# Fast Privacy-Preserving Punch Cards

Saba Eskandarian
Stanford University

## ABSTRACT

Loyalty programs in the form of punch cards that can be redeemed for benefits have long been a ubiquitous element of the consumer landscape. However, their increasingly popular digital equivalents, while providing more convenience and better bookkeeping, pose a considerable risk to consumer privacy. This paper introduces a privacy-preserving punch card protocol that allows firms to digitize their loyalty programs without forcing customers to submit to corporate surveillance. We also present a number of extensions that allow our scheme to provide other privacy-preserving customer loyalty features.

Compared to the best prior work, we achieve a 14× reduction in the computation and a 25× reduction in communication required to perform a "hole punch," a 62× reduction in the communication required to redeem a punch card, and a 394× reduction in the computation time required to redeem a card. Much of our performance improvement can be attributed to removing the reliance on pairings present in prior work, which has only addressed this problem in the context of more general loyalty systems. By tailoring our scheme to punch cards and related loyalty systems, we demonstrate that we can reduce communication and computation costs by orders of magnitude.

## 1 INTRODUCTION

Punch cards that can be redeemed for rewards after a number of purchases are a widely-used incentive for customer loyalty. Although these time-tested loyalty schemes remain popular, they are increasingly being replaced with digital equivalents that reside in mobile apps instead of physical wallets. The benefits of going digital for business owners include stronger defenses against counterfeit cards, a more convenient customer experience, and better bookkeeping around the popularity and efficacy of their loyalty program [11, 27].

Unfortunately, digital loyalty programs also introduce a myriad new opportunities for customers' privacy to be violated [11, 32], e.g., by linking customer behavior across transactions. This kind of tracking can be conducted by the business itself, a third-party loyalty service, or a malicious actor who gains access in a data breach. Thus any firm who wants to protect customer privacy should attempt to ensure that its digital loyalty program does not collect unnecessary data. But is it possible to digitize the traditional punch card without damaging customer privacy?

One approach to this problem is via standard anonymous credential techniques [13, 14, 16]. Ecash systems [2, 12] or even the uCentive system [34], which is specifically designed for loyalty programs, can be used to give a customer an unlinkable token for each purchase. However, storage and computation costs to hold and redeem a token in these systems must be linear in the number of "hole punches" a customer acquires.

A recent line of work, beginning with the Black Box Accumulation (BBA) of Jager and Rupp [29], removes this linear dependence on the number of hole punches. Although individual hole punches are unlinkable in the original BBA scheme, the processes of issuing and redeeming a punch card are not. This shortcoming is rectified in the later BBA+ and Updatabale Anonymous Credential Systems (UACS) works by Hartung et al. [28] and Blomer et al. [5], as well as the recent improvements of Bobolz et al. [6], all of which additionally extend the idea of black box accumulation to support a broader set of functionalities.

This work introduces new protocols specifically designed to support privacy-preserving digital punch cards. By focusing specifically on the requirements of punch cards and similar points-based loyalty programs, we are able to make both qualitative and quantitative improvements over prior work. Unlike the works listed above, our main protocol does not rely on pairings, enabling significant performance improvements. Moreover, by stepping away from previous abstractions used for punch cards, we can handle punch card issuance *non-interactively*, meaning that a customer can generate a new, unpunched card without any interaction with the server. As an ancillary benefit, this removes a potential denial of service opportunity in prior systems, where a customer could register many punch cards without actually needing to earn any punches.

In terms of performance, our scheme reduces the client side computation required to generate a new punch card by 280× compared to prior work (in addition to not requiring interaction with the server), reduces the total client and server computation times to perform a card punch by 14×, and reduces the time to redeem a card by 394×. Communication costs to punch and redeem a card are also reduced by 25× and 62×, respectively.

Our core protocol is quite simple. To generate a punch card, a client picks a random secret and hashes it to a point in an elliptic curve group using a hash function modeled as a random oracle [3, 21]. To receive a hole punch, the client masks this group element and sends it to the server, who sends it back raised to a server-side secret value, along with a proof that this was done honestly. Finally, after several punches, the client redeems the card by sending the unmasked version along with the initial random secret to the server. The server checks that the group element submitted matches the hash of the random secret raised to the appropriate exponent. It also checks that the punch card being redeemed has not been redeemed before. Since the server is not involved in card issuance and only ever sees separately masked versions of the card, it cannot link a redeemed card to any past transaction. We prove, in the Algebraic Group Model (AGM) [23], that a malicious customer cannot successfully claim more rewards that it is entitled to.

We also present a number of extensions to our main scheme that allow us to handle variations on the typical punch card. For example, we can handle special promotions where users get multiple punches, programs where purchases receive a fixed number of points instead of a single punch, and even private ticketing systems. Our most involved extension allows customers to merge the points on two punch cards without revealing anything to the server about the individual punch cards being merged. This extension uses pairings, but it still maintains the other advantages of our protocol and outperforms prior work, albeit by a smaller margin.

Our schemes are implemented in Rust with an Android wrapper for testing on mobile devices, and all our code and raw performance data are open source at https://github.com/SabaEskandarian/PunchCard.

## 2 DESIGN GOALS

This section describes our goals for a punch card scheme. We give security definitions and contrast the goals of our work with those of closely related works.

### 2.1 Functionality Goals

A punch card scheme consists of three components. First, a client running on a customer's phone should be able to create a new punch card. Next, the client and a server running a loyalty program can interact in order for the server to give the client a "hole punch." Finally, a client can submit a completed punch card to the server for verification, and the server will accept valid punch cards that have not already been redeemed. The server keeps a database DB of previously redeemed cards to make sure a client doesn't redeem the same card multiple times. After verifying a card, the server can give the client some out-of-band reward. In general, each of these steps can be a multi-round interactive protocol between the two parties. However, since all our protocols involve exactly one round, we present the syntax of a punch card scheme below as consisting of individual algorithms instead of interactive protocols.

A *punch card scheme* defined with respect to a security parameter $\lambda$ is defined as follows.

- $\mathsf{ServerSetup}(1^\lambda) \rightarrow \mathsf{sk}, \mathsf{pk}, \mathsf{DB}$: On input a security parameter $\lambda$, the initial server setup produces server public and secret keys, as well as an empty database to record previously redeemed punch cards.
- $\mathsf{Issue}(1^\lambda) \rightarrow \mathsf{psk}, p$: On input a security parameter $\lambda$, the Issue algorithm generates new punch card $p$ and a punch card secret psk.
- $\mathsf{ServerPunch}(\mathsf{sk}, \mathsf{pk}, p) \rightarrow p', \pi$: On input the server keys and a punch card, ServerPunch outputs an updated punch card $p'$ and a proof $\pi$ that the punch card $p$ was updated correctly.
- $\mathsf{ClientPunch}(\mathsf{pk}, \mathsf{psk}, p, p', \pi) \rightarrow \mathsf{psk}', p'' \text{or} \bot$: Given the public key, a punch card secret psk, the accompanying punch card $p$, a server-updated punch card value $p'$, and a proof $\pi$, ClientPunch outputs an updated secret psk$'$ and card $p''$ if the proof $\pi$ is accepted and $\bot$ otherwise.
- $\mathsf{ClientRedeem}(\mathsf{psk}, p) \rightarrow \mathsf{psk}', p'$: Given a punch card secret psk and the corresponding punch card $p$, ClientRedeem outputs an updated secret psk$'$ and card $p'$ that are ready to be sent to the server for redemption.
- $\mathsf{ServerVerify}(\mathsf{sk}, \mathsf{pk}, \mathsf{DB}, \mathsf{psk}, p, n) \rightarrow 1/0, \mathsf{DB}'$: on input the server keys, redeemed card database, a punch card, the accompanying secret, and an integer $n \in \mathbb{Z}$ determining the required number of punches for redemption, ServerVerify outputs a bit determing whether or not the punch card is accepted and an updated database DB$'$.

Correctness for a punch card scheme is defined in a straightforward way. An honestly generated punch card that has received $n$ punches should be accepted by an honest server. This should hold true even after many punch cards have been generated and redeemed.

*Definition 2.1 (Correctness).* We say that a punch card scheme is *correct* if for

$$\mathsf{sk}, \mathsf{pk}, \mathsf{DB}_0 \leftarrow \mathsf{ServerSetup}(1^\lambda)$$

and any $n \in \mathbb{Z}$, the following set of operations, repeated sequentially $N = \mathrm{poly}(\lambda)$ times, results in $b_j = 1$ for all $j \in [N]$ with all but negligible probability in $\lambda$.

```
$(\mathsf{psk}_0, p_0) \leftarrow \mathsf{Issue}(1^\lambda)$
for $i \in [n]$ :
  $p_i', \pi_i \leftarrow \mathsf{ServerPunch}(\mathsf{sk}, \mathsf{pk}, p_i)$
  $\mathsf{psk}_{i+1}, p_{i+1} \leftarrow \mathsf{ClientPunch}(\mathsf{pk}, \mathsf{psk}_i, p_i, p_i', \pi_i)$
$\mathsf{psk}, p \leftarrow \mathsf{ClientRedeem}(\mathsf{psk}_n, p_n)$
$b_j, \mathsf{DB}_{j+1} \leftarrow \mathsf{ServerVerify}(\mathsf{sk}, \mathsf{pk}, \mathsf{DB}_j, \mathsf{psk}, p, n)$
```

The functionality we desire from our punch cards is at a high level similar to that offered by black box accumulation (BBA) [29]. Although we offer a similar functionality, we will do so with stronger security guarantees and significantly improved performance. On the other hand, BBA+ [28], UACS [5], and Bobolz et al. [6] offer additional features that might be useful in other kinds of loyalty programs, such as reducing balances and partially spending accrued rewards. These features enable other applications, but, as described in Section 1, they render the solutions less effective for the original punch card problem. Bobolz et al. introduce the possibility of recovering from a partially completed spend that gets interrupted mid-protocol, e.g., due to a communication or hardware fault. Our scheme avoids the potential for this problem entirely because redemption only requires a single message from the client to the server.

