Batch Signatures, Revisited

Carlos Aguilar-Melchor
SandboxAQ
carlos.aguilar@sandboxaq.com

Martin R. Albrecht
SandboxAQ
martin.albrecht@sandboxaq.com

Thomas Bailleux
SandboxAQ
thomas.bailleux@sandboxaq.com

Nina Bindel
SandboxAQ
nina.bindel@sandboxaq.com

James Howe
SandboxAQ
james.howe@sandboxaq.com

Andreas Hülsing
Eindhoven University of Technology
andreas@huelsing.net

David Joseph
SandboxAQ
david.joseph@sandboxaq.com

Marc Manzano
SandboxAQ
marc.manzano@sandboxaq.com

ABSTRACT

We revisit batch signatures (previously considered in a draft RFC, and used in multiple recent works), where a single, potentially expensive, "inner" digital signature authenticates a Merkle tree constructed from many messages. We formalise a construction and prove its unforgeability and privacy properties.

We also show that batch signing allows us to scale slow signing algorithms, such as those recently selected for standardisation as part of NIST's post-quantum project, to high throughput, with a mild increase in latency. We demonstrate the practical efficiency of batch signing in the context of TLS. For the example of Falcon-512 in TLS, we can increase the amount of connections per second by a factor 3.2x, at the cost of an increase in the signature size by $\sim$ 14% and the median latency by $\sim$ 25%, where both are ran on the same 30 core server.

We also discuss applications where batch signatures allow us to increase throughput and to save bandwidth. For example, again for Falcon-512, once one batch signature is available, the additional bandwidth for each of the remaining $N-1$ is only 82 bytes.

1 INTRODUCTION

Unkeyed and symmetric cryptography is known to be significantly cheaper than asymmetric cryptography from a computational perspective. Indeed, hash functions, stream or block ciphers typically require between a few cycles [1] to a few hundred cycles [12], whereas key establishment and digital signature primitives require between tens of thousands to hundreds of millions of cycles [6]. In situations where a substantial volume of signatures must be handled – e.g. a Hardware Security Module (HSM) renewing a large set of short-lived certificates or a load balancer terminating a large number of TLS connections per second – this may pose serious limitations on scaling these and related scenarios.

These challenges are amplified by upcoming public-key cryptography standards: In July 2022, the US National Institute of Standards and Technology (NIST) announced four algorithms for post-quantum cryptography (PQC) standardisation. In particular, three digital signature algorithms, namely Dilithium [16], Falcon [21], and SPHINCS$^+$ [14], were selected, and migration from current standards to these new algorithms is already underway [27]. One of the key issues when considering migrating to PQC is that the computational costs of the new digital signature algorithms are significantly higher than those of ECDSA; the fastest currently-deployed primitive for signing. This severely impacts the ability of systems to scale and inhibits their migration to PQC, especially in higher-throughput settings.

For instance, at a 128-bit security level, using the standard SUPERCOP platform benchmarks on a Core i7 Tigerlake processor [6], ECDSA over an Edwards curve requires 85K cycles for signing. The equivalent Dilithium, Falcon and SPHINCS$^+$ signatures need 272K, 570K and 25M cycles, respectively (considering the fastest alternative among existing variants for each).[1] The performance gap between ECDSA and the three PQC alternatives is vast. Furthermore, there are good reasons to choose Falcon or SPHINCS$^+$ over Dilithium for certain scenarios, which increases the gap further: Falcon provides smaller signatures and verification key sizes which makes it a strong contender in networking applications and SPHINCS$^+$ relies on conservative security assumptions which are appropriate for long-term security.

In 2020 an RFC draft [2] proposed *Batch Signing for TLS* to solve existing scalability challenges of classical digital signature standards in a high-throughput TLS setting. In this approach, one expensive "inner" signing operation signs the root of a Merkle tree constructed from a batch of messages. Then, the final signature for each message contains the sibling nodes of a message to recover the Merkle tree's root and the original "inner" signature. This represents a logarithmic increase in the signature size but asymptotically reduces the amortised cost to a few hash computations. The draft was not finalised, and thus has now become a deprecated TLS working group document [3]. While the proposal was motivated by classical signature standards and TLS, the approach can be generalised and used with any signature scheme, e.g. with one of the PQC schemes, and in a myriad of additional settings.

Other recent works have considered using Merkle trees to reduce (amortised) signature size in certificates by signing them in batches and using the fact that they all share the same "inner" signature. A recent work [9] focuses on *stateful* signatures targeting such signature size reduction. Stateful signature schemes are capable of producing small signatures, which are ideal for use cases such as certificate authorities, but at the expense of a more involved design, with the critical need for state management. A *stateless* approach, in contrast, generalises more easily and allows

[1] See also the discussion of Falcon's performance in Section 2.3.

for a more flexible design applicable to a plethora of use cases for increasing the throughput of signature schemes. A recent RFC draft [4] proposes a stateless system to reduce certificate sizes by defining CAs that only sign certificates in batches and rely on a Certificate Transparency channel to deliver the large "inner" signatures.

These works highlight the increasing community interest in batch signatures.

1.1 Contributions

In this work, we study batch signatures and provide (i) a refinement of the construction from [2] to reduce the size by removing the collision-resistance property from the requirements of the hash function used to build the Merkle tree, (ii) a formal treatment of the unforgeability and privacy of batch signatures, (iii) a description of several settings in which batch signatures can have a positive impact in terms of either throughput increase or bandwidth reduction and (iv) an empirical study into batch signatures for both classical and PQC schemes in the context of TLS and certificate signing. Overall, our performance study indicates that, while the approach introduces a logarithmic overhead in signature sizes (cf. Table 1) and signing latency, it significantly reduces the CPU burden (cf. Table 2) allowing us to scale to a larger number of signatures per second compared with the plain approach using the same number of cores. Moreover, our benchmarks also indicate that in some applications batch signatures result in significant bandwidth savings.

In more detail, after some preliminaries in Section 2, we formally define batch signing and its security properties in Section 3. In particular, in addition to the usual unforgeability property, we also define batch privacy notions that essentially control the leakage of information due to signing in batches. We define two variants of batch privacy, with our construction achieving the weaker one. We then specify our batch signing scheme in Section 4, which is essentially a refined version of that in [2]. The main difference is that we do not need to rely on collision resistance but instead on target collision resistance [5], allowing us to pick smaller parameters and thus reduce the signature size. We prove the security properties of our construction in Section 5. We describe some real-world applications where batch signatures can provide significant improvements in Section 6 and finally describe our implementation for the TLS scenario in Section 7.

Table 1: Batch signature sizes for a targeted security level λ.

Scheme	λ	$\lvert vk \rvert$	$\lvert \sigma \rvert$	N	$\lvert sig \rvert$	$\lvert sig_c \rvert$
ECDSA P256	128	64	64	32	162	98
Dilithium2	128	1312	2420	32	2518	98
Dilithium5	256	2592	4595	32	4693	98
Falcon-512	128	897	666	16	748	82
Falcon-1024	256	1793	1280	32	1378	98
Falcon-512-fpemu	128	897	666	16	758	82
Falcon-1024-fpemu	256	1793	1280	16	1362	82

All sizes are in bytes. Batch signature size is given in the column $\lvert sig \rvert$, verification key size in $\lvert vk \rvert$, "inner" signature size in $\lvert \sigma \rvert$; all for a batch of size N. The compressed batch signature size, assuming the inner signature for multiple batch signatures is cached, is given in column $\lvert sig_c \rvert$. We assume two bytes are used to encode N in sig.

Signature sizes (or certificates) grow by fewer than one hundred bytes. This represents, for the algorithms considered (except ECDSA), at most ten percent when considering signatures and at most five percent when considering certificates (signatures + verification keys).

Table 2: Handshakes per second and latency for different percentiles in TLS using different signing algorithms.

Scheme and Instantiation		Handshakes Per Second	Latency (ms) med	p90	p99	Signing Cores
ECDSA P256	Plain	39,000	1.2	1.3	1.5	1.2
	MT N=32	49,000	1.3	1.7	2.7	1.0
Dilithium2	Plain	29,000	1.6	1.8	2.1	2.9
	MT N=32	50,000	1.8	2.2	2.7	1.0
Dilithium5	Plain	25,000	1.9	2.2	2.4	4.4
	MT N=32	43,000	2.2	2.6	3.2	1.0
Falcon-512	Plain	28,000	1.1	1.3	1.5	7.8
	MT N=16	43,000	1.5	1.8	2.5	2.0
Falcon-1024	Plain	24,000	2.0	2.1	2.3	13.1
	MT N=32	43,000	2.2	2.5	3.3	2.0
Falcon-512 (fpemu)	Plain	5,000	5.1	5.2	6.0	20.0
	MT N=16	16,000	6.4	7.6	8.4	8.0
Falcon-1024 (fpemu)	Plain	2,600	9.9	10.0	11.0	22.5
	MT N=16	8,200	12.0	15.0	17.0	8.0

All experiments are run on a 30 core machine with HyperThreading disabled. The results are presented for a 'plain' multi-threaded implementation (pool of as many threads as CPU cores, with select/poll handling), and for the Merkle Tree (MT) approach with a limit N to the maximum size of a tree. The amount of cores used for signatures is estimated for the plain approach, out of the computational cost of one signature, and fixed (by reserving cores explicitly) for the Merkle Tree approach.

