2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)

Splitting and Aggregating Signatures in Cryptocurrency Protocols

S. Sharmila Deva Selvi[1], Arinjita Paul[1], C. Pandu Rangan[1], Siva Dirisala[2], and Saswata Basu[2]

[1]Department of Computer Science and Engineering, IIT Madras, India
Email: {sharmila,arinjita,prangan}@cse.iitm.ac.in

[2]0chain LLC, San Jose, USA
Email: {siva,saswata}@0chain.net

***Abstract*—The blockchain technology and a vast amount of cryptocurrency related activities have generated an unprecedented level of interest among the public. However, even at the entry level, cryptocurrency users need to deal with the complex task of key management. In this paper, we propose a simple way to manage a user's private key, under a reasonable assumption that the user has two devices at his disposal (say a laptop and a mobile phone). We refer to our strategy as *key splitting*. Since these cryptographic keys are used for generating digital signatures, we should take a closer look at the signature schemes that would perform best under key splitting. At the operational level, scalability is one of the main challenges faced by the users and developers. While there are fundamental issues like consensus that challenge scalability, we focus on the computational efficiency in a block formation. Aggregation of signatures is one of the effective solutions to this problem. To this end, we observe that none of the existing signature schemes work well for BOTH key splitting and aggregation. The current popular schemes such as the ones used in Bitcoin or Schnorr's scheme implemented over Elliptic curves are neither suitable for aggregation nor can their keys be split in a convenient and meaningful way. A detailed theoretical and empirical analysis shows that the BLS short signature scheme is best suited for achieving both key splitting and aggregation.**

***Keywords*-Blockchain, key management, wallet, signature, scalability.**

I. Introduction

The real-world as well as the academic studies on cryptocurrencies and the block chain technology are among the most significant and trendy developments of Information Technology. Block-chain technology is witnessing an exponential growth in interest and technical advancement at this point of time. While these areas are witnessing an unprecedented growth and attention, their deployments face major hurdles at several fronts. One of the major concerns related to this technology is scalability and in general efficiency/reliability of the whole operation. For instance, every user in this community, sooner or later, directly or indirectly, is forced to deal with challenges of maintaining and managing the cryptographic keys that are used. The subtleties and challenges involved in key generation, maintenance and management are well known in security industry and both cryptographic and policy based solutions have been devised in the past. However, in the context of cryptocurrencies, we still do not have satisfactory solutions that would help scalability or ease of use. The second major concern is related to computational efficiency of the tasks performed during the execution of the protocols. One of the most computationally intense and most frequently used cryptographic primitives in blockchain technology is digital signatures. The users need to generate every transaction with appropriate authentication done on the transaction and the minors or validators need to verify/validate the same multiple number of times. In this paper, we focus on the signing process at the users end and verification process at the block formation/validation end.

In order to handle the challenges and complexities of key management, a number of techniques were proposed and deployed in different cryptocurrencies. In Bitcoin core, the keys are maintained in local storage. A typical user will have an access to a wallet software of his choice and use the same to authenticate transactions he is generating. As wallets generate the digital signature, it requires an access to the private key of the user. While this speeds up the wallet operations, the presence of a key for a long time in a system that is online increases its vulnerability. Off-line storage and air gapped storages are used by systems such as Armory [1]. Password protected wallets are deployed by certain systems but they do not provide any security against a malware that might read the key strokes etc. Third party hosted wallets are also suggested to remove the pains of key management to a novice user but then it requires enormous amount of trust in a third party. A detailed analysis on various techniques that are currently used in practice together with limitations in their usability is reported in [7].

In view of the shortcomings of the existing systems, we take a fresh look at key generation and management using two systems that may be available with a typical user. Our proposal is simple, easy to implement, secure and offers protections against theft/loss of the systems. Given that a typical user may have at his disposal several devices(atleast two, say a laptop and a mobile phone/notepad), is it possible to handle key management with relative ease? Specifically, we are interested in "splitting the private key" into several components and store each in a device so that:

1) We have adequate protection even in the case of

978-1-7281-1264-0/19/$31.00 ©2019 IEEE
DOI 10.1109/DAPPCON.2019.00021

loss/corruption of a component of a key.

2) Even in the case of loss/theft of the device and subsequent abuse of the key component available in the device.
3) Signature generation must involve all the split components.
4) The individual components of the signatures generated in each device is secure on its own and does not lead to any attacks and key exposure.

We note that the (BLS) short signature scheme of Boneh, Lynn and Shacham [6] is quite amenable for such split-ups and we describe a way in which an effective split-up be achieved. Also, such split-up is not possible in Schnorr signature [12] or Elliptic Curve Digital Signature Algorithm (ECDSA) [9] without sharing information between the two devices that generate the partial signatures.

The transaction generation as well as the block formation/validation involve running computationally intense signing and verification algorithms. Typically, the block size is kept small by design in order to speed up the communication and in small blocks, it is observed that signatures occupy a significant amount of space. For example, it is estimated that nearly 40% of the transcript space is occupied by signatures in case of bitcoin [14]. The computations involved in some of the deployed signature schemes are found to be very complex. For instance, the most widely used ECDSA combines the long term and short term keys in a non linear fashion and that directly contributes to its inefficiency [14]. Moreover, each block formation calls for verification of a number of signatures (the signatures found in the transactions chosen for pooling) and when the block is broadcast, again the validation process calls for huge number of signature verifications at every node of the network. In this context, aggregate verification offers an efficient solution. In signature aggregation/verification, we combine several signatures into one "super" signature and carry out the verification only on the super signature rather than on the individual signature. Thus we see a dramatic drop in the verification cost of n signatures to the cost of verifying one signature. This clearly saves space and a significant amount of computing time. While the aggregation is a neat idea leading to efficiency, it is unfortunate that not all signature schemes are suitable for aggregation. For some schemes, aggregation may not reduce the cost at all and the verification cost may remain the same without any reductions. The Schnorr or the ECDSA signature does not offer any natural way to aggregate due to the independent randomness deployed in different signatures on different messages. In fact, it is shown that any attempt to aggregate may even lead to serious attack as shown in [14]. In view of this fact, fresh attempts are made to create aggregatable signatures and the Gamma Γ- signature scheme in [14] is the most recent one. However, the keys of this scheme cannot be split in any convenient way. Moreover, empirical analysis shown in Table IV shows that BLS signature is far more efficient than the Γ-signature scheme.

