemma 1 it follows that $\Pr\left[\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma}(\lambda) \leq (1,t^*)\right] = \frac{1}{q^{t^*}}$ and:

$$\begin{aligned} Adv^{\mathsf{progEUF}}_{\mathcal{A},\Sigma}(\lambda) &= \Pr\left[\mathsf{Exp}^{\mathsf{progEUF}}_{\mathcal{A},\Sigma}(\lambda) = (1,t^*)\right] - (1 - \alpha_{\mathsf{prog}}(t^*)) \\ &\leq \frac{1}{q^{t^*}} - \left(1 - \left(1 - \frac{1}{q^{t^*}}\right)\right) = 0 \ . \end{aligned}$$

□

4.3 Combining Progressive And Efficient Verification

We observe that progressive verifications obtained with our transformation can be split into two parts: a one-time, computationally intensive, setup ($\mathsf{ProgStep}_0$); and an efficient online verification ($\mathsf{ProgStep}_1$ to $\mathsf{ProgStep}_T$). This gives rise to custom (intentionally adjustable) verification soundness, which from the application perspective makes post-quantum secure verification accessible to a larger range of devices, and from the theoretical perspective draws interesting connections between classical, information-theoretic and post-quantum security notions. We include a more detailed discussion on this in the Appendix C.

5 Conclusions And Future Work

We presented a study on how to achieve efficient and progressive verification for digital signatures. In addition to putting forth these notions and formal models for them, we presented two compilers that allow one to realize efficient (resp. progressive) verification for a wide class of existing constructions including lattice-based and multivariate base. 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* progressive verification? We address this question in a positive way in Appendix C , albeit in weaker security model. A solution with full fledged progressive 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 progressive 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., $\mathsf{ProgVer}$ includes only $\mathsf{ProgVer}_0$ and $\mathsf{ProgVer}_1$). 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 (RIT17-0035), the Swiss National Science Foundation under the SNSF project number 182452 and the Postdoc.Mobility grant number 203075, 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 RED2018-102321-T, and by the Madrid Regional Government under project BLOQUES (ref. S2018/TCS-4339). Part of this work was made while C.B. was at IBM Research - Zurich (CH) and visiting the University of Aarhus (DK).

References

1. F. Armknecht, P. Walther, G. Tsudik, M. Beck, and T. Strufe. Promacs: Progressive and resynchronizing macs for continuous efficient authentication of message streams. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 211–223, 2020.
2. M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *2013 ACM SIGSAC CCS.*, pages 863–874. ACM, 2013.
3. R. Bendlin, S. Krehbiel, and C. Peikert. How to share a lattice trapdoor: Threshold protocols for signatures and (H)IBE. In *ACNS*, 2013.
4. D. J. Bernstein. A secure public-key signature system with sxtremely fast verification.
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 *PKC*, pages 1–16. Springer, 2011.
7. X. Boyen. Lattice mixing and vanishing trapdoors: A framework for fully secure short signatures and more. In *PKC*, pages 499–517. Springer, 2010.
8. D. Cash, D. Hofheinz, E. Kiltz, and C. Peikert. Bonsai trees, or how to delegate a lattice basis. In *EUROCRYPT*. Springer, 2010.
9. D. Catalano, D. Fiore, and B. Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In *Advances in Cryptology – CRYPTO*, 2014.
10. Y. Chen, A. Lombardi, F. Ma, and W. Quach. Does fiat-shamir require a cryptographic hash function? In T. Malkin and C. Peikert, editors, *CRYPTO*, 2021.
11. 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.
12. J. Ding and D. Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *ACNS*, LNCS, 2005.
13. D. Fiore, A. Mitrokotsa, L. Nizzardo, and E. Pagnin. Multi-key homomorphic authenticators. In *ASIACRYPT*, 2016.
14. M. Fischlin. Progressive verification: The case of message authentication. In *International Conference on Cryptology in India*, pages 416–429. Springer, 2003.
15. R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.
16. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *ACM STOC*, 2008.
17. S. Gorbunov, V. Vaikuntanathan, and D. Wichs. Leveled fully homomorphic signatures from standard lattices. In *STOC*, pages 469–477. ACM, 2015.
18. S. Gorbunov and D. Vinayagamurthy. Riding on asymmetry: Efficient ABE for branching programs. In *ASIACRYPT*, LNCS, 2015.
19. S. Katsumata and S. Yamada. Group signatures without NIZK: from lattices in the standard model. In *Advances in Cryptology - EUROCRYPT*, 2019.
20. L. Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International, 1979.
21. D. V. Le, M. Kelkar, and A. Kate. Flexible signatures: Making authentication suitable for real-time environments. In *ESORICS*. Springer, 2019.
22. A. Loveless, R. Dreslinski, B. Kasikci, and L. T. X. Phan. Igor: Accelerating byzantine fault tolerance for real-time systems with eager execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.
23. V. Lyubashevsky. Lattice signatures without trapdoors. In *EUROCRYPT*, 2012.
24. D. Micciancio and C. Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *EUROCRYPT*, 2012.
25. M. S. E. Mohamed and A. Petzoldt. RingRainbow - An efficient multivariate ring signature scheme. In *Progress in Cryptology - AFRICACRYPT*, LNCS, 2017.
26. T. Plantard, A. Sipasseuth, C. Dumondelle, and W. Susilo. Drs: diagonal dominant reduction for lattice-based signature. In *PQC Standardization Conference*, 2018.
27. A. Sipasseuth, T. Plantard, and W. Susilo. Using freivalds' algorithm to accelerate lattice-based signature verifications. In *ISPEC*. Springer, 2019.
28. A. R. Taleb and D. Vergnaud. Speeding-up verification of digital signatures. *Journal of Computer and System Sciences*, 2020.