One way in which our setting differs fundamentally that of BBA+, UACS, and Bobolz et al. is the way in which we prevent a punch card from being redeemed more than once. In our setting, the server has access to a database of all previously redeemed cards when deciding whether or not to accept a new punch card submitted for verification. BBA+, UACS, and Bobolz et al. consider an *offline double spending* scenario where the server may not have access to such a database but must be able to identify clients who have double spent punch cards after the fact. We do not pursue this goal for three reasons, listed in order of increasing importance below.

(1) Not necessary: point-of-sale terminals often require an internet connection to work, so synchronizing spent punch cards between different locations of a firm with multiple branches can happen online with less performance cost than an offline verification approach.
(2) Prohibitively expensive: the performance cost of checking whether a punch card was double spent in prior work is

```
REALPRIV($\lambda, \mathcal{A}$):
1. $T \leftarrow \{\}$
2. $c \leftarrow 0$
3. $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathcal{A}_1(\lambda)$
4. $b \leftarrow \mathcal{A}_2^{O_{\mathsf{issue}}, O_{\mathsf{punch}}, O_{\mathsf{redeem}}}(\lambda)$
5. Output $b$
```

The experiment REALPRIV($\lambda, \mathcal{A}$) makes use of the following oracles, which have access to shared state $T$ keeping track of issued punch cards and the public key pk, subject to the restriction that $O_{\mathsf{redeem}}$ is only called once on each input id.

```
$O_{\mathsf{issue}}(\lambda)$:
1. $\mathsf{psk}, p \leftarrow \mathsf{Issue}(1^\lambda)$
2. $T[c] \leftarrow (\mathsf{psk}, p)$
3. $c \leftarrow c + 1$
4. Output $c, p$

$O_{\mathsf{punch}}(\mathsf{id}, p', \pi)$:
1. if $\mathsf{id} \notin T$, output $\perp$
2. $(\mathsf{psk}, p) \leftarrow T[\mathsf{id}]$
3. $(\mathsf{psk}', p'') \leftarrow \mathsf{ClientPunch}(\mathsf{pk}, \mathsf{psk}, p, p', \pi)$
4. if $(\mathsf{psk}', p'') \neq \perp$, then $T[\mathsf{id}] \leftarrow (\mathsf{psk}', p'')$
5. Output $p''$

$O_{\mathsf{redeem}}(\mathsf{id})$:
1. if $\mathsf{id} \notin T$, output $\perp$
2. $(\mathsf{psk}, p) \leftarrow T[\mathsf{id}]$
3. $\mathsf{psk}', p' \leftarrow \mathsf{ClientRedeem}(\mathsf{psk}, p)$
4. Output $\mathsf{psk}', p'$
```

**Figure 1: Real privacy experiment**

```
IDEALPRIV($\lambda, \mathcal{A}, \mathcal{S}$):
1. $T \leftarrow \{\}$
2. $c \leftarrow 0$
3. $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathcal{A}_1(\lambda)$
4. $b \leftarrow \mathcal{A}_2^{O_{\mathsf{issue}}, O_{\mathsf{punch}}, O_{\mathsf{redeem}}}(\lambda)$
5. Output $b$
```

The experiment IDEALPRIV($\lambda, \mathcal{A}, \mathcal{S}$) makes use of the following oracles, which have access to shared state $T$ keeping track of issued punch cards, the public key pk, and $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$, subject to the restriction that $O_{\mathsf{redeem}}$ is only called once on each input id.

```
$O_{\mathsf{issue}}(\lambda)$:
1. $p \leftarrow \mathcal{S}_1(1^\lambda)$
2. $T[c] \leftarrow (0, p)$
3. $c \leftarrow c + 1$
4. Output $c, p$

$O_{\mathsf{punch}}(\mathsf{id}, p', \pi)$:
1. if $\mathsf{id} \notin T$, output $\perp$
2. $(c_{\mathsf{id}}, p) \leftarrow T[\mathsf{id}]$
3. $p'' \leftarrow \mathcal{S}_2(\mathsf{pk}, p, p', \pi)$
4. if $p'' \neq \perp$, then $T[\mathsf{id}] \leftarrow (c_{\mathsf{id}} + 1, p'')$
5. Output $p''$

$O_{\mathsf{redeem}}(\mathsf{id})$:
1. if $\mathsf{id} \notin T$, output $\perp$
2. $(c_{\mathsf{id}}, p) \leftarrow T[\mathsf{id}]$
3. $\mathsf{psk}', p' \leftarrow \mathcal{S}_3(\mathsf{sk}, c_{\mathsf{id}})$
4. Output $\mathsf{psk}', p'$
```

**Figure 2: Ideal privacy experiment**

prohibitive, requiring at least one exponentiation for each previously redeemed punch card. This would be about 8 orders of magnitude slower than the hash table lookup required in our setting (as measured on our evaluation setup).

(3) Requires real-world identity: identifying the human user who double spent a punch card in a way that the person can be penalized requires some notion of real-world identity tied to the punch card client. This means that any loyalty system providing such a feature would require a user's real-world identity in order to operate. This violates our original goal of making a punch card loyalty program digital with no damage to user privacy.

## 2.2 Security Goals

At a high level, a punch card scheme must provide two kinds of security guarantees. First, it must protect client privacy such that the server learns nothing from messages sent by the client. Second, it must be sound in that no client can redeem more rewards than it has honestly accrued through valid hole punches authorized by the server.

We define privacy using a simulation-based definition. This means that in order for privacy to be satisfied, there must exist a *simulator* algorithm that can generate the view of the punch card server without access to client-side secrets. Informally, if the server can't distinguish between the output of the simulator and a real client, then it surely can't learn anything from interacting with a real client because it could have received the same information by running the simulator on its own.

Our privacy definition defines real and ideal privacy experiments, both of which begin with the challenger initializing an empty table $T$ mapping unique integer identifiers to punch cards and a counter $c \leftarrow 0$ that is incremented each time a new punch card is issued. The adversary is allowed to pick server secret and public keys (sk, pk), and then it is allowed to interact with oracles $O_{\mathsf{issue}}$, $O_{\mathsf{punch}}$, and $O_{\mathsf{redeem}}$ which play the role of the client in the punch card scheme. In the real privacy experiment, these oracles act as wrappers around the Issue, ClientPunch, and ClientRedeem functions, simply calling the functions on the requested punch card (identified by an id number chosen at issuance) and performing bookkeeping when punch cards are issued, updated, or redeemed. The ideal privacy experiment replaces each of these functions with calls to simulator algorithms $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ which *have no access* to punch card secrets. At the end of each experiment, the adversary outputs a distinguishing bit $b$.

*Definition 2.2 (Privacy).* Let $\Pi$ be a punch card scheme. Then for a security parameter $\lambda$, and for every adversary $\mathcal{A}$ made up of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$, there exists a simulator $\mathcal{S}$ made up of algorithms $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ such that the outputs of the experiments REALPRIV($\lambda, \mathcal{A}$) (Figure 1) and IDEALPRIV($\lambda, \mathcal{A}, \mathcal{S}$) (Figure 2) are computationally indistinguishable.

In particular, we say that a punch card scheme $\Pi$ has *privacy* if there exists a negligible function $\mathsf{negl}(\cdot)$ such that for any efficient

```
$\underline{\mathsf{SOUND}(\lambda, \mathcal{A})}$:
1. $\mathsf{sk}, \mathsf{pk}, \mathsf{DB} \leftarrow \mathsf{ServerSetup}(1^\lambda)$
2. $c_{\mathsf{punch}} \leftarrow 0$
3. $c_{\mathsf{redeem}} \leftarrow 0$
3. $\mathcal{A}^{O_{\mathsf{punch}}, O_{\mathsf{redeem}}}(\lambda, \mathsf{pk})$
4. if $c_{\mathsf{redeem}} > c_{\mathsf{punch}}$, output 1. Otherwise, output 0.
```

The experiment $\mathsf{SOUND}(\lambda, \mathcal{A})$ makes use of the following oracles, which all have access to the shared state $c_{\mathsf{punch}}, c_{\mathsf{redeem}}, \mathsf{sk}, \mathsf{pk}, \mathsf{DB}$.

```
$\underline{O_{\mathsf{punch}}(p)}$:
1. $p', \pi \leftarrow \mathsf{ServerPunch}(\mathsf{sk}, \mathsf{pk}, p)$
2. $c_{\mathsf{punch}} \leftarrow c_{\mathsf{punch}} + 1$
3. Output $(p', \pi)$
$\underline{O_{\mathsf{redeem}}(\mathsf{psk}, p, n)}$:
1. $b, \mathsf{DB}' \leftarrow \mathsf{ServerVerify}(\mathsf{sk}, \mathsf{pk}, \mathsf{DB}, \mathsf{psk}, p, n)$
2. if $b = 1$:
3.     $c_{\mathsf{redeem}} \leftarrow c_{\mathsf{redeem}} + n$
4.     $\mathsf{DB} \leftarrow \mathsf{DB}'$
5. Output $b$
```

**Figure 3: Soundness experiment**

adversary $\mathcal{A}$, we have

$$\Big|\Pr\big[\mathsf{REALPRIV}(\lambda, \mathcal{A}) = 1\big] - \Pr\big[\mathsf{IDEALPRIV}(\lambda, \mathcal{A}, \mathcal{S}) = 1\big]\Big| < \mathsf{negl}(\lambda).$$

Our soundness definition resembles that of BBA [29], which requires that a malicious client can only redeem as many punches as it has accrued. Aside from modifying the syntax of the definition to match our own, we have also modified it to allow the adversary to interleave hole punches and redemptions instead of requiring that all redemptions occur at the end of the protocol.

