The Pythia PRF Service

Adam Everspaugh**, Rahul Chatterjee***, Samuel Scott****, Ari Juels*†*, and Thomas Ristenpart*‡
*University of Wisconsin–Madison, {ace,rchat}@cs.wisc.edu
**Royal Holloway, University of London, sam.scott.2012@live.rhul.ac.uk
†Jacobs Institute, Cornell Tech, juels@cornell.edu
‡Cornell Tech, ristenpart@cornell.edu

Abstract

Conventional cryptographic services such as hardware-security modules and software-based key-management systems offer the ability to apply a pseudorandom function (PRF) such as HMAC to inputs of a client's choosing. These services are used, for example, to harden stored password hashes against offline brute-force attacks.

We propose a modern PRF service called PYTHIA designed to offer a level of flexibility, security, and ease-of-deployability lacking in prior approaches. The keystone of PYTHIA is a new cryptographic primitive called a *verifiable partially-oblivious PRF* that reveals a portion of an input message to the service but hides the rest. We give a construction that additionally supports efficient bulk rotation of previously obtained PRF values to new keys. Performance measurements show that our construction, which relies on bilinear pairings and zero-knowledge proofs, is highly practical. We also give accompanying formal definitions and proofs of security.

We implement PYTHIA as a multi-tenant, scalable PRF service that can scale up to hundreds of millions of distinct client applications on commodity systems. In our prototype implementation, query latencies are 15 ms in local-area settings and throughput is within a factor of two of a standard HTTPS server. We further report on implementations of two applications using PYTHIA, showing how to bring its security benefits to a new enterprise password storage system and a new brainwallet system for Bitcoin.

1 Introduction

Security improves in a number of settings when applications can make use of a cryptographic key stored on a remote system. As an important example, consider the compromise of enterprise password databases. Best practice dictates that passwords be hashed and salted before storage, but attackers can still mount highly effective brute-force cracking attacks against stolen databases.

Well-resourced enterprises such as Facebook [38] have therefore incorporated remote cryptographic operations to harden password databases. Before a password is stored or verified, it is sent to a *PRF service* external to the database. The PRF service applies a cryptographic function such as HMAC to client-selected inputs under a service-held secret key. Barring compromise of the PRF service, its use ensures that stolen password hashes (due to web server compromise) cannot be cracked using an offline brute-force attack: an attacker must query the PRF service from a compromised server for each password guess. Such online cracking attempts can be monitored for anomalous volumes or patterns of access and throttled as needed.

While PRF services offer compelling security improvements, they are not without problems. Even large organizations can implement them incorrectly. For example, Adobe hardened passwords using 3DES but in ECB mode instead of CBC-MAC (or another secure PRF construction) [23], a poor choice that resulted in disclosure of many of its customers' passwords after a breach. Perhaps more fundamental is that existing PRF services do not offer graceful remediation if a compromise is detected by a client. Ideally it should be possible to cryptographically erase (i.e., render useless via key deletion) any PRF values previously used by the client, without requiring action by end users and without affecting other clients. In general, PRF services are so inaccessible and cumbersome today that their use is unfortunately rare.

In this paper, we present a next-generation PRF service called PYTHIA to democratize cryptographic hardening. PYTHIA can be deployed within an enterprise to solve the issues mentioned above, but also as a public, multi-tenant web service suitable for use by any type of organization or even individuals. PYTHIA offers several security features absent in today's conventional PRF services that are critical to achieving the scaling and flexibility required to simultaneously support a variety of clients and applications. As we now explain, achieving these features necessitated innovations in both cryptographic primitive design and system architecture.

Key features and challenges. We refer to an entity using PYTHIA as a *client*. For example, a client might be a web server that performs password-based authentication for all of its end users. Intuitively, PYTHIA allows such a client to query the service and obtain the PRF output $Y = F_k(t, m)$ for a message m and a tweak t of the client's choosing under a client-specific secret key k held by the service. Here, the tweak t is typically a unique identifier for an end user (e.g., a random salt). In our running password storage example, the web server stores Y in a database to authenticate subsequent logins.

PYTHIA offers security features that at, first glance, sound mutually exclusive. First, PYTHIA achieves message privacy for m while requiring clients to reveal t to the server. Message privacy ensures that the PRF service obtains no information about the message m; in our password-storage example, m is a user's password. At the same time, though, by revealing t to the PRF service, the service can perform fine-grained monitoring of related requests: a high volume or otherwise anomalous pattern of queries on the same t would in our running example be indicative of an ongoing brute-force attack and might trigger throttling by the PRF service.

By using a unique secret key k for each client, PYTHIA supports individual key rotation should the value Y be stolen (or feared to be stolen). With traditional PRF services and password storage, such key rotation is a headache, and in many settings impractical, because it requires transitioning stored values $Y_1, \ldots, Y_n$ (one for each user account) to a new PRF key. The only way to do so previously was to have all n users re-enter or reset their passwords. In contrast, the new primitive employed for F_k in PYTHIA supports fast key rotation: the server can erase k, replace it with a new key k', and issue a compact (constant-sized) token with which the client can quickly update all of its PRF outputs. This feature also enables forward-security in the sense that the client can proactively rotate k without disrupting its operation.

PYTHIA provides other features as well, but we defer their discussion to Section 2. Already, those listed above surface some of the challenging cryptographic tensions that PYTHIA resolves. For example, the most obvious primitive on which to base PYTHIA is an oblivious PRF (OPRF) [27], which provides message privacy. But for rate-limiting, PYTHIA requires clients to reveal t, and existing OPRFs cannot hide only a portion of a PRF input. Additionally, the most efficient OPRFs (c.f., [28]) are not amenable to key rotation. We discuss at length other related concepts (of which there are many) in Section 7.

Partially-oblivious PRFs. We introduce *partially oblivious PRFs* (PO-PRFs) to rectify the above tension between fine-grained key management and bulk key management and achieve a primitive that supports batch key rotation. We give a PO-PRF protocol in the random oracle model (ROM) similar to the core of the identity-based non-interactive key exchange protocol of Sakai, Ohgishi, and Kasahara [44]. This same construction was also considered as a left-or-right constrained PRF by Boneh and Waters [13]. That said, the functionality achieved by our PO-PRF is distinct from these prior works and new security analyses are required. Despite relying on pairings, we show that the full primitive is fast even in our prototype implementation.

In addition to a lack of well-matched cryptographic primitives, we find no supporting formal definitions that can be adapted for verifiable PO-PRFs. (Briefly, previous definitions and proofs for fast OPRFs rely on hashing in the ROM before outputting a value [16, 28]; in our setting, hashing breaks key rotation.) We propose a new assumption (a one-more bilinear decisional Diffie-Hellman assumption), give suitable security definitions, and prove the security of the core primitive in PYTHIA under these definitions (in the appendix; complete results in [25]). Our new definitions and technical approaches may be of independent interest.

Using PYTHIA in applications. We implement PYTHIA and show that it offers highly practical performance on Amazon EC2 instances. Our experiments demonstrate that PYTHIA is practical to deploy using off-the-shelf components, with combined computation cost of client and server under 12 milliseconds. A single 8-core virtualized server can comfortably support over 1,000 requests per second, which is already within a factor of two of a standard HTTPS server in the same configuration. (Our PYTHIA implementation performs all communication over TLS.) We discuss scaling to handle more traffic volume in the body; it is straightforward given current techniques.

We demonstrate the benefits and practicality of PYTHIA for use in a diverse set of applications. First is our running example above: we build a new password-database system using a password "onion" that combines parallelized calls to PYTHIA and a conventional key hashing mechanism. Our onion supports PYTHIA key rotation, hides back-end latency to PYTHIA during logins (which is particularly important when accessing PYTHIA as a remote third-party service), and achieves high security in a number of compromise scenarios.

Finally, we show that PYTHIA provides valuable features for different settings apart from enterprise password storage. We implement a client that hardens a type

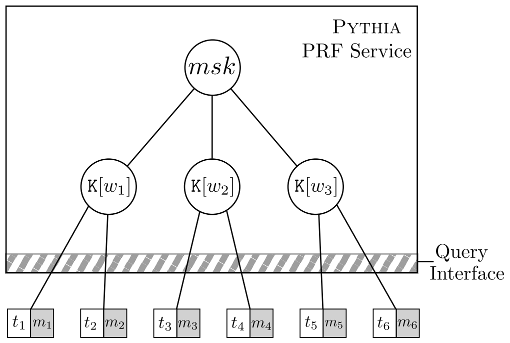

Figure 1: Diagram of PRF derivations enabled by PYTHIA. Everything inside the large box is operated by the server, which only learns tweaks and not the shaded messages.

of password-protected virtual-currency account called a "brainwallet" [14]; use of PYTHIA here prevents offline brute-force attacks of the type that have been common in Bitcoin.

Our prototype implementation of PYTHIA is built with open-source components and itself is open-source. We have also released Amazon EC2 images to allow companies, individuals, and researchers to spin-up PYTHIA instances for experimentation.