29. R. Tsabary. An equivalence between attribute-based signatures and homomorphic signatures, and new constructions for both. In *Theory of Cryptography TCC*, 2017.
30. Q. Wang, H. Khurana, Y. Huang, and K. Nahrstedt. Time valid one-time signature for time-critical multicast data authentication. In *IEEE INFOCOM 2009*.

Appendix

A Concrete Leakage Estimates

Let $rk(\mathbf{A})$ denote the rank of a matrix $\mathbf{A}$, and let $\langle \mathbf{v} \rangle_q$ be the hyperplane of $\mathbb{Z}_q^n$ generated by a vector $\mathbf{v} \in \mathbb{Z}_q^n$.

Lemma 2. *Let $\mathcal{C} := \{\mathbf{C} \in \mathbb{Z}_q^{k\times n} \ : \ rk(\mathbf{C}) = k\}$ be the set of $k \times n$ matrices that are full rank and have entries in $\mathbb{Z}_q$ (as introduced in the proof of Lemma 1). For any given set of vectors $\{\mathbf{w}_1, \dots, \mathbf{w}_{q_V+1}\} \subseteq \mathbb{Z}_q^n$ such that $\mathbf{w}_i \notin \langle \mathbf{w}_j \rangle_q$ for every $i \neq j$, define the collection of sets $\{\mathcal{H}_i := \{\mathbf{C} \in \mathcal{C} \ : \ \mathbf{C} \cdot \mathbf{w}_i = 0 \mod q\}\}_{i=0}^{q_V+1} \subseteq \mathcal{C}$ containing the matrices having the corresponding $\mathbf{w}_i$ in their right kernel and $\mathcal{H}_0 = \{\varnothing\}$. It holds that*

$$\left| \mathcal{C} \setminus \left(\bigcup_{j=0}^{i-1} \mathcal{H}_j \right) \right| \geq |\mathcal{H}_1| \cdot \left(\frac{q^n - 1}{q^{n-k} - 1} - (i-1) \right) \quad \forall \ i = 1, \dots, q_V. \tag{4}$$

Proof of Lemma 2 It is easy to see that the cardinality of $\mathcal{C}$ is:

$$|\mathcal{C}| = (q^n - 1) \cdot (q^n - q) \cdot (q^n - q^2) \cdot \ldots \cdot (q^n - q^{k-1}) = q^{\frac{k(k-1)}{2}} \cdot \prod_{j=0}^{k-1} (q^{n-j} - 1).$$

as we have full freedom for how to pick the first row of $\mathbf{C}$ (except for $\mathbf{c}_1 \neq 0$); for the second row, we can pick any vector $\mathbf{c}_2$ that is in $\mathbb{Z}_q^n$ but not in the span of the previous row of $\mathbf{C}$ (to keep the matrix full rank), and so on. The formula after the final equality follows from decomposing each $(q^n - q^j)$ factor as $q^j(q^{n-j} - 1)$ and observing that $\prod_{j=1}^{k-1} q^j = q^{\frac{k(k-1)}{2}}$. The same reasoning applies to computing the cardinality of a generic $\mathcal{H}_i$ (for $i > 0$), after noticing that the rows of the matrices $\mathbf{C} \in \mathcal{H}_i$ must be picked (as linearly independent vectors) from the $(n-1)$-dimensional hyperplane orthogonal to $\mathbf{w}_i$; thus

$$|\mathcal{H}_i| = \prod_{j=0}^{k-1} (q^{n-1} - q^j) = q^{\frac{k(k-1)}{2}} \cdot \prod_{j=0}^{k-1} (q^{(n-1)-j} - 1) = q^{\frac{k(k-1)}{2}} \cdot \prod_{j=1}^{k} (q^{n-j} - 1)$$

This proves the base case ($i = 1$) of the bound in (4): $|\mathcal{C} \smallsetminus \varnothing| = |\mathcal{H}_1| \cdot \frac{q^n-1}{q^{n-k}-1}$, since, compared to $|\mathcal{H}|$, $|\mathcal{C}|$ has the additional factor $j = 0$ and the missing factor $j = k$. Concretely this means that: $\Pr[\mathsf{bad}_1] \leq \frac{q^{n-k}-1}{q^n-1}$. For $i = 2$, again $|\mathcal{C} \setminus \mathcal{H}_1| = |\mathcal{C}| - |\mathcal{H}_1| = |\mathcal{H}_1| \left(\frac{q^n-1}{q^{n-k}-1} - 1 \right)$. For $i = 3$, we remove from the pool of eligible $\mathbf{C}$ all those matrices in $\mathcal{H}_1 \cup \mathcal{H}_2$, i.e., that have either $\mathbf{w}_1$ or $\mathbf{w}_2$ in their right kernel.[7] In other words, matrices composed by only rows orthogonal to $\mathbf{w}_1$ or to $\mathbf{w}_2$. The hyperplanes $\mathbf{w}_1^\perp$ and $\mathbf{w}_2^\perp$ both have dimension $n - 1$, and since we are in a space of dimension n, they must intersect in a subspace of dimension $n - 2$. For a tighter bound we use:

[7] The vectors $\mathbf{w}_i$ are assumed not to be multiples of one another. Otherwise, $\mathcal{A}$ does not extract new information from a rejection, i.e., there is no additional leakage.

$|\mathcal{H}_2 \setminus \mathcal{H}_1| = |\mathcal{H}_2| - |\mathcal{H}_1 \cap \mathcal{H}_2|$ and recall that $0 \le |\mathcal{H}_2| - |\mathcal{H}_1 \cap \mathcal{H}_2| \le |\mathcal{H}_1|$. Hence after the second rejected query the number of possible $\mathbf{C}$ becomes:

$$\begin{aligned}
&|\mathcal{C} \setminus (\mathcal{H}_1 \cup \mathcal{H}_2)| = |\mathcal{C}| - |\mathcal{H}_1| - |\mathcal{H}_2| + |\mathcal{H}_1 \cap \mathcal{H}_2| \\
&= |\mathcal{H}_1| \cdot (\frac{q^n - 1}{q^{n-k} - 1} - 2) + |\mathcal{H}^{(n-2)}| \\
&= |\mathcal{H}_1| \cdot (\frac{q^n - 1}{q^{n-k} - 1} - 2) + q^{k(k-1)/2} \prod_{j=2}^{k-1} (q^{n-j} - 1) \\
&= |\mathcal{H}_1| \cdot (\frac{q^n - 1}{q^{n-k} - 1} - 2 + \frac{1}{(q^{n-1} - 1)(q^{n-k} - 1)}) \ge |\mathcal{H}_1| \cdot (\frac{q^n - 1}{q^{n-k} - 1} - 2).
\end{aligned}$$

Remark that $\mathbf{C}$ could be still composed by some elements of $\mathcal{H}_1$ *and* some of $\mathcal{H}_2 \setminus \mathcal{H}_1$; this would be consistent with $\mathcal{A}$'s view at this point.

We can now proceed by induction, assuming (4) holds for the query index i, prove it for $i + 1$.

$$\begin{aligned}
\left| \mathcal{C} \setminus \bigcup_{j=0}^{i} \mathcal{H}_j \right| &= \left| \mathcal{C} \setminus \left((\bigcup_{j=0}^{i-1} \mathcal{H}_j) \cup \mathcal{H}_i \right) \right| \\
&= |\mathcal{C}| - \left| \bigcup_{j=0}^{i-1} \mathcal{H}_j \right| - |\mathcal{H}_i| + \left| \left(\bigcup_{j=0}^{i-1} \mathcal{H}_j \right) \cap \mathcal{H}_j \right| \\
&= \left| \mathcal{C} \setminus \bigcup_{j=0}^{i-1} \mathcal{H}_j \right| - |\mathcal{H}_1| + \left| \left(\bigcup_{j=0}^{i-1} \mathcal{H}_j \right) \cap \mathcal{H}_j \right| \ge \left| \mathcal{C} \setminus \bigcup_{j=0}^{i-1} \mathcal{H}_j \right| - |\mathcal{H}_1| \\
&\ge |\mathcal{H}_1| \cdot \left(\frac{q^n - 1}{q^{n-k} - 1} - i + 1 \right) - |\mathcal{H}_1| = |\mathcal{H}_1| \cdot \left(\frac{q^n - 1}{q^{n-k} - 1} - i \right).
\end{aligned}$$

□

B Examples Of Efficient Verification

Because any instantiation of our compiler is completely determined by the four subroutines **parse** pk, GetM, Check, and GetZV, in what follows we explain only how these four algorithms work. The complete descriptions of offVer and onVer are derived using the general structure in Figure 4.

B.1 From Lattices

We present concrete instantiations of our compiler for two categories of LBS: 'hash & sign' with representative the GPV08 signature [16], and 'Boyen/BonsaiTree' style with representative MP12 [24].

