Trust-minimizing BDHKE-based e-cash mint using secure hardware and distributed computation

Antonín Dufka
dufkan@mail.muni.cz
Masaryk University
Brno, Czech Republic

Jakub Janků
jjanku@mail.muni.cz
Masaryk University
Brno, Czech Republic

Petr Švenda
svenda@fi.muni.cz
Masaryk University
Brno, Czech Republic

Abstract

The electronic cash (or e-cash) technology based on the foundational work of Chaum [7] is emerging as a scalability and privacy layer atop of expensive and traceable blockchain-based currencies. Unlike trustless blockchains, e-cash designs inherently rely on a trusted party with full control over the currency supply. Since this trusted component cannot be eliminated from the system, we aim to minimize the trust it requires.

We approach this goal from two angles. Firstly, we employ misuse-resistant hardware to mitigate the risk of compromise via physical access to the trusted device. Secondly, we divide the trusted device's capabilities among multiple independent devices, in a way that ensures unforgeability of its currency as long as at least a single device remains uncompromised. Finally, we combine both these approaches to leverage their complementary benefits.

In particular, we surveyed blind protocols used in e-cash designs with the goal of identifying those suitable for misuse-resistant, yet resource-constrained devices. Based on the survey, we focused on the BDHKE-based construction suitable for the implementation on devices with limited resources. Next, we proposed a new multi-party protocol for distributing the operations needed in BDHKE-based e-cash and analyzed its security. Finally, we implemented the protocol for the JavaCard platform and demonstrated the practicality of the approach by measuring its performance on a physical smartcard.

CCS Concepts

• **Security and privacy → Key management**; **Hardware-based security protocols**; *Authorization*; *Multi-factor authentication.*

Keywords

trust-minimization, smartcards, multi-party computation, e-cash

ACM Reference Format:
Antonín Dufka, Jakub Janků, and Petr Švenda. 2024. Trust-minimizing BDHKE-based e-cash mint using secure hardware and distributed computation. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30-August 2, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3664476.3670889

ARES 2024, July 30-August 2, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3670889

1 Introduction

Decades-old designs of untraceable electronic cash [7] are resurfacing as a second-layer solution to the anonymity and scalability issues in modern, blockchain-based cryptocurrencies, allowing for fully untraceable and cheap transactions with instant confirmation.

Instead of relying on the integrity of a public ledger to solve the double-spending problem, an e-cash system relies on a centralized server, called *mint*, that controls the issuance and redemption of its e-cash tokens. The ownership of a token corresponds to the knowledge of a mint's signature on a unique number that has not yet been redeemed. Once the signature is presented to the mint, it adds the signed number to its database of already used tokens, and the corresponding token may no longer be redeemed. Until a token is redeemed, it can be transacted among users in the system. To achieve untraceability of transactions, issuance and redemption of a token should be unlinkable, which is realized using blind signatures [7].

A major disadvantage of e-cash systems is their complete reliance on a trusted mint which has full control over the supply of e-cash tokens. If the mint misbehaves, it may issue an arbitrary number of new tokens. The misbehavior can be detected by using various auditing techniques, but they are only reactive measures, and in the meantime, the underlying resources could be drained.

In contrast, a more preventative measure is to minimize the risk of token forgery by reducing the trust in the mint. We consider two approaches to this trust minimization: 1) utilizing secure tamper-proof hardware for performing security-critical operations and 2) dividing the mint capabilities among multiple independent devices.

Secure tamper-proof hardware is designed to be resistant to various physical attacks, so even in the hands of untrustworthy parties, the devices should be hard to compromise. Furthermore, as the devices are typically programmed to perform just the security-critical operations and no other, they feature only a minimal attack surface, making it harder for the attacker to find and exploit its vulnerabilities even if present.

A common issue of secure tamper-proof hardware, especially smartcards, is that their computational resources are quite limited. The devices are typically equipped with specialized cryptographic coprocessors to accelerate a limited set of predefined operations, which can be performed efficiently, but other computations cannot benefit from them. For this reason, we conducted a survey of building blocks of e-cash systems, focusing on the suitability of their implementation in resource-constrained devices.

The survey revealed that two building blocks are particularly suitable for the implementation on smartcards—pairing-free blind signatures, e.g., [12, 15, 17, 29], and a non-signature blind protocol called BDHKE [32]. The former includes a whole class of blind

signature protocols that do not rely on bilinear pairings, and the latter is a blind protocol that does not support public verifiability yet provides the functionality needed in an e-cash system. Since BDHKE is the most efficient construction from the mint's side and is also compatible with the blind BLS protocol [4], a popular building block of e-cash systems, we focused on it for further analysis.

The second approach—distributed computation—allows for minimizing the trust put in the mint by splitting its capabilities among multiple independent devices in a way that all of them need to agree on an operation before it can be completed. We propose a new protocol for securely realizing the BDHKE functionality distributed among n parties, aiming at the highest security configuration that remains secure as long as there is a single non-corrupted party. We analyzed the protocol's security by reducing the problem of a token forgery to the one-more gap Diffie-Hellman problem [16] in the random oracle model [3].

Since both approaches are orthogonal, combining them yields complementary benefits and further reduces the reliance on trust. Additionally, the combination provides an option to use hardware produced by different manufacturers to address the potential threat of supply chain attacks in the form of hardware backdoors [19].

We implemented the protocol on JavaCard-based smartcards both in the single-party and distributed variants and proposed an optimization allowing a further increase in performance by offloading certain computations to the host device. We evaluated the performance of the implementation on a physical smartcard, demonstrating the practicality of the solution. The source code is publicly available and released under a permissive license[1].

Paper contributions:

- Survey of blind protocols for e-cash systems focusing on their suitability for resource-constrained devices.
- Design of a new multi-party protocol for distributing the BDHKE functionality, which is also compatible with the multi-party (blind) BLS scheme.
- Security analysis of the protocol by constructing a reduction of token forgery to the one-more gap Diffie-Hellman problem [16] in the random oracle model [3].
- Proposal of an approach for secure offloading the used hash-to-curve operation to increase performance on smartcards.
- Open-source implementation of the protocol for the JavaCard platform and its evaluation on a physical smartcard.

1.1 Related Work

The outcome of this work is related to an earlier work by Mavroudis et al. [19], who used a set of smartcards to perform joint signing, asymmetric decryption, and randomness generation. Their goal was to achieve secure computation even in the presence of backdoored components by relying on hardware produced by different manufacturers. We extend this idea to e-cash mints.

The distributed computation of blind protocols has been widely analyzed, with several designs for many constructions like blind Schnorr-based signatures [12], blind BLS [31], or blind RSA [8, 25]. Nevertheless, to the best of our knowledge, no prior work has focused on the distributed computation of BDHKE. Compared to the other distributed blind protocols, our protocol relies on the one-more gap Diffie-Hellman hardness assumption, does not require a pairing-friendly curve and features short outputs consisting of just a single group element.

Tomescu et al. proposed [30] a complex system involving a distributed e-cash mint based on Pointcheval-Sanders signatures [24] that can withstand corruption up to a third of servers by utilizing threshold cryptography and Byzantine fault tolerance protocol. Baudet et al. [2] proposed a performant and scalable Byzantine fault tolerant protocol for electronic payments that involves partially-trusted sharded authorities. Rial and Piotrowska [27] designed a distributed e-cash system that enables offline transactions and features threshold issuance of the tokens. Apart from academic publications, Fedimint[2] is a recently evolving project that utilizes blind BLS signatures [4] to distribute computation and eliminate trust in the mint. Compared to these solutions, we utilize a different approach aiming for compatibility with resource-constrained devices communicating over a star topology.

