Efficient Aggregatable BLS Signatures with Chaum-Pedersen Proofs

Jeff Burdges, Oana Ciobotaru, Syed Lavasani, and Alistair Stewart

Web3 Foundation

Abstract. BLS signatures have fast aggregated signature verification but slow individual signature verification. We propose a three part optimisation that dramatically reduces CPU time in large distributed systems using BLS signatures: First, public keys should be given in both source groups $\mathbb{G}_2$ and $\mathbb{G}_1$, with a proof-of-possession [1] check for correctness. Second, aggregated BLS signatures should carry their particular aggregate public key in $\mathbb{G}_2$, so that verifiers can do both hash-to-curve and aggregate public key checks in $\mathbb{G}_1$. Third, individual non-aggregated BLS signatures should carry short Chaum-Pedersen DLEQ proofs of correctness [2], so that verifying individual signatures no longer requires pairings, which makes their verification much faster. We prove security for these optimisations. The proposed scheme is implemented and benchmarked to compare with classical BLS scheme.

1 Introduction

BLS signatures introduced by Boneh, Lynn and Shacham [3] are a popular aggregatable signature scheme. It has short aggregate signatures that can be efficiently verified. However due to the heavier cryptography involved, i.e., pairings on elliptic curves, individual signatures are much slower to verify compared to e.g. Schnorr or ECDSA signatures.

Aggregation of BLS signatures [3] simplifies some distributed systems, usually by being exportable proofs of byzantine agreement. After agreement and aggregation occurs, the aggregate signature saves foreign aggregate verifiers both compute and bandwidth over checking numerous slow signatures, especially at the scale of Ethereum's hundreds of thousands of signers. Because individual signature verification is very slow, either most nodes incur the high verification costs for every other node's signatures, or, else, system designers choose more centralised gossip flavours which may harm liveness.

As a rule, today one always does BLS signatures on curves with type III pairings, i.e. the pairing takes as input elements of two different groups $\mathbb{G}_1$ and $\mathbb{G}_2$. Typically, such as with the standard BLS12-381 curve, these have very different performance characteristics with $\mathbb{G}_2$ having much slower arithmetic and hash-to-curve, due to being defined over an extension field. In fact, recent progress against the discrete log problem in extension fields [4] could precipitate adopting pairings with a higher

embedding degree, which would slow $\mathbb{G}_2$ further. We want the parts of our protocol that need to be efficient to use all or mostly $\mathbb{G}_1$ operations.

In this work we propose individual BLS signatures carry Chaum-Pedersen [2] DLEQ proofs of their correctness, done in $\mathbb{G}_1$, so that verifying them no longer requires pairings or $\mathbb{G}_2$ operations. Moreover, a classical BLS signature places the public key and signature on opposite groups of the pairing (for type III pairings), but the choice of which of the groups is used for the keys or signatures creates trade offs. We mitigate these trade offs by proposing the following:

Protocol Sketch: First, we provide the public key on both source groups-like $pk = (sk \cdot g_1, sk \cdot g_2)$. We enforce this public key structure during the public key validation phase present in many aggregate BLS protocols in order to protect against rogue-key attacks [1] by adding proof-of-possession check [1] via a pairing check.
Second, we create BLS signatures $\sigma = sk \cdot H(m)$ using the much faster $\mathbb{G}_1$ hash-to-curve $H(m)$, but also provide a Chaum-Pedersen proof π_{sig} that $\log_{H(m)}(\sigma) = \log_{g_1}(pk_1)$. Verifying the correctness of σ is thus reduced to verifying the correctness of π_{sig}, which avoids any expensive pairing operations.

Third, an aggregator node provides both the aggregate BLS signature, as well as aggregate signer keys $apk_2 \in \mathbb{G}_2$, by summing the individual signatures and second source group public keys. At this point, aggregate verifiers only need to compute apk_1 as the sum of first source group public keys and check that apk_2 was correctly computed with only two additional scalar multiplications in $\mathbb{G}_1$. In the end, aggregate verifiers save time despite these multiplications because their hash-to-curve runs on $\mathbb{G}_1$, far faster than on $\mathbb{G}_2$. Even when using alternative methods for computing apk_1 that involve custom SNARKs such as in [5], the aggregate verifiers still require operations only in the smaller and faster source group.
We remark that a natural variation on our protocol exists in which one finds an even faster non-pairing curve $\mathcal{S}$ having the same group order as $\mathbb{G}_1$, publishes triplet public keys like $pk = (sk \cdot g_1, sk \cdot g_2, sk \cdot S)$, in which S generates $\mathcal{S}$, and then DLEQ proofs use pk_3 for performance. We do not discuss this variant because it falls under our security arguments without additional mathematical complexities, aside from choosing curve parameters for $\mathcal{S}$.