Efficient Verification for GPV08 [16]. 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 $PK.\mathsf{aux} = (\mathcal{H}, \beta)$, i.e., a description of a full-domain hash function $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_q^n$ and the norm bound $\beta \in \mathbb{R}$. The Check procedure is exactly as in the original verification (enforcing the norm bound β on the signature). The GetM algorithm takes in input the public matrix

$PK = \mathbf{A}$, and tails to it the identity matrix: $\mathbf{M} = [\mathbf{A}| - \mathbf{I}_n]$. The GetZV routine returns the matrix $\mathbf{Z}'$ (explained momentarily) and the vector $\mathbf{v} = [\boldsymbol{\sigma}|\mathcal{H}(\mu) \cdot \mathbf{1}_{1\times n}]$. The matrix $\mathbf{Z}'$ is made up of the same 'randomized key' vectors produced by GetZ during the offline verification, i.e., $\mathbf{z}'_j = \mathbf{z}_j \leftarrow \mathbf{c}_j\mathbf{M} = [\mathbf{c}_j\mathbf{A}| - \mathbf{c}_j]$. 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 [24]. The **parse** pk procedure assigns $PK \leftarrow \mathbf{A} = [\tilde{\mathbf{A}}|\mathbf{A}_0 \mid \dots \mid \mathbf{A}_\ell] \in \mathbb{Z}_q^{n\times(\bar{m}+n\lceil\log q\rceil\ell)}$ (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 $PK.\mathsf{aux} = (\mathbf{u}, \beta)$. The Check procedure is exactly as in the original verification (enforcing the norm bound β 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 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 [24]). 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.

B.2 From Multivariate Equations

For signatures based on multivariate equations we take Rainbow [11,12] 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 [11]. 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 consider the field $\mathbb{F} = \mathbb{F}_{2^r}$. 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., $\tilde{\mathbf{s}} \leftarrow [\mathbf{s}[1]^2, \mathbf{s}[1]\mathbf{s}[2], \dots, \mathbf{s}[n-1]\mathbf{s}[n], \mathbf{s}[n]^2]$.

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.

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 (1). In brief, a Σ^E achieves $(\mathsf{r}_0, \mathsf{e}_0)$-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 between the cost of running the offline verification once and using its outcome in r_0 online verifications, over the cost of running 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* (e.g., 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 [27]. Table 2 collects the common notation, while Table 1 displays a summary of our findings, that we motivate below.

Table 2: Parameters involved in the performance analysis of our compiler for efficient verification.

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 for signature with $\mathbf{Mv}$-style verification, e.g., [5,11,12,16,24,17], is dominated by a matrix-vector multiplication. Let $n = rows(\mathbf{M})$ and $m = cols(\mathbf{M})$, with $m \geq n$.

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 4.1 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 1 collects the figures for three representative classes of signature schemes, if we apply our compiler for efficient verification at 128 bit of security. The values are extrapolated as explained in the reminder of the section. In detail, k_0 depends on the signature Σ as it is the minimal value of the confidence level k for which Σ^E is existentially unforgeable; k_0 determines the length of the svk. The value r_0 is the minimum number of verifications to run in order to achieve a concrete efficiency gain of e_0. Thus, lower values of e_0 and r_0 correspond to better efficiency gains. The last column in Table 1 displays the ratio k_0/n that essentially tells how much *cheaper* onVer is compared to the original verification Ver (ignoring the one-time cost of running offVer). 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 (i.e., onVer is expected to be about 99× faster).

For convenience we categorize signatures according to the size of their underlying algebraic structure.

The modulo 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. [17] and by Fiore et al. [13]. In this case, using our compiler (with some caveats, as we show in the next section) yields that the advantage in the cmvEUF experiment (as per Definition 4) is negligible in the security parameter λ for any confidence level $k \geq k_0 = 1$. However, in [13,17] 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[8].We computed parameters for this family of schemes according to Albrecht et al.'s methodology[9]. 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 reached by our compiler for [13,17]. Recall formula given in Definition 3 for concrete amortized efficiency:

$$\frac{\mathsf{cost}\big(\mathsf{offVer}(\mathsf{pk}, k_0)\big) + \mathsf{r} \cdot \mathsf{cost}\big(\mathsf{onVer}(\mathsf{svk}, \mu, \boldsymbol{\sigma})\big)}{\mathsf{r} \cdot \mathsf{cost}\big(\mathsf{Ver}(\mathsf{pk}, \mu, \boldsymbol{\sigma})\big)} < 1$$

[8] In [13] 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 $t = 1$.

[9] Albrecht, Curtis, Deo, Davidson, Player, Postlethwaite, Virdia, Wunderer. Estimate all the LWE, NTRU schemes! in SCN 2018.

where k_0 is the chosen accuracy level and we are interested in finding the pair of values $(\mathsf{r}_0, \mathsf{e}_0)$ where r_0 is the minimum non-negative integer r for which this inequality hods, and $\mathsf{e}_0 < 1$ is a tight upperbound on the cost in the left hand side of the inequality when $\mathsf{r} = \mathsf{r}_0$. Notably, the better the efficiency of the offline/online verification, the smaller the value of e_0. In the case of signatures with $\mathbf{Mv}$-style verification it is easy to see that the left hand side of the inequality can be approximated with $\frac{k_0 nm + \mathsf{r} k_0 m}{rnm} = \frac{k_0}{\mathsf{r}} + \frac{k_0}{n}$. Setting $k = k_0 = 1$ and $n = 256$ we want to extract the minimum r_0 for which $1/\mathsf{r}_0 + 1/256$ is smaller than 1, formally $\mathsf{r}_0 = \min\{\mathsf{r} \in \mathbb{Z}_{>0} \mid 1/\mathsf{r} + 1/256 < 1\}$. It is easy to see that $\mathsf{r}_0 = 2$ suffices and we get $1 > \mathsf{e}_0 = \mathsf{0.504} > \mathsf{1/2 + 1/256}$. 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 running 2 standard verifications, while preserving the security level. Moreover, for this set of parameters $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k_0}{n} = \frac{1}{256} < 0.004$, 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 $99\times$ faster than Ver.