2 Preliminaries

In this section, we describe the used notation and define digital signatures and proofs of discrete logarithm equality.

2.1 Notation

A group description is a triplet $(\mathbb{G}, p, G)$, where $\mathbb{G}$ is a group of prime order p, and G is its selected generator. A group description can be generated using a probabilistic polynomial-time (PPT) algorithm GrGen that on input 1^λ outputs a group description with λ-bit prime p. We use the additive notation for the group operation and denote group elements in upper-case. Conversely, we use lower-case to denote $\mathbb{Z}_p$ elements. Uniform sampling of an element e from a non-empty set S is denoted as $e \leftarrow\$\ S$. By $\mathcal{A}(x)$, we denote the set of outputs of probabilistic algorithm $\mathcal{A}$ given input x. We reserve n for the number of protocol participants.

2.2 Digital signatures

Digital signature DS is a tuple of probabilistic polynomial-time algorithms (DS.Setup, DS.KeyGen, DS.Sign, DS.Verify) such that:

- $\mathsf{DS.Setup}(1^\lambda) \rightarrow \mathsf{par}$: On input a security parameter λ outputs a set of public parameters par which are implicitly provided to all other algorithms.
- $\mathsf{DS.KeyGen}() \rightarrow (\mathsf{sk}, \mathsf{pk})$: Generates and outputs a key pair (sk, pk), where sk is the private key and pk is the public key.
- $\mathsf{DS.Sign}(\mathsf{sk}, m) \rightarrow \sigma$: On input a private key sk and a message m, signs the message using the private key and outputs signature σ.
- $\mathsf{DS.Verify}(\mathsf{pk}, m, \sigma) \rightarrow b$: On input a public key pk, a message m, and a signature σ, output a bit b indicating whether the signature is valid with respect to the message and the public key.

A digital signature should satisfy *correctness*, i.e., it should hold for all $m \in \{0,1\}^*$ and $(\mathsf{sk}, \mathsf{pk}) \leftarrow\$\ \mathsf{DS.KeyGen}()$, $\Pr[\mathsf{DS.Verify}(\mathsf{pk}, m, \mathsf{DS.Sign}(\mathsf{sk}, m)) = 1] = 1$, and existential *unforgeability* against the chosen message attack [21].

[1] Available at https://github.com/crocs-muni/JCMint.

[2] https://fedimint.org/.

2.3 Proofs of discrete logarithm equality

Non-interactive zero-knowledge proof of discrete logarithm equality DLEQ is a pair of probabilistic polynomial-time algorithms (DLEQ.Prove, DLEQ.Verify) such that:

- DLEQ.Prove$(d, X, P) \to \pi$: On input scalar d and elements X, P, compute elements $Y = dX, Q = dP$ and output a zero-knowledge proof π of the relationship $\log_X Y = \log_P Q$.
- DLEQ.Verify$(\pi, X, Y, P, Q) \to b$: On input proof π and elements X, Y, P, Q output a bit b indicating whether the proof is valid with respect to the elements.

A non-interactive zero-knowledge proof of discrete logarithm equality should satisfy *completeness*, *soundness*, and *zero-knowledge* [21].

We will instantiate DLEQ with Chaum-Pedersen proof [10], which we briefly recall. The proof system works with a group description $(\mathbb{G}, p, G)$ and uses a hash function $H_\pi : \mathbb{G}^6 \to \mathbb{Z}_p$. The DLEQ.Prove$(d, X, P)$ first samples $r \leftarrow\$\ \mathbb{Z}_p$ and computes $Y = dX$, $Q = dP$, $A = rX$, $B = rP$, queries $c = H_\pi(X, Y, P, Q, A, B)$, computes $s = r - cd \pmod p$, and sets $\pi = (c, s)$. The DLEQ.Verify(π, X, Y, P, Q) decomposes $(c, s) = \pi$, computes $A = sX + cY$, $B = sP + cQ$, and outputs 1 if $c = H_\pi(X, Y, P, Q, A, B)$, otherwise 0.

Such a DLEQ instantiation allows, in the random oracle model [3], for proof forgery in simulations, which will be useful in reduction proof in Section 4.4. To forge a proof for inputs X, Y, P, Q, the simulator samples $s, c \leftarrow\$\ \mathbb{Z}_p^2$, computes $A = sX+cY$ and $B = sP+cQ$ and programs the oracle to output c for $H_\pi(X, Y, P, Q, A, B)$.

3 Electronic cash concepts

The concept of untraceable electronic cash was originally introduced by Chaum as an application of blind signatures [7, 9]. The proposal involves a trusted party issuing and redeeming tokens against its own local ledger, containing a set of redeemed tokens to prevent double-spending.

Mint implementations typically provide an interface for three basic operations: 1) issuing a new token, 2) redeeming a token, and 3) swapping an old token for a fresh one. The first two operations are self-explanatory. The third operation is needed to commit transactions between users. During a transaction, the payee should immediately swap the received token for a fresh one; otherwise, the payer, knowing the same information, could still redeem the token before the payee. For this reason, transactions require online interaction with the mint. Beyond these operations, other application-specific operations can be provided, but they can be constructed from these basic ones, so we do not discuss them further.

Token issuing and redeeming is typically performed in connection to a certain action performed by an external component, e.g., depositing or withdrawing the underlying collateral. To enable integration with such components without binding to a particular solution, we consider a signature-based interface to communicate these actions securely.

The interface assumes that the public keys of the mint and the external component are exchanged upon their initialization. With the interface, the external party must authorize each token issuance by creating a signature for a token request. The mint later verifies the signature, and if it is valid, a new token is created. Similarly, after the mint completes a redemption, it creates a signature confirming the successful redemption. This signature can be verified by the external party to ensure the validity of the redemption. For an illustration of such an interaction on an example usage with a custodian holding the collateral, see Figure 1.

3.1 Blind protocol

E-cash systems have been instantiated with various blind protocols, some of which rely only on private verifiability [32]. Therefore, we include this notion and define a blind protocol as follows.

A blind protocol BP is a tuple of probabilistic polynomial-time algorithms (BP.Setup, BP.KeyGen, $\{\mathsf{BP.Sign_j}\}_{j=1}^{r}$, BP.Verify), parameterized by the number of exchanged messages r, such that:

- BP.Setup$(1^\lambda) \to$ par: On input a security parameter λ outputs a set of public parameters par which are implicitly provided to all other algorithms.
- BP.KeyGen$() \to$ (sk, pk): Generates and outputs a key pair (sk, pk), where sk is the private key and pk is the public key.
- $\mathsf{BP.Sign}_j(\mathsf{st}, \mathsf{in}) \to (\mathsf{st}', \mathsf{out})$: On input a state st and input in perform a single step of the blind signing protocol and output new state st′ and output out. The state st′ is always stored by the party computing the $\mathsf{BP.Sign}_j$ procedure and input to its next invocation $\mathsf{BP.Sign}_{j+2}$, while the output out is provided to the other party which uses it as its input in computing $\mathsf{BP.Sign}_{j+1}$. The initial state of the user/issuer contains pk/sk, respectively. The user always computes the last round $\mathsf{BP.Sign}_r$ whose output out $= (m, T)$ contains the message m and the corresponding token T.
- BP.Verify$(\mathsf{sk}, m, T) \to b$: On input a *private* key sk, a message m, and a token T, output a bit b indicating whether the token is valid with respect to the message and the key.