The number of handshakes per second is roughly doubled for fast algorithms, and multiplied by a factor between three and four for slow algorithms. Latency (99th percentile) is increased by roughly fifty percent (one millisecond for fast algorithms and up to six milliseconds for the slower ones).

2 PRELIMINARIES

We write $x \leftarrow y$ for assigning y to x and $x \leftarrow\!\!\$\ \mathcal{D}$ for sampling x from some distribution $\mathcal{D}$. If $\mathcal{D}$ is a finite set, we assume the uniform distribution over this set. We write PPT for probabilistic polynomial time and BQP for bounded-error quantum polynomial time.

2.1 Hash Functions

In this work we consider *tweakable* hash functions. These are keyed hash functions that take an additional input which can be thought of as a domain separator (while the key or public parameter serves as a separator between users). When used right, tweakable hash functions allow to tightly achieve target collision resistance even in multi-target settings where an adversary wins when they manage to attack one out of many targets.

Definition 2.1 (Tweakable Hash Function [5]). Let $n, m \in \mathbb{N}$, let $\mathcal{P}$ be the public parameters space and $\mathcal{T}$ the tweak space. A *tweakable hash function* is a tuple of algorithm $\mathrm{H} = (\mathrm{KeyGen}, \mathrm{Eval})$ such that:

$\underline{\text{SM-TCR}^{\mathcal{A}}_{\text{H}}(k,\lambda)}$

$P \leftarrow \text{KeyGen}(1^\lambda)$

$S \leftarrow \mathcal{A}_0^{\text{H}(P,\cdot,\cdot)}(1^\lambda)$

/ $Q = \{(t_0,\mu_0),\ldots,(t_{k-1},\mu_{k-1})\}$ queries submitted to $\text{H}(P,\cdot,\cdot)$

for $i,\ell \in \{0,\ldots,k-1\}, i \neq \ell$ **do**

  if $t_i = t_\ell$: **return** $\perp$

$(j,\mu) \leftarrow \mathcal{A}_1(1^\lambda, S, P, Q)$

return $0 \leq j < k \wedge \mu \neq \mu_j \wedge \text{H}(P,t_j,\mu_j) = \text{H}(P,t_j,\mu)$

Figure 1: Single-function, Multi-Target Collision Resistance for distinct tweaks (SM-TCR).

Commit	RoR(x)
$b \leftarrow\!\$\ \{0,1\}$	$k \leftarrow\!\$\ \{0,1\}^\lambda$
$b' \leftarrow \mathcal{A}^{\text{RoR}(\cdot)}()$	$y_0 \leftarrow \text{F}(k,x)$
return $b = b'$	$y_1 \leftarrow\!\$\ \{0,1\}^n$
	return y_b

Figure 2: One-time Pseudorandom Function (OT-PRF)

KeyGen the setup function takes the security parameter 1^λ and outputs a (possibly empty) public parameter p. We write $p \leftarrow \text{KeyGen}(1^\lambda)$.

Eval the evaluation function takes public parameters p, a tweak t, an input $x \in \{0,1\}^m$ and returns a hash value h. We write $h \leftarrow \text{Eval}(p,t,x)$ or simply $h \leftarrow \text{H}(p,t,x)$. This is a deterministic function.

In what follows, we will avoid relying on plain collision resistance but target collision resistance of tweakable hash functions.

Definition 2.2 (Target Collision Resistant Hash Function [5]). An efficient tweakable hash function $\text{H} = (\text{KeyGen}, \text{Eval})$ is called single-function multiple-targets target-collision resistant for distinct tweaks (SM-TCR) if the advantage $\text{Adv}^{\text{sm-tcr}}_{\mathcal{A},\text{H}}(k,\lambda)$ of any (PPT/BQP) algorithms $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ that define up to k targets in the SM-TCR experiment defined in Figure 1 is negligible with

$$\text{Adv}^{\text{sm-tcr}}_{\mathcal{A},\text{H}}(k,\lambda) \coloneqq \Pr\left[\text{SM-TCR}^{\mathcal{A}}_{\text{H}}(k,\lambda) \Rightarrow 1\right].$$

We will also rely on one-time pseudrandomness to argue privacy.

Definition 2.3 (OT-PRF). Let $n, m \in \mathbb{N}$, $\text{F} : \{0,1\}^\lambda \times \{0,1\}^m \rightarrow \{0,1\}^n$ be a keyed function. We define

$$\text{Adv}^{\text{ot-prf}}_{\mathcal{A},\text{F}}(\lambda) \coloneqq \Pr[\text{OT-PRF}^{\mathcal{A}}_{\text{F}}(\lambda) \Rightarrow 1]$$

for $\text{OT-PRF}^{\mathcal{A}}_{\text{F}}(\lambda)$ as in Figure 2 and say F is an OT-PRF if no PPT/BQP adversary $\mathcal{A}$ has non-negligible advantage $\text{Adv}^{\text{ot-prf}}_{\mathcal{A},\text{F}}(\lambda)$.

2.2 Digital Signatures

Definition 2.4 (Signature Scheme). A signature scheme S consists of three PPT algorithms (KeyGen, Sign, Verify) such that:

$\underline{\text{EUF-CMA}^{\mathcal{A}}_{\text{S}}(\lambda) \text{ / BEUF-CMA}^{\mathcal{A}}_{\text{S}}(\lambda)}$

$Q \leftarrow \emptyset$;

$\text{vk}, \text{sk} \leftarrow \text{KeyGen}(1^\lambda)$;

$(\mu^\star, \sigma^\star) \leftarrow \mathcal{A}^{\text{Sign}}(\text{vk})$; / EUF-CMA

$(\mu^\star, \sigma^\star) \leftarrow \mathcal{A}^{\text{BSign}}(\text{vk})$; / BEUF-CMA

return $(\mu^\star, \cdot) \notin Q \wedge \text{Verify}(\text{vk}, \sigma^\star, \mu^\star) = 1$

$\underline{\text{Sign}(\mu)}$

$\sigma \leftarrow \text{Sign}(\text{sk}, \mu)$

$Q \leftarrow Q \cup \{(\mu, \sigma)\}$

return σ

$\underline{\text{BSign}(\mathcal{M})}$

$\mathcal{S} \leftarrow \text{Sign}(\text{sk}, \mathcal{M})$

for $0 \leq j < |\mathcal{M}|$ **do**

  $q_j \leftarrow (\mathcal{M}[j], \mathcal{S}[j])$

  $Q \leftarrow Q \cup \{q_j\}$

return $\mathcal{S}$

Figure 3: Existential Unforgeability under Chosen Message Attacks for Signatures (EUF-CMA) and Batch Signatures (BEUF-CMA).

KeyGen The key generation algorithm is a randomised algorithm that takes as input a security parameter 1^λ and outputs a pair (vk, sk), the *verification key* and *signing key*, respectively. We write $(\text{vk}, \text{sk}) \leftarrow \text{KeyGen}(1^\lambda)$.

Sign The signing algorithm takes as input a signing key sk, a message μ and outputs a signature σ. We write this as $\sigma \leftarrow \text{Sign}(\text{sk}, \mu)$. The signing algorithm may be randomised or deterministic. We may write $\sigma \leftarrow \text{Sign}(\text{sk}, \mu; r)$ to unearth the used randomness explicitly.

Verify The verification algorithm takes as input a verification key vk, a signature σ and a message μ and outputs a bit b, with $b = 1$ meaning the signature is valid and $b = 0$ meaning the signature is invalid. Verify is a deterministic algorithm. We write $b \leftarrow \text{Verify}(\text{vk}, \sigma, \mu)$.

We require that except with negligible probability over $(\text{vk}, \text{sk}) \leftarrow \text{KeyGen}(1^\lambda)$, it holds that $\text{Verify}(\text{vk}, \text{Sign}(\text{sk}, \mu), \mu) = 1$ for all μ.