In addition to the above proposed protocol which we present in detail in Section 2.2, we include a formal definition of signature aggregation in Section 2.1 and we also provide a security proof for our instantiation in Section 2.2. In Section 3 we present the comparison of the efficiency of the proposed scheme with classic BLS scheme.

Related Work: Alternative constructions for BLS signatures as well as other defence mechanisms against rogue key attacks [1], exist and we

briefly review both below. First, aggregation of BLS signatures for different messages have been studied before (e.g., [6]). In this case, rouge-key attacks are not a threat anymore (and, hence, PoPs are not a necessity anymore), but signature verification is computationally more expensive than in our case, requiring $O(n)$ parings for n different messages. Second, alternative aggregatable BLS signatures exist (e.g., [7]) where both the aggregated public key have their size independent of the number of signers and the signature verification is comparable to our variant. However, for the blockchain use cases we envision (for example the accountable light client system from [5]) we prefer our scheme detailed in Section 2.2: its corresponding key aggregation is a simple sum of the individual signers' public keys, while, in [7] the key aggregation operation involves more expensive scalar multiplications every time the key aggregation is performed.

2 Our Aggregatable Signature Scheme

2.1 Secure Signature Aggregation

An aggregatable signature scheme compresses signatures issued using possibly different signing keys into one signature. In this work we use an aggregatable signature scheme making explicit use of proofs-of-possession (PoPs) [1] in order to protect against rogue-key attacks [1].

Definition 1. *(Aggregatable Signature Scheme) An aggregatable signature scheme consists of the following tuple of algorithms (AS.Setup, AS.GenerateKeypair, AS.VerifyPoP, AS.Sign, AS.AggregateKeys, AS.AggregateSignatures, AS.Verify):*

- $pp \leftarrow AS.Setup(\lambda)$*: a setup algorithm that, given a security parameter* λ*, outputs public protocol parameters* pp*.*
- $((pk, \pi_{PoP}), sk) \leftarrow AS.GenerateKeypair(pp)$*: a key pair generation algorithm that outputs a secret key* sk*, and the corresponding public key* pk *together with a proof-of-possession* π_{PoP} *of the secret key.*
- $0/1 \leftarrow AS.VerifyPoP(pp, pk, \pi_{PoP})$*: a public key verification algorithm that, given a public key* pk *and a proof-of-possession* π_{PoP}*, outputs* 1 *if* π_{PoP} *is valid for* pk *and* 0 *otherwise.*
- $\sigma \leftarrow AS.Sign(pp, sk, m)$*: a signing algorithm that, given a secret key* sk *and a message* $m \in \{0,1\}^*$*, returns a signature* σ*.*
- $apk \leftarrow AS.AggregateKeys(pp, (pk_i)_{i=1}^n)$*: a public key aggregation algorithm that, given a vector of public keys* $(pk_i)_{i=1}^n$*, returns an aggregate public key* apk*.*
- $asig \leftarrow AS.AggregateSignatures(pp, (\sigma_i)_{i=1}^n)$*: a signature aggregation algorithm that, given a vector of signatures* $(\sigma_i)_{i=1}^n$*, returns an aggregate signature* $asig$*.*
- $0/1 \leftarrow AS.Verify(pp, apk, m, asig)$*: a signature verification algorithm that, given an aggregate public key* apk*, a message* $m \in \{0,1\}^*$*, and an aggregate signature* σ*, returns 1 or 0 to indicate if the signature is valid.*

We say ($AS.Setup$, $AS.GenerateKeypair$, $AS.VerifyPoP$, $AS.Sign$, $AS.AggregateKeys$, $AS.AggregateSignatures$, $AS.Verify$) is an aggregatable signature scheme if it satisfies perfect completeness, completeness for aggregation *and* unforgeability *as defined below.*

Perfect Completeness An aggregatable signature scheme ($AS.Setup$, $AS.GenerateKeypair$, $AS.VerifyPoP$, $AS.Sign$, $AS.AggregateKeys$, $AS.AggregateSignatures$, $AS.Verify$) has perfect completeness if for any message $m \in \{0,1\}^*$ and any $n \in \mathbb{N}$ it holds that:

$$
\begin{aligned}
&Pr[AS.Verify(pp, apk, m, asig) = 1 \ \wedge \ \forall i \in [n] \ AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 1 \mid \\
&pp \leftarrow AS.Setup(\lambda), \\
&((pk_i, \pi_{PoP,i}), sk_i) \leftarrow AS.GenerateKeypair(pp), \ i = 1, \ldots, n \\
&apk \leftarrow AggregateKeys(pp, (pk_i)_{i=1}^n), \\
&\sigma_i \leftarrow AS.Sign(pp, sk_i, m), \ i = 1, \ldots, n, \\
&asig \leftarrow AS.AggregateSignatures(pp, (\sigma_i)_{i=1}^n)] = 1.
\end{aligned}
$$

We note that an aggregatable signature scheme with perfect completeness implies the underlying signature scheme has perfect completeness.