BLS Keys	**Device 1**	**Device 2**
Private Key: $x \in \mathbb{Z}_q^*$ Public key: $X = xP$ $\sigma = xH(m)$		
	Accept mnemonic$_1$ $x_1 = H_1(\text{mnemonic}_1)$ $x_2 = x - x_1$ Store x_1 and x_2P Send x_2 to device 2	
		Accept $Passcode$ $s_1 = x_2 - H_1(Passcode)$ $y = H_2(x_2, Passcode)$ Store s_1, y

Table I: Our Key-Split algorithm

Device 2	**Device 1**
Accept Passcode $x_2 = s_1 + H_1(Passcode)$ $y' = H_2(x_2, Passcode)$ Check if $y' \stackrel{?}{=} y$ $\sigma_2 = x_2H(m)$ Send σ_2 to device 1	
	Check if $\hat{e}(\sigma_2, P) \stackrel{?}{=} \hat{e}(H(m), x_2P)$ $\sigma = x_1H(m) + \sigma_2 = x_1H(m) + x_2H(m)$ $= xH(m)$ Output σ as the signature for m

Table II: Signing Workflow

Thus we show that BLS scheme can be tinkered to accommodate key splitting at the users end and can be aggregated for verification at the block generation end. We have presented the details of signature scheme based on key splitting and aggregate verification. We have also carried out performance analysis and compared the running times of our approach with that of the Schnorr, ECDSA and Γ-signature schemes.

II. Definition

In this section, we describe the syntactical definition of a signature scheme that provides key split-up and aggregation.

A. Signature Scheme with Key Split-up

A signature scheme with key split-up consists of the following algorithms:

- **Setup**(λ): On input of a security parameter λ, this algorithm generates the public parameters $Params$.
- **KeyGen**($Params$): On input of the public parameters $Params$, this algorithm generates a public-private key pair (PK, SK) for a user.
- **Key-split**($SK, Params$): On input of a private key SK and public parameters $Params$, this algorithm generates two partial private keys SK_1 and SK_2 for device 1 and device 2 respectively. This algorithm is run by the user.
- **Sign-1**($m, SK_1, Params$): On input of a message m and the partial private key SK_1, this algorithm outputs the partial signature σ_1. This algorithm is run on device 1.

- **Sign-2**($m, SK_2, Params$): On input of a message m and the partial private key SK_2, this algorithm outputs the partial signature σ_2. This algorithm is run on device 2.
- **Sign-Combine**($\sigma_1, \sigma_2, Params$): On input of the partial signatures σ_1 and σ_2 and the public parameters $Params$, this algorithm combines the signatures and outputs a compressed short signature σ. This algorithm is run on device 1.
- **Verify**($m, \sigma, PK, Params$): On input of an aggregated signature σ, a public key PK and the public parameters, the algorithm returns "VALID" if σ is a valid signature on message m and public key PK. Else, it will output "INVALID".

B. Aggregate Signature Scheme with Key Split-up

An aggregate signature scheme with key split-up consists of the following algorithms along with the **Setup()**, **Key-Gen()**, **Key-Split()**, **Sign-1()**, **Sign-2()**, **Verify()** algorithms shown in Section II-A to enable signature aggregation.

- **Aggregate-Sign**(Σ=$\{(m_i, \sigma_i, PK_i)\}_{i=1\ to\ n}, Params$): On input of message m_i, public key PK_i and signature σ_i of n different users and the public parameters $Params$, this algorithm generates and outputs the aggregate signature δ. This algorithm can be run by any user who possess Σ.
- **Aggregate-Verify**(δ, $\{m_i, PK_i\}_{i=1\ to\ n}, Params$): On input of an aggregate signature δ, the message, public key pair (m_i, PK_i) of n different users and the public parameters $Params$, the algorithm returns "VALID" if δ is a valid aggregate signature on message-key pairs $\{m_i, PK_i\}_{i=1 to n}$. Else, it will output "INVALID".

III. Security Model

A. Security Model for Signature with Key Split-up

The security model for signature schemes with key split-up is same as that of the traditional signature schemes, by replacing the sign oracle modified into $Sign$-1 and $Sign$-2. The challenger C provides the forger $\mathcal{F}$ with a public key PK_T. The forger $\mathcal{F}$ can ask one of the partial private keys corresponding to PK_T of its choice. A signature scheme with key split-up is secure against existential forgery under the adaptive chosen message attack, if no probabilistic polynomial time algorithm $\mathcal{F}$ has a non-negligible advantage in the following game:

- **Setup Phase**: The challenger C generates the public parameters and sends it to the forger $\mathcal{F}$. The challenger C also provides the forger $\mathcal{F}$ with a public key PK_T. $\mathcal{F}$ now decides to request one part of the private key of its choice.
- **Training Phase**: The forger $\mathcal{F}$ adaptively queries partial signatures on public key PK_T with various messages of his choice.
- **Forgery Phase**: Finally, $\mathcal{F}$ produces a valid message, signature pair (m_T, σ_T) signed under public key PK_T on a message m_T.