A blind protocol should satisfy *one-more unforgeability*, just like a blind signature [7, 26], a weakened variant of *blindness* assuming an honest-but-curious signer due to lacking public verifiability, and a modified correctness allowing for verification with the private key, i.e., for every $m \in \{0,1\}^*$ and $(\mathsf{pk}, \mathsf{sk}) \leftarrow\$\ \mathsf{BP.KeyGen}()$ the exchange of messages during $\mathsf{BP.Sign}_j$ results in T such that $\Pr[\mathsf{BP.Verify}(\mathsf{sk}, m, T) = 1] = 1$.

We have surveyed blind protocols commonly used in e-cash construction with the aim of identifying the ones suitable for computation on resource-constrained devices. The original blind signature used in the e-cash proposal was based on the RSA algorithm [8, 25]. However, its outputs are inferior in size and efficiency to comparable elliptic-curve-based schemes, which can have up to 8x smaller outputs at the same 128-bit security level.

Currently, the most compact and feature-rich blind protocols are constructed using pairing-friendly elliptic curves [4, 6, 18, 24], allowing for very efficient signing. However, their verification is significantly more costly, as it involves pairing function computation, which is not feasible for resource-constrained devices.

A middle ground between these categories is offered by pairing-free elliptic-curve-based blind signatures [12, 15, 17, 29], which produce outputs with around double the size of pairing-based schemes, but with verification time of the same order of magnitude as signing. Unfortunately, due to the threat of the ROS attack [15], the schemes involve a more complex design and rely on a proof in the algebraic group model [14]. Still, these schemes are likely a viable option for resource-constrained devices.

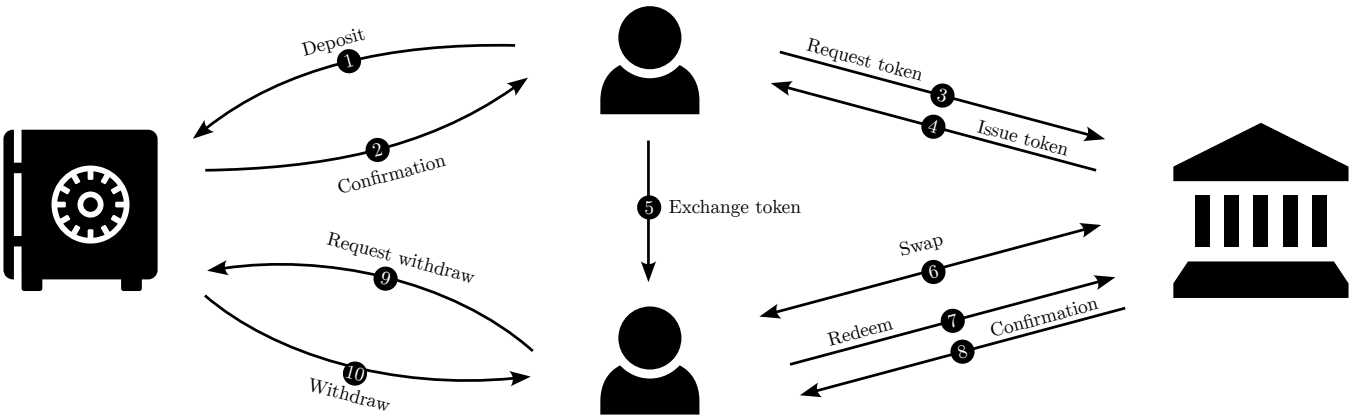

Figure 1: An example of a mint backed by an external custody service. A user wanting to enter the system first deposits money ❶ to the custody service. In response, the custody service signs ❷ the user's request for issuing an appropriate number of e-cash tokens. The user then sends ❸ this signature along with the request to the mint, which will verify it and issue the e-cash tokens ❹. Subsequently, the tokens can be transferred among users of the system ❺, who validate the tokens with the mint and swap them ❻ for fresh ones. Once a user wishes to withdraw the tokens, she requests a redemption by the mint ❼. The mint verifies the tokens and confirms the success by issuing a signature ❽. The user then provides ❾ this signature to the custody service, which in response releases ❿ an appropriate amount of the backing asset to the user.

Another alternative suitable for resource-constrained devices is blind protocols that lack public verifiability. These protocols can sometimes be constructed by transferring a pairing-based blind signature into the pairing-free setting and verifying the output by its reconstruction. An example of such a scheme is Blind Diffie-Hellman Key Exchange (BDHKE) [32], although it was not originally constructed this way. The verification of this scheme is as efficient as signing, which is even more efficient than that of pairing-free blind signatures. For these reasons, we focus on BDHKE in the rest of this work and expound on it in the following subsection.

3.1.1 Blind Diffie-Hellman Key Exchange. The BDHKE protocol was originally mentioned on a cypherpunk mailing list by David Wagner [32]. It was proposed as an alternative for the blind signing protocol based on RSA used in the original e-cash proposal, and it has been recently utilized in a project called Cashu[3], developing an e-cash system backed by Bitcoin [22].

The protocol is defined in Figure 2 as an instance of a blind protocol. To get a token using BDHKE, a user generates a random message m of λ bits, which is then hashed to a group element using $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ and blinded by adding a blinding factor rG, where r is a random scalar from $\mathbb{Z}_p$. The blinded point C is then sent to the signer, who multiplies it by its private key k and returns the resulting blinded token T'. Once the user receives the response, she can unblind it and get the token as $T = T' - rK$, where K is the public key of the signer. To verify the unblinded token later, the signer needs to be provided (m, T) and then checks whether $kH_{\mathbb{G}}(m) = T$ holds. We are not aware of a published security analysis of the protocol, so for completeness, we provide it in Appendix A.

The protocol is reminiscent of blind BLS [4], and it is virtually the same, only lacking public verifiability, as it does not utilize a group with computable bilinear pairings [5]. The only party that can verify the output is the party that controls the private key.

The non-verifiability can be resolved by extending the protocol, as Jarecki et al. [16] have shown. They have proposed a construction similar to BDHKE to which they added public verifiability using a non-interactive zero-knowledge proof of discrete logarithm equality [10]. This solves the problem only for the token recipient, as the verification requires the blinding factor, but allows to make the protocol blind even in the malicious setting.

[3] https://cashu.space/.

$\mathsf{Setup}(1^\lambda)$	$\mathsf{KeyGen}()$	$\mathsf{Sign}_1(\mathsf{st}, \mathsf{in})$
$(\mathbb{G}, p, G) \leftarrow \mathsf{GrGen}(1^\lambda)$	$k \leftarrow\$ \mathbb{Z}_p$	$m \leftarrow\$ \{0,1\}^\lambda$
Select $H_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$	$K \leftarrow kG$	$r \leftarrow\$ \mathbb{Z}_p$
$\mathsf{par} \leftarrow (\mathbb{G}, p, G, H_{\mathbb{G}}, \lambda)$	**return** (k, K)	$C \leftarrow H_{\mathbb{G}}(m) + rG$
return par		**return** $((\mathsf{st}, m, r), C)$

$\mathsf{Sign}_2(\mathsf{st}, \mathsf{in})$	$\mathsf{Sign}_3(\mathsf{st}, \mathsf{in})$	$\mathsf{Verify}(k, m, T)$
$k \leftarrow \mathsf{st}$	$(K, m, r) \leftarrow \mathsf{st}$	$V \leftarrow kH_{\mathbb{G}}(m)$
$C \leftarrow \mathsf{in}$	$T' \leftarrow \mathsf{in}$	**return** $V = T$
$T' \leftarrow kC$	$T \leftarrow T' - rK$	
return $(\perp, T')$	**return** $(\perp, (m, T))$	

Figure 2: Definition of BDHKE blind protocol.