Completeness for Aggregation An aggregatable signature scheme ($AS.Setup$, $AS.GenerateKeypair$, $AS.Verify$, $AS.Sign$, $AS.AggregateKeys$, $AS.AggregateSignatures$, $AS.Verify$) has completeness for aggregation if, for every adversary $\mathcal{A}$

$$
\begin{aligned}
&Pr[AS.Verify(pp, apk, m, asig) = 1 \ (***) \mid pp \leftarrow AS.Setup(\lambda), \\
&((pk_i, \pi_{PoP,i})_{i=1}^n, m, (\sigma_i)_{i=1}^n) \leftarrow \mathcal{A}(pp), \\
&\forall i \in [n], AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 1 \quad (*), \\
&\forall i \in [n], AS.Verify(pp, pk_i, m, \sigma_i) = 1 \quad (**), \\
&apk \leftarrow AS.AggregateKeys(pp, (pk_i)_{i=1}^n), \\
&asig \leftarrow AS.AggregateSignatures(pp, (\sigma_i)_{i=1}^n)(***)] = 1 - negl(\lambda).
\end{aligned}
$$

Unforgeable Aggregatable Signature For an aggregatable signature scheme ($AS.Setup$, $AS.GenerateKeypair$, $AS.VerifyPoP$, $AS.Sign$, $AS.AggregateKeys$, $AS.AggregateSignatures$, $AS.Verify$) the advantage of an adversary against unforgeability is defined by

$$Adv_{\mathcal{A}}^{forge}(\lambda) = Pr[Game_{\mathcal{A}}^{forge}(\lambda) = 1], \ where$$

$Game_{\mathcal{A}}^{forge}(\lambda)$:
$pp \leftarrow AS.Setup(\lambda)$
$((pk^*, \pi_{PoP}^*), sk^*) \leftarrow AS.GenerateKeypair(pp)$
$Q \leftarrow \emptyset$
$((pk_i, \pi_{PoP,i})_{i=1}^n, m, asig) \leftarrow \mathcal{A}^{OSign}(pp, (pk^*, \pi_{PoP}^*))$
If $pk^* \notin \{pk_i\}_{i=1}^n \vee m \in Q$, *then return* 0
For $i \in [n]$
    If $AS.VerifyPoP(pp, pk_i, \pi_{PoP,i}) = 0$ *return* 0
$apk \leftarrow AS.AggregateKeys(pp, (pk_i)_{i=1}^n)$
Return $AS.Verify(pp, apk, m, asig)$

and

$OSign(m_j)$:
$\sigma_j \leftarrow AS.Sign(pp, sk^*, m_j)$
$Q \leftarrow Q \cup \{m_j\}$
Return σ_j

and $\mathcal{A}^{OSign}$ denotes the adversary $\mathcal{A}$ with access to oracle $OSign$.
We say an aggregatable signature scheme is unforgeable if for all efficient adversaries $\mathcal{A}$ it holds that $Adv_{\mathcal{A}}^{forge}(\lambda) \leq negl(\lambda)$.

2.2 Aggregatable BLS Signatures

In the following, we instantiate the aggregatable signature definition given above with a scheme inspired by the BLS signature scheme [3] and its follow-up variants [1,7]. Because, in general, multisignatures are susceptible to so-called "rogue-key attacks" which can be mounted whenever the adversary is allowed to choose his public keys arbitrarily, in order to protect against such rogue-key attacks, we enhance our multisignature instantiation with proofs-of-possessio as defined in [1]. In turn, we instantiate our proofs-of-possession with BLS signatures in the first source group.

Instantiation 1. *(Aggregatable BLS Signatures) We call aggregatable BLS signatures the following instantiation of aggregatable signatures:*

- $(\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, \mathbb{G}_T, e, H, H_{PoP}, H_{DLEQ,sig})$ *from* pp *where* $pp \leftarrow AS.Setup(\lambda)$, *where* $\mathbb{G}_1$, $\mathbb{G}_2$, g_1, g_2, $\mathbb{G}_T$, e *are the first and second source groups, their generators and the associated pairing for some BLS elliptic curve* E, *respectively, and* $H : \{0,1\}^* \rightarrow \mathbb{G}_1$, $H_{PoP} : \{0,1\}^* \rightarrow \mathbb{G}_1$ *and* $H_{DLEQ,sig} : \{0,1\}^* \rightarrow \mathbb{Z}_r^*$ *are three hash functions.*
- $(pk_1, pk_2, sk, \sigma_{PoP}) \leftarrow AS.GenerateKeypair(pp)$, *where* $sk \xleftarrow{\$} \mathbb{Z}_r^*$ *and* $pk_1 = sk \cdot g_1 \in \mathbb{G}_1$ *and* $pk_2 = sk \cdot g_2 \in \mathbb{G}_2$ *and* $\sigma_{PoP} = sk \cdot H_{PoP}(pk_2)$ *and* r *is the size of the scalar field of elliptic curve* E.