*Definition 2.3 (Soundness).* Let $\Pi$ be a punch card scheme. Then for a security parameter $\lambda$ and adversary $\mathcal{A}$, we define the soundness experiment $\mathsf{SOUND}(\lambda, \mathcal{A})$ in Figure 3. We say that a punch card scheme $\Pi$ satisfies *soundness* if there exists a negligible function $\mathsf{negl}(\cdot)$ such that for any efficient adversary $\mathcal{A}$, we have

$$\Pr[\mathsf{SOUND}(\lambda, \mathcal{A}) = 1] < \mathsf{negl}(\lambda).$$

As in BBA, this definition does not capture whether or not a client can transfer value from one punch card to another or merge separate, partially filled punch cards to redeem a single, larger card. In fact, it is not entirely clear if this kind of card merging is a malicious behavior to be avoided or a beneficial feature to be desired. This kind of merging appears to be difficult to do in our main construction, but we show how to extend our scheme to allow a limited degree of merging in Section 4.

## 3 PRIVACY-PRESERVING PUNCH CARDS

This section describes our main punch card scheme. In addition to its quantitative improvements over prior work, which we measure in Section 5, our scheme has a number of other desirable properties:

- Whereas all prior works make use of pairings, either because they rely on Groth-Sahai proofs [26] or Pointcheval-Saunders signatures [36], our punch card scheme does not require pairings.
- We require no communication at all to issue a new punch card – a client can do this on its own without server involvement. This removes a potential denial of service opportunity present in prior work, where a client could initiate a number of punch cards without making any purchases, thereby making the server incur unnecessary storage and computation at no cost to the malicious client.
- Our redemption process involves a client sending a single message to the server, so there is no potential for the process to be interrupted mid-protocol and no need for a recovery process of the form proposed by Bobolz et al. [6].

### 3.1 Main Construction

**A basic scheme**. We will begin with a bare-bones version of our scheme that provides neither privacy nor soundness. From this starting point, we will gradually build up to our actual scheme. Throughout, we will work in a group $G$ of prime order $q$.

To set up the initial scheme, the server chooses a secret $\mathsf{sk} \in \mathbb{Z}_q$, and a client chooses a group element $p_0 \xleftarrow{R} G$ to represent the punch card. To receive a hole punch, the client sends $p_i$ to the server, who returns $p_{i+1} \leftarrow p_i^{\mathsf{sk}}$. To redeem a card after $n$ punches, the client submits $p_0$ and $p_n$ to the server, who accepts if $p_n = p_0^{\mathsf{sk}^n}$ and $p_0$ has not been previously used in a redeemed card.

**Adding privacy**. The scheme above clearly provides no privacy because the server can link the different times it sees a punch card. We can make punches made on the same card unlinkable by only sending the server *masked* versions of the punch card, in a way reminiscent of standard oblivious PRF constructions [22, 35]. The punch card is always masked with a fresh value $m \xleftarrow{R} \mathbb{Z}_q$ before being sent to the server, so the server only sees $p' \leftarrow p^m$, not $p$ itself. The mask $m$ is removed (via exponentiation by $1/m$) before the next mask is applied. This means that the server sees a different random group element each time it punches a card. Moreover, an honest server only sees a random group element $p \in G$ and $p^{\mathsf{sk}^n}$ at redemption time.

Unfortunately, this does not actually suffice to provide privacy against an actively malicious server. Consider a malicious server who always follows the scheme above, but during one hole punch (for a client it later wishes to re-identify) it uses a different secret $\mathsf{sk}' \xleftarrow{R} \mathbb{Z}_q$ so that $\mathsf{sk} \neq \mathsf{sk}'$ except with negligible probability. Then when an unsuspecting client attempts to redeem its punch card, instead of submitting $p, p^{\mathsf{sk}^n}$, it really submits $p, p^{\mathsf{sk}^{n-1}\mathsf{sk}'}$, allowing the server to identify it.

We can handle the attack above by having the server give a zero knowledge proof of knowledge that it has honestly punched a card. To facilitate this, we require the server setup to also output a public key $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$, for some publicly known generator $g \in G$. Then the server can prove at punching time that it is returning a punch card $p'$ such that $p' = p^{\mathsf{sk}}$, i.e., that $p, \mathsf{pk}, p'$ form a DDH tuple [20]. This can be proven efficiently with a generic Chaum-Pedersen proof [17] made non-interactive in the random oracle model [3, 21].

The server generates the proof $\pi$ and sends it to the client along with the punched card $p_{i+1}$. The client rejects the updated card if the proof does not verify. We denote proofs using the notation of Camenisch and Stadler [15], where $ZKPK\{(\mathsf{sk}), \mathsf{pk} = g^{\mathsf{sk}}, p' = p^{\mathsf{sk}}\}$ represents the Chaum-Pedersen proof, and require the standard zero knowledge and existential soundness properties [7].

**Adding soundness**. The two modifications above ensure that the scheme provides privacy, but it still fails to provide soundness, as a malicious client can redeem more points than it has received punches. Consider a client who at first honestly follows the protocol and redeems a punch card by submitting $p_0, p_n$. Next, it submits a masked $p_n$ for another punch and gets back $p_{n+1}$. Finally, it submits $p_1, p_{n+1}$ as another valid punch card. According to the scheme described thus far, the server would accept this punch card redemption, meaning that the malicious client can redeem $2n$ punches even though it only received $n+1$ punches.

The attack above works because the client can choose any group element it wants as $p_0$. We modify our scheme to provide soundness by forcing clients to generate $p_0$ as the output of a hash function modeled as a random oracle $H : \{0,1\}^{\lambda} \to G$. In particular, instead of choosing a random $p_0$, the client chooses a random $u \leftarrow \{0,1\}^{\lambda}$ and sets $p_0 \leftarrow H(u)$. When redeeming a punch card, instead of sending $p_0, p_n$, the client sends $u, p_n$, and the server checks that $p_n = H(u)^{\mathsf{sk}^n}$. Since the hash function is modeled as a random function, a malicious client cannot find the preimage of a group element under $H$, eliminating the attack.

With this defense, our scheme now provides both privacy and soundness. We formalize our construction as follows.

**Construction 1** (Punch Card Scheme)**.** *Let $G$ be a group of prime order $q$ with generator $g \in G$, a let $H$ be a hash function $H : \{0,1\}^* \to G$, modeled as a random oracle.*

*We construct our punch card scheme as follows:*

- $\mathsf{ServerSetup}(1^{\lambda}) \to \mathsf{sk}, \mathsf{pk}, \mathsf{DB}$*: Select random* $\mathsf{sk} \xleftarrow{R} \mathbb{Z}_q$ *and set* $\mathsf{pk} \leftarrow g^{\mathsf{sk}} \in G$. *Initialize* $\mathsf{DB}$ *as an empty hash table, and return* $\mathsf{sk}$, $\mathsf{pk}$, *and* $\mathsf{DB}$.
- $\mathsf{Issue}(1^{\lambda}) \to \mathsf{psk}, p$*: First, select a random secret* $u \xleftarrow{R} \{0,1\}^{\lambda}$ *and a random masking value* $m \xleftarrow{R} \mathbb{Z}_q$. *Then compute* $p \leftarrow H(u)^m \in G$. *Let* $\mathsf{psk} \leftarrow (u, m)$. *Return* $\mathsf{psk}, p$.
- $\mathsf{ServerPunch}(\mathsf{sk}, \mathsf{pk}, p) \to p', \pi$*: Compute* $p' \leftarrow p^{\mathsf{sk}}$ *as well as the proof of knowledge* $\pi \leftarrow ZKPK\{(\mathsf{sk}), \mathsf{pk} = g^{\mathsf{sk}}, p' = p^{\mathsf{sk}}\}$. *Output* $p', \pi$.
- $\mathsf{ClientPunch}(\mathsf{pk}, \mathsf{psk}, p, p', \pi) \to \mathsf{psk}', p''$ *or* $\perp$*: First, verify the proof* $\pi$. *If verification fails, output* $\perp$. *Otherwise, begin by interpreting* $\mathsf{psk}$ *as* $(u, m)$. *Then sample a new random masking value* $m' \xleftarrow{R} \mathbb{Z}_q$ *and compute* $p'' \leftarrow (p')^{m'/m}$. *Set* $\mathsf{psk}' \leftarrow (u, m')$, *and output* $\mathsf{psk}', p''$.
- $\mathsf{ClientRedeem}(\mathsf{psk}, p) \to \mathsf{psk}', p'$*: Begin by interpreting* $\mathsf{psk}$ *as* $(u, m)$ *with* $u \in \{0,1\}^{\lambda}$ *and* $m \in Z_q$. *Then compute* $p' \leftarrow p^{1/m} \in G$. *Return* $u$ *(as* $\mathsf{psk}'$*) and* $p'$.
- $\mathsf{ServerVerify}(\mathsf{sk}, \mathsf{pk}, \mathsf{DB}, \mathsf{psk}, p, n) \to 1/0, \mathsf{DB}'$*: Check whether* $p = H(\mathsf{psk})^{\mathsf{sk}^n}$ *and whether* $\mathsf{psk} \in \mathsf{DB}$. *If the first check returns true and the second returns false, insert* $\mathsf{psk}$ *into* $\mathsf{DB}$ *and return* 1, $\mathsf{DB}$. *Otherwise, return* 0, $\mathsf{DB}$.