B. Security Model for Aggregate Signature with Key Split-up

Our game based definition of the security of aggregate signature schemes with key split is an adaption of the aggregate chosen-key security model by Boneh $et\ al.$ [5]. In this model, the challenger C provides the aggregated forger $\mathcal{F}$ with a single public key PK_T and gives $\mathcal{F}$ the power to select all public keys except the challenger public key PK_T. An aggregate signature scheme with key split-up is secure against existential forgery under the adaptive chosen message attack, if no probabilistic polynomial time algorithm $\mathcal{F}$ has a non-negligible advantage in the following game:

- **Setup Phase**: The challenger C generates the public parameters and sends it to the aggregated forger $\mathcal{F}$. The challenger C also provides the forger $\mathcal{F}$ with a public key PK_T and access to one of the partial private key corresponding to SK_T of $\mathcal{F}'s$ choice.
- **Training Phase**: The forger $\mathcal{F}$ adaptively queries for signatures and partial signatures on public key PK_T with various messages of his choice.
- **Forgery Phase**: Finally, $\mathcal{F}$ outputs $n = k+1$ additional public keys $PK_1, PK_2, \cdots, PK_{n-1}, PK_T$, messages $m_1, m_2, \cdots m_{n-1}, m_T$ and an aggregated signature δ signed under public keys $PK_1, PK_2, \cdots, PK_k, PK_T$ on the messages $m_1, m_2, \cdots m_k, m_T$ respectively. Here it should be noted that $\mathcal{F}$ has generated all the public keys $PK_1, PK_2, \cdots, PK_k$ except PK_T and the challenger C does not know the private key corresponding to $PK_1, PK_2, \cdots, PK_k$.

The forger $\mathcal{F}$ wins the game if the aggregate signature δ is a valid aggregate of the signatures on the messages $m_1, m_2, \cdots m_k, m_T$ signed under public keys $PK_1, PK_2, \cdots, PK_k, PK_T$ respectively, such that the signature on message m_T under PK_T (i.e. both $Sign$-$1(m_T, PK_T)$ and $Sign$-$2(m_T, PK_T)$) has not been queried upon by $\mathcal{F}$.

Definition 1. *An aggregate forger $\mathcal{F}(t, q_H, q_s, \epsilon)$ breaks an aggregate signature scheme with key split-up if $\mathcal{F}$ runs in time atmost t, making q_H hash queries, q_S signing queries and his advantage Adv_{AggSig_A} is atleast ϵ. An aggregate signature scheme with key split-up is $(t, q_H, q_s, \epsilon)-$secure against existential forgery in the aggregate chosen key model if no (t, q_H, q_s, ϵ) forger breaks it.*

IV. Short Signature using Bilinear Pairing

The Boneh-Lynn-Shacham(BLS) signature scheme [6] is a signature scheme based on bilinear pairing of an elliptic curve group. Signatures produced by the BLS signature scheme are short signatures. The signature scheme is provably secure (the scheme is existentially unforgeable under

adaptive chosen-message attacks) assuming both the existence of random oracles and the intractability of the computational Diffie-Hellman problem in a gap Diffie-Hellman group.

The BLS signature scheme consists of the algorithms, **Setup()**, **KeyGen()**, **Sign()** and **Verify()**.

BLS Scheme:

- **Setup(κ)** :On input of the security parameter κ this algorithm will perform the following,
  - Choose a large prime q.
  - Choose two cyclic groups $\mathbb{G}$ and $\mathbb{G}_T$ of order q, where $\mathbb{G}$ is additive group and $\mathbb{G}_T$ is a multiplicative group.
  - Choose the generator $P \in \mathbb{G}$.
  - Define the bilinear pairing $\hat{e} : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$.
  - Define cryptographic hash function $H : \{0,1\}^{l_m} \rightarrow \mathbb{G}$, where l_m is the size of the message.
  - Output the system parameters $Params$ = $\langle q, P, H, \hat{e} \rangle$.
- **KeyGen($Params$)**This protocol is run by the user. This will output the public and private key of the user.
  - Choose $x \in {\mathbb{Z}_q}^*$.
  - Set $X = xP \in \mathbb{G}$.
  - Output public key $PK = \langle X \rangle$ and private key $SK = \langle x \rangle$.
- **Sign($m, SK, Params$)**: This protocol is run by user in the mobile. On input of message m and private key x, this will output the signature σ computed as shown below:
  - Compute $\sigma = xH(m, PK) \in \mathbb{G}$.
  - Output σ.
- **Verify($m, \sigma, PK, Params$)**: This protocol can be run by any user who possess the message m, signature σ on message m and public key PK, this algorithm will perform the following:
  - Compute $H = H(m, PK) \in \mathbb{G}$.
  - If $\hat{e}(H, X) \stackrel{?}{=} \hat{e}(\sigma, P)$ then output VALID, else output INVALID.