– $0/1 \leftarrow AS.VerifyPoP(pp, pk_1, pk_2, \sigma_{PoP})$, *where* $AS.VerifyPoP$ *outputs* 1 *if the following holds:*

$$e(H_{PoP}(pk_2) + t \cdot g_1, pk_2) = e(\sigma_{PoP} + t \cdot pk_1, g_2),$$

with $t \xleftarrow{\$} \mathbb{Z}_r$. *If the verification above does not pass, then* $AS.VerifyPoP$ *outputs* 0.

– $(\sigma, \pi_{DLEQ,sig}) \leftarrow AS.Sign(pp, sk, m)$: *where* $\sigma = sk \cdot H(m) \in \mathbb{G}_1$ *and* $\pi_{DLEQ,sig} \leftarrow Prove_{DLEQ,sig}(g_1, m, pk_1, \sigma, sk)$.

– $apk_1 \leftarrow AS.AggregateKeys(pp, (pk_1^{(i)}, pk_2^{(i)})_{i=1}^n)$, *where* $apk_1 = \sum_{i=1}^n pk_1^{(i)}$.

– $asig \leftarrow AS.AggregateSignatures(pp, (pk_1^{(i)}, pk_2^{(i)})_{i=1}^n, (\sigma^{(i)}, \pi_i)_{i=1}^n)$, *where* $asig = (\sigma^{(1)}, \pi_1)$ *if* $n = 1$ *and* $asig = (\sum_{i=1}^n \sigma^{(i)}, apk_2, \perp)$ *if* $n > 1$, *where* $apk_2 = \sum_{i=1}^n pk_2^{(i)}$.

– $0/1 \leftarrow AS.Verify(pp, apk_1, m, asig)$, *where* $AS.Verify$ *outputs* 1 *if either* $asig_3 = \perp$ *and* $e(asig_1 + t \cdot apk_1, g_2) = e(H(m) + t \cdot g_1, asig_2)$, *with* $t \xleftarrow{\$} \mathbb{Z}_r^*$ *or there exists no component* $asig_3$ *and* $Verify_{DLEQ,sig}(g_1, m, apk_1, asig_1, asig_2) = 1$; *in all other cases, it outputs* 0.

Above we have used the following argument system

$$\mathcal{PS}_{DLEQ,sig} = (KeyGen_{DLEQ,sig}, Prove_{DLEQ,sig}, Verify_{DLEQ,sig}) \quad (1)$$

– $(\mathbb{G}_1, g_1, H_{DLEQ,sig}) \leftarrow KeyGen_{DLEQ,sig}(\lambda)$ *as a subprotocol of* $AS.Setup(\lambda)$.

– $\pi_{DLEQ,sig} = (c, s) \leftarrow Prove_{DLEQ,sig}(g_1, m, pk_1, \sigma, sk)$ *where* $k \xleftarrow{\$} \mathbb{Z}_r^*$, $A = k \cdot g_1$, $B = k \cdot H(m)$, $c = H_{DLEQ,sig}(g_1, m, pk_1, \sigma, A, B)$, $s = k - c \cdot sk \mod r$.

– $0/1 \leftarrow Verify_{DLEQ,sig}(g_1, m, pk_1, \sigma, (c, s))$, *where* $Verify_{DLEQ,sig}$ *outputs 1 if* $c = H_{DLEQ,sig}(g_1, m, pk_1, \sigma, A'', B'')$ *where* $A'' = s \cdot g_1 + c \cdot pk_1$ *and* $B'' = s \cdot H(m) + c \cdot \sigma$ *and it outputs* 0 *otherwise.*

Note: It is easy to show that $\mathcal{PS}_{DLEQ,sig}$ is a zero-knowledge non-interactive argument of knowledge for relation $\mathcal{R}_{DLEQ,sig}$, where

$$\mathcal{R}_{DLEQ,sig} = \{(\mathbb{G}_1, g_1, m, pk_1, \sigma); sk) : pk_1 = sk \cdot g_1, \sigma = sk \cdot H(m)\},$$

and $\mathbb{G}_1$ is generated by g_1.