Observe that the asymptotic complexity of almost every operation in our punch card scheme depends only on the security parameter $\lambda$, with two exceptions. The first excpetion is that operations on DB have amortized time complexity $O(\lambda)$, but in the worst case a read/write to DB could depend on the number of previously redeemed punch cards. The other exception is the exponentiation $\mathsf{sk}^n$ performed in ServerVerify, where $O(\log n)$ group operations are required. However, since the same $n$ is often used for every punch card in practice, the server could precompute $\mathsf{sk}^n$ to remove the logarithmic dependence on $n$.

## 3.2 Security

We now discuss the security of our constructions. We begin by proving the privacy of our punch card scheme.

Theorem 3.1. *Assuming the existential soundness of the Chaum-Pedersen proof system, our punch card scheme has privacy (Definition 2.2) in the random oracle model.*

Proof. We begin by describing the simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$.

- $\mathcal{S}_1(1^{\lambda}) \to p$: This simulator samples and outputs a random group element $p \xleftarrow{R} G$.
- $\mathcal{S}_2(\mathsf{pk}, p, p', \pi) \to p''/\perp$: This simulator verifies the proof $\pi$ that $p, \mathsf{pk}, p'$ form a DDH tuple and outputs $\perp$ if verification fails. Otherwise, it samples and outputs a random group element $p'' \xleftarrow{R} G$.
- $\mathcal{S}_3(\mathsf{sk}, c_{\mathsf{id}}) \to \mathsf{psk}', p'$: This simulator samples a random string $\mathsf{psk}' \xleftarrow{R} \{0,1\}^{\lambda}$ and computes $p' \leftarrow H(\mathsf{psk}')^{\mathsf{sk}^{c_{\mathsf{id}}}}$. It outputs $\mathsf{psk}', p'$.

Next, we show through a short series of hybrids that $\mathsf{REALPRIV}(\lambda, \mathcal{A}) \approx_c \mathsf{IDEALPRIV}(\lambda, \mathcal{A}, \mathcal{S})$ for our punch card scheme.

**H0**: This hybrid is the real privacy experiment $\mathsf{REALPRIV}(\lambda, \mathcal{A})$.

**H1**: In this hybrid, we add an abort condition to the execution of the experiment. The experiment aborts and outputs 0 if $\mathcal{S}_2$ outputs $p'' \neq \perp$ (i.e., it accepts the proof $\pi$) but it is not the case that $\mathsf{pk} = g^{\mathsf{sk}} \wedge p' = p^{\mathsf{sk}}$.
This hybrid is indistinguishable from $H0$ by the soundness of the Chaum-Pedersen proof system. In particular, an adversary $\mathcal{A}$ who can distinguish between $H0$ and $H1$ can be used by an algorithm $\mathcal{B}$ to break the soundness of the proof system as follows. $\mathcal{B}$ plays the role of the adversary in the soundness game for the Chaum-Pedersen proof, and plays the role of the challenger to $\mathcal{A}$ in either $H0$ or $H1$ with probability 1/2 each. Whenever $\mathcal{A}$ causes experiment $H1$ to abort due to the check introduced in this hybrid, $\mathcal{B}$ submits the proof $\pi$ and the statement $\mathsf{pk} = g^{\mathsf{sk}} \wedge p' = p^{\mathsf{sk}}$ to the soundness challenger. Otherwise, $\mathcal{B}$ outputs $\perp$.
The algorithm $\mathcal{B}$ described above breaks the soundness of the Chaum-Pedersen proof with the same advantage that $\mathcal{A}$ distinguishes between $H0$ and $H1$. To see why, observe that the only difference in the view of $\mathcal{A}$ between $H0$ and $H1$ occurs when $H1$ aborts. Thus $\mathcal{A}$ must cause the experiment to abort with probability at least equal to its distinguishing advantage between $H0$ and $H1$. But whenever $H1$ aborts, $\mathcal{B}$ has a statement and proof that violate the soundness of the Chaum-Pedersen proof, so it wins the soundness game with the same advantage.

**H2**: In this hybrid, the challenger switches to record-keeping in the table $T$ in the way IDEALPRIV does and replaces calls to Issue, ClientPunch, and ClientRedeem with calls to $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, respectively.
This hybrid is indistinguishable from $H1$ because the distribution of the adversary $\mathcal{A}$'s view is identical in the two hybrids. We will establish this by considering the oracles $\mathcal{O}_{\mathsf{issue}}$, $\mathcal{O}_{\mathsf{punch}}$, and $\mathcal{O}_{\mathsf{redeem}}$ one at a time.

- $\mathcal{O}_{\mathsf{issue}}() \to c, p$: In $H1$, the value $p$ returned by this oracle is determined by $m \xleftarrow{R} \mathbb{Z}_q, u \xleftarrow{R} \{0,1\}^\lambda, p \leftarrow H(u)^m \in G$, which, since $H$ is modeled as a random oracle, corresponds to a uniformly random element of $G$. In $H2$, the value $p$ is directly chosen as a uniformly random element $p \xleftarrow{R} G$. In both hybrids, $c$ is simply the next value of a counter that is incremented with each query. Thus the distribution of the output of the oracle is identical across the two hybrids.
- $\mathcal{O}_{\mathsf{punch}}(\mathsf{id}, p', \pi) \to p''/\bot$: In both $H1$ and $H2$, the oracle verifies $\pi$ and outputs $\bot$ if verification fails (and the game aborts if verification succeeds for a false statement). Thus we only need to consider cases where the proof verifies, i.e., when $\mathsf{pk} = g^{\mathsf{sk}} \wedge p' = p^{\mathsf{sk}}$. In this case, $H1$ selects a random $m' \xleftarrow{R} \mathbb{Z}_q$ and outputs $p'' \leftarrow (p')^{m'/m} \in G$, which is distributed uniformly at random in $G$. In $H2$, the value of $p''$ is directly chosen as a uniformly random value $p'' \xleftarrow{R} G$. Thus the distribution of the output of the oracle is identical across the two hybrids.
- $\mathcal{O}_{\mathsf{redeem}}(\mathsf{id}) \to \mathsf{psk}', p'$: In $H1$, this oracle returns the secret $u$ used to generate the punch card stored at $T[\mathsf{id}]$ as well as the value of that punch card $p$ after removing the last mask $m$ to get $p' \leftarrow p^{1/m}$. The value of $u$ is distributed uniformly at random in $\{0,1\}^\lambda$. The value of $p'$ is equal to $H(u)$ raised to the server secret sk as many times as there was a successful call to $\mathcal{O}_{\mathsf{punch}}(\mathsf{id}, \cdot, \cdot)$ – that is, a call whose output was not $\bot$. This is the case because in each such call, the punch card value stored in $T$ is raised to sk and its mask is replaced with a new one. The final unmasking operation $p' \leftarrow p^{1/m}$ results in a punch card value $p' = H(u)^{\mathsf{sk}^n}$, where $n$ is the number of successful calls to $\mathcal{O}_{\mathsf{punch}}(\mathsf{id}, \cdot, \cdot)$.
  In $H2$, $u$ clearly has the same distribution as in $H1$ because in $\mathcal{S}_3$ it is sampled directly as $u \xleftarrow{R} \{0,1\}^\lambda$. The value $p'$ also has the same distribution as in $H1$ because the table $T$ gradually keeps count of the number $c_{\mathsf{id}}$ of successful calls to $\mathcal{O}_{\mathsf{punch}}(\mathsf{id}, \cdot, \cdot)$, so $\mathcal{S}_3$ can compute $p' \xleftarrow{R} H(u)^{\mathsf{sk}^{c_{\mathsf{id}}}}$ directly.

**H3**: This hybrid is identical to $H2$ except the abort condition introduced in $H1$ is removed. As was the case in $H1$, this hybrid is indistinguishable from the preceding hybrid by the soundness of the Chaum-Pedersen proof system. It also corresponds to the ideal privacy game IDEALPRIV($\lambda, \mathcal{A}, \mathcal{S}$), completing the proof.

□

Having proven privacy, we now turn to soundness. We prove the soundness of our scheme in the algebraic group model (AGM) [23], where for every group element the adversary produces, it must also give a representation of that group element in terms of elements it has already seen. This is a strictly weaker model (in the sense that it puts fewer restrictions on the adversary) than the widely-used generic group model [38], in which some of the prior works on privacy-preserving loyalty programs have been proven secure [5, 6]. Our proof relies on the $q$-discrete log assumption, which assumes the computational hardness of winning the following game.

*Definition 3.2 (q-discrete log game).* The $q$-discrete log game for a group $G$ of prime order $p$ is played between a challenger $C$ and an adversary $\mathcal{A}$. The challenger $C$ samples $x \xleftarrow{R} \mathbb{Z}_p$ and sends $g^x, g^{x^2}, ..., g^{x^q}$ to $\mathcal{A}$. The adversary $\mathcal{A}$ responds with a value $z \in \mathbb{Z}_p$, and the challenger outputs 1 iff $z = x$.

Depending on the concrete group in which the assumption is made, the $q$-discrete log game could be vulnerable to Brown-Gallant-Cheon attacks [10, 18], which reduce the security of the assumption by a factor of $\sqrt{q}$. Fortunately this attack only negligibly affects the security of the scheme, as $q$ is at most a polynomial in the security parameter $\lambda$.