A. BLS Scheme with key split-up for secure wallet:

- **Setup(κ)** :On input of the security parameter κ this algorithm will perform the following.
  - Choose a large prime q.
  - Choose two cyclic groups $\mathbb{G}$ and $\mathbb{G}_T$ of order q, where $\mathbb{G}$ is additive group and $\mathbb{G}_T$ is a multiplicative group.
  - Choose the generator $P \in \mathbb{G}$.
  - Define the bilinear pairing $\hat{e} : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$
  - Define cryptographic hash function $H : \{0,1\}^{l_m} \times \mathbb{G} \rightarrow \mathbb{G}$, where l_m is the size of the message.
  - Output the system parameters $Params$ = $\langle q, P, H, \hat{e} \rangle$.
- **KeyGen($Params$)**This protocol is run by the user. This will output the public and private key of the user.
  - Choose $x \in {\mathbb{Z}_q}^*$.
  - Set $X = xP \in \mathbb{G}$.
  - Output public key $PK = \langle X \rangle$ and private key $SK = \langle x \rangle$.
- **Key-Split($SK, Params$)**: This protocol is run by the user. On input of private key x, this algorithm will output the two private keys SK_1 and SK_2 corresponding to private key SK.
  - Choose $x_1 \in {\mathbb{Z}_q}^*$
  - Set $x_2 = (x - x_1) \in {\mathbb{Z}_q}^*$
  - Output private key pair $SK_2 = x_2$ and private key $SK_1 = x_1$. Here SK_1 is the private key corresponding to device 1 and SK_2 is the private key for device 2.
- **Sign-1($m, SK_1, Params$)**: This protocol is run by user in the device 1. On input of message m and private key x_1, this will output the signature σ_1 computed as below:
  - Compute $\sigma_1 = x_1 H(m, PK) \in \mathbb{G}$.
  - Output σ_1.
- **Sign-2($m, SK_2, Params$)**: This protocol is run by user in the device 2. On input of message m and private key x_2, this algorithm will output the signature σ_2 computed as below:
  - Compute $\sigma_2 = x_2 H(m, PK) \in \mathbb{G}$.
  - Output σ_2.
- **Sign-Combine(σ_1, $\sigma_2, Params$)**: This protocol is run by user on device 1 with the partial signature σ_1 of device 1, partial signature σ_2 from device 2 and public parameters *Params* as inputs and outputs the signature σ on message m by the private key SK corresponding to the public key PK.
  - Output $\sigma = \sigma_1 + \sigma_2 = x_1 H(m, PK) + x_2 H(m, PK) = xH(m, PK)$.
- **Verify($m, \sigma, PK, Params$)**: This protocol can be run by any user who possess the message m, a signature σ of the message m and the public key PK. This algorithm will perform the verification as following:
  - Compute $H = H(m, PK) \in \mathbb{G}$.
  - If $\hat{e}(H, X) \stackrel{?}{=} \hat{e}(\sigma, P)$ then output VALID, else output INVALID.

B. BLS Scheme with key split-up and aggregation

The aggregation mechanism is helpful during the verification of n signatures generated by n different users on n different messages $m_1, m_2, \cdots, m_n$. The algorithms **Setup()**, **KeyGen()**, **Key-Split()**, **Sign-1()**, **Sign-2()**, **Verify()** are same as that of the scheme given in Section IV-A. It has two more algorithms for generating aggregate signatures and verification of aggregate signatures, i.e., **Aggregate-Sign()** and **Aggregate-Verify()** respectively. The description of both the algorithms is given below:

- **Aggregate-Sign**(Σ=$\{(m_i, \sigma_i, PK_i)\}_{i=1\,to\,n}, Params$): This protocol is run by an user who has all the signatures in Σ as input and outputs the aggregate signature δ on message m_i by the private key SK_i corresponding to the public key PK_i, for $i = 1$ to n.
  - If **Verify**$(m_i, \sigma_i, PK_i, Params)$ is VALID for all signatures σ_i in Σ then output:
$$\delta = \sum_{i=1}^{n} \sigma_i.$$
  - Else, Output $\perp$.
- **Aggregate-Verify(** δ, $\{m_i, PK_i\}_{i=1\,to\,n}, Params$**)**: This protocol can be run by any user who possess the messages m_i, public keys PK_i, and an aggregate signature δ. This algorithm will perform the signature verification as follows:
  - For $(i = 1\ to\ n)$, compute $H_i = H(m_i, PK_i) \in \mathbb{G}$.
  - If $\prod_{i=1}^{n} \hat{e}(H_i, X_i) \stackrel{?}{=} \hat{e}(\delta, P)$ then output VALID, else output INVALID.

V. Security Proof

A. Security Proof of our Signature Scheme with Key Split-up

In this section we formally prove the security of the signature scheme with key split-up against existential forgery under adaptive chosen-message attack in the random oracle model. We formally show that our scheme is secure and its security follows from the Computational Diffie-Hellman (CDH) assumption in $(\mathbb{G}, \mathbb{G}_T)$.

Definition 2. ***Computational Diffie-Hellman (CDH) assumption****: The Computational Diffie-Hellman (CDH) assumption in $\mathbb{G}$ is, given a tuple of elements $(P, aP, bP) \in \mathbb{G}$, where a,b $\in_R \mathbb{Z}_q^*$, there exists no polynomial time adversary which can compute abP in $\mathbb{G}$, with a non-negligible advantage.*

Theorem 1. *Let $(\mathbb{G}, \mathbb{G}_T)$ be a (t', ϵ')-CDH group of order p. Then our signature scheme is $(t, q_{s_1}, q_{s_2}, q_{H_1}, \epsilon)$-secure against existential forgery under the adaptive chosen-message attack in the random oracle model, for all t and ϵ that satisfies,*

$$\epsilon' \geq \frac{\epsilon}{e(1 + q_{s_2})},\ \textit{and}$$
$$t' \leq t + (q_{H_1} + q_{s_1} + q_{s_2} + \mathcal{O}(1))$$

Proof: Suppose $\mathcal{F}$ is a forger algorithm that $(t, q_{s_1}, q_{s_2}, q_{H_1}, \epsilon)$-breaks our signature scheme on $(\mathbb{G}, \mathbb{G}_T)$ in time t. Then, we show how to construct a t'-time algorithm C that solves the CDH problem on $(\mathbb{G}, \mathbb{G}_T)$ with probability atleast ϵ'. The existence of such polynomial time solver for CDH problem is not possible, hence the existence of polynomial time attacker for the signature scheme is also not possible