We rely on the standard notion of existential unforgeability under chosen message attacks:

Definition 2.5 (EUF-CMA). We define

$$\text{Adv}^{\text{euf-cma}}_{\mathcal{A},\text{S}}(\lambda) \coloneqq \Pr[\text{EUF-CMA}^{\mathcal{A}}_{\text{S}}(\lambda) \Rightarrow 1]$$

for $\text{EUF-CMA}^{\mathcal{A}}_{\text{S}}(\lambda)$ as in Figure 3 and say a signature scheme S is EUF-CMA secure if no PPT/BQP adversary $\mathcal{A}$ has non-negligible advantage $\text{Adv}^{\text{euf-cma}}_{\mathcal{A},\text{S}}(\lambda)$.

Remark. *In our construction, the signature scheme takes as inputs and outputs* batches *of messages and signatures, respectively. We formally define batch signature schemes and their security (*BEUF-CMA*) in Section 3.*

2.3 Falcon Signature Scheme

Since our flagship demonstrator is the composition of our scheme with Falcon [20] (based on the GPV paradigm [11]), we give a stylised description in Figure 4, since this suffices for our purposes here. Let (TrapGen, SampD, SampPre) be PPT algorithms with the following syntax and properties [10, 11, 17]:

- $(\boldsymbol{A}, \mathrm{td}) \leftarrow \mathrm{TrapGen}(1^\eta, 1^\ell, q, \mathcal{R}, \beta)$ takes dimensions $\eta, \ell \in \mathbb{N}$, a modulus $q \in \mathbb{N}$, a ring $\mathcal{R}$, and a norm bound $\beta \in \mathbb{R}$. It generates a matrix $\boldsymbol{A} \in \mathcal{R}_q^{\eta \times \ell}$ and a trapdoor td. For any $n \in \mathrm{poly}(\lambda)$ and $\ell \geq \mathrm{lhl}(\mathcal{R}, \eta, q, \beta)$, the distribution of $\boldsymbol{A}$ is within $\mathrm{negl}(\lambda)$ statistical distance to the uniform distribution on $\mathcal{R}_q^{\eta \times \ell}$.
- $\boldsymbol{u} \leftarrow \mathrm{SampD}(1^\eta, 1^\ell, \mathcal{R}, \beta')$ with $\ell \geq \mathrm{lhl}(\mathcal{R}, \eta, q, \beta)$ outputs an element in $\boldsymbol{u} \in \mathcal{R}^\ell$ with norm bound $\beta' \geq \beta$. We have that $\boldsymbol{v} \coloneqq \boldsymbol{A} \cdot \boldsymbol{u} \bmod q$ is within $\mathrm{negl}(\lambda)$ statistical distance to the uniform distribution on $\mathcal{R}_q^\eta$.
- $\boldsymbol{u} \leftarrow \mathrm{SampPre}(\mathrm{td}, \boldsymbol{v}, \beta')$ with $\ell \geq \mathrm{lhl}(\mathcal{R}, \eta, q, \beta)$ takes a trapdoor td, a vector $\boldsymbol{v} \in \mathcal{R}_q^\eta$, and a norm bound $\beta' \geq \beta$. It samples $\boldsymbol{u} \in \mathcal{R}^\ell$ satisfying $\boldsymbol{A} \cdot \boldsymbol{u} \equiv \boldsymbol{v} \bmod q$ and $\|\boldsymbol{u}\| \leq \beta'$. Furthermore, $\boldsymbol{u}$ is within $\mathrm{negl}(\lambda)$ statistical distance to $\boldsymbol{u} \leftarrow \mathrm{SampD}(1^\eta, 1^\ell, \mathcal{R}, \beta')$ conditioned on $\boldsymbol{v} \equiv \boldsymbol{A} \cdot \boldsymbol{u} \bmod q$.

$\mathrm{KeyGen}(1^\lambda)$

$\boldsymbol{A}, \mathrm{td} \leftarrow \mathrm{TrapGen}(1^1, 1^2, q, \mathcal{R}, \beta)$

return $\mathrm{vk}_i = \boldsymbol{A}, \mathrm{sk}_i = \mathrm{td}$

$\mathrm{Sign}(\mu_j, \mathrm{sk}_i; r)$

$\boldsymbol{y}_j \leftarrow \mathrm{SampPre}(\mathrm{td}, H(\mu_j, r), \beta')$

return $\boldsymbol{y}$

$\mathrm{Verify}(\sigma_j, \mu_j, \mathrm{vk}_i)$

return $\left\|\boldsymbol{y}_j\right\| \overset{?}{\leq} \beta' \wedge H(\mu_j) \overset{?}{\equiv} \boldsymbol{A} \cdot \boldsymbol{y}_j$

Figure 4: Falcon signatures [11, 21]

Assumption 2.6. *The Falcon signature scheme is EUF-CMA secure. In particular, no (quantum) adversary exists to forge messages with cost $\ll 2^{128}$ for Falcon-512 and no such adversary exists with cost $\ll 2^{256}$ for Falcon-1024.*

Performance. Consider Falcon-512 which minimises the signature size among the NIST selected post-quantum signature algorithms. An optimised implementation beats RSA-2048 signing by roughly a factor of five [6]. Critically, however, this optimised implementation relies on constant-time double-precision floating point arithmetic. This is not completely out of reach, as demonstrated by constant-time Falcon implementations [19] on several different CPUs, working around several CPU instructions' behaviours. However, the long-term reliability of this approach is less certain than for bit or integer operations. That is, future instructions or optimisations might prevent the desired constant-time behaviour. Furthermore, many CPUs to date simply lack fast constant-time double-precision arithmetic [13].

On systems where no sufficiently constant-time floating point unit is available or where floating-point arithmetic is avoided for the reasons mentioned above, floating-point arithmetic can be emulated (in constant time) at a hefty – approximately 20x – overhead [19].

3 BATCH SIGNATURES

We formally define batch signatures.

Definition 3.1 (Batch Signature Scheme). A batch signature scheme S consists of three PPT algorithms (KeyGen, BSign, Verify) such that:

KeyGen The key generation algorithm is a randomised algorithm that takes as input a security parameter 1^λ and outputs a pair $(\mathrm{vk}, \mathrm{sk})$, the *verification key* and *signing key*, respectively. We write $(\mathrm{vk}, \mathrm{sk}) \leftarrow \mathrm{KeyGen}(1^\lambda)$.

BSign The batch signing algorithm takes as input a signing key sk, a list of messages $\mathcal{M} = \{\mu_i\}$ and outputs a list of signatures $\mathcal{S} = \{\mathrm{sig}_i\}$. We write this as $\mathcal{S} \leftarrow \mathrm{BSign}(\mathrm{sk}, \mathcal{M})$. The signing algorithm may be randomised or deterministic. We may write $\mathcal{S} \leftarrow \mathrm{BSign}(\mathrm{sk}, \mathcal{M}; r)$ to unearth the used randomness explicitly.

Verify The verification algorithm takes as input a verification key vk, a signature sig and a message μ and outputs a bit b, with $b = 1$ meaning the signature is valid and $b = 0$ meaning the signature is invalid. Verify is a deterministic algorithm. We write $b \leftarrow \mathrm{Verify}(\mathrm{vk}, \mathrm{sig}, \mu)$.

We require that except with negligible probability over $(\mathrm{vk}, \mathrm{sk}) \leftarrow \mathrm{KeyGen}(1^\lambda)$, for all $\mathcal{M} \coloneqq \{\mu_i\}$ and $\mathcal{S} \leftarrow \mathrm{BSign}(\mathrm{sk}, \{\mu_i\})$ it holds that $\forall\, \mathrm{sig}_i \in \mathcal{S} : \mathrm{Verify}(\mathrm{vk}, \mathrm{sig}_i, \mu_i) = 1$.

Definition 3.2 (EUF-CMA for Batch Signature Schemes). We define

$$\mathrm{Adv}^{\text{euf-cma}}_{\mathcal{A},\mathrm{S}}(\lambda) \coloneqq \Pr[\text{BEUF-CMA}^{\mathcal{A}}_{\mathrm{S}}(\lambda) \Rightarrow 1]$$

for $\text{BEUF-CMA}^{\mathcal{A}}_{\mathrm{S}}(\lambda)$ as in Figure 3 and say a batch-signature scheme S is EUF-CMA secure if no PPT/BQP adversary $\mathcal{A}$ has non-negligible advantage $\mathrm{Adv}^{\text{euf-cma}}_{\mathcal{A},\mathrm{S}}(\lambda)$.