Theorem 2. *Assuming that co-CDH holds for* e *(see Appendix for a reminder of this assumption) and* H, H_{PoP} *and* $H_{DLEQ,sig}$ *are modelled as random oracles, then instantiation 1 is an aggregatable signature scheme as per definition 1.*

Proof. The *perfect completeness* property is very easy to prove.

Regarding *completeness for aggregation*, the non-trivial case is when $n > 1$. Let $\mathcal{A}_1$ be an efficient adversary trying to break this property. Due to our instantiation, we have the following explicit notation $(pk_i, \pi_{PoP,i})_{i=1}^n = ((pk_1^{(i)}, pk_2^{(i)}), \sigma_{PoP}^{(i)})_{i=1}^n$. Since $(*)$ holds, then by the Schwartz-Zippel lemma $e(g_1, pk_2^{(i)}) = e(pk_1^{(i)}, g_2), \forall i \in [n]$, which, due to the bilinearity of pairing e and the fact that the associated source groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are cyclic, it implies that $\forall i \in [n]$, $\exists\ (sk_i)_{i=1}^n \in (\mathbb{Z}_r)^n$

such that $pk_1^{(i)} = sk_i \cdot g_1$ and $pk_2^{(i)} = sk_i \cdot g_2$. Since $(**)$ holds, then due to the existential soundness of $\mathcal{PS}_{DLEQ,sig}$, except with negligible probability, there exist $(sk_i)_{i=1}^n \in (\mathbb{Z}_r)^n$ such that $pk_1^{(i)} = sk_i \cdot g_1$ and $\sigma^{(i)} = sk_i \cdot H(m)$. The above properties together with $(***)$ in turn, imply that, except with negligible probability, $apk_1 = (\sum_{i=1}^n sk_i) \cdot g_1$, $asig_2 = (\sum_{i=1}^n sk_i) \cdot g_2$ and $asig_1 = (\sum_{i=1}^n sk_i) \cdot H(m)$. This, in turn, implies that the aggregated signature verification $(***)$ holds, except with negligible probability.

Regarding *unforgeability*, let $\mathcal{A}_2$ be an efficient adversary trying to break this property (see definition 1). Next, we follow a similar proof technique as detailed for theorem 15.2, part b in [8] where given a successful adversary $\mathcal{A}_2$ against unforgeability one can construct a successful adversary $\mathcal{B}$ against the co-CDH assumption as follows:
Adversary $\mathcal{B}$ is given a tuple $(u_1 = \alpha \cdot g_1, u_2 = \alpha \cdot g_2, v_1 = \beta \cdot g_1)$ where $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_r$, as in the co-CDH attack game (see Appendix); $\mathcal{B}$ needs to compute $z_1 = \alpha\beta \cdot g_1 = \alpha \cdot v_1$. First, $\mathcal{B}$ sends the public key $(pk^* = (u_1, u_2), \sigma^*_{PoP})$ to the forger $\mathcal{A}_2$, where $\sigma^*_{PoP} = \delta^* \cdot u_1$ and $\delta^* \xleftarrow{\$} \mathbb{Z}_r$, $H_{PoP}(pk_2{}^*) = \delta^* \cdot g_1$. Hence $\sigma^*_{PoP} = \alpha \cdot H_{PoP}(pk_2{}^*)$.
Afterwards, $\mathcal{A}_2$ makes a sequence of queries: Q_{ro} hash queries to H, Q_{sig} signature queries, Q_{pop} queries to H_{PoP}, and $Q_{DLEQ,sig}$ queries to $H_{DLEQ,sig}$. To these queries, adversary $\mathcal{B}$ responds as follows:

- Overall, hash queries to H are handled by $\mathcal{B}$ by first choosing a random $\omega \in \{1, \ldots, Q_{ro}\}$. (Note that among the Q_{ro} hash queries there may be one message that is not part of the signature queries from $\mathcal{A}_2$ but is output as an alleged forgery by $\mathcal{A}_2$. Using random value ω, $\mathcal{B}$ is trying to guess the index of that precise message queried by $\mathcal{A}_2$.) Then, for $j = 1, 2, ..., Q_{ro}$, when $\mathcal{A}_2$ issues hash query number j (i.e., query for $H(m_j)$), $\mathcal{B}$ responds by:
  - if $j \neq \omega$ then $\mathcal{B}$ chooses $\rho_j \xleftarrow{\$} \mathbb{Z}_r$ and sets $H(m_j) := \rho_j \cdot g_1$,
  - if $j = \omega$ then $\mathcal{B}$ sets $H(m_\omega) = v_1$.
- Signing queries m_j are answered as follows: The first component $\sigma^{(j)}$ of the individual signature for a message m_j is $\rho_j \cdot u_1 = \alpha \cdot H(m_j)$. This can be answered correctly by $\mathcal{B}$ as long as he guesses the correct index ω defined above and, also, since all the corresponding values ρ_j are chosen and known by him. Regarding the second component of the individual signature, i.e., $\pi^{(j)}_{DLEQ,sig}$, this can be computed by $\mathcal{B}$ using the fact that it can program the oracle for $H_{DLEQ,sig}$ and, also, since the non-interactive argument of knowledge $\mathcal{PS}_{DLEQ,sig}$ has zero-knowledge. Indeed, $\pi^{(j)}_{DLEQ,sig}$ is computed as (c_j, s_j) where c_j, s_j are chosen uniformly at random in $\mathbb{Z}_r^*$, setting $A_j = s_j \cdot g_1 + c_j \cdot u_1$ and $B_j = s_j \cdot H(m_j) + c_j \cdot \sigma^{(j)}$ and finally $\mathcal{B}$ records in the corresponding table the value for the simulated random oracle $H_{DLEQ,sig}$ in $(g_1, m_j, u_1, \sigma^{(j)}, A_j, B_j)$ as equal to c_j.
- $H_{DLEQ,sig}(g_1, m', pk_1^{(j)}, \sigma^{(j)}, A_j^{'}, B_j^{'})$ queries are received, stored in a table and retrieved as consistent queries to a simulated random oracle.
- $H_{PoP}(m_j)$ queries for $m_j \neq pk_2^*$ by choosing $\delta_j \xleftarrow{\$} Z_r$ and setting $H_{PoP}(m_j) = v_1 + \delta_j \cdot g_1$; $H_{PoP}(pk_2^*)$ has already been defined.

Eventually, $\mathcal{A}_2$ outputs a valid aggregate forgery

$$(((pk_1^{(i)}, pk_2^{(i)}), \sigma_{PoP}^{(i)})_{i=1}^n, m, asig)$$

where

$$\forall i \in [n], e(H_{PoP}(pk_2{}^{(i)}) + t_i \cdot g_1, pk_2{}^{(i)}) = e(\sigma_{PoP}^{(i)} + t_i \cdot pk_1{}^{(i)}, g_2) \quad (10),$$

with $t_i \xleftarrow{\$} \mathbb{Z}_r, \forall i \in [n]$ and

$$AS.Verify(pp, \sum_{i=1}^{n} pk_1^{(i)}, m, asig) = 1 \quad (20)$$

and $pk^* = (u_1, u_2)$ is among the public keys and m was not signed before.

- Case 1: $n = 1$. In this case, the valid forgery has the form

$$(pk_1^{(1)} = u_1, pk_2^{(1)} = u_2, \sigma_{PoP}^*, m, \sigma^*).$$

  If index ω was guessed correctly by $\mathcal{B}$, then by definition of H above $H(m) = v_1$; moreover, by the definition of valid forgery, we also have

$$Verify_{DLEQ,sig}(g_1, m, pk_1^* = u_1 = \alpha \cdot g_1, \sigma^*, \pi^*) = 1. \qquad (30)$$

  Due to knowledge soundness of the argument system $\mathcal{PS}_{DLEQ,sig}$ with respect to simulated oracle H, (30) implies that, with overwhelming probability, $\sigma^* = \alpha \cdot H(m)$. But since $H(m) = v_1$, we conclude the value $z_1 = \alpha \cdot v_1$ that $\mathcal{B}$ needs to output in the co-CDH game is σ^*, so $\mathcal{B}$ can indeed output z_1 with the help of $\mathcal{A}_2$. This concludes the proof in this case.
- Case 2: $n > 1$. To conclude the proof, $\mathcal{B}$ uses a similar technique as in theorem 15.2 part b in [8] for computing the first component σ^* of signature which corresponds to public key pk^* by dividing out of $asig_1$ all the first components of every individual signature (corresponding to every public key $pk^{(i)} \neq pk^*$). Note that there are two types of signatures: type I corresponding to public key $pk^{(i)} = pk^* = (u_1, u_2)$ and type II corresponding to all other public keys $pk^{(i)} \neq (u_1, u_2)$. For type I signature, $\mathcal{B}$ only needs to know how many times the public key (and, consequently, the respective signature) is repeated in the output of $\mathcal{A}_2$. For type II signatures, let $\alpha_i \in \mathbb{Z}_r$ be such that $pk_2^{(i)} = \alpha_i \cdot g_2$. Due to (10), this implies that $pk_1{}^{(i)} = \alpha_i \cdot g_1$ and $\sigma_{PoP}^{(i)} = \alpha_i \cdot H_{PoP}(pk_2{}^{(i)})$. $\mathcal{B}$ can compute for a public key $pk^{(i)} \neq (u_1, u_2)$ the corresponding signature's first component as $\sigma^{(i)} = \sigma_{PoP}^{(i)} - \delta_i \cdot pk_1^{(i)}$. It is clear that

$$\sigma^{(i)} = \sigma_{PoP}^{(i)} - \delta_i \cdot pk_1^{(i)} = \alpha_i \cdot v_1 + \alpha_i \delta_i \cdot g_1 - \alpha_i \delta_i \cdot g_1 = \alpha_i \cdot v_1 = \alpha_i \cdot H(m).$$