Let P be the generator of $\mathbb{G}$. Algorithm C is provided with the challenge instance $(P, aP, bP) \in \mathbb{G}$ whose goal is to generate $abP \in \mathbb{G}$. Forger $\mathcal{F}$ interacts with the challenger C and the challenger will answer all the queries asked by the Forger in the following way :

- **Setup Phase :** Challenger C starts by giving $\mathcal{F}$ the common reference string $(P, \mathbb{G}, \mathbb{G}_T)$ and the public key $PK_T = \langle X_T = (s_1 + s_2 a)P) \rangle$, where s_1, s_2 is chosen at random from $\mathbb{Z}_q^*$. The corresponding private key $x_T = s_1 + s_2 a$, which is not known to the challenger. Now, $\mathcal{F}$ will request C the partial key SK_1 or SK_2 which $\mathcal{F}$ wishes to compromise . If he decides to get access to partial private key SK_1 then $x_1 = s_1$ and $x_2 = as_2$. If $\mathcal{F}$ decides to get access to partial private key SK_2 then $x_1 = as_2$ and $x_2 = s_1$. Note that C does not know the value of either x_1 or x_2. For simplicity we assume that $\mathcal{F}$ decides to compromise partial private key x_1.
- **Training Phase :** During this phase, $\mathcal{F}$ is given access to the following oracles provided by the challenger C:
  - H query: C handles the hash queries of the forger $\mathcal{F}$ by maintaining a list L_H consisting of tuples defined as $\langle m, PK, c, h, H_m \rangle$. Initially the list is empty and is updated as explained below. When $\mathcal{F}$ queries the oracle H with a message $m \in \{0,1\}^{l_m}$ and public key PK as input, C responds in the following way:
    - If a tuple $\langle m, PK, c, h, H_m \rangle$ already exists in the L_H list, then C responds with $H(m, PK) = H_m \in \mathbb{G}$.
    - Otherwise, C picks a random $h \in \mathbb{Z}_q^*$ and flips a coin $c \in \{0, 1\}$ such that $Pr[c = 0] = \gamma$ (defined later) and sets $H_m = hbP$ if $c = 1$. Else, it sets $H_m = hP$. Also, $\mathcal{C}$ stores the tuple $\langle m, PK, c, h, H_m \rangle$ in L_H list and output H_m.
  - Sign-1 query: When a signature query is generated for device 1, C does the following:
    1) C queries $H(m, PK)$ to obtain H_m.
    2) C computes $\sigma_1 = x_1 H_m$.
    3) C sends σ_1 as the signature to $\mathcal{A}$.
  - Sign-2 query: When a signature query is generated for device 2, C does the following:
    1) C checks whether (m, PK) is already present L_H list. If (m, PK) belongs to L_H list, then:
       - C retrieves the tuple $\langle m, PK, c, h, H_m \rangle$ from L_H list.
       - If $c = 0$, sets $\sigma_2 = s_2 haP = as_2 H_m = x_2 H_m$.
       - Else, Aborts.
    2) If (m, PK) does not belong to L_H list, then:
       - C picks a random $h \in \mathbb{Z}_q{}^*$
       - C sets $H_m = hP$ and $c = 0$.
       - Also, C stores $\langle m, PK, c, h, H_m \rangle$ in L_H list.

* C computes σ_2= s_2haP=$(as_2)hP$=as_2H_m =x_2H_m.

3) C sends σ_2 as the signature to $\mathcal{F}$.

- **Forgery Phase :** On getting sufficient training, algorithm $\mathcal{F}$ produces a message-signature pair (m^*, σ^*) such that σ^* is not obtained by querying the signing oracle for a message m^* and σ^* is valid. Now, C computes the solution to the hard problem as shown below.
  - C checks L_H for a tuple $\langle m^*, PK, c, h, H_{m^*}\rangle$.
  - C computes $T = (hs_2)^{-1}(\sigma^* - hs_1P)$. This correctly generates the solution to the hard problem $T = abP$ as below.

$$\begin{aligned} T &= (hs_2)^{-1}(\sigma^* - hs_1P) \\ &= (hs_2)^{-1}(x(hP) - hs_1P) \\ &= (hs_2)^{-1}((s_1 + s_2a)hP - hs_1P) \\ &= (hs_2)^{-1}(s_1hP + s_2abhP - hs_1P) \\ &= (hs_2)^{-1}(s_2abhP) \\ &= abP \\ &= LHS. \end{aligned}$$

  Thus, no forgery is possible by $\mathcal{F}$ in polynomial time with a non-negligible advantage.
- **Probability Analysis:** We calculate the probability with which $\mathcal{C}$ aborts during the simulation. Let *Abort* denote the event that $\mathcal{C}$ aborts during the game and q_{s_2} denote the number of queries made to the *Sign-2* oracle. We note that $\mathcal{C}$ does not abort in the following events:
  - E_1: $c = 0$ in a *Sign-2* query of *Training* phase.
  - E_1: $c^* = 1$ in the *Forgery* phase.