The modulo q is a large 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. [16], Boyen [7], and its improved version by Micciancio and Peikert [24], Boneh and Franklin's linearly homomorphic signature [6], Rainbow [12] and LUOV [5]. For the lattice-based constructions, in order to guarantee a negligible advantage in the cmvEUF experiment (see Definition 4) we need to set an appropriate value of $k \geq k_0 > 1$. 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 $\boldsymbol{\sigma} \in \mathbb{Z}_q^m$. 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 cmvEUF advantage to be negligible it must hold that $q^{-k} \leq 2^{-\lambda}$. Hence, to estimate k, one needs to first fix the value of λ, compute the corresponding q that can guarantee such level of security, and then extract the minimum value k_0 for which the above relation holds.

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[10]. 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 [24], we use the parameters provided in Figure 2 in [24]. Since in [24] they set $\delta = 1.007$, to ensure a fair comparison, we compute the parameters Gentry et al.'s signature [16] for the same value of δ. As a result, we observed that for this δ all of the schemes require about the same modulo $q = 2^{30}$ (for $n = 256$). For this set of parameters, our efficient verification provides 80 (resp. 128; 250) bits of security with just $k_0 = 3$ (resp. $k_0 = 5$; $k_0 = 9$). Thus our compiler achieves a $(4, 0.77)$-concrete amortized efficiency (resp.$(6, 0.86)$; $(10, 0.94)$), and a concrete tradeoff between onVer and Ver of $k_0/n < 0.02$ (resp. 0.02; 0.04). In particular, for the lower security settings this means that onVer is about $98\times$ faster than Ver.

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

[10] Gama and Nguyen. Predicting lattice reduction, in EUROCRYPT, 2008.

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 $k_0 = 32$, since $q^{-k_0} = 2^{-4\cdot 32} = 2^{-128}$. yields that the minimum number of repetitions r_0 to achieve amortized efficiency (i.e., for which we have $k_0/\mathsf{r} + k_0/n < 1$) 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 $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k_0}{n} = \frac{32}{64} = 0.5$, 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 $k_0 = 16$, since $q^{-k_0} = 2^{-8\cdot 16} = 2^{-128}$. For the highest NIST security category, [11] 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 $k_0 = 16$ 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.

B.3 Generalization To Signatures With Properties

Efficient verification can be easily generalized to the case of signatures with different security notions, such as *strong* or *selective* unforgeability, or with advanced properties. This is of particular interest for LBS, where the versatility of well-established hardness assumptions has already given life to a variety of constructions under different security models and realizing advanced properties, including *homomorphic* [17], *threshold* [3], *constrained* [29] and *indexed attribute based* signatures [19]; and yet relying on an **Mv**-style verification (as introduced in the beginning of the section, and displayed in Figure 3).

Signatures with properties require more complex security definitions than plain existential unforgeability. Figure 6 provides a generic formalism to unify the description of the unforgeability experiments for signatures with properties. In a nutshell the common requirements are:

$\mathsf{Exp}^{\mathsf{generic-unf}}_{\mathcal{A},\Sigma}(\lambda)$

$val \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pval}, \mathsf{sval}) \leftarrow \mathsf{Setup}(1^\lambda)$

$\mathsf{st}_K \leftarrow \emptyset,\ \mathsf{st} \leftarrow \emptyset$

$(\mu^*, \boldsymbol{\sigma}^*, \mathsf{aux}^*) \leftarrow \mathcal{A}^{OK(\,\cdot\,;\mathsf{sval},\mathsf{st}_K),\ O(\,\cdot\,;\mathsf{st})}(\mathsf{pval}, val)$

$b \leftarrow \mathsf{WinCond}(\mu^*, \boldsymbol{\sigma}^*, \mathsf{aux}^*, \mathsf{pval}, \mathsf{st}_k, \mathsf{st}, val)$

$\textbf{if } \mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*, \mathsf{aux}^*) = 1 \ \wedge\ b$

$\quad \textbf{return } 1$

$\textbf{else return } 0.$

Fig. 6: Generic description of the unforgeability under adaptive chosen message attacks experiment for signatures with properties.

1. If the signature guarantees *selective* unforgeability, the first step in the experiment is for $\mathcal{A}$ to declare the target messages for the forgery; in Figure 6 this is handled via the *val* variable. If unforgeability is adaptive, *val* is set to $\perp$.
2. A setup phase, where a probabilistic routine (denoted Setup in Figure 6) generates a set of secret values sval –handed over to the oracles– and other public auxiliary values pval, that include verification keys, delivered to the adversary.
3. 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.).
4. 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 additional checks to ensure the signature is actually a forgery (and is not trivially derivable from the adversary's view, e.g., because it was 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, and on the type of unforgeability: If selective, it checks that the queries and the forgery are consistent with the values *val* declared at the beginning of the game. If existential, it checks that μ^* was not queried to the signing oracle. If strong, it checks that the queries to the signing oracle are all distinct.