The following proposition is immediate, by simply calling Sign for all $\mu_i \in \mathcal{M}$. We call this the *naïve construction.*

Proposition 3.3. *Every EUF-CMA secure signature scheme can be turned into a EUF-CMA secure batch signature scheme.*

We also define two privacy notions for batch signatures. These assert that no efficient adversary can distinguish whether signatures were signed in the same batch or not. A weak variant of privacy only guarantees that signatures from the same batch do not leak anything about a message for which no signature is made available.

Definition 3.4 ((Weak) Batch Privacy). We define

$$\mathrm{Adv}^{\text{batch-priv}}_{\mathcal{A},\mathrm{S}}(\lambda) \coloneqq \left|\Pr[\text{BATCH-PRIV}^{\mathcal{A}}_{\mathrm{S}}(\lambda) \Rightarrow 1] - 1/2\right|$$

and

$$\mathrm{Adv}^{\text{wbatch-priv}}_{\mathcal{A},\mathrm{S}}(\lambda) \coloneqq \left|\Pr[\text{wBATCH-PRIV}^{\mathcal{A}}_{\mathrm{S}}(\lambda) \Rightarrow 1] - 1/2\right|$$

for the games defined in Figure 5 and say a signature scheme S has (weak) batch privacy if no PPT/BQP adversary $\mathcal{A}$ has non-negligible advantage $\mathrm{Adv}^{\text{(w)batch-priv}}_{\mathcal{A},\mathrm{S}}(\lambda)$.

Our construction in Section 4 achieves wBATCH-PRIV but not BATCH-PRIV and thus establishes that there are schemes achieving the former but not the latter. Next, we establish that an adversary breaking wBATCH-PRIV can also break BATCH-PRIV.

Lemma 3.5. *Let $\mathcal{A}$ be an adversary against* wBATCH-PRIV *with* $\mathrm{Adv}^{\text{wbatch-priv}}_{\mathcal{A},\mathrm{S}}(\lambda)$. *Then there is an adversary $\mathcal{B}$ against* BATCH-PRIV *with advantage*

$$\mathrm{Adv}^{\text{batch-priv}}_{\mathcal{B},\mathrm{S}}(\lambda) \geq 1/2 \cdot \mathrm{Adv}^{\text{wbatch-priv}}_{\mathcal{A},\mathrm{S}}(\lambda)$$

$\text{BATCH-PRIV}^{\mathcal{A}}_{\text{S}}(\lambda)$	$\textsc{Sign}(\mathcal{M})$
$b \leftarrow\$ \{0,1\}$;	**if** $b = 0$ **then**
$b^\star \leftarrow \mathcal{A}^{\textsc{Sign}}(\text{vk})$;	**for** $\mu_i \in \mathcal{M}$ **do**
return $b^\star = b$	$\{\text{sig}_i\} \leftarrow \text{BSign}(\text{sk}, \{\mu_i\})$;
	$\mathcal{S} \leftarrow \{\text{sig}_i\}_{0 \le i < \lvert\mathcal{M}\rvert}$
	else
	$\mathcal{S} \leftarrow \text{BSign}(\text{sk}, \mathcal{M})$;
	return $\mathcal{S}$

$\text{wBATCH-PRIV}^{\mathcal{A}}_{\text{S}}(\lambda)$	$\textsc{Sign}(\mathcal{M}, i, \{\mu_0, \mu_1\})$
$b \leftarrow\$ \{0,1\}$;	**if** $i \ge \lvert\mathcal{M}\rvert \vee i < 0$ **then return** $\perp$
$b^\star \leftarrow \mathcal{A}^{\textsc{Sign}}(\text{vk})$;	$\mathcal{M}_i \leftarrow \mu_b$ / i-th message is μ_b
return $b^\star = b$	$\mathcal{S} \leftarrow \text{BSign}(\text{sk}, \mathcal{M})$;
	$\mathcal{S}_i \leftarrow \perp$ / delete i-th signature
	return $\mathcal{S}$

Figure 5: (Weak) Batch Privacy.

Proof. To construct the adversary $\mathcal{B}$ against BATCH-PRIV, we use the BATCH-PRIV signing oracle to simulate the call to BSign. For this, we sample a bit c to decide what set $\mathcal{M}$ to submit to BATCH-PRIV signing oracle. When the adversary outputs $c^\star = c$ we output $b^\star = 1$, otherwise we output $b^\star = 0$.

To bound $\text{Adv}^{\text{batch-priv}}_{\mathcal{B},\text{S}}(\lambda)$ note that if $b = 0$ the signatures returned by the BATCH-PRIV signing oracle are independent of μ_0 and μ_1 by construction and thus the advantage of $\mathcal{A}$ is zero. If $b = 1$ then our signing oracle faithfully emulates the wBATCH-PRIV signing oracle. Thus,

$$\begin{aligned}
\text{Adv}^{\text{batch-priv}}_{\mathcal{B},\text{S}}(\lambda) &= \left|\Pr[\text{BATCH-PRIV}^{\mathcal{B}}_{\text{S}}(\lambda) \Rightarrow 1] - 1/2\right| \\
&= 1/2 \cdot \left|\Pr[\text{wBATCH-PRIV}^{\mathcal{A}}_{\text{S}}(\lambda) \Rightarrow 1 \mid b = 0] - 1/2\right| \\
&\quad + 1/2 \cdot \left|\Pr[\text{wBATCH-PRIV}^{\mathcal{A}}_{\text{S}}(\lambda) \Rightarrow 1 \mid b = 1] - 1/2\right| \\
&= 0 + 1/2 \cdot \left|\Pr[\text{wBATCH-PRIV}^{\mathcal{A}}_{\text{S}}(\lambda) \Rightarrow 1 \mid b = 1] - 1/2\right| \\
&= 0 + 1/2 \cdot \text{Adv}^{\text{wbatch-priv}}_{\mathcal{A},\text{S}}(\lambda)
\end{aligned}$$

□

Finally, we note that BATCH-PRIV is achievable:

Proposition 3.6. *The naïve construction of batch signatures from the Falcon signature scheme is batch private.*

4 CONSTRUCTION

Our construction relies on a Merkle tree. When addressing nodes in a Merkle tree of height h with N leaves, we may label nodes and leaves in the tree by their position: $n_{i,k}$ is the i-th node at height k, counting from left to right and from bottom upwards (i.e. leaves are on height 0 and the root is on height h). We illustrate this in Figure 6.

Let $\text{S} = (\text{KeyGen}, \text{Sign}, \text{Verify})$ be a digital signature scheme as defined in Definition 2.4, let H be a tweakable hash function as defined in Definition 2.1. We define our batch signature scheme

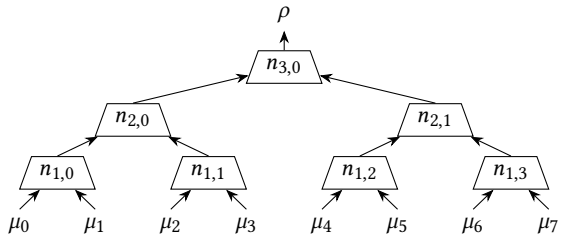


Figure 6: A Merkle tree and addressing scheme.

Algorithm 1 $\text{BSign}(\text{sk}, M = [\mu_0, \mu_1, \ldots, \mu_{N-1}])$ for $N = 2^n$

$T \leftarrow [\,]$
$\text{id} \leftarrow\$ \{0,1\}^\lambda$ ▷ Tree identifier
for $0 \le i < N$ **do** ▷ Generate N leaves
  $r_i \leftarrow\$ \{0,1\}^\lambda$
  $T[0,i] \leftarrow \text{H}(\text{id},\ 0 \mid i,\ r_i \mid \mu_i)$
end for
$h \leftarrow \log_2 N$
for $0 \le k < h$ **do**
  for $0 \le j < 2^{h-k-1}$ **do** ▷ Build tree
    $\text{left},\ \text{right} \leftarrow T[k, 2j],\ T[k, 2j+1]$
    ▷ id is public parameter, $(1 \mid (k+1) \mid j)$ is tweak
    $T[k+1, j] \leftarrow \text{H}(\text{id},\ 1 \mid (k+1) \mid j,\ \text{left} \mid \text{right})$
  end for
end for
$\text{root} \leftarrow T[h, 0]$
$\sigma \leftarrow \text{S.Sign}(\text{sk}, \text{id} \mid \text{root} \mid N)$
for $0 \le i < N$ **do** ▷ Generate user signature
  $\text{path}_i \leftarrow []$
  for $0 \le k < \log_2 N$ **do**
    $j \leftarrow \lfloor i/2^k \rfloor$
    if $j \bmod 2 = 0$ **then**
      $\text{path}_i[k] = T[k, j+1]$
    else
      $\text{path}_i[k] = T[k, j-1]$
    end if
  end for
  $\text{sig}_i \leftarrow (\text{id}, N, \sigma, i, r_i, \text{path}_i)$
end for
return $\{\text{sig}_0, \text{sig}_1, \ldots, \text{sig}_{N-1}\}$

$\text{BaS} = (\text{KeyGen}, \text{BSign}, \text{Verify})$ with $\text{KeyGen} := \text{S.KeyGen}$ and BSign and Verify as in Algorithms 1 and 2 respectively.