We now state and prove our soundness theorem.

Theorem 3.3. *Assuming the zero-knowledge property of the Chaum-Pedersen proof system and the q-discrete log assumption in G, our punch card scheme has soundness (Definition 2.3) in the algebraic group model with random oracles.*

Proof. Since $q$ already refers to the order of the group $G$, we will refer to the $N$-discrete log assumption throughout this proof. The high-level idea of the proof is to program random oracle queries with re-randomizations of powers of $g^x$ given by the $N$-discrete log challenger. Then, whenever a punch card is given by the adversary, the algebraic adversary must also give a representation of the punch card $p$ in terms of group elements it has seen before. As such, the challenger can pick out the $g^{x^i}$ component and replace it with $g^{x^{i+1}}$ in its response. Then a punch card that is accepted before receiving $n$ punches must include a second representation of $g^{x^n}$, allowing us to solve for $x$.

We now formalize the proof idea sketched above. Our proof proceeds through a series of hybrids.

**H0**: This hybrid is the soundness experiment SOUND($\lambda, \mathcal{A}$).

**H1**: In this hybrid, we replace the proof $\pi$ output by $\mathcal{O}_{\mathsf{punch}}$ with a simulated proof.
The the zero-knowledge property of the Chaum-Pedersen proof guarantees that the proof can be simulated. Since hybrids $H0$ and $H1$ are identical save for the real proof in $H0$ and the simulated proof in $H1$, the output of an adversary $\mathcal{A}$ who distinguishes between $H0$ and $H1$ can also be used to distinguish between a real and simulated proof with the same advantage.

**H2**: In this hybrid, we add an abort condition to the execution of the experiment. The experiment aborts and outputs 0 if the $\mathcal{O}_{\mathsf{redeem}}(\mathsf{psk}, p, n)$ oracle ever outputs $b = 1$ when it receives a value of psk that the adversary has not previously queried from the random oracle $H$.
This hybrid is indistinguishable from $H1$ because the probability of an adversary successfully triggering this abort

condition is negligible in $\lambda$ and there are no other differences between $H1$ and $H2$. In order for the $O_{\text{redeem}}$ oracle to output $b = 1$, it must be the case that $\text{ServerVerify}(\text{sk}, \text{pk}, \text{DB}, \text{psk}, p, n)$ outputs 1, which means that $p = H(\text{psk})^{\text{sk}^n}$. But since $H$ is modeled as a random function and $H(\text{psk})$ has not been queried before, its output is chosen uniformly at random in $G$, that is, $H(\text{psk}) \xleftarrow{R} G$. But then $H(\text{psk})^{\text{sk}^n}$ is also distributed uniformly at random in $G$, and the probability $\Pr[p = H(\text{psk})^{\text{sk}^n}] \leq \text{negl}(\lambda)$.

**H3**: In this hybrid, we modify how the challenger computes the output of $\text{ServerPunch}(\text{sk}, \text{pk}, p)$ and of the random oracle $H$. Recall that since $\mathcal{A}$ is an algebraic adversary, every group element it sends is accompanied by a representation in terms of the previous group elements it has seen: the generator $g$, returned punch cards $p'_1, ..., p'_Q$ for the $Q$ queries it has made to the $O_{\text{punch}}$ oracle, and random oracle outputs $H_1, ..., H_{Q'}$ for the $Q'$ random oracle queries it has made.
Let $g_i = g^{\text{sk}^i}$ for $i \in \mathbb{Z}$. Whenever the adversary $\mathcal{A}$ makes a call to the oracle $H$ on a previously unqueried point $u$, the challenger samples $r \xleftarrow{R} \mathbb{Z}_q$ and sets $H(u) \leftarrow g_1^r$. Since $r$ is distributed uniformly at random in $\mathbb{Z}_q$, so is $H(u)$.
Next, whenever $\mathcal{A}$ makes a call to the oracle $O_{\text{punch}}(p)$, instead of setting $p' \leftarrow p^{\text{sk}}$, the challenger looks at the algebraic representation of $p$ submitted by $\mathcal{A}$ and replaces each occurence of $g_i$ with $g_{i+1}$, including replacing $g$ with $g_1$. Since the only elements $\mathcal{A}$ has seen are $g$, random oracle outputs, and the previous results of $O_{\text{punch}}$, the challenger can keep track of which elements contain which $g_i$ as it sends them to $\mathcal{A}$. The outputs of $O_{\text{punch}}(p)$ in $H3$ are identical to the outputs in $H2$, because the process described here results in the same group element $p'$ that would be represented by $p^{\text{sk}}$.
Since all the changes in $H3$ result in identically distributed outputs as in $H2$, the two hybrids are indistinguishable.

From $H3$, we can prove that any algebraic adversary $\mathcal{A}$ who wins the soundness game can be used by an algorithm $\mathcal{B}$, described below, to break the $N$-discrete log assumption in $G$. Algorithm $\mathcal{B}$ plays the role of the adversary in the $N$-discrete log game while simultaneously playing the role of the challenger in $H3$. Algorithm $\mathcal{B}$ simulates $H3$ exactly to $\mathcal{A}$, except that it uses the $N$-discrete log challenge messages $g^x, g^{x^2}, ..., g^{x^N}$ as the values of $g_i$. That is, $g_i = g^{x^i}$. Moreover, it sets $\text{pk} \leftarrow g_1$ in the setup phase. Observe that the $g_i$ are distributed identically as in $H3$, so this is a perfect simulation of $H3$ with $x$ playing the role of sk. The value of $N$ required in the assumption depends on the maximum number of sequential punches $\mathcal{A}$ requests on the same group element.

Now, if $\mathcal{A}$ wins the soundness game, it means that $c_{\text{redeem}} > c_{\text{punch}}$. This, in turn, implies that there was some successful punch card redemption ServerVerify where the accepted value of $p$ had not been previously punched $n$ times, i.e., the representation of $p$ does not contain $g_{n+1}$. But since successful verification requires that $p = H(\text{psk})^{x^n} = (g_1^r)^{x^n} = g_{n+1}^r$, and the algebraic adversary $\mathcal{A}$ must give a representation of $p$, we now have two different representations of $g_{n+1} = g^{x^{n+1}}$, which together yield a degree-$n+1$ equation in $x$. This equation can be solved for $x$ using standard techniques [39], allowing $\mathcal{B}$ to recover $x$ and win the $N$-discrete log game. □

## 4 MERGING PUNCH CARDS

Having described our main construction, we now consider another feature sometimes enjoyed by physical punch cards that we may want to reproduce digitally: merging partially-filled cards. Just as in real life, it is possible to "merge" two punch cards by redeeming them separately and taking into account the sum of the number of punches across the two cards. However, this process reveals the number of punches held by each card at redemption time, information that the customer may want to hide. We can hide the value of the two cards being merged by resorting to pairings.

*Definition 4.1 (Pairings [7]).* Let $G_0, G_1, G_T$ be three cyclic groups of prime order $q$ where $g_0 \in G_0$ and $g_1 \in G_1$ are generators. A *pairing* is an efficiently computable function $e : G_0 \times G_1 \rightarrow G_T$ satisfying the following properties:

- Bilinear: for all $u, u' \in G_0$ and $v, v' \in G_1$ we have
$$e(u \cdot u', v) = e(u, v) \cdot e(u', v)$$
and
$$e(u, v \cdot v') = e(u, v) \cdot e(u, v')$$
- Non-degenerate: $g_T \leftarrow e(g_0, g_1)$ is a generator of $G_T$.

When $G_0 = G_1$, we say that the pairing is a *symmetric pairing*. We refer to $G_0$ and $G_1$ as the *pairing groups* and refer to $G_T$ as the *target group*.

Using a symmetric pairing, we can quite simply merge two punch cards without revealing the number of punches on each. Before redeeming punch cards $p_0$ and $p_1$ which have $i$ and $j$ punches, respectively, with $i + j = n$, the client computes $p \leftarrow e(p_0, p_1)$. To redeem a merged card, the client sends the server the merged punch card $p$ along with $u_0$ and $u_1$, the secrets for the two punch cards merged into $p$. The server checks that $e(H(u_0)^{\text{sk}^n}, H(u_1))$. The bilinear property of the pairing ensures that $e(p_0, p_1) = e(H(u_0)^{\text{sk}^i}, H(u_1)^{\text{sk}^j}) = e(H(u_0)^{\text{sk}^n}, H(u_1))$. We can even hide whether or not a redeemed punch card is merged by generating a fresh punch card before redemption and merging a complete card with it.

The performance of symmetric pairings is far worse than that of asymmetric pairings, so we would like to have a scheme that works for asymmetric pairings as well. Unfortunately, directly converting the idea above to asymmetric pairings meets with some difficulties. Since each punch card must belong to either $G_0$ or $G_1$, we can only merge pairs of cards where $p_0 \in G_0$ and $p_1 \in G_1$. But this is a decision that must be made when a card is first issued, restricting punch cards to being merged with cards that belong to the other pairing group.

We resolve this problem by splitting each punch card into two components, one in each pairing group. Each component behaves as a punch card in the original scheme. Generating a punch card is similar to the original scheme, but the secret $u \xleftarrow{R} \{0,1\}^\lambda$ is hashed by two different functions $H_0 : \{0,1\}^\lambda \rightarrow G_0$ and $H_1 : \{0,1\}^\lambda \rightarrow G_1$. Each hole punch repeats the punch protocol of the original scheme twice, once in $G_0$ and once in $G_1$. Redeeming a card requires merging the $G_0$ and $G_1$ components of the two cards with each other as above, and since the client has a version of each punch card in both groups, it can merge them as before.