  Since $\sigma^{(i)}$ is a correct signature, it verifies

$$e(\sigma^{(i)}, g_2) = e(H(m), pk_2{}^{(i)}) \quad (40)$$

  Moreover, let d be the number of repetitions of pk^* in the output of $\mathcal{A}_2$. If s is the product of all the signatures that $\mathcal{B}$ can compute

(i.e., for public keys different from pk^*), then from (20) we obtain $e(s, g_2) = e(H(m), \sum_{i=1}^{n} {pk_2}^{(i)} - d \cdot {pk_2}^*)$ and, together with (40), this implies $e(d \cdot \sigma^*, g_2) = e(asig - s, g_2) = e(H(m), {pk_2}^*)^d$. Finally, using also the definitions of $H(m)$, ${pk_2}^*$ and z_1 we have

$$e(asig, g_2) = e(s, g_2) \cdot e(asig - s, g_2) = e(s, g_2) \cdot e(H(m), {pk_2}^*)^d =$$
$$= e(s, g_2) \cdot e(z_1, g_2)^d = e(s, g_2) \cdot e(d \cdot z_1, g_2).$$

Hence, $d \cdot \sigma^* = asig - s = d \cdot z_1$. The value z_1 that $\mathcal{B}$ needs to output is computable as $d^{-1} \cdot (asig - s)$.

3 Benchmarks

The implementation of the scheme proposed in this paper using Rust programming language can be found as part of the Web3 Foundation BLS Library [9]. The BLS library uses Arkworks Framework [10] as the backend to perform curve and arithmetic operations. The curve used in for benchmarking is the BLS12-377 curve introduced in [11].
The benchmarks have been measured using a 'cargo bench' running on a single thread on an 'Intel(R) Xeon(R) E5-2440 0 @ 2.40GHz' CPU.

3.1 Verification using aggregated $\mathbb{G}_1$ public keys

Table 1 lays out the details of times needed to perform operations related to aggregate the signers' public keys and verifying the aggregated signature signed by 100 signers or 1000 signers respectively. In both scenarios we assume that an aggregated signature in $\mathbb{G}_1$ and a list of signers' public keys is handed to the verifier.

In the first case, in accordance with aggregated BLS signature verification, the verifier has the list of signers' public keys in $\mathbb{G}_2$ and aggregates them to check the validity of the given signature. In the second case the aggregated public key of the signers in $\mathbb{G}_2$ is additionally handed to the verifier. The verifier has the list of signers' public keys in $\mathbb{G}_1$ and aggregates them to check the validity of the signature using the scheme proposed in this work.

Scheme	**1000 Signers**	**10000 Signers**
Aggregate Public keys in $\mathbb{G}_2$ and Verify (standard BLS)	10.815 ms	69.062 ms
Aggregate Public keys in $\mathbb{G}_1$ and Verify (our scheme)	6.815 ms	22.159 ms

Table 1: Verification time for 1000/10000 aggregated signatures using public keys in $\mathbb{G}_1$ and $\mathbb{G}_2$ on BLS12-377 Curve.

The benchmarks in Table 1 demonstrate that when the number of signers increases, the advantage of performing the aggregation of public keys in

$\mathbb{G}_1$ becomes more prominent. This result is expected because the scheme presented in this work performs only two extra scalar multiplications in $\mathbb{G}_1$ compared to standard BLS verification with aggregate public key in $\mathbb{G}_2$. Thus, when the number of signers increases, the benefit from having to perform a large number of elliptic curve additions in the faster $\mathbb{G}_1$ group compared to the same number of additions in the slower $\mathbb{G}_2$ group overcompensates for the overhead cost of two extra scalar multiplications in the case of our scheme.

3.2 Verification of individual BLS signatures via verification of Chaum-Pedersen Proofs

Table 2 illustrates the efficiency of verification when using additional Chaum-Pedersen proofs that accompany the BLS signatures and the BLS public keys in the first source group $\mathbb{G}_1$.

Scheme	**1000 Signatures**
BLS Verification (applying pairings with each individual signature)	4534 ms
Chaum-Pedersen verification for BLS signature and key (our scheme)	2480 ms

Table 2: Verification time for 1000 signatures verified individually (non-aggregated) using BLS Verification vs using the Chaum-Pedersen proofs.