  We have $Pr[\neg Abort] \geq \gamma^{q_{s_2}}(1-\gamma)$, which has a maximum value at $\gamma_{OPT} = \frac{q_{s_2}}{1+q_{s_2}}$. Using γ_{OPT}, we obtain:

$$Pr[\neg Abort] \geq \frac{1}{e(1+q_{s_2})},$$

  where, e is the base of the natural logarithm. Note that the simulation of the random oracles is perfect. Therefore, $\mathcal{C}$ solves the CDH problem in $(\mathbb{G}, \mathbb{G}_T)$ with probability:

$$\epsilon' \geq \frac{\epsilon}{e(1+q_{s_2})}.$$

  Note that C solves the hard problem after the forger $\mathcal{F}$ makes q_H queries to the H oracle, q_{s_1} queries to the $Sign$-1 oracle, q_{s_2} queries to the $Sign$-2 oracle and generates a forged signature. Also, given the forged signature, the challenger extracts the solution to the CDH problem with $\mathcal{O}(1)$ computation. Therefore the total time t' taken by C in solving the hard problem is as below:

$$t' \leq t + (q_H + q_{s_1} + q_{s_2} + \mathcal{O}(1))$$

  If t is a polynomial, it would imply that t' is also a polynomial which contradicts the assumption that $(\mathbb{G}, \mathbb{G}_T)$ is a CDH group pair. This completes the proof of the theorem.

■

B. Security Proof of our Aggregate Signature Scheme with Key Split-up

In this section we formally prove the security of the signature scheme with key split-up against existential forgery under adaptive chosen-message attack in the random oracle model. We formally show that our scheme is secure and its security follows from the CDH assumption in $(\mathbb{G}, \mathbb{G}_T)$.

Theorem 2. *Let $(\mathbb{G}, \mathbb{G}_T)$ be a (t', ϵ')-CDH group of order p. Then our signature scheme is $(t, q_{s_1}, q_{s_2}, q_{H_1}, \epsilon)$-secure against existential forgery under the adaptive chosen-message attack in the random oracle model, for all t and ϵ that satisfies,*

$$\epsilon' \geq \frac{\epsilon}{e(1+q_{s_2})}, \text{ and}$$
$$t' \leq t + (q_{H_1} + q_{s_1} + q_{s_2} + \mathcal{O}(1))$$

Proof: Suppose $\mathcal{F}$ is a forger algorithm that $(t, q_{s_1}, q_{s_2}, q_{H_1}, \epsilon)$-breaks our signature scheme on $(\mathbb{G}, \mathbb{G}_T)$ in time t. Then, we show how to construct a t'-time algorithm C that solves the CDH problem on $(\mathbb{G}, \mathbb{G}_T)$ with probability atleast ϵ'. The existence of such polynomial time solver for CDH problem is not possible, hence the existence of polynomial time attacker for the signature scheme is also not possible

Let P be the generator of $\mathbb{G}$. Algorithm C is provided with the challenge instance $(P, aP, bP) \in \mathbb{G}$ whose goal is to generate $abP \in \mathbb{G}$. Forger $\mathcal{F}$ interacts with the challenger C and the challenger will answer all the queries asked by the Forger in the following way :

- **Setup Phase:**This phase is similar to the **Setup Phase** in V-A.
- **Training Phase :** During this phase, $\mathcal{F}$ is given access to the following oracles provided by the challenger C:
  - H query: C handles the hash queries of the forger $\mathcal{F}$ by maintaining a list L_H consisting of tuples defined as $\langle m, PK, c, h, H_m\rangle$. Initially the list is empty and is updated as explained below. When $\mathcal{F}$ queries the oracle H with a message $m \in \{0,1\}^{l_m}$ and public key PK as input, C responds in the following way:
    * If $(PK \neq PK_T)$:
      · If a tuple $\langle m, PK, c, h, H_m\rangle$ already exists in the L_H list, then C responds with $H(m, PK) = H_m \in \mathbb{G}$.
      · C picks a random $h \in \mathbb{Z}_q^*$ sets $H_m = hP$, Also, C stores the tuple $\langle m, PK, -, h, H_m\rangle$ in L_H list and output H_m.

* If ($PK = PK_T$):
  - If a tuple $\langle m, PK, c, h, H_m \rangle$ already exists in the L_H list, then C responds with $H(m, PK) = H_m \in \mathbb{G}$.
  - Otherwise, C picks a random $h \in \mathbb{Z}_q^*$ and flips a coin $c \in \{0, 1\}$ such that $Pr[c = 0] = \gamma$ (defined later) and sets $H_m = hbP$ if $c = 1$, Else, it sets $H_m = hP$. Also, C stores the tuple $\langle m, PK, c, h, H_m \rangle$ in L_H list and outputs H_m.

– Sign-1 query: When a Sign-1 is requested by $\mathcal{F}$, C does the following:
  1) C queries $H(m, PK)$ to obtain H_m.
  2) C computes $\sigma_1 = x_1 H_m$.
  3) C sends σ_1 as the signature to $\mathcal{A}$.

– Sign-2 query: When a Sign-2 is requested by $\mathcal{F}$, the challenger $\mathcal{C}$ does the following:
  1) C checks whether (m, PK) is already present L_H list. If (m, PK) belongs to L_H list, then
     * C retrieves the tuple $\langle m, PK, c, h, H_m \rangle$ from L_H list.
     * If $c = 0$, sets $\sigma_2 = s_2 haP = as_2 H_m = x_2 H_m$.
     * Else aborts.
  2) If (m, PK) does not belong to L_H list, then
     * C picks a random $h \in \mathbb{Z}_q^*$
     * C sets $H_m = hP$ and $c = 0$.
     * Also, C stores $\langle m, PK, c, h, H_m \rangle$ in L_H list.
     * C computes $\sigma_2 = s_2 haP = (as_2)hP = as_2 H_m = x_2 H_m$.
  3) C sends σ_2 as the signature to $\mathcal{F}$.