We formalize this sketch of a solution below. We replace the ClientRedeem algorithm from our punch card syntax with a new ClientMergeRedeem algorithm that merges two punch cards before redeeming them.

**Construction 2** (Mergable Punch Card Scheme). *Let $G_0, G_1, G_T$ be groups of prime order $q$ with generators $g_0 \in G_0, g_1 \in G_1$, and let $H_0, H_1$ be hash functions $H_0 : \{0,1\}^* \to G_0, H_1 : \{0,1\}^* \to G_1$, modeled as random oracles. We construct our punch card scheme as follows:*

- $\mathsf{ServerSetup}(1^\lambda) \to \mathsf{sk}, \mathsf{pk}, \mathsf{DB}$*: Select random* $\mathsf{sk} \xleftarrow{R} \mathbb{Z}_q$ *and set* $\mathsf{pk}_0 \leftarrow g_0^{\mathsf{sk}} \in G_0, \mathsf{pk}_1 \leftarrow g_1^{\mathsf{sk}} \in G_1$. *Initialize* DB *as an empty hash table, and return* $\mathsf{sk}, \mathsf{pk} = (\mathsf{pk}_0, \mathsf{pk}_1)$, *and* DB.
- $\mathsf{Issue}(1^\lambda) \to \mathsf{psk}, p$*: First, select a random secret* $u \xleftarrow{R} \{0,1\}^\lambda$ *and random masking values* $m_0 \xleftarrow{R} \mathbb{Z}_q, m_1 \xleftarrow{R} \mathbb{Z}_q$. *Then compute* $p_0 \leftarrow H_0(u)^{m_0} \in G_0, p_1 \leftarrow H_1(u)^{m_1} \in G_1$. *Let* $\mathsf{psk} \leftarrow (u, m_0, m_1)$. *Return* $\mathsf{psk}, p = (p_0, p_1)$.
- $\mathsf{ServerPunch}(\mathsf{sk}, \mathsf{pk}, p) \to p', \pi$*: First, interpret* pk *as* $(\mathsf{pk}_0, \mathsf{pk}_1)$ *and* $p$ *as* $(p_0, p_1)$. *Compute* $p_0' \leftarrow p_0^{\mathsf{sk}}, p_1' \leftarrow p_1^{\mathsf{sk}}$ *as well as the proofs of knowledge* $\pi_0 \leftarrow ZKPK\{(\mathsf{sk}), \mathsf{pk}_0 = g_0^{\mathsf{sk}}, p_0' = p_0^{\mathsf{sk}}\}$ *and* $\pi_1 \leftarrow ZKPK\{(\mathsf{sk}), \mathsf{pk}_1 = g_1^{\mathsf{sk}}, p_1' = p_1^{\mathsf{sk}}\}$. *Output* $p' = (p_0', p_1'), \pi = (\pi_0, \pi_1)$.
- $\mathsf{ClientPunch}(\mathsf{pk}, \mathsf{psk}, p, p', \pi) \to \mathsf{psk}', p''\text{or}\bot$*: First, interpret* psk *as* $(u, m_0, m_1)$, $p$ *as* $(p_0, p_1)$, $p'$ *as* $(p_0', p_1')$, *and* $\pi$ *as* $(\pi_0, \pi_1)$ . *Next, verify the proofs* $\pi_0$ *and* $\pi_1$. *If either verification fails, output* $\bot$. *Then sample new random masking values* $m_0' \xleftarrow{R} \mathbb{Z}_q, m_1' \xleftarrow{R} \mathbb{Z}_q$ *and compute* $p_0'' \leftarrow (p_0')^{m_0'/m_0}, p_1'' \leftarrow (p_1'^{m_1'/m_1})$. *Finally, output* $\mathsf{psk}' = (u, m_0', m_1'), p'' = (p_0'', p_1'')$.
- $\mathsf{ClientMergeRedeem}(\mathsf{psk}, p, \mathsf{psk}', p') \to \mathsf{psk}'', p''$*: Begin by interpreting* psk *as* $(u, m_0, m_1)$, psk′ *as* $(u', m_0', m_1')$, $p$ *as* $(p_0, p_1)$, *and* $p'$ *as* $(p_0', p_1')$. *Then compute* $p'' \leftarrow e(p_0^{1/m_0}, (p_1')^{1/m_1'}) \in G_T$. *Return* $\mathsf{psk}'' = (u, u')$ *and* $p''$.
- $\mathsf{ServerVerify}(\mathsf{sk}, \mathsf{pk}, \mathsf{DB}, \mathsf{psk}, p, n) \to 1/0, \mathsf{DB}'$*: Begin by interpreting* psk *as* $(u, u')$. *Then perform the following checks:*
  *(1)* $p = e(H_0(u)^{\mathsf{sk}^n}, H_1(u'))$
  *(2)* $u \in \mathsf{DB}$
  *(3)* $u' \in \mathsf{DB}$
  *If the first check returns true and the other checks return false, insert* $u$ *and* $u'$ *into* DB *and return* 1, DB. *Otherwise, return* 0, DB.

Although not included in our formal construction, our scheme could be extended to allow more punches to occur on a merged card so long as the client indicates that it is a merged card being punched and the punch/proof occur over elements in $G_T$. Note that this scheme only allows for two punch cards to merged. Our general strategy for merging punch cards could be extended to more than two cards using multilinear maps [8, 19, 24], but a construction that allows merging of more than two cards while only relying on efficient standard primitives would require new techniques. This is an interesting problem for future work to address.

We now state and prove our security theorems for the mergable punch card scheme. The only change required in the security games to account for the change from ClientRedeem to ClientMergeRedeem is that the redeem oracle in the privacy game takes in two ids instead of just one and passes both corresponding punch cards to ClientMergeRedeem.

Theorem 4.2. *Assuming the existential soundness of the Chaum-Pedersen proof system, our mergable punch card scheme has privacy in the random oracle model.*

Proof (sketch). We begin by describing the simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$.

- $\mathcal{S}_1(1^\lambda) \to p$: This simulator samples and outputs two random group elements $p_0 \xleftarrow{R} G_0$ and $p_1 \xleftarrow{R} G_1$.
- $\mathcal{S}_2(\mathsf{pk}, p, p', \pi) \to p''/\bot$: This simulator interprets $\pi = (\pi_0, \pi_1)$ and verifies both proofs, outputting $\bot$ if either verification fails. Otherwise, it samples and outputs two random group elements $p_0'' \xleftarrow{R} G_0$ and $p_1'' \xleftarrow{R} G_1$.
- $\mathcal{S}_3(\mathsf{sk}, c_{\mathsf{id}}, c_{\mathsf{id}'}) \to \mathsf{psk}', p'$: This simulator samples two random strings $u \xleftarrow{R} \{0,1\}^\lambda, u' \xleftarrow{R} \{0,1\}^\lambda$ and computes $p' \leftarrow e(H_0(u)^{\mathsf{sk}^{c_{\mathsf{id}}}}, H_1(u')^{\mathsf{sk}^{c_{\mathsf{id}'}}})$. It outputs $\mathsf{psk} \leftarrow (u, u')$ and $p'$.

Next, we show through a short series of hybrids that $\mathsf{REALPRIV}(\lambda, \mathcal{A}) \approx_c \mathsf{IDEALPRIV}(\lambda, \mathcal{A}, \mathcal{S})$ for our mergable punch card scheme. The rest of proof of this theorem is very similar to that of Theorem 3.1. The main difference is that the soundness of the Chaum-Pedersen proof system needs to be invoked separately in each of $G_0$ and $G_1$. Thus we only sketch the steps of the hybrid argument below.

**H0**: This hybrid is the real privacy experiment $\mathsf{REALPRIV}(\lambda, \mathcal{A})$.
**H1**: In this hybrid, we add an abort condition to the execution of the experiment. The experiment aborts and outputs 0 if $\mathcal{S}_2$ outputs $p'' \neq \bot$ (i.e., it accepts the proofs $\pi_0$ and $\pi_1$) but it is not the case that $\mathsf{pk}_0 = g_0^{\mathsf{sk}} \wedge p_0' = p_0^{\mathsf{sk}}$.
**H2**: In this hybrid, we add an abort condition to the execution of the experiment. The experiment aborts and outputs 0 if $\mathcal{S}_2$ outputs $p'' \neq \bot$ (i.e., it accepts the proofs $\pi_0$ and $\pi_1$) but it is not the case that $\mathsf{pk}_1 = g_1^{\mathsf{sk}} \wedge p_1' = p_1^{\mathsf{sk}}$.
**H3**: In this hybrid, the challenger switches to record-keeping in the table $T$ in the way IDEALPRIV does and replaces calls to Issue, ClientPunch, and ClientMergeRedeem with calls to $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, respectively.
**H4**: This hybrid is identical to $H3$ except the abort condition introduced in $H2$ is removed.
**H5**: This hybrid is identical to $H4$ except the abort condition introduced in $H1$ is removed. It also corresponds to the ideal privacy game $\mathsf{IDEALPRIV}(\lambda, \mathcal{A}, \mathcal{S})$, completing the proof.

□

Next, we prove the soudness of our scheme. Since our new scheme uses pairings, we use an *asymmetric q*-discrete log assumption, which assumes the computational hardness of winning the following game.