2 Overview and Challenges

We now give a high-level overview of PYTHIA, the motivations for its features, what prior approaches we investigated, and the threat models we assume. First we fix some terminology and a high-level conceptual view of what a PRF service would ideally provide. The service is provisioned with a master secret key msk. This will be used to build a tree that represents derived sub-keys and, finally, output values. See Figure 1, which depicts an example derivation tree associated with PYTHIA as well as which portions of the tree are held by the server (within the large box) and which are held by the client (the leaves). Keys of various kinds are denoted by circles and inputs by squares.

From the msk we derive a number of *ensemble keys*. Each ensemble key is used by a client for a set of related PRF invocations — the ensemble keys give rise to isolated PRF instances. We label each ensemble key in the diagram by $\mathtt{K}[w]$. Here w indicates a client-chosen *ensemble selector*. An *ensemble pre-key* $\mathtt{K}[w]$ is a large random value chosen and held by the server. Together, msk and $\mathtt{K}[w]$ are used to derive the ensemble key $k_w = \text{HMAC}(msk, \mathtt{K}[w])$. A table is necessary to support cryptographic erasure of (or updates to) individual ensemble keys, which amounts to deleting (or updating) a table entry.

Each ensemble key can be used to obtain PRF outputs of the form $F_{k_w}(t,m)$ where F is a (to-be-defined) PRF keyed by k_w, and the input is split into two parts. We call t a *tweak* following [30] and m the message. Looking ahead t will be made public to PYTHIA while m will be private. This is indicated by the shading of the PRF output boxes in the figure.

Deployment scenarios. To motivate our design choices and security goals, we relay several envisioned deployment scenarios for PYTHIA.

Enterprise deployment: A single enterprise can deploy PYTHIA internally, giving query access only to other systems they control. A typical setup is that PYTHIA fields queries from web servers and other public-facing systems that are, unfortunately, at high risk of compromise. PRF queries to PYTHIA harden values stored on these vulnerable servers. This is particularly suited to storing check-values for passwords or other low-entropy authentication tokens, where one can store $F_{k_w}(t,m)$ where t is a randomly chosen, per-user identifier (a salt) and m is the low-entropy password or authentication token. Here w can be distinct for each server using PYTHIA.

Public cloud service: A public cloud such as Amazon EC2, Google Compute Engine, or Microsoft Azure can deploy PYTHIA as an internal, multi-tenant service for their customers. Multi-tenant here means that different customers query the same PYTHIA service, and the cloud provider manages the service, ensemble pre-key table, etc. This enables smaller organizations to obtain the benefits of using PYTHIA for other cloud properties (e.g., web servers running on virtual machine instances) while leaving management of PYTHIA itself to experts.

Public Internet service: One can take the public cloud service deployment to the extreme and run PYTHIA instances that can be used from anywhere on the Internet. This raises additional performance concerns, as one cannot rely on fast intra-datacenter network latencies (sub-millisecond) but rather on wide-area latencies (tens of milliseconds). The benefit is that PYTHIA could then be used by arbitrary web clients, for example we will explore this scenario in the context of hardening brainwallets via PYTHIA.

One could tailor a PRF service to each of these settings, however it is better to design a single, application-agnostic service that supports all of these settings simultaneously. A single design permits reuse of open-source implementations; standardized, secure-by-default configurations; and simplifies the landscape of PRF services.

Security and functionality goals. Providing a single suitable design requires balancing a number of security and functionality goals. The most obvious requirements are for a service that: provides low-latency protocols (i.e., single round-trip and amenable for implementation as simple web interfaces); scales to hundreds of millions of ensembles; and produces outputs indistinguishable from random values even when adversaries can query the service. To this list of basic requirements we add:

- *Message privacy*: The PRF service must learn nothing about m. Message privacy supports clients that require sensitive values such as passwords to remain private even if the service is compromised, or to promote psychological acceptability in the case that a separate organization (e.g., a cloud provider) manages the service.
- *Tweak visibility*: The server must learn tweak t to permit fine-grained rate-limiting of requests.[1] In the password storage example, a distinct tweak is assigned to each user account, allowing the service to detect and limit guessing attempts against individual user accounts.
- *Verifiability:* A client must be able to verify that a PRF service has correctly computed F_{k_w} for a ensemble selector w and tweak/message pair t, m. This ensures, after first use of an ensemble by a client, that a subsequently compromised server cannot surreptitiously reply to PRF queries with incorrect values.[2]
- *Client-requested ensemble key rotations:* A client must be permitted to request a rotation of its ensemble pre-key $\mathtt{K}[w]$ to a new one $\widehat{\mathtt{K}[w]}$. The server must be able to provide an update token Δ_w to roll forward PRF outputs under $\mathtt{K}[w]$ to become PRF outputs under $\widehat{\mathtt{K}[w]}$, meaning that the PRF is *key-updatable* with respect to ensemble keys. Additionally, Δ_w must be *compact*, i.e., constant in the number of PRF invocations already performed under w. Clients can mandate that rotation requests be authenticated (to prevent malicious key deletion). A client must additionally be able to *transfer* an ensemble from one selector w to another selector w'.
- *Master secret rotations:* The server must be able to rotate the master secret key msk with minimal impact on clients. Specifically, the PRF must be key-updatable with respect to the master secret key msk so that PRF outputs under msk can be rolled forward to a new master secret $\widehat{msk}$. When such a rotation occurs, the server must provide a compact update token δ_w for each ensemble w.

[1]In principle, the server need only be able to link requests involving the same t, not learn t. Explicit presentation of t is the simplest mechanism that satisfies this requirement.

[2]This matters, for example, if an attacker compromises the communication channel but not the server's secrets (msk and $\mathtt{K}[w]$). Such an attacker must not be able to convince the client that arbitrary or incorrect values are correct.

- *Forward security:* Rotation of an ensemble key or master secret key results in complete erasure of the old key and the update token.

Two sets of challenges arise in designing PYTHIA. The first is cryptographic. It turns out that the combination of requirements above are not satisfied by any existing protocols we could find. Ultimately we realized a new type of cryptographic primitive was needed that proves to be a slight variant of oblivious PRFs and blind signatures. We discuss the new primitive, and our efficient protocol realizing it, in the next section. The second set of challenges surrounds building a full-featured service that provides the core cryptographic protocol, which we treat in Section 4.

3 Partially-oblivious PRFs

We introduce the notion of a (verifiable) partially-oblivious PRF. This is a two-party protocol that allows the secure computation of $F_{k_w}(t, m)$, where F is a PRF with server-held key k_w and t, m are the input values. The client can verify the correctness of $F_{k_w}(t, m)$ relative to a public key associated to k_w. Following our terminology, t is a tweak and m is a message. We say the PRF is partially oblivious because t is revealed to the server, but m is hidden from the server.

Partially oblivious PRFs are closely related to, but distinct from, a number of existing primitives. A standard oblivious PRF [27], or its verifiable version [28], would hide both t and m, but masking both prevents granular rate limiting by the server. Partially blind signatures [1] allow a client to obtain a signature on a similarly partially blinded input, but these signatures are randomized and the analysis is only for unforgeability which is insufficient for security in all of our applications.

We provide more comparisons with related work in Section 7 and a formal definition of the new primitive in Appendix B. Here we will present the protocol that suffices for PYTHIA. It uses an admissible bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ over groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q, and a pair of hash functions $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \{0,1\}^* \to \mathbb{G}_2$ (that we will model as random oracles). More details on pairings are provided in Appendix B. A secret key k_w is an element of $\mathbb{Z}_p$. The PRF F that the protocol computes is:

$$F_{k_w}(t, m) = e\big(H_1(t), H_2(m)\big)^{k_w} .$$

This construction coincides with the Sakai, Ohgishi, and Kasahara [44] construction for non-interactive identity-based key exchange, where t and m would be different identities and k_w a secret held by a trusted key authority. Likewise, this construction is equivalent to the left-or-right constrained PRF of Boneh and Waters [13]. The

```
PRF-Cl $(w,t,m)$                                PRF-Srv $(msk)$
$r \xleftarrow{\$} \mathbb{Z}_q$
$x \leftarrow H_2(m)^r$
                        $\xrightarrow{w,t,x}$
                                $\tilde{x} \leftarrow e(H_1(t),x)$
                                $k_w \leftarrow \mathrm{HMAC}(msk, \mathtt{K}[w])$
                                $p_w \leftarrow g^{k_w}$
                                $y \leftarrow \tilde{x}^{k_w}$
                                $\pi \xleftarrow{\$} \mathrm{ZKP}(\mathrm{DL}_g(p_w) = \mathrm{DL}_{\tilde{x}}(y))$
                        $\xleftarrow{p_w, y, \pi}$
If $p_w$ matches &
$\pi$ verifies then
    Ret $y^{1/r}$
Else Ret $\bot$
```

Figure 2: The partially-oblivious PRF protocol used in PYTHIA. The value π is a non-interactive zero-knowledge proof that the indicated discrete logs match. The client also checks that p_w matches ones seen previously when using selector w.

contexts of these prior works are distinct from ours and our analyses will necessarily be different, but we note that all three settings similarly exploit the algebraic structure of the bilinear pairing. See Section 7 for further discussion of related work.

The client-server protocol that computes $F_{k_w}(t,m)$ in a partially-oblivious manner is given in Figure 2. There we let g be a generator of $\mathbb{G}_1$. We now explain how the protocol achieves our requirements described in the last section.