3.2 Mint model

We model mint as a stateful system $\mathcal{M}$ parameterized by a digital signature scheme DS and a blind protocol BP. The system is initialized with its private key k, redeem confirmation private key k', external party's public key E, and an empty ledger L. The system provides three operations:

- $\mathsf{Issue}(C, \sigma) \to T$: On input a blinded challenge C and its signature σ, $\mathcal{M}$ checks if $\mathsf{DS.Verify}(E, C, \sigma) = 1$, and proceeds with the $\mathsf{BP.Sign_j}$ message exchange to produce a token T.
- $\mathsf{Redeem}(m, T) \to \sigma$: On input a message m and a signed token T, $\mathcal{M}$ checks that $m \notin L$, updates $L \leftarrow L \cup \{m\}$, and if $\mathsf{BP.Verify}(k, m, T) = 1$, outputs $\mathsf{DS.Sign}(k', m)$.
- $\mathsf{Swap}(m, T, C) \to T'$: On input a message m, a signed token T, and a challenge C, checks that $m \notin L$, updates $L \leftarrow L \cup \{m\}$, and if $\mathsf{BP.Verify}(k, m, T) = 1$, produces a new token T' using the $\mathsf{BP.Sign_j}$ message exchange.

The operations can consist of multiple sub-operations that are invoked in a sequence before obtaining the output. In case a verification within the operations fails, or the sub-operations are invoked in an incorrect order, no output may be given. DS, BP, and the public keys of k and k' are publicly available.

4 Distributing mint functionality

The risk of compromise of an e-cash system can be decreased by distributing its mint functionality among multiple independent partial mints in a way that unless all of them agree with an operation, the operation cannot be successfully completed. Such a setting requires the private key to be split among the partial mints using a secure secret sharing and the produced shares to be used by the parties in a secure multi-party computation protocol without reconstructing the complete private key in a single place.

We consider a setup of n partial mints $\mathcal{M}_i$, where each is initialized with its private key k_i, public keys of all partial mints $K_1, \ldots, K_n$ such that the mint key is defined as $K = \sum_{i=1}^{n} K_i$, analogously for the redeeming key k' and the corresponding public keys, and a public key of an external component E. Additionally, the partial mints can communicate with each other via a mediator that can arbitrarily alter or drop messages (and thus cause a denial of service). Beyond that, the partial mints behave exactly as regular mints; only DS and BP are instantiated with a multi-party protocol.

4.1 Distributing BDHKE

We propose a new protocol for secure distributed computation of BDHKE, called dBDHKE, in a way suitable for resource-constrained devices. Interestingly, the proposed solution can also be used to distribute the blind BLS protocol [4] to devices that hold key shares but do not support pairing function computation.

The distributed variant of Sign operation is from mint's side (Sign_2) the same as single-party one, since its outputs can be aggregated non-interactively, just like in the BLS scheme [31]. The aggregation is an additional task that needs to be performed by the party performing the protocol in Sign_3, who receives partial tokens from the partial mints $T'_1, \ldots, T'_n$ and aggregates them as $T' = \sum_{i=1}^{n} T'_i$ before their unblinding.

Distributing the Verify computation is more complex, as produced tokens cannot be directly verified without having access to the complete private key $k = \sum_{i=1}^{n} k_i$. Therefore, we propose a two-round protocol, during which the token is recreated and complemented by non-interactive zero-knowledge proofs of its correct construction. Using these proofs, partial mints can verify that the provided token was valid. However, since the token has been recreated during this process, it may no longer be accepted in the future,

$\mathsf{Verify}_1(\mathsf{st}, \mathsf{in})$

$((K_j)_{j=1}^n, k_i) \leftarrow \mathsf{st}$
$(m, T) \leftarrow \mathsf{in}$
$V_i \leftarrow k_i H_{\mathbb{G}}(m)$
$\pi_i \leftarrow \mathsf{DLEQ.Prove}(k_i, G, H_{\mathbb{G}}(m))$
return $((m, T, (K_j)_{j=1}^n), (V_i, \pi_i))$

$\mathsf{Verify}_2(\mathsf{st}, \mathsf{in})$

$(m, T, (K_j)_{j=1}^n) \leftarrow \mathsf{st}$
$(V_j, \pi_j)_{j=1}^n \leftarrow \mathsf{in}$
for $j \in \{1, \ldots, n\}$ **do**
  if $\mathsf{DLEQ.Verify}(\pi_j, G, K_j, H_{\mathbb{G}}(m), V_j) \neq 1$ **then**
    return 0
done
return $\sum_{j=1}^{n} V_j = T$

Figure 3: Definition of dBDHKE verification procedures.

independently of the verification result. The application utilizing the protocol needs to enforce this behavior, which is achieved in the mint case by adding the token to the ledger.

The verification protocol splits the Verify operation into two procedures ($\mathsf{Verify}_1, \mathsf{Verify}_2$), defined in Figure 3. In the first procedure, verifiers (partial mints) are initialized with their state containing their private key k_i and public keys K_j of all parties, and input consisting of a message m and a token T to verify. Using its key share k_i, each verifier computes a verifying token V_i and a DLEQ proof π_i of its correct construction. The procedure outputs a state for the next round, consisting of the verified token and public keys of other parties, and a message to transmit, consisting of the verifying token V_i and proof π_i. The messages are then transmitted to other protocol participants.

The second procedure of each party is invoked with its state from the previous round and messages of all parties as input. The procedure verifies all received proofs and checks that the reconstructed point corresponds to the token that is being verified. The result of the verification is output.

4.1.1 Correctness. The token issued with the distributed protocol is a valid token verifiable with the private key $k = \sum_{i=1}^{n} k_i$ as shown in the following equation:

$$T = T' - rK = \sum_{i=1}^{n} T'_i - rK = \sum_{i=1}^{n} k_i(H_{\mathbb{G}}(m) + rG) - rK$$
$$= k(H_{\mathbb{G}}(m) + rG) - rK = kH_{\mathbb{G}}(m) + rK - rK = kH_{\mathbb{G}}(m).$$

The distributed verification is correct since for a valid token T of message m, $T = \sum_{i=1}^{n} V_i = \sum_{i=1}^{n} k_i H_{\mathbb{G}}(m) = kH_{\mathbb{G}}(m)$, which satisfies the verification equation.

4.2 Distributing DS

To fully distribute the trust of all mint operations, even the DS.Sign during Redeem operation needs to be computed using a multi-party protocol. Multi-party signing is a common technique for

which many standard protocols exist, e.g., [4, 23]. Focusing on the compatibility with resource-constrained devices, we consider the Schnorr multi-signature scheme by Nick et al. [23], as it can be efficiently constructed and verified using smartcards. The protocol consists of two communication rounds, which can be interleaved with the verification rounds to eliminate the extra communication. In case the verification on smartcards is not needed, BLS signatures [4] may be a better solution, as they are more efficient to compute and can be aggregated non-interactively.

4.3 DOS mitigation

Since dBDHKE requires a token to be added to the ledger during verification, independently of its validity, an attacker may attempt to spam the system with invalid tokens and fill up the available memory. This is particularly troublesome for devices with limited storage, which may need to rely on various storage offloading techniques, significantly degrading the system's performance.

This problem can be addressed in an account-based system by constraining the user's number of verification attempts within a time window or by sanctioning users repeatedly attempting to verify invalid tokens. Although such systems do not enable direct tracing of users' transactions, their metadata may reveal partial information about users' balances unless additional precautions to avoid this analysis are taken.