*Definition 4.3 (asymmetric q-discrete log game).* The $q$-discrete log game for groups $G_0, G_1$ of prime order $p$ is played between a challenger $C$ and an adversary $\mathcal{A}$. The challenger $C$ samples $x \xleftarrow{R} \mathbb{Z}_p$ and sends $g_0^x, g_0^{x^2}, ..., g_0^{x^q}, g_1^x, g_1^{x^2}, ..., g_1^{x^q}$ to $\mathcal{A}$. The adversary $\mathcal{A}$ responds with a value $z \in \mathbb{Z}_p$, and the challenger outputs 1 iff $z = x$.

THEOREM 4.4. *Assuming the zero-knowledge property of the Chaum-Pedersen proof system and the asymmetric q-discrete log assumption in $G_0$ and $G_1$, our mergable punch card scheme has soundness (Definition 2.3) in the algebraic group model with random oracles.*

PROOF (SKETCH). Since $q$ already refers to the order of the groups $G_0, G_1, G_T$, we will refer to the asymmetric $N$-discrete log assumption throughout this proof. The majority of proof of this theorem is very similar to that of Theorem 3.3. The main difference in the hybrids is that several hybrids need to be repeated to account for each punch card being made up of two group elements instead of one. Thus we only sketch the steps of the hybrid argument below and focus on the last step of the argument.

**H0**: This hybrid is the soundness experiment $\mathsf{SOUND}(\lambda, \mathcal{A})$.

**H1**: In this hybrid, we replace the proof $\pi_0$ output by $O_{\mathsf{punch}}$ with a simulated proof.

**H2**: In this hybrid, we replace the proof $\pi_1$ output by $O_{\mathsf{punch}}$ with a simulated proof.

**H3**: In this hybrid, we add an abort condition to the execution of the experiment. The experiment aborts and outputs 0 if the $O_{\mathsf{redeem}}(\mathsf{psk}, p, n)$ oracle ever outputs $b = 1$ when it receives a value of $u$ or $u'$ that the adversary has not previously queried from both random oracles $H_0$ and $H_1$.

**H4**: In this hybrid, we modify how the challenger computes the output of ServerPunch$(\mathsf{sk}, \mathsf{pk}, p)$ and of the random oracle $H$. Let $g_{0,i} = g_0^{\mathsf{sk}^i}$ and $g_{1,i} = g_1^{\mathsf{sk}^i}$ for $i \in \mathbb{Z}$. Recall that since $\mathcal{A}$ is an algebraic adversary, every group element it sends is accompanied by a representation in terms of the previous group elements it has seen. Just as we did in the proof of Theorem 3.3, instead of punching cards by raising $p_0^{\mathsf{sk}}$ and $p_1^{\mathsf{sk}}$, we examine the algebraic representation of $p_0, p_1$ submitted by the adversary and replace each instance of $g_{0,i}$ or $g_{1,i}$ with $g_{0,i+1}$ or $g_{1,i+1}$, respectively. Also, whenever the adversary $\mathcal{A}$ makes a call to the oracles $H_j$ (for $j \in 0, 1$) on a previously unqueried point $u$, the challenger samples $r \xleftarrow{R} \mathbb{Z}_q$ and sets $H_j(u) \leftarrow g_{j,1}^r$.

From $H4$, we can prove that any algebraic adversary $\mathcal{A}$ who wins the soundness game can be used by an algorithm $\mathcal{B}$, described below, to break the $N$-discrete log assumption in either $G_0$ or $G_1$. Algorithm $\mathcal{B}$ plays the role of the adversary in the asymmetric $N$-discrete log game while simultaneously playing the role of the challenger in $H4$. Algorithm $\mathcal{B}$ simulates $H4$ exactly to $\mathcal{A}$, except that it uses the asymmetric $N$-discrete log challenge messages $g_0^x, g_0^{x^2}, ..., g_0^{x^N}, g_1^x, g_1^{x^2}, ..., g_1^{x^N}$ as the values of $g_{0,i}$ and $g_{1,i}$. That is, $g_{0,i} = g_0^{x^i}$ and $g_{1,i} = g_1^{x^i}$. Moreover, it sets $\mathsf{pk} \leftarrow (g_{0,1}, g_{1,1})$ in the setup phase. Observe that all $g_{0,i}$ and $g_{1,i}$ are distributed identically as in $H4$, so this is a perfect simulation of $H4$ with $x$ playing the role of sk. The value of $N$ required in the assumption depends on the maximum number of sequential punches $\mathcal{A}$ requests on the same group element.

Now, if $\mathcal{A}$ wins the soundness game, it means that $c_{\mathsf{redeem}} > c_{\mathsf{punch}}$. This, in turn, implies that there was some successful punch card redemption ServerVerify where the accepted value of $p$ had not been previously punched $n$ times between the two merged cards. Let $a$ and $b$ the number of times each of the two merged cards had been punched before redemption, so we have $a + b < n$.

The successful verification requires that

$$p = e(H_1(u)^{x^n}, H_2(u')) = e(g_{0,1}^{rx^n}, g_{1,1}^{r'}) = e(g_{0,1}^{rx^{a'}}, g_{1,1}^{r'x^{b'}})$$

for any $a', b'$ where $a' + b' = n$, and the algebraic adversary must give a representation of $p$ (it can include a pairing in this representation). It must be true that one of $a'$ or $b'$ is greater than $a$ or $b$, respectively, because $a + b < n$. Thus the representation of $p$ must not include either $g_{0,a'+1}^r$ or $g_{1,b'+1}^{r'}$ because one of those values will not have been given to $\mathcal{A}$. This means we now have two different representations of one of these elements, which together yield a degree $a' + 1$ or $b' + 1$ equation in $x$. This equation can be solved for $x$ using standard techniques [39], allowing $\mathcal{B}$ to recover $x$ and win the asymmetric $N$-discrete log game. □

## 5 IMPLEMENTATION AND EVALUATION

We implemented our main punch card scheme from Section 3 as well as the mergable punch card scheme from Section 4. Our implementation is written in Rust with a Java wrapper to run the Rust code on Android devices. The implementation of the main punch card scheme relies on the `curve25519-dalek` [33] crate which implements curve25519 [4], and the mergable punch card scheme uses the `pairing-plus` [25] crate, which provides an implementation of BLS12-381 curves [9]. Our implementation and raw evaluation data are available at https://github.com/SabaEskandarian/PunchCard.

We carried out our evaluation with the client running on a Google Pixel (first generation) phone and the server running on a laptop with an Intel i5-8265U processor @ 1.60GHz. All data reported on our scheme comes from an average of at least 100 trials. ServerVerify was run with $n = 10$ punches on each redeemed punch card and an empty database DB of used cards. We repeated the test of the main scheme with a database of 1,000,000 used cards and saw no significant difference between that and the test with an empty database, leading us to conclude that the hash table lookup does not dominate the cost of the ServerVerify algorithm.

Figure 1 shows the running time of each of the algorithms in our main punch card construction as well as the amount of in-protocol data sent by the party running the algorithm in a punch card system. The data sent in ServerSetup refers to the size of the public key which must be communicated to clients. Observe that we do not require the client to communicate any data in order for a new punch card to be issued. The most costly operation, punching a card, requires less than 5ms between the client and server combined, and all other operations require less than 1ms. Commmunication is under 200 Bytes for all operations.

Figure 2 shows the same information for the mergable punch card scheme. The mergable scheme runs considerably more slowly than the main scheme. This is the result of 1) more work being required in the mergable scheme, 2) group operations being more costly in pairing groups, and 3) the heavily optimized library used for curve25519 (`curve25519-dalek`) in the implementation of the main scheme. Group elements in pairing groups are also larger than in curve25519. In curve25519, the size of a group element $g \in G$ is 32 Bytes, but in the BLS12-381 curves, $g_0 \in G_0$ requires 48 Bytes, $g_1 \in G_1$ requires 96 Bytes, and $g_T \in G_T$ requires 576 Bytes.

**Comparison to prior work**. We compare our punch card scheme to the loyalty systems BBA+ [28], UACS [5], and Bobolz et al. [6].

|  | ServerSetup | Issue | ServerPunch | ClientPunch | ClientRedeem | ServerVerify |
|---|---|---|---|---|---|---|
| Computation Time (ms) | 0.019 | 0.304 | 0.134 | 4.314 | 0.890 | 0.064 |
| Data Sent (Bytes) | 32 | 0 | 128 | 32 | 64 | 0 |

**Table 1: Computation and communication costs for our main punch card scheme.**

|  | ServerSetup | Issue | ServerPunch | ClientPunch | ClientMergeRedeem | ServerVerify |
|---|---|---|---|---|---|---|
| Computation Time (ms) | 1.09 | 34.97 | 4.33 | 137.79 | 36.43 | 4.00 |
| Data Sent (Bytes) | 144 | 0 | 496 | 144 | 640 | 0 |

**Table 2: Computation and communication costs for our mergable punch card scheme using pairings.**

|  | Issuing a Card | Punching a Card | Redeeming a Card |
|---|---|---|---|
| BBA+ scheme | 115.27 | 385.61 | 375.73 |
| UACS scheme | 86 | 127 | 454 |
| Bobolz et al. scheme | 130 | 64 | 1254 |
| Our main scheme | 0.304 (282.99× faster) | 4.448 (14.4× faster) | 0.954 (393.8× faster) |
| Our mergable scheme | 34.97 (2.5× faster) | 142.12 | 40.43 (9.3× faster) |

**Table 3: Computation time (in milliseconds) for our schemes and prior work. Speedups shown in parentheses refer to improvement over best prior work.**

|  | Issuing a Card | Punching a Card | Redeeming a Card |
|---|---|---|---|
| BBA+ scheme | 992 | 4048 | 3984 |
| Our main scheme | 0 | 160 (25.3× reduction) | 64 (62.3× reduction) |
| Our mergable scheme | 0 | 640 (6.3× reduction) | 640 (6.2× reduction) |

**Table 4: Communication (in Bytes) in our schemes and BBA+ [28], the only prior work to record communication costs. Our schemes incur no communication to issue a new card and achieve order of magnitude improvements for other operations. The pairing-based scheme requires more communication because pairing group elements are larger and punching a card requires twice as many elements communicated.**

We do not compare to the original BBA work [29] because its performance is strictly worse than the works to which we do compare. We use the performance numbers reported by each prior work. Performance numbers for UACS and Bobolz et al., were also recorded with a Google Pixel phone but used a computer with a stronger i7 processor. BBA+ only reports the client-side cost of each of its protocols and uses a OnePlus 3 phone. In order to better capture the total cost of using each approach, we combine client and server costs to give the overall computation cost of each scheme. However, the distribution of cost between the client and server is similar for all works, with the mobile device incurring most of the computation cost.