Blinding the message: In our protocol, the client blinds the message m, hiding it from the server, by raising it to a randomly selected exponent $r \xleftarrow{\$} \mathbb{Z}_q$. As $e\big(H_1(t), H_2(m)^r\big) = e\big(H_1(t), H_2(m)\big)^r$, the client can unblind the output y of PRF-Srv by raising it to $1/r$. This protocol hides m unconditionally, as $H_2(m)^r$ is a uniformly random element of $\mathbb{G}_2$.

Verifiability: The protocol enables a client to verify that the output of PRF-Srv is correct, assuming the client has previously stored p_w. The server accompanies the output y of the PRF with a zero-knowledge proof π of correctness.

Specifically, for a public key $p_w = g^{k_w}$, where g is a generator of $\mathbb{G}_1$, the server proves $\mathrm{DL}_g(p_w) = \mathrm{DL}_{\tilde{x}}(y)$. Standard techniques (see, e.g., Camenisch and Stadler [17]) permit efficient ZK proofs of this kind in the random oracle model. [3] The notable computational costs for the server are one pairing and one exponentiation in $\mathbb{G}_T$; for the client, one pairing and two exponentiations in $\mathbb{G}_T$. [4]

Efficient key updates: The server can quickly and easily update the key k_w for a given ensemble selector w by replacing the table entry $s = \mathtt{K}[w]$ with a new, randomly selected value s', thereby changing $k_w = \mathrm{HMAC}(msk, s)$ to $k'_w = \mathrm{HMAC}(msk, s')$. It can then transmit to the client an update token of the form $\Delta_w = k'_w / k_w \in \mathbb{Z}_q$.

The client can update any stored PRF value $F_{k_w}(t,m) = e\big(H_1(t), H_2(m)\big)^{k_w}$ by raising it to Δ_w; it is easy to see that $F_{k_w}(t,m)^{\Delta_w} = F_{k'_w}(t,m)$.

The server can use the same mechanism to update msk, which requires generating a new update token for each w and pushing these tokens to clients as needed.

Unblinded variants. For deployments where obliviousness of messages is unnecessary, we can use a faster, unblinded variant of the PYTHIA protocol that dispenses with pairings. The only changes are that the client sends m to the server, there is no unblinding of the server's response, and, instead of computing

$$\tilde{x} \leftarrow e(H_1(t), x)$$

the server computes

$$\tilde{x} \leftarrow H_3(t \,\|\, m) .$$

All group operations in this unblinded variant are over a standard elliptic curve group $\mathbb{G} = \langle g \rangle$ of order q and we use a hash function $H_3 : \{0,1\}^* \to \mathbb{G}$.

An alternative unblinded construction would be to have the server apply the Boneh-Lynn-Shacham short signatures [12] to the client-submitted $t \,\|\, m$; verification of correctness can be done using the signature verification routine, and we can thereby avoid ZKPs. This BLS variant may save a small amount of bandwidth.

These unblinded variants provide the same services (verifiability and efficient key updates) and security with the obvious exception of the secrecy of the message m. In some deployment contexts an unblinded protocol may be sufficient, for example when the client can maintain state and submit a salted hash m instead of m directly. In this context, the salt should be held as a secret on the client and never sent to the server.

4 The PYTHIA Service Design

Figure 3 gives the high-level API exposed by PYTHIA to a client. We now describe its functions in terms of the lifecycle of an ensemble key. We assume a security parameter n specifying symmetric key lengths; a typical choice would be $n = 128$.

[3]Some details: The prover picks $v \xleftarrow{\$} \mathbb{Z}_q$ and then computes $t_1 = g^v$ and $t_2 = \tilde{x}^v$ and $c \leftarrow H_3(g, p_w, \tilde{x}, y, t_1, t_2)$. Let $u = v - c \cdot k$. The proof is $\pi = (c, u)$. The verifier computes $t'_1 = g^u \cdot p_w^c$ and $t'_2 = \tilde{x}^u y^c$. It outputs true if $c = H_3(g, p_w, \tilde{x}, y, t'_1, t'_2)$.

[4]The client's pairing can be pre-computed while waiting for the server's reply.

Command	Description
$\mathsf{Init}(w\ [, options])$	Create table entry $\mathtt{K}[w]$ (for ensemble key k_w)
$\mathsf{Eval}(w, t, m)$	Return PRF output $F_{k_w}(t, m)$
$\mathsf{Reset}(w,\ authtoken)$	Update $\mathtt{K}[w]$ (and thus k_w); return update token Δ_w
$\mathsf{GetAuth}(w)$	Send one-time authentication token $authtoken$ to client

Figure 3: The basic PYTHIA API.

We defer to later sections the underlying client-server protocols and to Appendix A details on key lifecycle management options, additional API calls for token management and ensemble transfer, and a discussion of master secret key rotation.

Ensemble initialization. To begin using the PYTHIA service, a client creates an ensemble key for selector w by invoking $\mathsf{Init}(w\ [, options])$. PYTHIA generates a fresh, random table entry $\mathtt{K}[w]$. Recall that ensemble key $k_w = \text{HMAC}(msk, \mathtt{K}[w])$. So Init creates k_w as a byproduct.

Ideally, w should be an unguessable byte string. (An easily guessed one may allow attackers to squat on a key selector, thereby mounting a denial-of-service (DoS) attack.) For some applications, as we explain below, this isn't always possible. If an ensemble key for w already exists, then the PYTHIA service returns an error to the client. Otherwise, the client receives a message signifying that initialization is successful.

Init includes a number of options we detail in Appendix A.

PRF evaluation. To obtain a PRF value, a client can perform an evaluation query $\mathsf{Eval}(w, t, m)$, which returns $F_{k_w}(t, m)$. Here t is a tweak and m is a message. To compute the PRF output, the client and server perform a one-round cryptographic protocol (meaning a single message from client to server, and one message back). We present details in Section 3, but remind the reader that t is visible to the server in the client-server protocol invoked by Eval, while m is blinded.

The server rate-limits requests based on the tweak t, and can also raise an alert if the rate limit is exceeded. We give example rate limiting policies in Section 5.

Ensemble-key reset. A client can request that an ensemble key k_w be reset by invoking $\mathsf{Reset}(w)$. This reset is accomplished by overwriting $\mathtt{K}[w]$ with a fresh, random value. The name service returns a compact (e.g., 256-bit) update token Δ_w that the client may use to update all PRF outputs for the ensemble. It stores this token locally, encrypted under a public key specified by the client, as explained below.

Note that reset *results in erasure of the old value of* k_w. Thus a client that wishes to delete an ensemble key k_w permanently at the end of its lifecycle can do so with a Reset call.

Reset is an authenticated call, and thus requires the following capability.

Authentication. To authenticate itself for API calls, the client must first invoke GetAuth, which has the server transmit an (encrypted) authentication token $authtoken$ to the client out-of-band. The token expires after a period of time determined by a configuration parameter in PYTHIA. Our current implementation uses e-mail for this, see Appendix A for more details. Of course, in some deployments one may want authentication to be performed in other ways, such as tokens dispensed by administrators (for enterprise settings) or simply given out on a first-come-first-serve basis for each ensemble identifier (for public Internet services).

4.1 Implementation

We implemented a prototype PYTHIA PRF service as a web application accessed over HTTPS. All requests are first handled by an nginx web server with uWsgi as the application server gateway that relays requests to a Django back-end. The PRF-Srv functionality is implemented as a Django module written in Python. Storage for the server's key table and rate-limiting information is done in MongoDB.

We use the Relic cryptographic library [2] (written in C) with our own Python wrapper. We use Barreto-Naehrig 254-bit prime order curves (BN-254) [4]. These curves provide approximately 128-bits of security.

In our experiments the service is run on a single (virtual) machine, but our software stack permits components (web server, application sever, database) to be distributed among multiple machines with updates to configuration files.

For the purpose of comparison, we implemented three variants of the PYTHIA service. The first two are the unblinded protocols described in Section 3. In these two schemes, the client sends m in the clear (possibly hashed with a secret salt value first) and the server replies with $y = H_1(t \,\|\, m)^k$. In the first scheme, denoted UNB, the server provides $p = g_1^k$ and a zero-knowledge proof where g_1 is a generator of $\mathbb{G}_1$. The second scheme, denoted BLS, uses a BLS signature for verification. The server provides $p = g_2^k$ where g_2 is a generator of $\mathbb{G}_2$ and the client verifies the response by computing and comparing the values: $e(y, g_2) = e(H_1(t \,\|\, m), p)$.

Our partially-oblivious scheme is denoted PO.

For the evaluation below we use a Python client implementing PRF-Cl for all three schemes using the same

	Time (µs)		
Group	Group Op	Exp	Hashing
$\mathbb{G}_1$	5.7	175	77
$\mathbb{G}_2$	6.7	572	210
$\mathbb{G}_T$	9.8	1145	–
pairing operation (e) takes 1005 µs			

Figure 4: Time taken by each operation in BN-254 groups. Hashing times are for 64-byte inputs.

libraries indicated above for the server and httplib2 to perform HTTPS requests.

4.2 Performance

For performance and scalability evaluation we hosted our PYTHIA server implementation on Amazon's Elastic Compute Cloud (EC2) using a c4.xlarge instance which provides 8 virtual CPUs (Intel Xeon third generation, 2.9GHz), 15 GB of main memory, and solid state storage. The web server, nginx, was configured with basic settings recommended for production deployment including one worker process per CPU.