Remark. *For clarity we restrict our presentation to a fixed, power-of-two batch size N. To handle batches that do not satisfy this, we break down too long lists of messages into several batches of size at most N. To handle batches of size less than N we can either pad the tree by repeating leaves or use incomplete trees (see e.g. "L-trees" in [8]). Since this is standard in the literature, we omit the details here.*

Algorithm 2 $\mathrm{Verify}(\mathrm{vk}, \mu, \mathrm{sig} = (\mathrm{id}, N, \sigma, i, r, \mathrm{path}))$

1: $h \leftarrow \mathrm{H}(\mathrm{id}, 0 \mid i, r \mid \mu)$
2: $k \leftarrow 0$;
3: **for** $1 \leq k < \log_2 N$ **do** ▷ Construct root
4: $j \leftarrow \lfloor i/2^k \rfloor$
5: **if** $j \bmod 2 = 0$ **then**
6: $h \leftarrow \mathrm{H}(\mathrm{id}, 1 \mid k \mid j,\ h \mid \mathrm{path}[k])$
7: **else**
8: $h \leftarrow \mathrm{H}(\mathrm{id}, 1 \mid k \mid j,\ \mathrm{path}[k] \mid h)$
9: **end if**
10: **end for**
11: **return** $\mathrm{S.Verify}(\mathrm{vk}, \mathrm{id} \mid h \mid N)$

5 SECURITY PROOF

Theorem 5.1. *Let* $\mathrm{S} = (\mathrm{KeyGen}, \mathrm{Sign}, \mathrm{Verify})$ *be a digital signature scheme as in Definition 2.4,* H *be a tweakable hash function as in Definition 2.1. Let* $\mathrm{BaS} = (\mathrm{KeyGen}, \mathrm{BSign}, \mathrm{Verify})$ *be the batch signature scheme with* BSign *and* Verify *defined in Algorithms 1 and 2, respectively. If there exists a (classical or quantum) adversary* $\mathcal{A}$ *that breaks* BEUF-CMA *of* BaS *(see Definition 2.5) with* q_s *queries to the signing oracles, then it holds that*

$$\mathrm{Adv}^{\text{BEUF-CMA}}_{\mathrm{BaS},\mathcal{A}}(\lambda) \leq \mathrm{Adv}^{\text{EUF-CMA}}_{\mathrm{S},\mathcal{B}}(\lambda) + q_s \cdot \mathrm{Adv}^{\text{SM-TCR}}_{\mathrm{H},C}(N, \lambda),$$

where $\mathcal{B}$ *makes* q_s *queries to its signing oracle.*

The idea of the proof is as follows. Assume adversary $\mathcal{A}$ forges a signature of BaS on some message. By definition of unforgeability this message has not been queried to the signing oracle. This enables us to distinguish two cases. Either the root (that is included in the signature) was part of a query response or not. If it has not been part of a response, we can extract a forgery for S. In the other case, there must be a collision somewhere in the hash tree which we can use to solve SM-TCR.

Proof. Let $\mathcal{A}$ be an adversary against BEUF-CMA of BaS. More concretely, assume that the adversary $\mathcal{A}$ gets the verification key vk, has access to a signing oracle, and outputs a signature $\mathrm{sig}^\star := (\mathrm{id}^\star, N^\star, \sigma^\star, i^\star, r^\star, \mathrm{path}^\star)$ for a message $\mu^\star$ that has not been queried before, i.e. $(\mu^\star, \cdot) \notin Q$. We proceed via a series of game hops. Throughout, we let Adv_i denote the advantage of $\mathcal{A}$ in Game_i. Also, we implicitly define root_i (and $\mathrm{root}^\star$) by sig_i (and $\mathrm{sig}^\star$) since they can be computed deterministically: it is the value that comes out of the authentication path evaluation in Verify, cf. up to Line 10 of Algorithm 2.

Game_0*:* BEUF-CMA *against* BaS. So

$$\mathrm{Adv}_0 = \mathrm{Adv}^{\text{BEUF-CMA}}_{\mathrm{BaS},\mathcal{A}}.$$

Game_1*: Excluding* S*-forgeries.* Game_1 is identical to Game_0 except that it aborts if $(\cdot, (\mathrm{id}^\star, \mathrm{root}^\star, N^\star, \cdots)) \notin Q$. Here, we use that $\mathrm{sig}^\star$ and sig_i implicitly define $\mathrm{root}^\star$ and root_i. In this case, $((\mathrm{id}^\star, \mathrm{root}^\star, N^\star), \sigma^\star)$ is an EUF-CMA forgery for S found by $\mathcal{A}$. In particular, we have that

$$\mathrm{S.Verify}(\mathrm{vk}, \sigma^\star, (\mathrm{id}^\star, \mathrm{root}^\star, N^\star)) = 1$$

if $\mathrm{Verify}(\mathrm{vk}, \mu^\star, \mathrm{sig}^\star) = 1$.

To bound the distance between both games, we construct an algorithm $\mathcal{B}$ that breaks EUF-CMA of S using $\mathcal{A}$. Given vk, $\mathcal{B}$ runs $\mathcal{A}(\mathrm{vk})$. It implements the signing oracle for $\mathcal{A}$ following Algorithm 1 with the only difference that it asks its own S-signing oracle to sign $(\mathrm{id}, \mathrm{root}, N)$ in line 15. Hence, $\mathcal{B}$ makes the same number of signing queries $\mathcal{A}$ makes. Consider the event that Game_1 aborts but Game_0 does not. We can bound this probability by $\mathcal{B}$'s advantage $\mathrm{Adv}^{\text{EUF-CMA}}_{\mathrm{S},\mathcal{B}}$ with q_s many queries. So

$$\mathrm{Adv}_0 - \mathrm{Adv}_1 \leq \mathrm{Adv}^{\text{EUF-CMA}}_{\mathrm{S},\mathcal{B}}.$$

Bounding Adv_1*: Forgery in the tree.* We now bound the probability that an adversary succeeds in Game_1. Note that if we did not abort in Game_1, we have that the id and root of the forgery $(\mathrm{id}^\star, \mathrm{root}^\star, N^\star)$ are identical to those of a tree that has been created during a signing query. Let this query be the j-th query $\mathcal{M}_j = \{\mu_0, \ldots, \mu_{N^\star-1}\}$ with response $\mathcal{S}_j$. Hence $(\mathrm{id}^\star, \mathrm{root}^\star, N^\star) = (\mathrm{id}_k, \mathrm{root}_k, N_k) \forall\ \mathrm{sig}_k \in \mathcal{S}_j$. Here, again, we implicitly define $\mathrm{root}^\star$ and root_k by $\mathrm{sig}^\star$ and sig_k. Also, given the fact that $\mu^\star, \mathrm{sig}^\star = (\mathrm{id}^\star, \sigma^\star, N^\star, i^\star, r_i^\star, \mathrm{path}_i^\star)$ is a forgery, by definition of BEUF-CMA, we must have that $\mu^\star \neq \mu_{i^\star}$. Running $\mathrm{Verify}(\mathrm{vk}, \mu^\star, \mathrm{sig}^\star)$ and $\mathrm{Verify}(\mathrm{vk}, \mu_{i^\star}, \mathrm{sig}_{i^\star})$ we note that this computes the same branch in two hash trees of same height and with identical roots but differing starting values. By the pigeonhole principle, this implies that there must be a collision in these paths which can be extracted.

We use the above observation to construct an adversary C against the SM-TCR-security of H. At the beginning of the game, C guesses which signing query j the collision will occur in. To answer the j-th signing query, instead of sampling id (Line 2), C builds the tree using calls to its $\mathrm{H}(P, \cdot, \cdot)$ oracle (where P is chosen by the SM-TCR challenger, see Figure 1). After finishing the tree, C requests P from the challenger before Line 15 and finishes Algorithm 1. Later, when the adversary $\mathcal{A}$ outputs a forgery $\mathrm{sig}^\star$ on $\mu^\star$, C extracts the collision using Verify as outlined above. The algorithm submits the colliding value from the forgery as the solution in the $\text{SM-TCR}^{\mathcal{A}}_{\mathrm{H}}(\lambda)$ game.