A more private system would utilize a group-based authentication, for example, based on group signatures [11], which only proves that the accessing user is one of the eligible users but does not allow identifying which one. This setting, however, is only suitable for preventing an attack by outsiders, as eligible users cannot be distinguished from each other.

In an entirely anonymous setting, the problem can be addressed by disincentivizing spamming by increasing the cost of making a request, similarly to the concept of HashCash [1]. For a distributed BDHKE-based mint, the technique can be applied by requiring the submitted token (output of $H_{\mathbb{G}}$) to have a defined prefix. Given such a requirement, users would need to search for an input that hashes to a value with the prefix before submitting a request. A mint then could efficiently verify whether a token satisfies the requirement by recomputing the hash before proceeding with the protocol. The complexity of the search could be parameterized so that the search would take the user significantly more time than the verification.

4.4 Security properties

In this section we show that the dBDHKE protocol with DLEQ instantiated using Chaum-Pedersen proof [10] satisfies the security against *one-more forgery*, as defined in Figure 5, assuming the hardness of the one-more gap Diffie-Hellman problem [16] in the random oracle model [3]. Privacy-related properties are inherited from the base scheme since users' interactions remain the same.

In the attacker model, we assume that the initial generation and distribution of private keys and corresponding public keys were performed correctly and consistently. Then, the attacker was given private keys of all but one honest party $\mathcal{M}_1$ to simulate the corruption of $n-1$ parties. The attacker has also an oracle access to $\mathcal{M}_1$'s procedures Sign_2, Verify_1, and Verify_2, and also random oracles $H_{\mathbb{G}}$ and H_π.

4.4.1 Hardness assumption. The one-more gap Diffie-Hellman problem [16] is defined in Figure 4. A challenger provides an adversary a group description $(\mathbb{G}, p, G)$ and a public key $K = kG$, where k is known only to the challenger. Additionally, the adversary has access to three oracles: Chal, which outputs a randomly sampled challenge C from $\mathbb{G}$, sDH, which on input C outputs kC, and sDDH, which on input C, T from $\mathbb{G}$ outputs 1 if and only if $kC = T$, i.e., (C, K, T) is a valid DDH triple. The adversary wins if it outputs more distinct valid solutions kC_i to challenges than is the number of queries made to the sDH oracle.

$\mathsf{OMGDH}^{\mathcal{A}}(\lambda)$

$(\mathbb{G}, p, G) \leftarrow \mathsf{GrGen}(1^\lambda)$
$k \leftarrow\$ \mathbb{Z}_p; K \leftarrow kG$
$c \leftarrow 0; l \leftarrow 0$
$(\hat{C}_j, U_j)_{j=1}^{c}$
  $\leftarrow\$ \mathcal{A}^{\mathsf{Chal},\mathsf{sDH},\mathsf{sDDH}}(\mathbb{G}, p, G, K)$
for $j \in \{1, \ldots, c\}$ **do**
 if $\hat{C}_j \neq C_j \vee U_j \neq kC_j$ **then**
  return 0
done
return $l < c$

$\mathsf{Chal}()$

$c \leftarrow c + 1$
$C_c \leftarrow\$ \mathbb{G}$
return C_c

$\mathsf{sDH}(C)$

$l \leftarrow l + 1$
return kC

$\mathsf{sDDH}(C, T)$

return $kC = T$

Figure 4: The (static) one-more gap Diffie-Hellman game.

$\mathsf{OMUF}^{\mathcal{A}}_{\mathsf{dBDHKE}}(\lambda)$

$(\mathbb{G}, p, G, H_{\mathbb{G}}, \lambda) \leftarrow \mathsf{Setup}(1^\lambda)$
Select $H_\pi : \mathbb{G}^6 \to \mathbb{Z}_p$
$(k_i, K_i)_{i=1}^{n} \leftarrow \mathsf{KeyGen}^n()$
$L \leftarrow \{\}; l \leftarrow 0$
$\mathcal{A}^{H_{\mathbb{G}}, H_\pi, \mathcal{M}_1}(K_1, (k_j)_{j=2}^{n})$
return $l < 0$

$\mathcal{M}_1.\mathsf{Sign}_2(C)$

$l \leftarrow l + 1$
$(\mathsf{st}, T') \leftarrow \mathsf{Sign}_2(k_1, C)$
return T'

$\mathcal{M}_1.\mathsf{Verify}_1(m, T)$

if $m \in L$ **then return** $\perp$
$L \leftarrow L \cup \{m\}$
$\mathsf{st} \leftarrow ((K_j)_{j=1}^{n}, k_1)$
$\mathsf{in} \leftarrow (m, T)$
$\mathsf{st}', \mathsf{out} \leftarrow \mathsf{Verify}_1(\mathsf{st}, \mathsf{in})$
return out

$\mathcal{M}_1.\mathsf{Verify}_2(\mathsf{proofs})$

if $st' = \perp$ **then return** $\perp$
$b \leftarrow \mathsf{Verify}_2(\mathsf{st}', \mathsf{proofs})$
$\mathsf{st}' \leftarrow \perp$
if b **then** $l \leftarrow l - 1$
return b

Figure 5: The dBDHKE one-more unforgeability game.

4.4.2 Reduction. Given a PPT adversary $\mathcal{A}$ that with non-negligible probability wins the OMUF game, we construct a PPT adversary $\mathcal{B}$ who wins the OMGDH game with non-negligible probability.

$\mathcal{B}$ is initialized with a group description $(\mathbb{G}, p, G)$, the challenger's public key K_1, and Chal, sDH and sDDH oracles. $\mathcal{B}$ initializes $\mathcal{A}$ using the same group description, K_1 as the honest party's public key, and private keys of all other $n-1$ parties $k_2, \ldots, k_n$ which $\mathcal{B}$ generated. $\mathcal{B}$ answers $\mathcal{A}$'s oracle queries as follows:

- $H_{\mathbb{G}}(m)$: $\mathcal{B}$ checks whether the oracle has been queried with the same input before. If not, it invokes its Chal() oracle and its output C is programmed into the $H_{\mathbb{G}}$ oracle. Then, the programmed value is output.
- $H_{\pi}(X, Y, P, Q, A, B)$: $\mathcal{B}$ checks whether the oracle has been queried with the same input before. If not, a fresh $c \leftarrow\$ \mathbb{Z}_p$ is sampled and programmed into the H_{π} oracle. Then, the programmed value is output.
- $\mathcal{M}_1.\mathsf{Sign}_2(C)$: $\mathcal{B}$ increments its counter l, relays the request to its sDH(C) oracle, and returns the output $k_1 C$.
- $\mathcal{M}_1.\mathsf{Verify}_1(m, T)$: $\mathcal{B}$ uses its sDDH oracle to check whether $V_1' = T - \sum_{j=2}^{n} k_j H_{\mathbb{G}}(m)$ is a valid token with respect to m and the challenger's key K_1. In case it is, the verifying point V_1 is set to the extracted point V_1'. Otherwise, $\mathcal{B}$ queries the sDH oracle for it. The DLEQ proof π_1 is forged by programming H_{π}, as described in Section 2.3. In case the value for given inputs is already set, $\mathcal{B}$ aborts. Finally, (V_1, π_1) is output.
- $\mathcal{M}_1.\mathsf{Verify}_2(\text{proofs})$: $\mathcal{B}$ checks that a verification is currently in progress and proceeds according to the OMUF game. In case a verification of the token with sDDH oracle failed, yet Verify_2 output 1, abort.