Latency. We measured client query latency for each protocol using two clients: one within the same Amazon Web Service (AWS) availability zone (also c4.xlarge) and one hosted at the University of Wisconsin–Madison with an Intel Core i7 CPU (3.4 GHz). We refer to the first as the LAN (local-area network) setting and the second as the WAN (wide-area network) setting. In the LAN settings we used the AWS internal IP address. All queries were made over TLS and measurements include the time required for clients to blind messages and unblind results (PO), as well as verify proofs provided by the server (unless indicated otherwise). All machines used for evaluation were running Ubuntu 14.04.

Microbenchmarks for group operations appear in Figure 4 and Figure 5 shows the timing of individual operations that comprise a single PRF evaluation. All results are mean values computed over 10,000 operations. These values were captured on an EC2 c4.xlarge instance using the Python profiling library line_profiler. The most expensive operations, by a large margin, are exponentiation in $\mathbb{G}_t$ and the pairing operation. By extension, PO sign, prove, and verify operations become expensive.

We measured latencies averaged over 1,000 PRF requests (with 100 warmup requests) for each scheme and the results appear in Figure 6. Computation time dominates in the LAN setting due to the almost negligible network latency. The WAN case with cold connections (no HTTP KeepAlive) pays a performance penalty due to the four round-trips required to set up a new TCP and TLS connection. While even 400 ms latencies are not prohibitive in our applications, straightforward engineering improvements would vastly improve WAN timing: using TLS session resumption, using lower-latency secure protocol like QUIC [43], or even switching to a custom UDP protocol (for an example one for oblivious PRFs, see [5]).

Server Op	Time (ms)		
Table	1.2		
Rate-limit	0.9		
	UNB	BLS	PO
Sign	0.3	0.3	1.5
Prove	0.5	0.3	2.5

Client Op	UNB	BLS	PO
Blind	-	-	0.3
Unblind	-	-	1.2
Verify	0.9	2.0	4.0

Figure 5: Computation time for major operations to perform a PRF evaluation. Table retrieves $K[w]$ from database; Rate-limit updates rate-limiting record in database; and Sign generates the PRF output;

	Latency (ms)					
	LAN			WAN		
Scheme	Cold	Hot	No π	Cold	Hot	No π
UNB	7.0	3.8	2.4	389	82	80
BLS	7.9	4.9	2.4	392	85	80
PO	14.9	11.8	5.2	403	96	84
RTT ping		0.1			82	

Figure 6: Average latency to complete a PRF-Cl with client-server communication over HTTPS. LAN: client and server in the same EC2 availability zone. WAN: server in EC2 US-West (California) and client in Madison, WI. Hot connections made with HTTP KeepAlive enabled; cold connections with KeepAlive disabled. No π: KeepAlive enabled; prove and verify computations are skipped.

Throughput. We used the distributed load testing tool autobench to measure maximum throughput for each scheme. We compare to a static page containing a typical PRF response served over HTTPS as a baseline. We used two clients in the same AWS region as the server. All connections were cold: no TLS session resumption or HTTP KeepAlive. The maximum throughput for a static page is 2,200 connections per second (cps); UNB and BLS 1,400 cps; and PO 1,350 cps. Thus our PYTHIA implementation can handle a large number of clients on a single EC2 instance. If needed, the implementation can be scaled with standard techniques (e.g., a larger number of web servers and application servers on the front-end with a distributed key-value store on the back-end).

Storage. Our implementation stores all ensemble prekey table (K) entries and rate-limiting information in MongoDB. A table entry is two 32 byte values: a SHA-

256 hash of the ensemble selector w and its associated value $\mathtt{K}[w]$. In MongoDB the average storage size is 195 bytes per entry (measured as the average of 100K entries), including database overheads and indexes. This implementation scales easily to 100 M clients with under 20 GB of storage.

To rate-limit queries, our implementation stores tweak values along with a counter and a timestamp (to expire old entries) in MongoDB. Tweak values are also hashed using SHA-256 which ensures entries are of constant length. In our implementation each distinct tweak requires an average of 144 bytes per entry (including overheads and indexes). Note however that rate limiting entries are purged periodically as counts are only required for one rate-limiting period. Our implementation imposes rate-limits at hour granularity. Assuming a maximum throughput of 2,000 requests per second, rate-limiting storage never exceeds 1 GB.

All told, with only 20 GB stored data, PYTHIA can serve over 100 M clients and perform rate-limiting at hour granularity. Thus fielding a database for PYTHIA can be accomplished on commodity hardware.

5 Password Onions

Web servers and other systems frequently store passwords in hashed form. A *password onion* is the result of additionally invoking a PRF service to harden the hash. In currently suggested onions, one sequentially combines local hashing and application of the PRF service.

We now present a service that we have implemented on top of PYTHIA for managing password onions. First, we describe the limitations of contemporary systems as exemplified by a recently disclosed architecture employed by Facebook [39]. Then we show how our password-onion system, which was easily engineered on top of PYTHIA, can address these limitations.

In what follows, we use the term "client" or "web server" to denote the server performing authentication and storing derived values from passwords and "PRF server" to denote the PYTHIA service.

5.1 Facebook password onion

An example of a contemporary system, used by Facebook, is given in Figure 7.[5] Their PRF service applies HMAC using a service-held secret and returns the result. In this architecture, an adversary that compromises the web server and the password hashes it stores must still

```
PW-Onion($pw$)
$h_1 \leftarrow$ MD5($pw$)
$sa \leftarrow_{\$} \{0,1\}^{160}$
$h_2 \leftarrow$ HMAC[SHA-1]($h_1, sa$)
$h_3 \leftarrow$ PRF-Cl($h_2$) $=$ HMAC[SHA-256]($h_2, msk$)
$h_4 \leftarrow$ scrypt($h_3, sa$)
$h_5 \leftarrow$ HMAC[SHA-256]($h_4$)
Ret ($sa, h_5$)
```

Figure 7: The Facebook password onion. PRF-Cl(h_2) invokes the Facebook PRF service HMAC[SHA-256](h_2, K_s) with PRF-service secret key K_s.

mount an online attack against the PRF service to compromise accounts. This is a big advance on the hashing-only practices that are commonly used.

The Facebook architecture nevertheless has some shortcomings. It is easy to see from Figure 7 that Facebook's system, like most contemporary PRF services, lacks several important features present in PYTHIA. One is message privacy: the Facebook PRF service applies HMAC to h_2. This is the salted hash of the password, and so learning the salt as well as compromising the PRF service suffices to re-enable offline brute-force attacks. This threat is avoided by PYTHIA due to blinding.

Another feature is batch key updates. In fact, the Facebook PRF service doesn't permit autonomous key updates at all, in the sense of an update to msk that can be propagated into PRF output updates. Should the client (password database) be compromised, the only way to reconstitute a hash in an existing password onion is to *wait until a user logs in and furnishes pw.* It is not clear whether the Facebook PRF service performs granular rate-limiting, although no such capability is indicated in [38]. PYTHIA, as we shall see, addresses all of these issues by design in our password onion system.

The Facebook onion also presents a subtle performance issue. By applying cryptographic primitives serially, the time to hash a password equals the time for local computations, call it t_{local}, *plus* the time for the round-trip PRF service call, call it t_{prf}. An attacker that compromises the web service and PRF service incurs no network latency, and thus may gain a considerable advantage in guessing time over an honest web server. In our PYTHIA-based password onion service, we address this issue by observing that it is possible to avoid serialization of key derivation functions on the web server and the PRF service call. That is, we introduce in our PYTHIA-based service the idea of *parallelizable password onions*.

[5]This figure is of "archaeological" interest. It appears that vulnerabilities in MD5 led to the addition of a layer of processing under SHA-1; when vulnerabilities were found in SHA-1, Facebook then added layers of SHA-256. As we explain later, full-blown replacement of MD5 and SHA-1 with SHA-256 was not easily accomplished.

```
UpParOnion($w, sa, pw$)
$z \leftarrow$ PBKDF($pw, sa$)
$u \leftarrow$ PRF-Cl($w, sa, pw$)
$h \leftarrow u^z$
Ret ($h, sa$)
```

Figure 8: An updatable, parallelizable password onion. PRF-Cl returns elements of a group $\mathbb{G}$. The value w is a unique PRF-service identifier for the web server (e.g., a random 256-bit string) and sa is a random per-user salt value.

5.2 PYTHIA password onion

The onion algorithm we construct for PYTHIA is shown in Figure 8. For PYTHIA, the output of PRF-Cl is an element of a group $\mathbb{G}_T$. To use this service, a web server stores (h, sa) upon password registration; it verifies a proffered password pw' by checking that UpParOnion$(w, sa, pw') = h$. Written out we have that:

$$h = u^z = e(H_1(sa), H_2(pw))^{k_w z}.$$

This design ensures that the key update functions in the PYTHIA API may be used to update onions as well. For example, to update an ensemble key k_w to k'_w, the service computes and furnishes to the web server an update token $\Delta_w = k'_w/k_w$. The web server may compute h^{Δ_w} for each stored value h.