We can bound the probability that $\mathcal{A}$ succeeds in Game_1 by C's advantage $\mathrm{Adv}^{\text{SM-TCR}}_{\mathrm{H},C}(N, \lambda)$ and the probability of C guessing the right query j. So

$$\mathrm{Adv}_1 \leq q_s \cdot \mathrm{Adv}^{\text{SM-TCR}}_{\mathrm{H},C}(N, \lambda).$$

Combining both bounds confirms the claimed statement. □

Remark. *We note that our proof is not tight due to the factor* q_s *incurred from guessing the right query to play the* SM-TCR *game with. It is plausible that this factor of* q_s *can be removed by a more careful analysis of the required* SM-TCR *property. More precisely, we use a different public parameter* id *for each tree. For a good tweakable hash function, a query under public parameter* id *should not leak any information about the outcome using a different parameter* $\mathrm{id}' \neq \mathrm{id}$*. Hence, an adversary should intuitively not gain any advantage from targeting multiple instances of* H *at the same time, as long as they use different public parameters as we do. We leave an analysis of this property for follow-up work.*

Theorem 5.2. *Let* $\mathrm{S} = (\mathrm{KeyGen}, \mathrm{Sign}, \mathrm{Verify})$ *be a digital signature scheme as in Definition 2.4,* H *be a tweakable hash function as in Definition 2.1. Let* $\mathrm{BaS} = (\mathrm{KeyGen}, \mathrm{BSign}, \mathrm{Verify})$ *be the batch*

signature scheme with BSign *and* Verify *defined in Algorithms 1 and 2, respectively. If there exists a (classical or quantum) adversary* $\mathcal{A}$ *that breaks* wBATCH-PRIV *of* BaS *(see Definition 3.4), then it holds that*

$$\mathrm{Adv}^{\text{wBATCH-PRIV}}_{\text{BaS},\mathcal{A}} \leq 2 \cdot \mathrm{Adv}^{\text{OT-PRF}}_{\text{F},\mathcal{B}},$$

for $\mathrm{F}(k, (\mathrm{id}, i, x)) \coloneqq \mathrm{H}(\mathrm{id},\ 0 \mid i,\ k \mid x)$.

Proof. On receipt of $\mathcal{M}$, we run the signing oracle as usual but call our RoR oracle with input (id, i, μ_b) in Line 5 of Algorithm 1. If the oracle returns random outputs (which happens with probability 1/2), the advantage of $\mathcal{A}$ is zero. Otherwise, if $\mathcal{A}$ returns the correct answer, this allows to distinguish F from random, which is bounded by $\mathrm{Adv}^{\text{OT-PRF}}_{\text{F},\mathcal{B}}$. □

6 APPLICATIONS

Batch signatures reduce the computational cost of signing by replacing one signature per message with fewer than two hashes per message and one signature per batch. They can thus be used to increase the throughput attainable at a given amount of computational power. In some applications, the amount of data that needs to be sent can be reduced in addition; namely, if a given entity (is aware that it) receives multiple signatures from the same batch. In this case, sending the signed root multiple times is redundant and we can asymptotically reduce the amount of received information to a few hashes per message.

In the following, we describe how to use batch signatures to reduce computational costs and communication costs in two operations: certificate generation (typically in an HSM) and transcript signature (typically in TLS). We then discuss two scenarios in which this can be particularly beneficial.

6.1 Computational Costs

As noted, we will consider two scenarios: HSMs that generate a large set of short-lived certificates, and server-side signing for TLS.

6.1.1 Hardware Security Modules. Generating a large set of certificates, for example when they are renewed for a group of entities, implies in general computing a signature per certificate, and thus can represent a significant computational burden. Moreover, those signatures are in general computed on HSMs, which are significantly slower than traditional CPUs. For example, where a modern commodity CPU can sign tens of thousands of messages per second with ECDSA [6], some widely used enterprise-grade cloud HSMs can just sign a few hundred messages per second [7]. Thus, short-lived certificates [26], for example, can put significant stress on HSMs, especially when certificate renewal concerns a large set of devices or containers (e.g. an Envoy mesh network with 10K to 100K containers [15]).

In such a setting, deploying a batch signing approach is quite straightforward. Interfacing with the HSM, an agent waits for a signal from the HSM indicating that it is ready to start a new signature. While waiting, the agent gathers incoming certificate requests that are hashed and a fixed size (e.g. of 32 leaves) Merkle tree is built. When the HSM signals "ready", the agent completes the Merkle tree with the appropriate number of zeroed leaves and sends it for signing to the HSM. In the opposite case, when the tree is full before the HSM is ready, the agent starts a new tree resulting in a queue of trees to be signed.

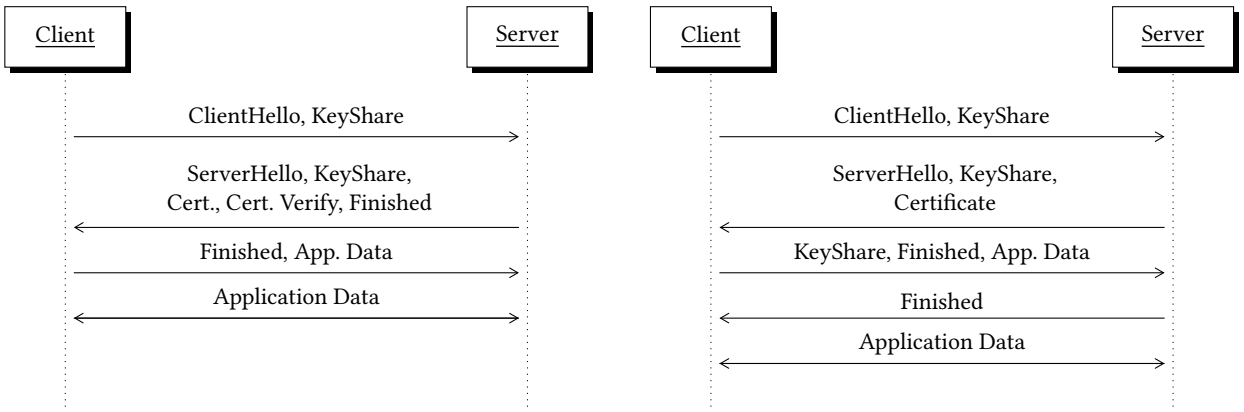
When the signature of the Merkle tree root is returned by the HSM, it is added to each certificate together with the sibling path associated to that request (see Line 26, Algorithm 1), resulting in the final certificate. Of course, this assumes that the certificate requester is able to verify batch signatures. Moreover, the CA generating the certificates using batch signatures needs to be updated accordingly. Naturally this increases the throughput at which certificates can be signed by roughly a factor equal to the batch size, as we need only one signature per 32 certificates. For example, in the cloud HSM setting we would pass for ECDSA from hundreds to hundreds of thousands of signatures per second.

We expect that this effect will be more pronounced in a post-quantum setting where signing operations are much more expensive (as mentioned above). However, hard performance figures for post-quantum signatures on HSMs are not yet available, so we cannot estimate the likely throughput.

6.1.2 Transport Layer Security. TLS, being one of the most popular and commonly used cryptographic protocols today, will also suffer substantial impact from the transition to post-quantum cryptography. Indeed, many of the recent benchmarks (e.g. [23]) show a significant performance penalty, especially on the computational and communication costs associated to signatures (in general done server-side), and the gap becomes much more apparent when considering packet loss [18]. This performance degradation in PQ TLS is incurred due to larger signature sizes and slower signing speeds. KEM sizes and performances while also worse are much closer to ECC in comparison. As a result, to circumvent the use of (PQ) signatures for authentication in TLS KEMTLS [22] was proposed which replaces static server authentication with a static KEM, so that only the involved KEM public keys need to be signed rather than the transcript. The results reported in [22] show a reduction in the bandwidth required for the client and server communications, as well as reducing the computational costs on the server's CPU.

However, despite the performance virtues of KEMTLS, it requires a number of significant infrastructure changes in order for it to fully reach fruition. Specifically, in order for KEMTLS to be used in practice, it will rely on changes to (i) include KEM public keys into a public-key infrastructure (PKI) and (ii) TLS implementations to operate with different state machines on both client and server sides. These points inhibit the design of a KEMTLS standard and its uptake compared with "plain" PQ TLS. We illustrate the messages exchanged in TLS 1.3 and in KEMTLS in Figures 7a and 7b.