Throughout $\mathcal{A}$'s execution, l corresponds to the difference between the number of queries made to the sDH oracle and the number of challenge solutions known by $\mathcal{B}$, as will be detailed in the following paragraphs. Consequently, if l turns negative and $\mathcal{B}$ does not abort, $\mathcal{B}$ is able to win the OMGDH game.

For each query to $\mathcal{M}_1.\mathsf{Sign}_2$, the l value is incremented since sDH was queried, and we have to assume that the output is not a solution to a challenge, as $\mathcal{A}$ could have used blinding.

When $\mathcal{A}$ invokes the verification sequence with a valid token, $\mathcal{B}$ is able to extract $k_1 H_{\mathbb{G}}(m)$ without performing an sDH query, as it can rely on the verification of the token using the sDDH oracle. Since the token was valid, l may be eventually decremented, but $\mathcal{B}$ can already answer the corresponding challenge.

When $\mathcal{A}$ invokes the verification sequence with an invalid token, $\mathcal{B}$ has to query the sDH oracle to provide the correct answer, but since the token is invalid, the verification will fail with overwhelming probability due to the soundness of the DLEQ system. In that case, l will not be decremented, and since m was already included in L, the reconstructed token is no longer acceptable for verification, and $\mathcal{B}$ learned the solution to the corresponding challenge.

If $\mathcal{A}$ attempts to invoke the verification with a message that is already present in L, it will immediately return without making any further queries or changes to l.

$\mathcal{A}$'s view in the simulation is indistinguishable from a real execution. When $\mathcal{A}$ terminates, $\mathcal{B}$ looks up all challenges C_i from the Chal oracle, which may have been either included in L, or unused. For the former, $\mathcal{B}$ looks up the corresponding solutions U_i obtained when simulating the $\mathcal{M}_1.\mathsf{Verify}_1$ procedure. For the latter, $\mathcal{B}$ queries its sDH oracle. Finally, $\mathcal{B}$ outputs all pairs (C_i, U_i). If the output of $\mathcal{A}$ is a valid solution to the OMUF game, then the output of $\mathcal{B}$ is a valid solution to the OMGDH problem.

Consequently, $\mathcal{B}$ wins the OMGDH whenever $\mathcal{A}$ wins the OMUF game and $\mathcal{B}$ does not abort. $\mathcal{B}$ aborts when it fails to program the H_{π} oracle with a DLEQ forgery, which occurs with probability at most q_{π}/p, where q_{π} is the number of queries made to the H_{π} oracle. The only other instance when $\mathcal{B}$ aborts is when there is an inconsistency between at least one of the DLEQ proofs and the sDDH oracle, each of which can occur with the probability of the soundness error ϵ_{π} of the DLEQ system, which is negligible. Therefore, both failure events occur only with negligible probability.

4.4.3 Avoiding trusted setup. In case the trusted setup is undesirable, the protocol can be modified to accommodate distributed key generation at the cost of an additional communication round.

Without knowing the keys $k_2, \ldots, k_n$, $\mathcal{B}$ would not be able to simulate answers for valid tokens during $\mathcal{M}_1.\mathsf{Verify}_1$ queries. In order to restore this ability, $\mathcal{B}$ would need to obtain the verification points of all parties before outputting its answer. This can be achieved by extending the protocol by an additional commitment round, during which $\mathcal{B}$ learns the verification points of other parties before having to submit its answer.

5 Implementation on smartcards

In order to obtain the maximal benefits of using tamper-proof devices, the devices need to run the full mint functionality as defined in Section 3.2, involving all three operations Issue, Redeem, and Swap, including the state management.

We implemented the BDHKE protocol on current JavaCard-based smartcards both in the single-party and the distributed variant. Since the JavaCard platform provides only a very restrictive interface to hardware-accelerated operations, we utilize techniques proposed in [20] and improved in [13] that enabled at least partial acceleration using coprocessor. However, compared to using a different smartcard platform (e.g., MULTOS) or relying on vendor-specific proprietary interfaces, the implementation is portable among JavaCard smartcards, and its source code can be fully public[4].

We assume that smartcards are provisioned in a secure environment, i.e., initially loaded with an authentic applet implementation and provided with its key and additional setup information, but afterward can be used even in an adversarial environment.

5.1 Single-party implementation

The single-party BDHKE protocol from the signer's side consists of two operations: scalar multiplication and hashing to a curve. Scalar multiplication with hardware-accelerated implementation is commonly supported by the latest JavaCard smartcards [33], so this operation can be computed close to the platform's native performance. However, hashing to a curve is more costly, as neither the algorithm nor its underlying operations are available in the JavaCard API.

There are many hash-to-curve algorithms, but a relatively straightforward one (and used by the Cashu project) is the following: compute the SHA256 hash of the input, concatenate a counter value to the hash output, and hash it again. The result is taken as an x-coordinate of a point. Afterward, the corresponding y-coordinate is computed. Since the probability of such a point being valid is only around 50%, it can fail. In that case, the counter is incremented, and the process is repeated until a valid point is found.

[4]The applet's source code is available at https://github.com/crocs-muni/JCMint.

This whole process can be implemented on a smartcard, but it is rather costly, as computing the y-coordinate requires several modular operations. These operations are relatively slow, especially when using modular arithmetic that is only partially hardware-accelerated. Although we implemented this operation fully on a smartcard, we also propose a computation offloading approach in the following section, which allows for a vastly improved performance.

5.1.1 Offloading hash-to-curve computation. Since smartcards need to be used with a typically much more computationally powerful host device, the cards can offload costly computations to their host as long as the card can efficiently verify the result.

The SHA256 computation can be done reasonably efficiently on a smartcard [33] and the costly part is the reconstruction of a point, which may need to be repeated several times due to the trial and error attempts. To avoid this, the host device can compute the output of the hash-to-curve operation and provide it to a smartcard. The smartcard can efficiently verify that the provided output is a valid point and that its x-coordinate is a hash of the input.

However, a problem with this approach is that it only works if a valid point is found in the first iteration. Otherwise, the card would not be able to know whether it received the first point that matched, which could result in an incorrect point being provided. If that would be the case, an attacker could convince the smartcard that a point has a different pre-image, causing double-spending.

The issue can be avoided by putting a constraint on the message that users use for their tokens. Messages that require more than just a single iteration will be considered invalid. This shifts more computation to users, as they have to search for such messages, but as they run much more powerful devices, they should not notice any delay as every attempt has around 50% chance to succeed.

5.2 Extension to multi-party setting

The multi-party setting adds the need for computation and verification of a zero-knowledge proofs of discrete-logarithm equality, and the addition of points to check whether the verifying points match the received token. The support for point addition is already available in the latest JavaCard-based smartcards, but both the proof computation and verification need to be implemented.

The DLEQ functionality instantiated with the Chaum-Pedersen construction [10] requires scalar multiplications, point additions, a hash function, and modular arithmetic. All these operations can be computed using the JCMathLib library [20], but the modular arithmetic exhibits significantly lower performance than a native implementation could.

During the first round of the verification protocol, only a single proof needs to be constructed. However, in the second round, all proofs of the other parties need to be verified, incurring an overhead linear in the number of parties. This computation becomes rather restrictive with larger sets of signers, as it involves costly operations.