Parallelization. Password verification here is *parallelizable* in the sense that z and u may be computed independently and then combined. Such parallel implementation of the onion achieves a password verification latency of $\max\{t_{local}, t_{prf}\}$ (plus a single exponentiation), as opposed to $t_{local} + t_{prf}$ in a serialized implementation.

A web server generally aims to achieve a verification latency equal to some latency target T that is high enough to slow offline brute-force attacks, but low enough not to burden users. For a parallelized onion a web server can meet its latency target by setting $t_{local}, t_{prf} \approx T$. At the same time an offline attacker that has compromised the web server and PYTHIA must perform about $t_{local}+t_F > T$ work to check a single password guess, where t_F is the computation time of F_{k_w} (i.e., t_{prf} minus network latency). An attacker can parallelize, but her total work still goes up relative to the serial onion approach for the same latency target T.

We estimate the security improvement of parallel onions over serial onions using our benchmarks from Section 4.2. We fix a login latency budget of $T = 300$ ms.[6] The latency costs for a PYTHIA query with hot connections are 12 ms (LAN) and 96 ms (WAN). If one performs computations serially with a fixed T then PBKDF computations need to be reduced by 4% (LAN) and 32% (WAN) compared to the parallel approach. In the event that the PYTHIA server and password database are compromised, the serial onion enables speedup of offline dictionary attacks by the same percentages.

[6]This is the default setting for Python's bcrypt and scrypt modules, though all PBDKFs are tunable so one can choose T to be any value desired.

Rate limiting and logging. The transparency of tweaks enables the PYTHIA PRF service in this setting to execute any of a wide range of rate-limiting policies with per-account visibility (in contrast to what may be in Facebook an account-blind PRF service). As an example demonstrating the flexibility of our architecture, in our implementation PYTHIA performs a tiered rate-limiting: for a given account (t), it limits queries to at most 10 per hour per account, and at most 300 per month. (In expectation, guessing a random 4-digit PIN would require 1.4 years under this policy.) It logs violations of these thresholds. In a production environment, it could also send alerts to security administrators.

We emphasize that a wide range of other rate-limiting policies is possible. We also point out that PYTHIA's rate limiting supplements that normally implemented at the web server for remote login requests. PYTHIA performs rate limiting and may issue alerts even if the web server is compromised.

Key update. The key update calls in the PYTHIA API, and the ability to rotate either k_w or msk efficiently, propagates up to the password onion service. Key updates instantly invalidate the web server's existing password database—a useful capability in case of compromise. A compromised database becomes useless to an attacker attempting to recover passwords, even with the ability to query PYTHIA. Using a key update token, the web server can then recover from compromise by refreshing its database.

We created a client simulator with MongoDB and the mongoengine Python module. With this we benchmarked key updates with 100,000 database entries. The client requested a key update from PYTHIA, received the update token Δ_w, and updated each database entry. The complete update required less than 1 ms per entry, and terminated in less than 97 seconds for all 100,000 entries. For a larger database we assume updates scale linearly, and so an update for 1 million users completes in under 17 minutes.

The web server need not need lock the database to perform updates; it can execute them in parallel with normal login operations. Doing so does require additional versioning information for each entry to indicate the version of k_w (in the simplest form, whether or not it has received the latest update).

6 Hardened Brainwallets

Brainwallets are a common but dangerous way to secure accounts in the popular cryptocurrency Bitcoin, as well as in less popular cryptocurrencies such as Litecoin. Here we describe how the PYTHIA service can be used directly as a means to harden brainwallets. This application showcases the ease with which a wide variety of applications can be engineered around PYTHIA.

How brainwallets work. Every Bitcoin account has an associated private / public key pair (sk, pk). The private key sk is used to produce digital (ECDSA) signatures that authorize payments from the account. The public key pk permits verification of these signatures. It also acts as an account identifier; a Bitcoin address is derived by hashing pk (under SHA-256 and RIPEMD-160) and encoding it (in base 58, with a check value).

Knowledge of the private key sk equates with control of the account. If a user loses a private key, she therefore loses control over her account. For example, if a high entropy key sk is stored exclusively on a device such as a mobile phone or laptop, and the device is seized or physically destroyed, the account assets become irrecoverable.

Brainwallets offer an attractive remedy for such physical risks of key loss. A brainwallet is simply a password or passphrase P memorized by a Bitcoin account holder. The private key sk is generated directly from P. Thus the user's memory serves as the only instrument needed to authorize access to the account.

In more detail, the passphrase is typically hashed using SHA-256 to obtain a 256-bit string $sk =$ SHA-256(P). Bitcoin employs ECDSA signatures on the `secp256k1` elliptic curve; with high probability ($\approx 1 - 2^{-126}$), sk is less than the group order, and a valid ECDSA private key. (Some websites employ stronger key derivation functions. For example, WrapWallet by keybase.io [29] derives sk from an XOR of each of PBKDF2 and scrypt applied to P and permits use of a user-supplied salt.)

Since a brainwallet employs only P as a secret, and does not necessarily use any additional security measures, an attacker that guesses P can seize control of a user's account. As account addresses are posted publicly in the Bitcoin system (in the "blockchain"), an attacker can easily confirm a correct guess. Brainwallets are thus vulnerable to brute-force, offline guessing attacks. Numerous incidents have come to light showing that brainwallet cracking is pandemic [14].[7]

[7] At one point, rumor had it that cracking brainwallets was more profitable than "mining,", the basic process of generating fresh Bitcoins.

6.1 A PYTHIA-hardened brainwallet

PYTHIA offers a simple, powerful means of protecting brainwallets against offline attack. Hardening P in the same manner as an ordinary password yields a strong key $\tilde{P}$ that can serve in lieu of P to derive sk.

To use PYTHIA, a user chooses a unique identifier id, e.g., her e-mail address, an account identifier $acct$, and a passphrase P. The identifier $acct$ might be used to distinguish among Bitcoin accounts for users who wish to use the same password for multiple wallets. The client then sends $(w = id, t = id \,\|\, acct, m = P)$ to the PYTHIA service to obtain the hardened value $F_{k_w}(t, m) = \tilde{P}$. Here, id is used both as an account identifier and as part of the salt. Message privacy in PYTHIA ensures that the service learns nothing about P. Then $\tilde{P}$ is hashed with SHA-256 to yield sk. The corresponding public key pk and address are generated in the standard way from sk [7].

PYTHIA forces a would-be brainwallet attacker to mount an online attack to compromise an account. Not only is an online attack much slower, but it may be rate-limited by PYTHIA and detected and flagged. As the PYTHIA service derives $\tilde{P}$ using a user-specific key, it additionally prevents an attacker from mounting a dictionary attack against multiple accounts. While in the conventional brainwallet setting, two users who make use of the same secret P will end up controlling the same account, PYTHIA ensures that the same password P produces distinct per-user key pairs.

Should an attacker compromise the PYTHIA service and steal msk and K, the attacker must still perform an offline brute-force attack against the user's brainwallet. So in the worst case, a user obtains security with PYTHIA at least as good as without it.

Additional security issues. A few subtle security issues deserve brief discussion:

- *Stronger KDFs:* To protect against brute-force attack in the event of PYTHIA compromise, a resource-intensive key-derivation function may be desirable, as is normally used in password databases. This can be achieved by replacing the SHA-256 hash of $\tilde{P}$ above with an appropriate KDF computation, or alternatively using an onion approach described in Section 5.
- *Denial-of-service:* By performing rate-limiting, PYTHIA creates the risk of targeted denial-of-service attacks against Bitcoin users. As Bitcoin is pseudonymous, use of an e-mail address as a PYTHIA key-selector suffices to prevent such attacks against users based on their Bitcoin addresses alone. Users also have the option, of course, of using a semi-secret id. A general DoS attack against

the PYTHIA service is also possible, but of similar concern for Bitcoin itself [8].

- *Key rotation:* Rotation of an ensemble key k_w (or the master key msk) induces a new value of $\tilde{P}$ and thus a new (sk, pk) pair and account. A client can handle such rotations in the naïve way: transfer funds from the old address to the new one.
- *Catastrophic failure of* PYTHIA*:* If a PYTHIA service fails catastrophically, e.g., msk or $\mathtt{K}$ is lost, then in a typical setting, it is possible simply to reset users' passwords. In the brainwallet case, the result would be loss of virtual-currency assets protected by the server—a familiar event for Bitcoin users [35]. This problem can be avoided, for instance, using a threshold implementation of PYTHIA, as mentioned in Section 6.2 or storing sk in a secure, offline manner like a safe-deposit box for disaster recovery.

6.2 Threshold Security

In order to gain both redundancy and security, we give a threshold scheme that can be used with a number of Pythia servers to protect a secret under a single password. This scheme uses Shamir's secret sharing threshold scheme [45] and gives (k, n) threshold security. That is, initially, n Pythia servers are contacted and used to protect a secret s, and then any k servers can be used to recover s and any adversary that has compromised fewer than k Pythia servers learns no information about s.