Note that for the BLS verification, we have applied pairings to each of the 1000 signatures[1] rather than aggregating those signatures and applying pairings to the aggregated signature. There are real world practical applications which require individual verification of BLS signatures, such as when individual signatures are gossiped to an aggregator node and the gossiper is supposed not to gossip invalid signatures. Similarly is the case of identifying the signer(s) accountable for the failure of an aggregated signature.

Considering that the pairing costs as much as eight naive $\mathbb{G}_1$ scalar multiplications in the Artworks library, we have benefited from about a 45% reduction of the cost of the verification of time of individual signatures. We note that there are various avenues to improve the efficiency of the scalar multiplications (such as employing efficient multi-scalar multiplication methods) which has not been accounted for and can make our scheme even more efficient.

4 Conclusion

In this paper we define, instantiate and prove the security properties for an aggregatable BLS signature scheme such that when used appropriately, it allows the verifier to essentially store or work only with short

[1] In our scheme, the first component of the individual signatures is in $\mathbb{G}_1$ and the signature is over the corresponding signer's $\mathbb{G}_2$ public key.

public keys and signatures because they are in the first target group of the associated BLS pairing. We have implemented our signature scheme and we have provided concrete benchmarks and an evaluation of its efficiency benefits. We are planning to improve the efficiency of the implementation by applying other optimisations related to multiple scalar multiplications.

References

1. T. Ristenpart and S. Yilek, "The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks," in *EUROCRYPT 2007*, pp. 228–245, 2007.
2. D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Annual international cryptology conference*, pp. 89–105, Springer, 1992.
3. D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *ASIACRYPT 2001*, pp. 514–532, 2001.
4. T. Kim and R. Barbulescu, "Extended tower number field sieve: A new complexity for the medium prime case," in *Advances in Cryptology – CRYPTO 2016*, pp. 543–571.
5. O. Ciobotaru, F. Shirazi, A. Stewart, and S. Vasilyev, "Accountable light clients for pos blockchains." Cryptology ePrint Archive, Report 2022/1205, 2022.
6. D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *EUROCRYPT 2003*, pp. 416–432, 2003.
7. D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *ASIACRYPT 2018*, pp. 435–464, 2018.
8. D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography*. 2020.
9. "Boneh-Lynn-Shacham (bls) signature library with signature and aggregation in G1," tech. rep., Web3.0 Technologies Foundation, `https://github.com/w3f/bls/tree/skalman-double-puclic-key-verify`, 2022.
10. Arkworks, "An ecosystem for developing and programming with zksnarks," tech. rep., Arkworks, `https://github.com/arkworks-rs`, 2022.
11. S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
12. S. Yonezawa, "Pairing-friendly curves," 2020. `https://tools.ietf.org/id/draft-yonezawa-pairing-friendly-curves-02.html`.
13. S. Galbraith, K. Paterson, and N. Smart, "Pairings for cryptographers." ePrint 2006/165, 2006.

A Appendix

Below we remind the reader the co-CDH assumption, which is a variation of the standard computational Diffie-Hellman assumption (CDH) for the

case when two groups are used. Let E be a pairing friendly elliptic curve and let $\mathbb{G}_1$, $\mathbb{G}_1$ and $\mathbb{G}_T$ be appropriately chosen subgroups of order r with g_1, g_2 generators for the first two subgroups, respectively. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a secure pairing [12,13].

Attack Game co-CDH For a given adversary $\mathcal{A}$ the attack runs as follows:

- The challenger computes $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_r$, $u_1 \leftarrow g_1^{\alpha}$, $u_2 \leftarrow g_2^{\alpha}$, $v_1 \leftarrow g_1^{\beta}$, $z_1 \leftarrow g_1^{\alpha\beta}$ and gives the tuple (u_1, u_2, v_1) to $\mathcal{A}$; α is used twice, once in $\mathbb{G}_1$ and in $\mathbb{G}_2$.
- The adversary $\mathcal{A}$ outputs some $\hat{z_1} \in \mathbb{G}_1$.

$\mathcal{A}$'s advantage in solving the co-CDH problem for e, denoted by

$$Adv^{coCDH}[\mathcal{A}, e]$$

is the probability that $\hat{z_1} = z_1$.

co-CDH Assumption We say that the co-CDH assumption holds for the pairing e if for all efficient adversaries $\mathcal{A}$, the quantity $Adv^{coCDH}[\mathcal{A}, e]$ is negligible. If e is a symmetric pairing, then $\mathbb{G}_1 = \mathbb{G}_2$ and $g_1 = g_2$ in which case the co-CDH assumption is identical to the standard CDH assumption.