Adapting the syntax and security experiment of efficient verification to signatures with properties is rather straightforward. Similarly, our compiler of Section 4.1 can be easily adjusted to work on signatures with properties and with **Mv**-style verification, as we discuss momentarily. Regarding security, the core part of the proof of Theorem 1 is information-theoretical, and therefore it does not significantly change when considering signatures that are only selectively unforgeable, or strongly unforgeable. In the following we analyze the impact of our compiler on the efficiency of some schemes whose verification is structured as in Figure 3: the constrained LBS in [29], the (indexed) attribute-based LBS in [19], the homomorphic LBS in [13], the threshold LBS in [3], and the multivariate-based ring signature RingRainbow [25]. This list is by no means an exhaustive list. Indeed, in this work we decided to ignore lattice-based signatures with properties that are obtained using the Fiat-Shamir with abort technique from [23], despite the fact that wherever the result of Chen et al. [10] is applicable, our compiler is too. The reason is that signatures with properties that rely on such technique are many, and the efficiency gain computation is similar to the one performed in Section B .

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 [29] includes an **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 $k = 9 \ll \lambda$ so that the advantage of $\mathcal{A}$ in Theorem 1 is $\frac{q_V+1}{q^k - q_V} < 1/2^{256}$. 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 $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k}{n} = \frac{8}{256} < 0.036$, i.e., the online verification requires about 3.6% 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 [19] and HS in [13] 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[11]. 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 [18]. The vectors $(\mathbf{Z}', \mathbf{v})$ used in the verification might (as in [13]) or might not (as in [19]) 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 B . Regarding the iABS, the suggested value of the modulo 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 already $k = 6$ guarantees that $\frac{q_V+1}{q^k - q_V} < 1/2^{305}$, thus 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: $\frac{\mathsf{cost}(\mathsf{onVer})}{\mathsf{cost}(\mathsf{Ver})} \leq \frac{k}{O(d \log d)} = \frac{6}{O(d \log d)}$. From this inequality is clear that already for a circuit of depth 4 the online verification only requires 75% 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. Bendlin, Krehbiel, and Peikert [3] 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 B).

[11] The construction of group signature in [19] has this iABS as building block, but it does not satisfy any of these conditions, as the verification circuit depends strongly on the signature. The authors did not find a straightforward way to modify this construction to have efficient verification without significantly impacting the signature length.

RingRainbow [25]. 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 **Mv**-style verification with the same technique used for Rainbow (cf. Section B). 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 [25]). 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 $\frac{q_V+1}{q^k - q_V} = 1/2^{256}$ for 128 bits of post-quantum security, which is ensured by $k \geq k_0 = 36$. Plugging these values in our amortized efficiency formula $\frac{k_0}{\mathsf{r}} + \frac{k_0}{n}$ (that is the formula derived from Definition 3 for signatures with **Mv**-style verification) yields that the minimum number of repetitions r_0 to achieve meaningful amortized efficiency is $\mathsf{r}_0 = 580$, and the corresponding amortized efficiency factor is $\mathsf{e}_0 0.8992 > 36/580 + 36/43$. In this case, our compiler produces an online verification such that $\frac{\mathsf{cost(onVer)}}{\mathsf{cost(Ver)}} = \frac{k_0}{n} = \frac{36}{43} < 0.86$, in other words, our compiler produces an online verification that requires only 86% of the computation required by the standard verification.

C Modeling Efficient & Progressive Signature Verification With R-Bounded Randomness Reuse

Given the two notions of efficient verification (Section 2) and progressive verification (Section 3), the question naturally rises whether it be possible to combine these frameworks and simultaneously realize progressive *and* efficient verification. Surprisingly, the naïve combination does not achieve high accuracy and unforgeability. This is essentially because efficiency demands reuse of svk, which makes the confidence function degrade with every new verification, while progressiveness allows for premature verification outcomes that may leak a substantial amount of information about svk. In order to formally handle this situation and overcome the issue described above, we introduce the concept of **pr**ogressive and **ef**ficient (pref) verification with r-bounded randomness reuse. Similarly to Definition 5 (progressive 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 svk can be reused while guaranteeing unforgeability. For correctness and security, both k and r are input to the confidence function, which now is named α_{pref}.