Preparation. The client chooses an ensemble key selector w, tweak t, password P, and contacts n Pythia servers to compute $q_i = \text{PRF-Cl}_i(w, t, P) \mod p$ for $0 < i \leq n$. The client selects a random polynomial of degree $k - 1$ with coefficients from $\mathbb{Z}_p^*$ where p is a suitably large prime: $f(x) = \sum_{j=0}^{k-1} x^j a_j$. Let the secret $s = a_0$. Next the client computes the vector $\Phi = (\phi_1, ..., \phi_n)$ where $\phi_i = f(i) - q_i$. The client durably stores the value Φ, but does not need to protect it (it's not secret). The client also stores public keys p_i from each Pythia server to validate proofs when issuing future queries.

Recovery. The client can reconstruct s if she has Φ by querying any k Pythia servers giving k values q_i. These q_i values can be applied to the corresponding Φ values to retrieve k distinct points that lie on the curve $f(x)$. With k points on a degree $k - 1$ curve, the client can use interpolation to recover the unique polynomial $f(x)$, which includes the curve's intercept $a_0 = s$.

Security. If an adversary is given Φ, w, t, the public keys p_i, a ciphertext based on s, and the secrets from $m < k$ Pythia servers, the adversary has no information that will permit her to verify password guesses offline. Compared to [45], this scheme reduces the problem of storing n secrets to having access to n secure OPRFs and durable (but non-secret) storage of the values Φ and public keys p_i.

Verification. Verification of server responses occurs within the Pythia protocol. If a server is detected to be dishonest (or goes out of service), it can be easily replaced by the client without changing the secret s. To replace a Pythia server that is suspected to be compromised or detected as dishonest, the client reconstructs the secret s using any k servers, executes Reset operations on all remaining servers: this effects a cryptographic erasure on the values Φ and $f(x)$. The client then selects a new, random polynomial, keeping a_0 fixed, and generates and stores an updated Φ' that maps to the new polynomial.

7 Related Work

We investigated a number of designs based on existing cryptographic primitives in the course of our work, though as mentioned none satisfied all of our design goals. Conventional PRFs built from block ciphers or hash functions fail to offer message privacy or key rotation. Consider instead the construction $H(t \,\|\, m)^{k_w}$ for $H : \{0,1\}^* \rightarrow \mathbb{G}$ a cryptographic hash function mapping onto a group $\mathbb{G}$. This was shown secure as a conventional PRF by Naor, Pinkas, and Reingold assuming decisional Diffie-Hellman (DDH) is hard in $\mathbb{G}$ and when modeling H as a random oracle [40]. It supports key rotations (in fact it is key-homomorphic [11]) and verifiability can be handled using non-interactive zero-knowledge proofs (ZKP) as in PYTHIA. But this approach fails to provide message privacy if we submit both t and m to the server and have it compute the full hash.

One can achieve message-hiding by using blinding: have the client submit $X = H(t \,\|\, m)^r$ for random $r \in \mathbb{Z}_{|\mathbb{G}|}$ and the server reply with X^{k_w} as well as a ZKP proving this was done correctly. The resulting scheme is originally due to Chaum and Pedersen [19], and suggested for use by Ford and Kaliski [26] in the context of threshold password-authenticated secret sharing (see also [3, 15, 20, 34]). There an end user interacts with one or more blind signature servers to derive a secret authentication token. If $\mathbb{G}$ comes equipped with a bilinear pairing, one can dispense with ZKPs. The resulting scheme is Boldyreva's blinded version [10] of BLS signatures [12]. However, neither approach provides granular rate limiting when blinding is used: the tweak t is hidden from the server. Even if the client sends t as well, the server cannot verify that it matches the one used to compute X and attackers can thereby bypass rate limits.

To fix this, one might use Ford-Kaliski with a separate secret key for each tweak. This would result in having a different key for each unique w, t pair. Message privacy is maintained by the blinding, and querying $w, t, H(t' \| m)^r$ for $t \neq t'$ does not help an attacker circumvent per-tweak rate limiting. But now the server-side storage grows in the number of unique w, t pairs, a client using a single ensemble w must now track N public keys when they use the service for N different tweaks, and key rotation requires N interactions with the PRF server to get N separate update tokens (one per unique tweak for which a PRF output is stored). When N is large and the number of ensembles w is small as in our password storage application, these inefficiencies add significant overheads.

Another issue with the above suggestions is that their security was only previously analyzed in the context of one-more unforgeability [42] as targeted by blind signatures [18] and partially blind signatures [1]. (Some were analyzed as conventional PRFs, but that is in a model where adversaries do not get access to a blinded server oracle.) The password onion application requires more than unforgeability because message privacy is needed. (A signature could be unforgeable but include the entire message in its signature, and this would obviate the benefits of a PRF service for most applications.) These schemes, however, can be proven to be one-more PRFs, the notion we introduce, under suitable one-more DDH style assumptions using the same proof techniques found in Appendix B.

Fully oblivious PRFs [27] and their verifiable versions [28] also do not allow granular rate limiting. We note that the Jarecki, Kiayias, and Krawczyk constructions of verifiable OPRFs [28] in the RO model are essentially the Ford-Kaliski protocol above, but with an extra hash computation, making the PRF output $H'(t \| m \| H(t \| m)^{k_w})$. Our notion of one-more unpredictability in the appendix captures the necessary requirements on the inner cryptographic component, and might modularize and simplify their proofs. Their transform is similar to the unique blind signature to OPRF transformation of Camenisch, Neven, and shelat [16]. None of these efficient oblivious PRF protocols support key rotations (with compact tokens or otherwise) as the final hashing step destroys updatability.

The setting of capture-resilient devices shares with ours the use of an off-system key-holding server and the desire to perform cryptographic erasure [32, 33]. They only perform protocols for encryption and signing functionalities, however, and not (more broadly useful) PRFs. They also do not support granular rate limiting and master secret key rotation.

Our main construction coincides with prior ones for other contexts. The Sakai, Ohgishi, and Kasahara [44] identity-based non-interactive key exchange protocol computes a symmetric encryption key as $e(H_1(ID_1), H_2(ID_2))^k$ for k a master secret held by a trusted party and ID_1 and ID_2 being the identities of the parties. See [41] for a formal analysis of their scheme. Boneh and Waters suggest the same construction as a left-or-right constrained PRF [13]. The settings and their goals are different from ours, and in particular one cannot use either as-is for our applications. Naïvely one might hope that returning the constrained PRF key $H_1(t)^{k_w}$ to the client suffices for our applications, but in fact this totally breaks rate-limiting. Security analysis of our protocol requires new techniques, and in particular security must be shown to hold when the adversary has access to a half-blinded oracle — this rules out the techniques used in [13, 41].

Key-updatable encryption [11] and proxy re-encryption [9] both support key rotation, and could be used to encrypt password hashes in a way supporting compact update tokens and that prevents offline brute-force attacks. But this would require encryption and decryption to be handled by the hardening service, preventing message privacy.

Verifiable PRFs as defined by [21, 22, 31, 36] allow one to verify that a known PRF output is correct relative to a public key. Previous verifiable PRF constructions are not oblivious, let alone partially oblivious.

Threshold and distributed PRFs [21, 37, 40] as well as distributed key distribution centers [40] enable a sufficiently large subset of servers to compute a PRF output, but previous constructions do not provide the granular rate limiting and key rotation we desire. However, it is clear that there are situations where applications would benefit from a threshold implementation of PYTHIA, for both redundancy and distribution of trust, as discussed in Section 6.2 for the case of brainwallets.

8 Conclusion

We presented the design and implementation of PYTHIA, a modern PRF service. Prior works have explored the use of remote cryptographic services to harden keys derived from passwords or otherwise improve resilience to compromise. PYTHIA, however, transcends existing designs to simultaneously support granular rate limiting, efficient key rotation, and cryptographic erasure. This set of features, which stems from practical requirements in applications such as enterprise password storage, proves to require a new cryptographic primitive that we refer to as a partially oblivious PRF.

Unlike a (fully) oblivious PRF, a partially oblivious PRF causes one portion of an input to be revealed to the server to enable rate limiting and detection of online brute-force attacks. We provided a bilinear-pairing based construction for partially oblivious PRFs that is highly efficient and simple to implement (given a pairings library), and also supports efficient key rotations. A formal proof of security is unobtainable using existing techniques (such as those developed for fully oblivious PRFs). We thus gave new definitions and proof techniques that may be of independent interest.

We implemented PYTHIA and show how it may be easily integrated it into a range of applications. We designed a new enterprise "password onion" system that improves upon the one recently reported in use at Facebook. Our system permits fast key rotations, enabling practical reactive and proactive key management, and uses a parallelizable onion design which, for a given authentication latency, imposes more computational effort on attackers after a compromise. We also explored the use of PYTHIA to harden brainwallets for cryptocurrencies.

Acknowledgements

The authors thank Kenny Paterson for feedback on an early draft of this paper. This work was supported in part by NSF grants CNS-1330308, CNS-1065134, and CNS-1330599, as well as a gift from Microsoft.

References

[1] Masayuki Abe and Tatsuaki Okamoto. Provably secure partially blind signatures. In *Advances in Cryptology–CRYPTO*. Springer, 2000.

[2] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[3] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *Computer and Communications Security*. ACM, 2011.

[4] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography*. Springer, 2006.

[5] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Dupless: server-aided encryption for deduplicated storage. In *USENIX Security*. USENIX, 2013.

[6] Mihir Bellare, Chanathip Namprempre, David Pointcheval, Michael Semanko, and Matthew Franklin. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3), 2003.