Table 1: The average computation time (ms) and standard deviation of signing a token (Sign_2) and verifying it (Verify_1, Verify_2) in dependence on the number of involved parties with and without hash-to-curve computation offloading (opt). In the single-party setting, verification can be done in a single round; therefore Verify_2 is empty. The sum of the time of the operations corresponds to the time a mint requires to perform the Swap operation. Measured on NXP JCOP4 P71 JavaCard with a 256-bit elliptic curve.

n	1	2	3	4	5
Sign_2	52 (±1)	52 (±0)	52 (±0)	52 (±0)	52 (±0)
Verify_1 (opt)	122 (±5)	342 (±10)	342 (±10)	339 (±11)	341 (±10)
Verify_1	784 (±285)	1002 (±303)	1026 (±326)	1028 (±331)	995 (±287)
Verify_2	—	287 (±1)	535 (±1)	787 (±1)	1035 (±1)
Total (opt)	174 (±5)	681 (±12)	930 (±11)	1178 (±12)	1428 (±11)
Total	836 (±286)	1340 (±304)	1613 (±327)	1866 (±332)	2082 (±288)

6 Performance evaluation

We measured the performance of our JavaCard implementation on NXP JCOP4 P71 smartcard[5] with a 256-bit curve, both in the single-party and the distributed variant with up to 5 parties, and also with and without hash-to-curve computation offloading. We measured separately the time required to issue and verify a new token using the (d)BDHKE protocol. The sum of those operations corresponds to the mint's Swap operation. All the measurements were repeated one hundred times and averaged. The results from the smartcard are presented in Table 1 and visualized in Figure 6.

Due to the used measurement approach, the results include communication overhead involving sending an APDU packet to the smartcard and receiving the response, requiring 6-108 ms, depending on the amount of transmitted data. On the contrary, they do not include possible delays caused by the mediator computation nor computation of other parties which can run in parallel.

The implementation turned out to be particularly efficient in the single-party setting with enabled optimizations, allowing for a token swap in just 174 (± 5) ms. This is close to the hardware's native performance level, as none of the involved operations require the use of reconstructed operations from JCMathLib. In case the constraint on valid points from Section 5.1.1 cannot be applied, the hash-to-curve computation cannot be offloaded and must be computed on a card. This can result in multiple attempts of reconstructing a point, each requiring around 232 (± 18) ms, leading to high variation and computation time, resulting in 836 (± 286) ms.

In the distributed setting, swapping a token involves both dBDHKE verification rounds and also issuing a new token after successful verification. This computation requires relying on the JCMathLib library independently on the hash-to-curve offloading.

[5] A widely available and performant smartcard with the JavaCard platform.

Table 2: The number of operations needed to perform in different parts of the protocol depending on the number of parties n. The last row shows the number of operations required in the verification of single-party BDHKE.

	$\mathbb{G}$ mult	$\mathbb{G}$ add	$\mathbb{Z}_p$ mult	$\mathbb{Z}_p$ add	H2C
Sign_2	1	0	0	0	0
Verify_1	3	0	1	1	1
Verify_2	$4n-4$	$3n-3$	0	0	0
Verify	1	0	0	0	1

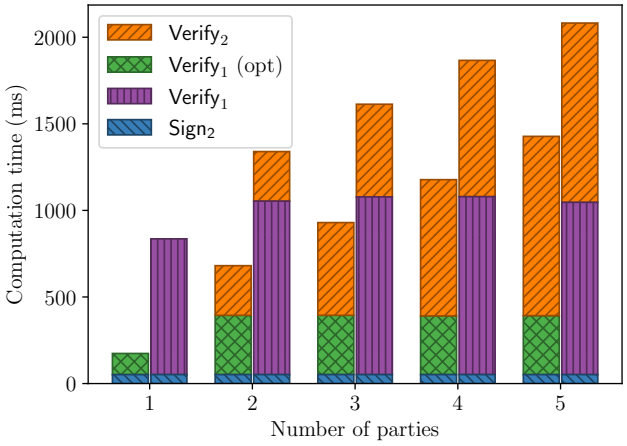

Figure 6: The time required to perform a token swap in dependence on the number of parties. Left columns show time with offloaded hash-to-curve computation and right columns without. Measured on NXP JCOP4 P71 smartcard with a 256-bit elliptic curve.

With the offloading enabled, a 2-party setup can swap a token in 681 ms and each additional party adds around 250 ms, being still under 1.5 s for 5 parties. Without the offloading, the average computation time is increased by around 700 ms independently of the number of parties.

The achieved results are acceptable for mints performing a single-digit number of transactions per minute. Furthermore, the throughput can be increased in solutions using multiple token denominations by adding smartcards dedicated to different denominations.

Still, the hardware is capable of achieving better results using a proprietary interface. We counted the number of mathematical operations used in the protocol (presented in Table 2) and estimated the potential native performance of the protocol based on the number of scalar multiplications ($\mathbb{G}$ mult), as it is by a large factor the most costly operation in the protocol. The JavaCard API exposes this operation directly; therefore, we can measure how fast it can perform. By extrapolating from this value, we estimate that in the two-party setting, the protocol should be computable on the platform at around 400 ms, and each additional party should add around 200 ms.

7 Conclusion

In this work, we have proposed an approach for trust-minimization of e-cash mints based on BDHKE, which is a very efficient yet not publicly verifiable blind protocol. The approach involves distributing the protocol computation among independent devices holding secret shares of the mint's key and computing the operations on tamper-proof devices.

Although distributed signing with BDHKE is straightforward, due to the tokens being aggregatable non-interactively, the verification is more complex, as BDHKE tokens are not publicly verifiable. Consequently, parties holding only a part of the key cannot verify it alone. To circumvent this issue, we proposed a multi-party protocol during which the token is recreated and complemented with a zero-knowledge proof of its validity.

The protocol is also of separate interest, as it is compatible with the BLS scheme, enabling the verification of multi-party BLS on resource-constrained devices that are not capable of computing the pairing function, in a setting where the verifiers hold shares corresponding to the private key.

We have implemented the protocol for the JavaCard platform and measured its performance on a physical smartcard NXP JCOP4 P71. The results show that BDHKE is the most practical in the single-party setting. However, even a two-party variant can swap a token in less than 700 ms, making the results practical for mints performing a single-digit number of token swaps per minute. Furthermore, we estimate that the two-party protocol could be computed in under half a second with native access to the platform.

As a future work, we plan to extend the approach to enable completing the computation with some fraction of parties missing by using Shamir's secret sharing [28] instead of additive secret sharing. Another perspective research direction is devising an approach for batching the DLEQ proofs to enable more efficient verification for large setups.

Acknowledgments

The authors were supported by the European Union under Grant Agreement No. 101087529 (CHESS). We also thank Martin Paljak for his comments on the initial idea and expertise with smartcards.

References

[1] Adam Back et al. 2002. Hashcash-a denial of service counter-measure. (2002).
[2] Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. 2023. Zef: low-latency, scalable, private payments. In *Proceedings of the 22nd Workshop on Privacy in the Electronic Society*. 1–16.
[3] Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*. 62–73.
[4] Alexandra Boldyreva. 2002. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *International Workshop on Public Key Cryptography*. Springer, 31–46.
[5] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.
[6] Jan Camenisch and Anna Lysyanskaya. 2004. Signature schemes and anonymous credentials from bilinear maps. In *Annual international cryptology conference*. Springer, 56–72.
[7] David Chaum. 1983. Blind Signatures for Untraceable Payments. In *Advances in Cryptology*, David Chaum, Ronald L. Rivest, and Alan T. Sherman (Eds.). Springer US, Boston, MA, 199–203.
[8] David Chaum. 1985. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM* 28, 10 (1985), 1030–1044.