Definition 8 (Progressive and Efficient Verification). *A signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Ver})$ admits a (r, k)-efficient and $(T, \alpha_{\mathsf{pref}})$-progressive verification realization $\Sigma^{F+E} = (\Sigma, \mathsf{prefVer})$ if there exist*

- *two positive integers:* r *(number of reuses of the secret randomness) and k (interruption step);*
- *an efficiently computable confidence function $\alpha_{\mathsf{pref}} : \{0, \dots, k\} \times \{0, \dots, \mathsf{r}\} \to [0, 1]$;*
- *a set of admissible sequences of states $\mathcal{S} = \{st^{(1)}, st^{(2)}, \dots\}$ (each sequence $st^{(j)}$ contains* $\mathsf{r} + 1$ *states st_i, i.e., $st^{(j)} = (\mathsf{st}_i)_{i=0}^{\mathsf{r}}$, $\mathsf{st}_0 = \varnothing$); and*
- *a progressive verification algorithm* $\mathsf{prefVer}$ *consisting of $k+1$ steps* $\mathsf{prefVer}_0, \dots, \mathsf{prefVer}_k$ *with the same syntax as in Definition 5.*

Definition 9 (r-Reuse k-Progressive Correctness). *Let Σ be a signature scheme that admits progressive and efficient verification realized by the tuple $(\mathsf{r}, k, \alpha_{\mathsf{pref}}, \mathsf{prefVer}, \mathcal{S})$. Then $\Sigma^{F+E} =$*

$(\Sigma, \mathsf{prefVer})$ *satisfies* (r, k)*-correctness if, 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* $(\mu_i, \boldsymbol{\sigma}_i)_{i=1}^{\mathsf{r}}$ *with* $\mu_i \in \mathcal{M}$ *and* $\boldsymbol{\sigma}_i$ *such that* $\mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 1$ *and for any sequence of interruption values* $(t_i)_{i=1}^{\mathsf{r}} \subseteq \{1, \ldots, k\}$*, it holds that:*

$$\Pr\left[\mathsf{prefVer}(\mathsf{st}_i, \mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i, t_i) = \alpha_{\mathsf{pref}}(t_i, i)\right] = 1 \quad \text{for all } i = 1, \ldots, \mathsf{r}.$$

Definition 10 (Concrete Amortized Efficiency). *A scheme* $\Sigma^{F+E} = (\Sigma, \mathsf{prefVer})$ *realizes* $(\mathsf{r}_0, \mathsf{e}_0)$*-concrete amortized efficiency if, for a given security parameter* λ*, for any key pair* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(\lambda)$*, for any pair tuple of pairs* $(\mu_i, \boldsymbol{\sigma}_i)$ *with* $\mu_i \in \mathcal{M}$ *and* $\boldsymbol{\sigma}_i$ *such that* $\mathsf{Ver}(\mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i) = 1$*, for any one sequence of admissible states* $(st_i)_{i=0}^{\mathsf{r}} \subseteq \mathcal{S}$*, there exist a small, real constant* $0 < \mathsf{e}_0 < 1$*, and a non-negative integer* r_0 *such that for every* $\mathsf{r} \geq \mathsf{r}_0$ *the following holds true:*

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

C.1 Bounded Efficient And Progressive Security

Figure 7 collects a description of our security game and experiment for existential unforgeability under adaptive chosen message attack for signatures with *progressive and efficient verification* (r-prefEUF).

$\mathsf{r\text{-}prefEUF}(\Sigma, \lambda, k)$

- $\mathsf{ctr} \leftarrow 0, \mathsf{st}_0 \leftarrow \varnothing, \mathsf{L}_S \leftarrow \varnothing$
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^{\lambda})$
- $(\mu^*, \boldsymbol{\sigma}^*, t^*) \leftarrow \mathcal{A}^{O\mathsf{Sign}, \mathsf{prefVer}}(\mathsf{pk}, \lambda)$
- **return** $(\mathsf{ctr}, \mu^*, \boldsymbol{\sigma}^*, t^*)$

$O\mathsf{prefVer}_k(\mathsf{st}_{\mathsf{ctr}}, \mathsf{pk}, \mu, \boldsymbol{\sigma}, t')$

- **if** $(\mathsf{ctr} \geq \mathsf{r})$ **return** $\perp$
- $t \leftarrow O\mathsf{Int}(t')$
- $\alpha \leftarrow \mathsf{prefVer}(\mathsf{st}_{\mathsf{ctr}}, \mu, \boldsymbol{\sigma}, t)$
- $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
- **return** α

$\mathsf{Exp}_{\mathcal{A}, \Sigma, \mathsf{r}}^{\mathsf{r\text{-}prefEUF}}(\lambda, k)$

- $(\mathsf{ctr}, \mu^*, \boldsymbol{\sigma}^*, t') \leftarrow \mathsf{r\text{-}prefEUF}(\Sigma, \lambda, k)$
- $\beta \leftarrow \mathsf{Ver}(\mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*)$
- $t^* \leftarrow O\mathsf{Int}(t')$
- $\alpha \leftarrow \mathsf{prefVer}_k(\mathsf{st}_{\mathsf{ctr}}, \mathsf{pk}, \mu^*, \boldsymbol{\sigma}^*, t^*)$
- **if** $(\mu^* \in \mathsf{L}_S \vee \alpha = \perp \vee \beta = 1)$
  - **return** $(0, 0)$
- **return** (ctr, t^*)

$O\mathsf{Sign}_{\mathsf{sk}}(\mu)$

- $\mathsf{L}_S \leftarrow \mathsf{L}_S \cup \{\mu\}$
- $\boldsymbol{\sigma} \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$
- **return** $\boldsymbol{\sigma}$

Fig. 7: Security model for existential unforgability under chosen message and progressive verification for signatures with stateful, (k, r)-efficient and progressive verification: queries security game, experiment and oracles.