[7] Technical background of version 1 Bitcoin addresses. https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses.

[8] Bitcoin wiki, "weaknesses". https://en.bitcoin.it/wiki/Weaknesses.

[9] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *Advances in Cryptology–EUROCRYPT*. Springer, 1998.

[10] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *Public Key Cryptography*. Springer, 2002.

[11] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In *Advances in Cryptology–CRYPTO*. Springer, 2013.

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology–ASIACRYPT*. Springer Berlin Heidelberg, 2001.

[13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology-ASIACRYPT*. Springer, 2013.

[14] Brainwallet. https://en.bitcoin.it/wiki/Brainwallet.

[15] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *Computer and Communications Security*. ACM, 2012.

[16] Jan Camenisch, Gregory Neven, and abhi shelat. Simulatable adaptive oblivious transfer. In *Advances in Cryptology–EUROCRYPT*. Springer Berlin Heidelberg, 2007.

[17] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical Report No. 260, Dept. of Computer Science, ETH Zurich, 1997.

[18] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology*. Springer, 1983.

[19] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Advances in Cryptology–CRYPTO*. Springer, 1993.

[20] Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. In *Advances in Cryptology–EUROCRYPT*. Springer, 2003.

[21] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In *Public Key Cryptography*. Springer, 2002.

[22] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography*. Springer, 2005.

[23] Paul Ducklin. Anatomy of a password disaster – Adobe's giant-sized cryptographic blunder, 2013. https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/.

[24] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology–CRYPTO*. Springer, 1985.

[25] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The Pythia PRF service. Full version of this paper. http://pages.cs.wisc.edu/~ace/papers/pythia.pdf.

[26] Warwick Ford and Burton S. Kaliski, Jr. Server-assisted generation of a strong secret from a password. In *International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, 2000.

[27] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography*. Springer, 2005.

[28] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-PAKE in the password-only model. In *Advances in Cryptology–ASIACRYPT*. Springer, 2014.

[29] Max Krohn and Chris Coyne. Wrap Wallet. https://keybase.io/warp.

[30] Moses Liskov, Ronald L Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology–CRYPTO*. Springer, 2002.

[31] Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in Cryptology–CRYPTO*. Springer, 2002.

[32] Philip MacKenzie and Michael K Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing*, 16(4), 2003.

[33] Philip MacKenzie and Michael K Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security*, 2(1), 2003.

[34] Philip MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In *Advances in Cryptology–CRYPTO*. Springer, 2002.

[35] R. McMillan. The inside story of Mt. Gox, bitcoin's $460 million disaster. *Wired*, 2014.

[36] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science*. IEEE, 1999.

[37] Silvio Micali and Ray Sidney. A simple method for generating and sharing pseudo-random functions, with applications to clipper-like key escrow systems. In *Advances in Cryptology–CRYPTO*. Springer, 1995.

[38] Alec Muffet. Facebook: Password hashing & authentication. Presentation at Real World Crypto, 2015.

[39] Allec Muffet. Facebook: Password hashing and authentication. https://video.adm.ntnu.no/pres/54b660049af94.

[40] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In *Advances in Cryptology–EUROCRYPT*. Springer, 1999.

[41] Kenneth G Paterson and Sriramkrishnan Srinivasan. On the relations between non-interactive key distribution, identity-based encryption and trapdoor discrete log groups. *Designs, Codes and Cryptography*, 52(2), 2009.

[42] David Pointcheval and Jacques Stern. Provably secure blind signature schemes. In *Advances in Cryptology–ASIACRYPT*. Springer, 1996.

[43] Jim Roskind. QUIC: Multiplexed stream transport over UDP. *Google working design document*, 2013.

[44] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. In *Cryptography and Information Security*, 2000.

[45] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

A Additional PYTHIA API details

Many PYTHIA-dependent services can benefit from additional API features and calls beyond the primary ones discussed in the body of the paper. (For example, the PYTHIA password onion system in Section 5 uses the Transfer API call.) We detail these other API features in this appendix.

Key-management options. The client can specify a number of options in the call Init regarding management of the ensemble key k_w. The client can provide a contact email address to which alerts and authentication tokens may be sent. (If no e-mail is given, no API calls requiring authentication are permitted at present and no alerts are provided. Later versions of PYTHIA will support other authentication and alerting methods.)

Selector option	Description
Email	Contact email for selector
Resettable	Whether client-requested rotations allowed
Limit	Establish rate-limit per t
Time-out	Date/time to delete k_w
Public-key	Key under which to encrypt and store update and authentication tokens
Alerts	Whether to email contact upon rate limit violation

Figure 9: Optional settings for establishing key selectors in PYTHIA.

Command	Description
$\mathsf{Transfer}(w, w' \,[, options])$	Creates new ensemble w'; outputs update token $\Delta_{w \to w'}$; resets k_w
$\mathsf{SendTokens}(w, authtoken)$	Sends stored update tokens to client
$\mathsf{PurgeTokens}(w, authtoken)$	Purges all stored update tokens for ensemble w

Figure 10: The PYTHIA API. The individual calls are explained in detail in the text.

The client can specify whether k_w should be resettable (default is "yes"). The client can specify a limit on the total number of F_{k_w} queries that should be allowed before resetting $\mathtt{K}[w]$ (default is unlimited) and/or an absolute expiration date and time in UTC at which point $\mathtt{K}[w]$ is deleted (default is no time-out). Either of these options overrides the resettable flag. The client can specify a public key pk_{cl} for a public-key encryption scheme under which to encrypt authentication tokens and update tokens (for Reset, Transfer, as described below, and for master secret key rotations). Finally, the client can request that alerts be sent to the contact email address in the case of rate limit violations. This option is ignored if no contact email is provided. The options are summarized in Figure 9.

PYTHIA also offers some additional API calls, given in Figure 10, which we now describe.

Ensemble transfer. A client can create a new ensemble w' (with the same options as in Init) while receiving an update token that allows PRF outputs under ensemble w to be rolled forward to w'. This is useful for importing a password database to a new server. The PYTHIA service returns an update token $\Delta_{w \to w'}$ for this purpose and stores it encrypted under pk_{cl}. For the case $w' = w$, this call also allows option updates on an existing ensemble w.

Update-token handling. The PYTHIA service stores update tokens encrypted under pk_{cl}, with accompanying timestamps for versioning. The API call SendTokens causes these to be e-mailed to the client, while PurgeTokens causes update-token ciphertexts to be deleted from PYTHIA.

Note that once an update token is deleted, old PRF values to which the token was not applied become cryptographically erased — they become random values unrelated to any messages. A client can therefore delete the key associated with an ensemble by calling Reset and PurgeTokens.

Master secret rotations. PYTHIA can also rotate its master secret key msk to a new key msk'. Recall that ensemble keys are computed as $k_w = \text{HMAC}(msk, \mathtt{K}[w])$, so rotation of msk results in rotation of all ensemble keys. To rotate to a new msk', the server computes k_w for all ensembles w with entries in $\mathtt{K}$, and stores δ_w encrypted under pk_{cl}. If no encryption key is set, then the token is stored in the clear. This is a forward-security issue while it remains, but only for that particular key ensemble. At this point msk is safe to delete. Clients can be informed of the key rotation via e-mail.

Subsequent SendTokens requests will return the resulting update token, along with any other stored update tokens for the ensemble. If multiple rotations occur between client requests, then these can be aggregated in the stored update token for each ensemble. This is trivial if they are stored in the clear (just multiply the new token against the old) and also works if they are encrypted with an appropriately homomorphic encryption scheme such as ElGamal [24].

B Formal Security Analyses

We provide formal security notions for partially oblivious PRFs, and proofs of security relative to them for our scheme from Section 3.

Partially-oblivious PRFs. A partially oblivious PRF protocol $\Pi = (\mathcal{K}, \text{PRF-Cl}, \text{PRF-Srv}, F)$ consists of the following. The key generation algorithm $\mathcal{K}$ outputs a public key and private key pair (pk, sk). We assume that from sk one can compute pk easily. The PRF-Srv algorithm takes input the secret key sk and a client request message (a bit string) and returns a server response message (another bit string). The client algorithm PRF-Cl takes inputs a tweak t and message m, can make a single call to PRF-Srv, and outputs a value. Finally we associate to the protocol a keyed function $F_{sk}\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. A scheme is correct if executing $\text{PRF-Cl}^{\text{PRF-Srv}_{sk}(\cdot)}(t,m)$ with fresh coins matches $F_{sk}(t,m)$ with probability one. In words, the protocol computes the appropriate function of t, m.

Bilinear pairing setups. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be groups all of order p that have associated to them an admissible bilinear pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Recall that for generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, there exists a generator $g_T \in \mathbb{G}_T$ such that $e(g_1^\alpha, g_2^\beta) = g_T^{\alpha\beta}$ for all $\alpha, \beta \in \mathbb{Z}_p$. As shorthand for below we refer to a pairing setup $\mathbb{G} = (g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and assume some compact description of $\mathbb{G}$ as a bit-string where appropriate.