A less invasive proposal would be to use batch signing for server-side computations. This approach goes back to [2] and is explored in this work. As in Algorithm 1, a server can amortise its signature computation costs by adding each incoming client to a Merkle tree, building the tree, and returning the signed root to each client, in addition to some auxiliary information. This then reduces the number of "inner" signature computations required (by a factor equal to the batch size), which is the major contributing factor for the high-throughput improvements shown in Table 2. This significantly improves the performance of PQ TLS without any major changes to the PKI.



(a) High-level overview of TLS 1.3 1-RTT handshake. Certificate Verify contains the signature.

(b) High-level overview of KEMTLS handshake.

Figure 7: TLS 1.3 and KEMTLS.

These improvements come at a few extra minor costs in TLS; these being a slight increase in the overall batch signature sizes (adding between 82 and 98 bytes, as shown in Table 1) compared to the non-batch version of the signature as well as a slight increase in latency (shown in Table 2) and computation for the client, which is due to the hash function calls when building the Merkle tree.

6.2 Bandwidth Reduction

As noted previously, in [4] a reduction in certificate sizes is proposed by a new type of CA which would exclusively sign a new type of batch oriented certificates. Such a CA would only be used together with a Certificate Transparency authority which changes the usual required flows for authentication. The benefits obtained in certificate size and verification/signature costs are significant. It also implies that the main criterion for being on a same batch are: being signed by the same CA, and being signed roughly at the same time.

In this section we consider what benefits we can obtain in a simpler setting, supposing just that a usual CA and its users can use multiple signing algorithms, one of them being a batch signature. In this case multiple certificates will be in the same batch when a user considers this beneficial. In this section we show how this can be beneficial (besides reducing the load of the CA). There are two flows in which we can expect bandwidth reduction for the entities with certificates belonging to a same signed batch.

6.2.1 First Flow: HSMs. The first flow is from the signing authority (again, typically, an HSM) to the entities corresponding to a same batch of signed certificates. Indeed, all the issued certificates have a signature that contains the same Merkle tree root signature but a different sibling path. Depending on the exact situation, it is then possible for the HSM to broadcast the root signature or the entire Merkle Tree and remove any information from the certificates that are sent back that can be reconstructed from the broadcasts.

6.2.2 Second Flow: TLS. The second flow is from the entities holding the certificates signed on the same batch to the entities receiving that certificate. This typically happens in TLS when the server sends their certificate to the client. In that setting, if the client is going to interact over the lifetime of the certificates with multiple servers from the same batch group, it can inform that the tree root signature is already known (for example in TLS with a variation to the TLS Cached Information Extension that allows to notify a server that some information is already known). The certificate can just contain the sibling path as the signature, leading to a bandwidth usage reduction (between 1KB and 3KB if using Falcon or Dilithium, and up to 30 KB if using SPHINCS$^+$).

6.3 Use-Cases

We consider here two more fleshed-out examples of situations where forming batches is natural and can produce significant gains.

6.3.1 Fleet of Load Balancers. To reduce downtime (e.g. because of a Denial-of-Service attack, server maintenance, etc.) and to improve scalability (e.g. to efficiently (geographic) distribute requests under heavy network traffic) load balancers are essential for most of today's web applications. At the same time, they often act as a TLS termination proxy, and as such decrypt, encrypt, and sign the incoming and outgoing HTTPS traffic to offload cryptographic computations from back-end server(s), see Figure 8. As a consequence, when cryptographic computations become more costly due to the transition to PQC, load balancers themselves may become a throughput bottle-neck. Implementing batch signatures could significantly increase their workload capacity.

In this use-case, we consider a fleet of load balancers belonging to a large cloud provider that renew their certificates periodically (say weekly). They form a natural group on the certificate renewal process and making a batch certificate signature request can significantly reduce the computational load on the associated HSMs, as described in Section 6.1.1, and the communications, as described at the beginning of this section. Most importantly, in such a setting, the fleet of load balancers would send full certificates only once per week and per user, and in all remaining connections load balancers will reduce the size of the certificates by 1 to 30KB. From a user perspective, if major cloud providers use this system, a user will only

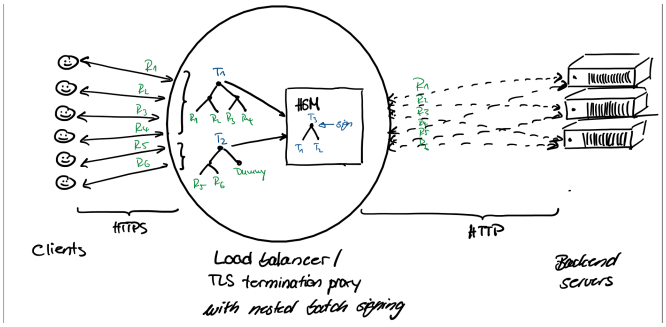


Figure 8: Client–Load Balancer–Backend server connection with nested batch signatures.

have to download full sized certificates a few times per week (for connections served through cloud provider load balancers).

Instead of considering the bandwidth reduction, this can be seen as a usability question for schemes with large signatures such as SPHINCS$^+$. This algorithm relies on mild assumptions, and thus is a good candidate for CAs. Unfortunately, increasing certificate sizes to tens of kilobytes can be considered too steep of a requirement. Batch signing can solve this in the load balancer setting (and similar situations) as full certificates are only sent exceptionally.

6.3.2 mTLS Mesh. The second scenario is a large-scale container mesh network that renews the *mutual TLS* (mTLS) certificates in its Envoy instances. In this scenario, by considering all the Envoy instances as a single batch, there is no need to ever transmit the root signature between peers, as every peer has the root signature in its own certificate (see Figure 9). Of course, during the generation of the certificates we also benefit from the already described computational and communication reduction for the HSM which can be quite significant for mesh networks with thousands or tens of thousands of containers. The storage requirements for the whole set of mesh certificates (in cached key servers that are used throughout mesh networks) is also greatly reduced, e.g. for a ten thousand container mesh using SPHINCS$^+$ signatures it would be reduced from hundreds of megabytes to hundreds of kilobytes.

7 IMPLEMENTATION & BENCHMARK SETUP

As a demonstrator for batch signatures we implement them for one of the most important use cases, TLS, and set up a load balancer benchmark. Here we provide details on our implementation and on the benchmark setup.

7.1 Implementation details

There are a number of components that make up the overall implementation for our proof-of-concept batch signing experiments in TLS. These allow us to estimate realistic conditions for secure network communications and thus accurately learn how effective our construction is under such conditions. These components are:

- `bsign_engine`: the Rust implementation of batch signing, which includes Merkle tree building and batch signing functionalities. We also have a benchmarking wrapper to produce results.

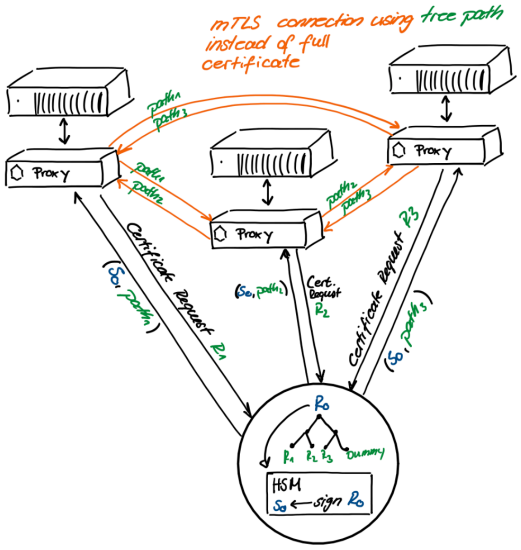


Figure 9: Mesh network with envoys using nested signatures within mTLS

- `openssl`: includes a fork of OpenSSL 1.1.1, using support from the liboqs library [25] and `batchtls_engine`.
- `tcpserver`: a TCP server, using `openssl` and `bsign_engine`.

The `bsign_engine` is the core component of our implementation. It is responsible for asynchronously gathering client requests into Merkle trees and signing the associated roots with a fixed number of signing threads.

We introduce two implementation-specific parameters, described below, which are used to tweak the performance of our implementation. This fine-tuning allows us to increase performance based on a number of factors, most importantly what signature algorithm is used, the server specifications, and the resulting latency.

Merkle Tree Size: This quantifies the amount of messages handled per batch signing transaction, which thus affects the latency and throughput of the server. Having a larger number of clients (messages as tree leaves) will reduce the average computation time per client, but at the same time will increase the size of the final signature (due to the longer path) and increase median and worst-case latency. We provide batch signature sizes in Table 1.

We parameterise the Merkle tree using the number of leaves in the tree and commonly fix this to a power-of-two for efficiency reasons, e.g. `MT_size` $= 2^5$ which produces a (balanced) tree of height 5. However, this parameter may need to be adapted to fit other hardware or performance constraints.