Definition 11 (r-Bounded Progressive Security (r-prefEUF)). *Let* Σ *be a signature scheme that admits a non-trivial realization of* (r, k)*-efficient and progressive verification* Σ^{F+E}*. Then, for a given security parameter* λ*,* Σ^{F+E} *is existentially unforgeable under adaptive chosen message attack with progressive and efficient verification attack* (r-prefEUF) *if for all efficient PPT adversaries* $\mathcal{A}$

the success probability in the r-prefEUF *experiment is:*

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

C.2 A Compiler For PREF MV-Style Verification

First of all we notice that our compiler for efficient verification described in Section 4.1 is trivially $\mathsf{poly}(\lambda)-$prefEUF (i.e., existentially unforgeable against an unbounded polynomial number of verification queries), by defining $\mathsf{prefVer}=(\mathsf{prefVer}_0,\mathsf{prefVer}_1)$ as shown in Figure 8. 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.

```
prefVer(st, pk, μ, σ, t)        k = 1
β ← 0
if t < 0
  return ⊥
if t > 1
  set t ← 1
for i = 0, t
  (st, β) ← prefVer_i(st, pk, μ, σ)
  if β = 0   return α = ⊥
return α = (1 − 1/q^t)
```

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

```
prefVer_0(st, pk, μ, σ)
parse st = (svk, ctr)
if st = ∅
  svk ← offVer(pk, 1)
  st ← (svk, 1)
else st ← (svk, ctr + 1)
return (st, 1)
```

```
prefVer_1(st, μ, σ)
parse st = (svk, ctr)
β ← onVer(st, μ, σ)
return (st, β)
```

Fig. 8: The trivial two-step prefVer solution based on our compiler for efficient verification (Figure 4).

We now present a compiler for signatures with **Mv**-style verification and $q=\mathsf{poly}(\lambda)$ that realizes efficient bounded progressive verification. This compiler builds on top of the two compilers presented in Section 4.1. Intuitively, the problem with progressive verification is that if interrupted after $t<k$ steps the process may erroneously accept an invalid signature with a non-negligible probability $\approx 1/q^t$. In Section 4.2 we mitigate this leakage of information between queries by refreshing the vectors in svk after every verification. This conservative approach clearly impacts efficiency. 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 progressive verifications.

Our compiler works essentially as the efficient verification compiler in Figure 5, except that the offVer algorithm (that generates a fresh svk) is run only *once every* r *verifications*. To further optimize the scheme, we replace the GetZV algorithm by k algorithms GetZV_i each of which is run by the corresponding $\mathsf{prefVer}_i$. 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. Each algorithm takes as input the corresponding i-th vectors $((\mathbf{c}_i,\mathbf{z}_i),PK.\mathsf{aux},\mu)$ and returns $(\mathbf{z}'_i,\mathbf{v}_i)$ that are defined according to the scheme considered:

```
prefVer(st, pk, μ, σ, t)
β ← 0
if t < 0
   return ⊥
if t > k
   set t ← k
parse st = (svk, ctr)
if st ≠ ∅ ∧ ctr > r
   set  st ← ∅
for i = 0, ..., t
   (st, β) ← prefVer_i(st, pk, μ, σ)
   if β = 0
      return α = ⊥
ctr ← ctr + 1  // update  st
return α = (1 − 1/q^t − ctr/q)
```

```
prefVer_0(st, pk, μ, σ)
parse st = (svk, ctr)
if st = ∅
   svk ← offVer(pk, k)
   st ← (svk, 0)  // ctr ← 0
return (st, 1)
```

```
prefVer_1(st, μ, σ)
parse st = (svk, ctr)
(z', v) ← GetZV_i(svk, μ, σ)
if z' · v = 0   mod q
   return (st, 1)
return (st, 0)
```

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

Fig. 9: Generic compiler to obtain efficient and progressive verification of signature schemes with $\mathbf{Mv}$-style verification and q polynomial in the security parameter.

GPV08 [16]: 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 [24]: the GetZV_i routine outputs $\mathbf{z}'_i = [\tilde{\mathbf{z}}_i \mid \mathbf{z}_i^0 + \sum_{j=1}^{\ell} \mu[j]\mathbf{z}_i^j|\mathbf{c}_i]$ and $\mathbf{v}_i = [\boldsymbol{\sigma}|\mathbf{u}]$.
Rainbow [11]: 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 $\alpha_{\mathsf{pref}}(\cdot, \cdot)$ is defined as:

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

r_0-concrete amortized efficiency.

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

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

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

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

This yields $\sum_{i=0}^{\mathsf{r}_0-1} \mathsf{cost}\big(\mathsf{prefVer}(\mathsf{st}_i, \mathsf{pk}, \mu_i, \boldsymbol{\sigma}_i, k)\big) = knm + \mathsf{r}_0 k(n+m)$. 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.