[9] David Chaum, Amos Fiat, and Moni Naor. 1990. Untraceable electronic cash. In *Advances in Cryptology—CRYPTO'88: Proceedings 8*. Springer, 319–327.

[10] David Chaum and Torben Pryds Pedersen. 1992. Wallet databases with observers. In *Annual international cryptology conference*. Springer, 89–105.

[11] David Chaum and Eugène Van Heyst. 1991. Group signatures. In *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*. Springer, 257–265.

[12] Elizabeth Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. 2023. Snowblind: A Threshold Blind Signature in Pairing-Free Groups. In *Annual International Cryptology Conference*. Springer, 710–742.

[13] Antonín Dufka and Petr Švenda. 2023. Enabling efficient threshold signature computation via java card API. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*. 1–10.

[14] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. 2018. The algebraic group model and its applications. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*. Springer, 33–62.

[15] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. 2020. Blind Schnorr signatures and signed ElGamal encryption in the algebraic group model. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*. Springer, 63–95.

[16] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. 2014. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 233–253.

[17] Julia Kastner, Julian Loss, and Jiayu Xu. 2022. The abe-okamoto partially blind signature scheme revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 279–309.

[18] Veronika Kuchta and Mark Manulis. 2015. Rerandomizable threshold blind signatures. In *Trusted Systems: 6th International Conference, INTRUST 2014, Beijing, China, December 16-17, 2014, Revised Selected Papers 6*. Springer, 70–89.

[19] Vasilios Mavroudis, Andrea Cerulli, Petr Svenda, Dan Cvrcek, Dusan Klinec, and George Danezis. 2017. A Touch of Evil: High-Assurance Cryptographic Hardware from Untrusted Components. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 1583–1600. https://doi.org/10.1145/3133956.3133961

[20] Vasilios Mavroudis and Petr Svenda. 2020. JCMathLib: wrapper cryptographic library for transparent and certifiable JavaCard applets. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 89–96.

[21] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. 2018. *Handbook of applied cryptography*. CRC press.

[22] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.

[23] Jonas Nick, Tim Ruffing, and Yannick Seurin. 2021. MuSig2: Simple two-round Schnorr multi-signatures. In *Annual International Cryptology Conference*. Springer, 189–221.

[24] David Pointcheval and Olivier Sanders. 2016. Short randomizable signatures. In *Topics in Cryptology-CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29-March 4, 2016, Proceedings*. Springer, 111–126.

[25] David Pointcheval and Jacques Stern. 1996. Provably secure blind signature schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 252–265.

[26] David Pointcheval and Jacques Stern. 2000. Security arguments for digital signatures and blind signatures. *Journal of cryptology* 13 (2000), 361–396.

[27] Alfredo Rial and Ania M Piotrowska. 2023. Compact and Divisible E-Cash with Threshold Issuance. *arXiv preprint arXiv:2303.08221* (2023).

[28] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[29] Stefano Tessaro and Chenzhi Zhu. 2022. Short pairing-free blind signatures with exponential security. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 782–811.

[30] Alin Tomescu, Adithya Bhat, Benny Applebaum, Ittai Abraham, Guy Gueta, Benny Pinkas, and Avishay Yanai. 2022. UTT: Decentralized ecash with accountable privacy. *Cryptology ePrint Archive* (2022).

[31] Duc Liem Vo, Fangguo Zhang, and Kwangjo Kim. 2003. A new threshold blind signature scheme from pairings. In *SCIS2003*. SCIS, 233–238.

[32] David Wagner. 1996. Chaumian ecash without RSA. https://cypherpunks.venona.com/date/1996/03/msg01848.html. (accessed on 2024-01-18).

[33] Petr Švenda, Rudolf Kvašňovský, Imrich Nagy, and Antonín Dufka. 2022. JCAlgTest: Robust Identification Metadata for Certified Smartcards.. In *SECRYPT*. 597–604.

$\mathsf{OMUF}^{\mathcal{A}}_{\mathsf{BDHKE}}(\lambda)$

$(\mathbb{G}, p, G, H_{\mathbb{G}}, \lambda) \leftarrow \mathsf{Setup}(1^{\lambda})$

$(k, K) \leftarrow \mathsf{KeyGen}(); l \leftarrow 0; L \leftarrow \{\}$

$(m_j, U_j)_{j=1}^{l+1} \leftarrow\!\!\$\ \mathcal{A}^{H_{\mathbb{G}}, H_{\pi}, \mathcal{M}}(K)$

for $j \in \{1, \ldots, l+1\}$ **do**

  if $m_j \in L \vee kH_{\mathbb{G}}(m_j) \neq U_j$ **then return** 0

  $L \leftarrow L \cup \{m_j\}$

done

return 1

$\mathcal{M}.\mathsf{Sign}_2(C)$

$l \leftarrow l + 1$

$(\mathsf{st}, T') \leftarrow \mathsf{Sign}_2(k, C)$

return T'

$\mathcal{M}.\mathsf{Verify}(m, T)$

return $\mathsf{Verify}(k, m, T)$

Figure 7: The BDHKE one-more unforgeability game.

A BDHKE security analysis

The BDHKE protocol satisfies *blindness* assuming honest-but-curious signer. It follows from the fact that in the Sign_2 the signer receives only random elements from $\mathbb{G}$, and, when the signer behaves according to the protocol, it receives only $kH(m) = T$ during verification, which yields no information about the corresponding signing query. The honest behavior can be enforced in the malicious setting by requiring a DLEQ proof to be provided along the output of Sign_2.

The *one-more unforgability* of BDHKE is achieved in the malicious setting, assuming the hardness of one-more gap Diffie-Hellman problem. Given a PPT adversary $\mathcal{A}$ that with non-negligible probability wins the OMUF game (Figure 7), we construct a PPT adversary $\mathcal{B}$ who wins the OMGDH game with non-negligible probability.

$\mathcal{B}$ is initialized with a group description $(\mathbb{G}, p, G)$, the challenger's public key K, and Chal, sDH and sDDH oracles. $\mathcal{B}$ initializes $\mathcal{A}$ using the same group description and K as the public key. $\mathcal{B}$ answers $\mathcal{A}$'s oracle queries as follows:

- $H_{\mathbb{G}}(m)$: $\mathcal{B}$ checks whether the oracle has been queried with the same input before. If not, it invokes its Chal() oracle and the output C is programmed into the oracle. Then, the programmed value is output.
- $\mathcal{M}.\mathsf{Sign}_2(C)$: $\mathcal{B}$ increments its counter l, relays the request to its $\mathsf{sDH}(C)$ oracle, and returns the output kC.
- $\mathcal{M}.\mathsf{Verify}(m, T)$: $\mathcal{B}$ checks using sDDH whether T is a valid token with respect to m and K, and outputs the result.

For each query to $\mathcal{M}.\mathsf{Sign}_2$, the l value is incremented since sDH was queried. When $\mathcal{A}$ invokes the verification oracle, the sDDH oracle is used to provide the answer, so no additional sDH query was needed. $\mathcal{A}$'s view in the simulation is indistinguishable from a real execution. When $\mathcal{A}$ terminates, $\mathcal{B}$ looks up all the remaining challenges from the Chal oracle and solves them using the sDH oracle. Then, $\mathcal{B}$ outputs all challenge-solution pairs. If the output of $\mathcal{A}$ is a valid solution to the OMUF game, then the output of $\mathcal{B}$ is a valid solution to the OMGDH problem. Consequently, $\mathcal{B}$ wins the OMGDH whenever $\mathcal{A}$ wins the OMUF game.