Figure 3 compares the performance numbers of our schemes against those of prior work. Our main scheme issues a card 282.99× faster than the best prior work (UACS), punches a card 14.4× faster than the best prior work (Bobolz et al.), and redeems a card 393.8× faster than the best prior work (BBA+). Although each prior work we compared to was the fastest in one of these three procedures, our scheme strictly dominates all of them by at least an order of magnitude. The performance improvement comes from removing the reliance on pairings and significantly reducing the number and complexity of zero knowledge proofs required in each operation.

Our mergable punch card scheme outperforms prior work in almost every category, and by a margin of 9.3× in card redemption. The Bobolz et al. scheme punches a card about twice as fast as our scheme, but they achieve this by removing zero knowledge proofs from the punch protocol and pushing them to redemption, which takes over a second (our redemption is 31× faster). An important difference to point out between our implementation and prior work is that while our implementation was done with BLS12-381 curves, which provide 128 bits of security, prior works all used BN curves [1, 30] that only provide 100 bits of security [31, 37].

Figure 4 compares the communication costs of our schemes with BBA+, the only prior work to report the communication costs incurred by their implementation. Unlike all prior work, our scheme requires *no communication* to issue a new card, and card punching and redemption require 25.3× and 62.3× less communication, respectively, than BBA+. For the mergable scheme, the improvements are reduced to about 6×, but even this scheme requires significantly less communication.

## 6 EXTENSIONS

We now briefly discuss extensions to our main punch card scheme that can allow it to be used in a wider variety of applications.

**Multi-punches**. Some loyalty programs sometimes offer extra punches on their punch cards as a special promotion. Others don't use a punch card at all, opting instead for a system where different transactions earn varying numbers of points. Our punch card scheme can easily be extended to handle these situations by having the server raise $p$ to $\mathsf{sk}^t$, where $t$ is the number of points being awarded for a given transaction.

Unfortunately, this kind of multi-punch raises a new security question. Most punch card schemes offer a fixed value $n$ at which point a card can be redeemed for some benefit, or perhaps a few values at which different kinds of rewards are unlocked. But the possibility of gaining more than one punch with a given transaction introduces the potential for a client to "overshoot" the required number of points. This does not pose an issue for functionality, because the client can just redeem a card with $n' > n$ punches and perhaps even get a new card with the remaining balance. However, this might introduce a privacy issue because the redemption reveals the total number of punches on a card, which is no longer always the exact same value for all clients. One way to eliminate this problem is to have the server send all possible values $p^{\mathsf{sk}}, p^{\mathsf{sk}^2}, ..., p^{\mathsf{sk}^t}$ when punching a card. This works well for settings where $t$ is small, e.g., a double-punch promotion. We leave the problem of an efficient solution for large $t$ for future work.

**Managing used card database size**. Our punch card scheme requires keeping a database DB of used punch card secrets $u$, stored in a hash table in our implementation. While this does not pose a performance problem because of the amortized constant time lookup in the hash table, the storage cost increases over time. Although at 128 bits per secret, it would take a long time for storage costs to become prohibitive, a high-volume punch card program may wish for a plan to eventually remove old punch cards from the database without allowing double spending.

One way to help reduce the long-term storage requirement is by adding extra information into the secret $u$. Since $u$ is ultimately passed through a hash function modeled as a random oracle, adding structured information before the random bits makes no difference in the security of the scheme (unless the structured information itself leaks something). Clients can be required to add an expiration date to the beginning of $u$. Then the card redemption would check whether the card being used is expired or not. To encourage clients to pick reasonable expiration dates, cards with expiration dates too far in the future could be rejected as well. Expiration dates used in $u$ could be standardized, e.g., to the first day of a given year, to prevent the date itself from leaking too much information about an individual customer's shopping habits.

**Private ticketing**. Our punch card scheme can also be viewed as a scheme for private ticketing, or, more generally, as a one-time use anonymous credential. To issue a ticket, the client generates a new punch card, and the server punches it. A ticket can reflect additional information (e.g., if a train ticket is first class or coach, which transit zones a ticket is valid for, etc.) by the number of punches added to the ticket. To record multiple pieces of information on the same ticket, the random oracle $H$ can be used to generate multiple group elements from $u$, each of which can hold a different number of punches. Since the punches cannot be linked to their redemption, a client can later present the ticket without linking it to the issuance process.

## 7 CONCLUSION

We have presented a new scheme for punch card loyalty programs that significantly outperforms all prior work both quantitatively and qualitatively. Our scheme does not require any server interaction for a client to receive a punch card, does not require pairings, and outperforms prior work in card issuance, punching, and redemption by 283×, 14.4×, and 394× respectively, strictly dominating the performance of all prior solutions to this problem. We have also shown several extensions to our main scheme, including a modified protocol that allows merging punch cards (using pairings) that still outperforms prior work. Our implementation is open source and available at https://github.com/SabaEskandarian/PunchCard.

## ACKNOWLEDGMENTS

We would like to thank Dan Boneh for several helpful conversations. This work was funded by NSF, DARPA, a grant from ONR, and the Simons Foundation. Opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

## REFERENCES

[1] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*, pages 319–331, 2005.

[2] Mira Belenkiy, Melissa Chase, Markulf Kohlweiss, and Anna Lysyanskaya. Compact e-cash and simulatable vrfs revisited. In *Pairing-Based Cryptography - Pairing 2009, Third International Conference, Palo Alto, CA, USA, August 12-14, 2009, Proceedings*, pages 114–131, 2009.

[3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, 1993.

[4] 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.

[5] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1671–1685, 2019.

[6] Jan Bobolz, Fabian Eidens, Stephan Krenn, Daniel Slamanig, and Christoph Striecks. Privacy-preserving incentive systems with highly efficient point-collection. In *Asia CCS*, 2020.

[7] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography (version 0.5)*. 2020. https://cryptobook.us.

[8] Dan Boneh and Alice Silverberg. Applications of multilinear forms to cryptography. *IACR Cryptology ePrint Archive*, 2002:80, 2002.

[9] Sean Bowe. BLS12-381: new zk-snark elliptic curve construction. https://electriccoin.co/blog/new-snark-curve/, 2017.

[10] Daniel R. L. Brown and Robert P. Gallant. The static diffie-hellman problem. *IACR Cryptology ePrint Archive*, 2004:306, 2004.

[11] Meta S. Brown. Loyalty programs and data mining.

[12] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 302–321, 2005.

[13] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, pages 93–118, 2001.

[14] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, pages 56–72, 2004.

[15] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 410–424, 1997.

[16] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.

[17] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 89–105, 1992.

[18] Jung Hee Cheon. Security analysis of the strong diffie-hellman problem. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, pages 1–11, 2006.

[19] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 476–493, 2013.

[20] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[21] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[22] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 303–324, 2005.

[23] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. 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*, pages 33–62, 2018.

[24] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 1–17, 2013.

[25] Jack Grigg, Riad Wahby, Sean Bowe, and Zhenfei Zhang. pairing-plus. https://github.com/algorand/pairing-plus, 2020.

[26] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, pages 415–432, 2008.

[27] Sarita Harbour. What are the pros and cons of punch cards for your small business? 2017.

[28] Gunnar Hartung, Max Hoffmann, Matthias Nagel, and Andy Rupp. BBA+: improving the security and applicability of privacy-preserving point collection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1925–1942, 2017.

[29] Tibor Jager and Andy Rupp. Black-box accumulation: Collecting incentives in a privacy-preserving way. *PoPETs*, 2016(3):62–82, 2016.

[30] K. Kasamatsu, S. Kanno, T. Kobayashi, and Y. Kawahara. Barreto-naehrig curves. Technical report, 2 2014.

[31] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 543–571, 2016.

[32] Ravie Lakshmanan. Loyalty programs cost you your personal data – are the rewards worth it? 2019.

[33] Isis Agora Lovecruft and Henry de Valence. curve25519-dalek. https://github.com/dalek-cryptography/curve25519-dalek, 2019.

[34] Milica Milutinovic, Italo Dacosta, Andreas Put, and Bart De Decker. ucentive: An efficient, anonymous and unlinkable incentives scheme. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 588–595, 2015.

[35] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51(2):231–262, 2004.

[36] David Pointcheval and Olivier Sanders. 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*, pages 111–126, 2016.

[37] Y. Sakemi, Lepidum, T. Kobayashi, and T. Saito. Pairing-friendly curves. Technical report, 3 2020.

[38] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 256–266, 1997.

[39] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, 2006.