Signing Threads: These are responsible for taking the ownership of a Merkle tree and for signing its root. When a signing thread is currently signing a Merkle tree root, it cannot handle the next tree. Thus, if the `bsign_engine` has a single signing thread, the worst observed latency will correspond to twice the time needed for a signature. By adding additional signing threads, the latency will get closer to the time needed for a signature. However, having too many signing threads may saturate the scheduling and may

slow down the engine, or become sub-optimal at the very least. Inversely, if we reduce the number of signing threads we also limit the number of cores used for signatures. In Table 2, we choose the number of signing threads/cores as the smallest number that allows to increase the number of handshakes significantly (with respect to the plain implementation) with a low latency increase.

We use the open-source Rust library, `ring` [24], for our ECDSA implementation and the open-source C library, `liboqs` [25], for our implementations of post-quantum signatures. One reason for using this library in particular is because it has integration into OpenSSL 1.1.1[2] which we patch for our experiments to work with the `bsign_engine`. The patch modifies the state machine of OpenSSL to inject batch signing under certain conditions, but in such a way that it remains functional with classic TLS when these conditions are not met. More specifically, our patch adds a new structure – `BATCHTLS_CTX` – to the OpenSSL context – `SSL_CTX` – to track the context of the engine. We use environment variables for setting the parameters in `bsign_engine` for simplicity, as opposed to adding new APIs on top of `SSL_CTX`.

These aforementioned components are the ingredients that make up our overall implementation for batch signing in TLS, once combined with a TCP server. Implementing it this way gives us a realistic environment in order to run our experiments; providing conditions that we would expect to see in the real world.

7.2 Benchmark Setup

We demonstrate the application of batch signatures for the TLS use case with the results for these shown in Table 2. For this setup we took between one and four client machines and a single server machine. Each of them uses a Google Cloud C2 instance which has an Intel 3.9 GHz Cascade Lake processor. The specific instance type we used was the `c2-standard-30`, which offers 30 (virtual) CPU threads, 120 GB memory, and a (max) egress bandwidth of 32 Gbps. We disable hyper-threading in order to have more stable tests.

The results in Table 2 labelled as 'plain' are taken from a multi-threaded implementation (with a pool of as many threads as computer cores and with select/poll handling). For the batch signing results, we use a limit on the maximum size of the Merkle tree. The amount of cores used for signatures is estimated for the plain approach out of the computational cost of one signature, and fixed (by reserving cores explicitly) for the batch signing approach.

The results in Table 2 provide both handshakes per second (essentially, throughput) and latency (for various percentiles). We kept Merkle tree sizes to either 16 or 32, since larger sizes incurred much higher latency costs. Indeed, for large trees the latency added is unrealistic, despite throughput being increased.

The performance results are presented in Table 2 and discussed on the associated caption. Results for SPHINCS$^+$ and a second benchmark in the HSM setting will be presented in a full version of this paper.

[2] see https://openquantumsafe.org/applications/tls.html.

REFERENCES

[1] Kahraman Akdemir, Martin Dixon, Patrick Fay Wajdi Feghali, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. *Breakthrough AES Performance with Intel® AES New Instructions.* Whitepaper. Intel.

[2] David Benjamin. 2020. *Batch Signing for TLS.* Internet-Draft draft-ietf-tls-batch-signing-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-tls-batch-signing/00/ Work in Progress.

[3] David Benjamin. 2022. private communication. (2022).

[4] David Benjamin, D O'Brien, and Bas Westerbaan. 2023. *Merkle Tree Certificates for TLS.* Internet-Draft draft-davidben-tls-merkle-tree-certs-00. Internet Engineering Task Force. https://www.ietf.org/id/draft-davidben-tls-merkle-tree-certs-00.html Work in Progress.

[5] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. 2019. The SPHINCS$^+$ Signature Framework. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2129–2146. https://doi.org/10.1145/3319535.3363229

[6] Daniel J Bernstein and Tanja Lange. 2018. SUPERCOP: System for unified performance evaluation related to cryptographic operations and primitives. https://bench.cr.yp.to/supercop.html. (2018).

[7] AWS CloudHSM. 2023. FAQS - Performance and capacity. https://aws.amazon.com/cloudhsm/faqs/#Performance_and_capacity. (2023). [Online; accessed 21-February-2023].

[8] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. 2008. Digital Signatures Out of Second-Preimage Resistant Hash Functions. In *Post-quantum cryptography, second international workshop, PQCRYPTO 2008*, Johannes Buchmann and Jintai Ding (Eds.). Springer, Heidelberg, 109–123. https://doi.org/10.1007/978-3-540-88403-3_8

[9] Andrew Fregly, Joseph Harvey, Burton S. Kaliski Jr., and Swapneel Sheth. 2022. Merkle Tree Ladder Mode: Reducing the Size Impact of NIST PQC Signature Algorithms in Practice. Cryptology ePrint Archive, Report 2022/1730. (2022). https://eprint.iacr.org/2022/1730.

[10] Nicholas Genise and Daniele Micciancio. 2018. Faster Gaussian Sampling for Trapdoor Lattices with Arbitrary Modulus. In *EUROCRYPT 2018, Part I (LNCS)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.), Vol. 10820. Springer, Heidelberg, 174–203. https://doi.org/10.1007/978-3-319-78381-9_7

[11] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. 2008. Trapdoors for hard lattices and new cryptographic constructions. In *40th ACM STOC*, Richard E. Ladner and Cynthia Dwork (Eds.). ACM Press, 197–206. https://doi.org/10.1145/1374376.1374407

[12] Shay Gueron and Vlad Krasnov. 2012. Parallelizing message schedules to accelerate the computations of hash functions. *Journal of Cryptographic Engineering* 2, 4 (2012), 241–253.

[13] James Howe and Bas Westerbaan. 2022. Benchmarking and Analysing the NIST PQC Finalist Lattice-Based Signature Schemes on the ARM Cortex M7. Cryptology ePrint Archive, Report 2022/405. (2022). https://eprint.iacr.org/2022/405.

[14] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. 2022. *SPHINCS+.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[15] Matt Klein. 2017. Lyft's Envoy: Experiences Operating a Large Service Mesh. SREcon17 Americas. (2017). available at https://www.usenix.org/sites/default/files/conference/protected-files/srecon17americas_slides_klein.pdf.

[16] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. 2022. *CRYSTALS-DILITHIUM.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[17] Daniele Micciancio and Chris Peikert. 2012. Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller. In *EUROCRYPT 2012 (LNCS)*, David Pointcheval and Thomas Johansson (Eds.), Vol. 7237. Springer, Heidelberg, 700–718. https://doi.org/10.1007/978-3-642-29011-4_41

[18] Christian Paquin, Douglas Stebila, and Goutam Tamvada. 2020. Benchmarking Post-quantum Cryptography in TLS. In *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, Jintai Ding and Jean-Pierre Tillich (Eds.). Springer, Heidelberg, 72–91. https://doi.org/10.1007/978-3-030-44223-1_5

[19] Thomas Pornin. 2019. New Efficient, Constant-Time Implementations of Falcon. Cryptology ePrint Archive, Report 2019/893. (2019). https://eprint.iacr.org/2019/893.

[20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2020. *FALCON.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[21] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2022. *FALCON.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[22] Peter Schwabe, Douglas Stebila, and Thom Wiggers. 2020. Post-Quantum TLS Without Handshake Signatures. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1461–1480. https://doi.org/10.1145/3372297.3423350

[23] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. 2020. Post-Quantum Authentication in TLS 1.3: A Performance Study. In *NDSS 2020*. The Internet Society.

[24] Brian Smith. 2023. Crate ring. https://github.com/briansmith/ring. (2023). [Online; accessed 24-February-2023].

[25] Douglas Stebila and Michele Mosca. 2016. Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project. In *SAC 2016 (LNCS)*, Roberto Avanzi and Howard M. Heys (Eds.), Vol. 10532. Springer, Heidelberg, 14–37. https://doi.org/10.1007/978-3-319-69453-5_2

[26] Emin Topalovic, Brennan Saeta, Lin-Shung Huang, Collin Jackson, and Dan Boneh. 2012. Towards Short-Lived Certificates. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP)*.

[27] Shalanda D. Young. 2022. National Security Memo on Promoting United States Leadership in Quantum Computing While Mitigating Risks to Vulnerable Cryptographic Systems (NSM-10). Executive Office of the President, Office of Management and Budget, Washington, DC, USA. (2022). https://www.whitehouse.gov/wp-content/uploads/2022/11/M-23-02-M-Memo-on-Migrating-to-Post-Quantum-Cryptography.pdf