The scheme. The partially-oblivious PRF at the core[8] of our bilinear pairing scheme from Section 3 is as follows for some fixed pairing setup $\mathbb{G}$. Let $H_1\colon \{0,1\}^* \to \mathbb{G}_1$ and $H_2\colon \{0,1\}^* \to \mathbb{G}_2$ be hash functions that we will later model as random oracles.

Key generation $\mathcal{K}$ picks a random exponent sk and computes a public key $pk = g_1^s k$. The PRF-Cl(t, m) algorithm computes a mask $r \leftarrow\!\!{\scriptstyle\$}\; \mathbb{Z}_p$ and sends t and $x = H_2(m)^r$ to the server. The PRF-Srv(sk, t, x) computes $y = e(H_1(t), x)^s k$ and a ZKP π that $\text{DL}_{g_1}(pk) = \text{DL}_{\tilde{x}}(y)$ where $\tilde{x} = e(H_1(t), x)$. It sends pk, y, π to the client, who verifies the ZKP, deletes it, and then outputs $y^{1/r}$. The correctness of the scheme follows from the correctness of the ZKP and the properties of the pairing.

The ZKP is used to ensure that a malicious server responds as per the protocol. In the following security analyses we focus primarily on malicious clients, and for simplicity analyze a simpler version of the protocol that omits the ZKP. The proofs found below can be extended to the full protocol by applying the zero-knowledge security of the proof systems that we use (i.e., use the zero-knowledge simulator to produce fake, but realistic-looking to the client, proofs).

B.1 Unpredictability Security

We define a one-more unpredictability security notion. It modifies one-more unforgeability [42] to be suitable for the setting of unpredictable functions (as opposed to publicly verifiable signatures). The game is shown in Figure 11. We associate to any protocol Π, adversary $\mathcal{A}$, and query number q the one-more-unpredictability advantage defined as

$$\mathbf{Adv}^{\text{om-unp}}_{\Pi,q}(\mathcal{A}) = \Pr\left[\,\text{om-UNP}^{\mathcal{A}}_{\Pi,q} \Rightarrow \mathsf{true}\,\right].$$

The probability here (and for games defined later below) is over all random coins used by the procedures and the adversary. The event refers to the probability that the value returned by the main procedure is true. In words, the definition requires that an adversary cannot produce ℓ outputs of the PRF using less than ℓ queries on partially-blinded inputs to the server. One can easily extend this notion to deal with full blinded inputs as well, but we will not need this.

[8]For brevity we omit key selectors here, and instead focus on analyzing security for a single key instance. Assuming properly generated keys for each selector, one can show that security for a single key instance implies security for many.

```
Game om-UNP$_\Pi^\mathcal{A}$
$(pk, sk) \leftarrow\$ \mathcal{K}$
$c \leftarrow 0$
$(t_1, m_1, \sigma_1), \ldots, (t_\ell, m_\ell, \sigma_\ell) \leftarrow\$ \mathcal{A}^{\text{PRF-Srv}, H_1, H_2}$
If $\exists i \neq j \,.\, (t_i, m_i) = (t_j, m_j)$ then Ret false
Ret $(\wedge_i(\sigma_i = F_{sk}^{H_1,H_2}(t_i, m_i)) \wedge c < \ell)$

PRF-Srv$(t, Y)$
$c \leftarrow c + 1$
Ret PRF-Srv$_{sk}^{H_1,H_2}(t, Y)$
```

Figure 11: Security game for one-more unpredictability.

This notion of security is sufficient for PYTHIA in applications where the output of the protocol is not stored, but rather used as an unforgeable credential such as with our hardened Brainwallet application (Section 6).

The security of our scheme is based on the following one-more bilinear computational Diffie-Hellman (BCDH) problem, an extension of the one-more CDH assumption given by Boldyreva [10]. To the best of our knowledge this assumption is new, but it is a straightforward adaptation of previous one-more assumptions [6, 10] to our setting. For a pairing setup $\mathbb{G}$, game om-BCDH$_\mathbb{G}$ is defined in Figure 12. In words, the adversary gets a group element $g_1^s k \in \mathbb{G}_1$ as well as target oracles $\text{Targ}_1, \text{Targ}_2$ that return random group elements in $\mathbb{G}_1, \mathbb{G}_2$ respectively. Finally the adversary can query a helper oracle Help that raises $\mathbb{G}_T$ elements to the k. To win, it must compute ℓ values $e(X_i, Y_j)^k$ for ℓ larger than the number of helper queries and each X_i, Y_j a unique pair of (distinct) values returned by the target oracle. Let $\mathbf{Adv}_\mathbb{G}^{\text{om-cdh}}(\mathcal{B}) = \Pr\left[\text{om-BCDH}_\mathbb{G}^\mathcal{B} \Rightarrow \mathsf{true}\right]$.

We have the following theorem establishing the one-more unpredictability of our scheme The proof is essentially identical to the proof of Boldyreva's blind signatures [10].

Theorem 1 *Let Π be the simplified partially oblivious PRF protocol for a pairing setup $\mathbb{G}$ and H_1, H_2 modeled as random oracles. Then for any one-more unpredictability adversary $\mathcal{A}$ making at most q* PRF-Srv *queries, we give in the proof below a one-more CDH adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_\Pi^{\text{om-unp}}(\mathcal{A}) \leq \mathbf{Adv}_\mathbb{G}^{\text{om-cdh}}(\mathcal{B})$$

where $\mathcal{B}$ runs in time that of $\mathcal{A}$ plus $\mathcal{O}(q)$ group operations.

Proof: We assume without loss of generality that $\mathcal{A}$ never repeats a query to either random oracle and makes a random oracle $H_1(t_i)$ and $H_2(m_i)$ query for each (t_i, m_i, σ_i) triple it outputs. The adversary $\mathcal{B}$ will work as follows when given inputs $\mathbb{G}, X$ and access to oracles $\text{Targ}_1, \text{Targ}_2, \text{Help}$. First, it runs $\mathcal{A}$. Whenever $\mathcal{A}$ makes

```
Game om-BCDH$_\mathbb{G}^\mathcal{B}$
$sk \leftarrow\$ \mathbb{Z}_p$
$q_h, q_{1,t}, q_{2,t} \leftarrow 0$
$(i_1, j_1, \sigma_1), \ldots, (i_\ell, j_\ell, \sigma_\ell) \leftarrow\$ \mathcal{A}^{\text{Targ}_1, \text{Targ}_2, \text{Help}}(\mathbb{G}, g_1^s k)$
If $q_h \geq \ell$ then Ret false
If $\exists \alpha \,.\, (i_\alpha > q_{1,t}) \vee (j_\alpha > q_{2,t})$ then Ret false
If $\exists \alpha \neq \beta \,.\, (i_\alpha, j_\alpha) = (i_\beta, j_\beta)$ then Ret false
Ret $\forall \alpha \,.\, e(X_{i_\alpha}, Y_{j_\alpha})^k = \sigma_\alpha$

Targ$_1$
$q_{1,t} \leftarrow q_{1,t} + 1$ ; $X_{q_{1,t}} \leftarrow\$ \mathbb{G}_1$ ; Ret $X_{q_{1,t}}$

Targ$_2$
$q_{2,t} \leftarrow q_{2,t} + 1$ ; $Y_{q_{2,t}} \leftarrow\$ \mathbb{G}_2$ ; Ret $Y_{q_{2,t}}$

Help$(Z)$
$q_h \leftarrow q_h + 1$ ; Ret $Z^{sk}$
```

Figure 12: Security game for a one-more BCDH assumption for bilinear pairing setting $\mathbb{G} = (g_1, g_2, g_t, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.

an $H_1(t)$ query, $\mathcal{B}$ queries Targ_1 to obtain a $\mathbb{G}_1$-element that we will denote $X[t]$, sets c_t to be the number of H_1 queries so far (including the current), and returns $X[t]$ to $\mathcal{A}$. Whenever $\mathcal{A}$ makes an $H_2(m)$ server query, $\mathcal{B}$ queries Targ_2, obtains a $\mathbb{G}_2$-element that we will denote $Y[m]$, sets d_m to be the number of H_2 queries so far (including the current), and returns $Y[m]$ to $\mathcal{A}$. Whenever $\mathcal{A}$ makes a PRF-Srv(t, Y) query, the adversary $\mathcal{B}$ computes $Z \leftarrow e(H_1(t), Y)$, and then queries Z to its helper oracle Help to obtain a value $\sigma \in \mathbb{G}_T$. It returns σ to $\mathcal{A}$.

Eventually $\mathcal{A}$ outputs a series of triples $(t_1, m_1, \sigma_1), \ldots, (t_q, m_q, \sigma_q)$. At this point adversary $\mathcal{B}$ outputs the sequence of pairs $(c_{t_1}, d_{m_1}, \sigma_1), \ldots, (c_{m_q}, d_{m_q}, \sigma_q)$.

Suppose $\mathcal{A}$ wins its game. Then it made at most $q - 1$ queries to PRF-Srv and so $\mathcal{B}$ makes at most $q - 1$ queries to Help. It is also the case that all predictions by $\mathcal{A}$ are for unique tag, message pairs, meaning that $\mathcal{B}$'s output will also be for unique pairs of targets. Finally, it is clear that correct predictions σ_i are also BCDH solutions. ▌

B.2 Pseudorandomness Security

See the full version of this paper [25] for *one-more PRF* security definitions and associated proofs.