- **Forgery Phase:** On getting sufficient training, algorithm $\mathcal{F}$ produces a n message-public pair $\{(m_i, PK_i), (m_T, PK_T)\}_{i=1\,to\,k}$, where (n=k+1) and a valid aggregate signature δ such that m_T, PK_T is not queried to both $Sign$-1 and $Sign$-2 oracle. Now, C computes the solution to the hard problem as shown below.
  – For all i=1 to k, $\mathcal{C}$ checks and retrieves h_i from the entry $\langle m_i, PK_i, -, h_i, H_{m_i} \rangle$ in L_H list.
  – C retrieves h corresponding to $\langle m_T, PK_T, c, h, H_{m_T} \rangle$ in L_H list.
  – Computes $T = (hs_2)^{-1}(\delta - \sum_{i=1}^{k}(h_i PK_i) - hs_1 P)$.

  This correctly generates the solution to the hard problem $T = abP$ as below.

$$
\begin{aligned}
T =& (hs_2)^{-1}(\delta - \sum_{i=1}^{k}(h_i PK_i) - hs_1 P) \\
=& (hs_2)^{-1}(\sum_{j=1}^{n}(x_j H_j) - \sum_{i=1}^{k}(h_i X_i) - hs_1 P) \\
=& (hs_2)^{-1}(\sum_{j=1}^{k}(x_j H_j) + x_T H_T \\
& \qquad - \sum_{i=1}^{k}(x_i H_i) - hs_1 P) \\
=& (hs_2)^{-1}(x_T H_T - hs_1 P) \\
=& (hs_2)^{-1}((s_1 + s_2 ab)hP - hs_1 P) \\
=& (hs_2)^{-1}(s_1 hP + s_2 abhP - hs_1 P) \\
=& (hs_2)^{-1}(s_2 abhP) \\
=& abP \\
=& LHS.
\end{aligned}
$$

  Thus, no forgery is possible by $\mathcal{F}$ in polynomial time with a non-negligible advantage.

- **Probability Analysis:** We calculate the probability with which $\mathcal{C}$ aborts during the simulation. Let *Abort* denote the event that $\mathcal{C}$ aborts during the game and q_{s_2} denote the number of queries made to the $Sign$-2 oracle. We note that $\mathcal{C}$ does not abort in the following events:
  – E_1: $c = 0$ in a $Sign$-2 query of *Training* phase.
  – E_1: $c^* = 1$ in the *Forgery* phase.

  We have $Pr[\neg Abort] \geq \gamma^{q_{s_2}}(1 - \gamma)$, which has a maximum value at $\gamma_{OPT} = \frac{q_{s_2}}{1+q_{s_2}}$. Using γ_{OPT}, we obtain:

$$Pr[\neg Abort] \geq \frac{1}{e(1+q_{s_2})},$$

  where e is the base of the natural logarithm. Note that the simulation of the random oracles is perfect. Therefore, $\mathcal{C}$ solves the CDH problem in $(\mathbb{G}, \mathbb{G}_T)$ with probability:

$$\epsilon' \geq \frac{\epsilon}{e(1 + q_{s_2})}.$$

  Note that C solves the hard problem after the forger $\mathcal{F}$ makes q_H queries to the H oracle, q_{s_1} queries to the $Sign$-1 oracle, q_{s_2} queries to the $Sign$-2 oracle and generates a forged signature. Also, given the forged signature, the challenger extracts the solution to the CDH problem with $\mathcal{O}(1)$ computation. Therefore the total time t taken by C in solving the hard problem is as below:

$$t' \leq t + (q_H + q_{s_1} + q_{s_2} + \mathcal{O}(1))$$

  If t' is a polynomial, it would imply that t is also a polynomial which contradicts the assumption that $(\mathbb{G}, \mathbb{G}_T)$ is a CDH group pair. This completes the proof of the theorem.

■

VI. Remark on Multi-signatures

Multi-signatures [4] is a special case of signature aggregation where the message to be signed is same for all

the parties involved in the signature generation process. Typically this calls for a coordinated way of generating the multi-signature and this is in contrast to the signatures in the transactions which are independently generated. While the BLS scheme works well for general aggregation, it suffers from a subtle flaw when used for aggregation or multi-signature in block-chain as pointed out in [8]. However, there is an easy fix by including the public key of the signer along with the message to be signed. This fix is also suggested in [8].

VII. Comparison of Schnorr, ECDSA, $\Gamma-$ Signatures and BLS based approaches

Based on the proposed protocol, we present the computational complexity and storage complexity for the generation and verification of a single signature and n signatures for Schnorr, ECDSA, $\Gamma-$ signatures and our approach in Table III. Note that, both Schnorr and ECDSA signature schemes do not allow key-aggregation nor key-split. Our signature scheme requires only one pairing operation for verifying an aggregate signature, which is more efficient than the Schnorr, ECDSA and $\Gamma-$ signatures, where the time taken increases with the number of signatures combined.

In Table IV, we provide the time taken (in microseconds (μs), milliseconds (ms) and seconds (s)) for the aggregation, verification of n signatures and also the time for verification of an n-signatures aggregation for values $n = 1, 100, 300, 400, 500$ and 1000 signatures. The different implementations on performance are tested under 2.4 GHz Intel Core i7 quad-core processor. The programming language used is Go language [2], and the programming tool is Goland 2018.2. For symmetric bilinear pairing operation, we have used the MCL Library in the implementation of our signature protocol, which is a portable and fast pairing-based cryptography library that supports optimal Ate pairing over BN curves and BLS12-381 curves [13]. The Schnorr signature is implemented on the edwards25519-curve [3], which is the current standard deployed in cryptocurrencies [10] for fast performances. We have implemented the ECDSA signature scheme on the Koblitz curve secp256k1 [11] over $\mathbb{F}_q$ as defined in FIPS 186-3 [9]. The $\Gamma-$ signature is also constructed on the secp256k1 curve as per the protocol [14]. Note that both the Schnorr and ECDSA scheme does not support aggregation. From the performance comparison, it is evident that our scheme is more efficient than the existing aggregatable $\Gamma-$ signature scheme, and also provides the added functionality of key split-up.

VIII. Conclusion

In this paper, we propose a simple key management scheme, called *key split-up* and showed that it is easy to realize that in BLS signature scheme. We also noted that both Schnorr and ECDSA does not allow any natural way to split their keys. Since it is known that aggregation is not possible/desirable for Schnorr or ECDSA, we have chosen the aggregatable $\Gamma-$ signature scheme that is the most recent. Even in this case, we note that BLS is more efficient than the Γ signature protocol.

Acknowledgment.: The authors would like to thank Dr. Rupesh Nasre for providing access to multi-core processors to conduct experimental studies as a part of this work.

References

[1] Armory. Armory Secure Wallet. https://www.bitcoinarmory.com/, 2016.

[2] Daniel J. Bernstein. The go programming language. https://golang.org/.

[3] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, pages 207–228, 2006.

[4] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, pages 435–464, 2018.

[5] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 416–432, 2003.

[6] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, pages 514–532, 2001.

[7] Shayan Eskandari, Jeremy Clark, David Barrera, and Elizabeth Stobert. A first look at the usability of bitcoin key management. *CoRR*, abs/1802.04351, 2015.

[8] Sergey Gorbunov. How not to use aggregate signatures in your blockchain. https:medium.com/@sergey_nog /how-not-to-use-aggregate-signatures -in-your-blockchain - 63e05be2cbbe.

[9] G Locke and P Gallagher. Fips pub 186-3: Digital signature standard (dss). *Federal Information Processing Standards Publication*, 3:186–3, 2009.

[10] Hartwig Mayer. Ecdsa security in bitcoin and ethereum: a research survey. *CoinFaabrik, June*, 28, 2016.

[11] C. Research. SEC 2: Recommended Elliptic Curve Domain Parameters 2010. http://www.secg.org/sec2-v2.pdf, 2010.

[12] Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

Feature	Schnorr [12]	ECDSA	$\Gamma-$Signature [14]	Our Approach
Size of 1 signature	$\|\mathbb{Z}_q\| + \|\mathbb{G}\|$	$2\|\mathbb{Z}_q\|$	$2\|\mathbb{Z}_q\|$	$\|\mathbb{G}\|$
Size of n signatures(aggregated)	$\|\mathbb{Z}_q\| + n\|\mathbb{G}\|$	(2n)$\|\mathbb{Z}_q\|$	$(n+1)\|\mathbb{Z}_q\|$	$\|\mathbb{G}\|$
Complexity of 1 signature Generation	RA	RA	RA	$2RA + PA$
Complexity of n signature Generation	$(n)RA$	$(n)RA$	(n)RA	$(2n)RA + nPA$
Complexity of 1 verification	$2RA + PA$	2RA	$2RA + PA$	2 $Pairing$
Complexity of n verification	$(n+1)RA + nPA$	(2n) RA +PA	(2n+1) RA+(n+1)PA	$(n+1)$ $Pairing$
Size of Public Key	$\|\mathbb{G}\|$	$\|\mathbb{G}\|$	$\|\mathbb{G}\|$	$\|\mathbb{G}\|$

Table III: The Efficiency comparisons of Schnorr,ECDSA, $\Gamma-$ Signatures and our Signature scheme based on BLS. Abbreviations: RA- Repeated Point Addition, PA- Point Addition

Schnorr Signature	Aggregation Cost	Verification Cost	Aggregation + Verification Cost
1 Signature	Not Possible	418.9 μs	-
100 Signatures	Not Possible	42.55 ms	-
300 Signatures	Not Possible	125.3 ms	-
400 Signatures	Not Possible	165.31 ms	-
500 Signatures	Not Possible	204.83 ms	-
1000 Signatures	Not Possible	411.36 ms	-
ECDSA Signature	Aggregation Cost	Verification Cost	Aggregation + Verification Cost
1 Signature	Not Possible	280.1 μs	-
100 Signatures	Not Possible	22.65 ms	-
300 Signatures	Not Possible	60.3 ms	-
400 Signatures	Not Possible	86.17 ms	-
500 Signatures	Not Possible	104.32 ms	-
1000 Signatures	Not Possible	211.62 ms	-
$\Gamma-$ **Signature**	Aggregation Cost	Verification Cost	Aggregation + Verification Cost
1 Signature	-	25.57 ms	25.57 ms
100 Signatures	2.59 s	2.61 s	5.2 ms
300 Signatures	7.58 s	7.62 s	15.2 s
400 Signatures	10.19 s	10.24 s	20.43 s
500 Signatures	12.81 s	12.97 s	25.78 s
1000 Signatures	27.6 s	27.67 s	55.3 s
Our Scheme	Aggregation Cost	Verification Cost	Aggregation + Verification Cost
1 Signature	-	1.12 ms	1.12 ms
100 Signatures	50.35 ms	1.10 ms	51.45 ms
300 Signatures	153.91 ms	1.12 ms	155.03 ms
400 Signatures	206.55 ms	1.17 ms	207.72 ms
500 Signatures	256.52 ms	1.13 ms	257.65 ms
1000 Signatures	504.96 ms	1.88 ms	506.84 ms

Table IV: Performance Evaluation of our Signature scheme, Schnorr Signature scheme, ECDSA scheme and Γ-signature scheme for aggregation and verification cost. Note that Schnorr and ECDSA do not support aggregation.

[13] Mitsunari Shigeo. Mcl-a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl.

[14] Yunlei Zhao. Aggregation of gamma-signatures and applications to bitcoin. Cryptology ePrint Archive, Report 2018/414, 2018. https://eprint.iacr.org/2018/414.