Secure Account Recovery for a Privacy-Preserving Web Service

Ryan Little
Boston University[*]

Lucy Qin
Georgetown University[†]

Mayank Varia
Boston University

Abstract

If a web service is so secure that it does not even know—and does not want to know—the identity and contact info of its users, can it still offer account recovery if a user forgets their password? This paper is the culmination of the authors' work to design a cryptographic protocol for account recovery for use by a prominent secure matching system: a web-based service that allows survivors of sexual misconduct to become aware of other survivors harmed by the same perpetrator. In such a system, the list of account-holders must be safeguarded, even against the service provider itself.

In this work, we design an account recovery system that, on the surface, appears to follow the typical workflow: the user types in their email address, receives an email containing a one-time link, and answers some security questions. Behind the scenes, the defining feature of our recovery system is that the service provider can perform email-based account validation without knowing, or being able to learn, a list of users' email addresses. Our construction uses standardized cryptography for most components, and it has been deployed in production at the secure matching system.

As a building block toward our main construction, we design a new cryptographic primitive that may be of independent interest: an oblivious pseudorandom function that can either have a fully-private input or a partially-public input, and that reaches the same output either way. This primitive allows us to perform online rate limiting for account recovery attempts, without imposing a bound on the creation of new accounts. We provide an open-source implementation of this primitive and provide evaluation results showing that the end-to-end interaction time takes 8.4-60.4 ms in fully-private input mode and 3.1-41.2 ms in partially-public input mode.

Content Warning: *This paper discusses, at a high level, the issue of sexual assault on college campuses, particularly in Section 1. From Section 3 onward, the paper is more focused on the technical design of the account recovery protocol.*

[*]Email addresses: {ryanlit,varia}@bu.edu
[†]Email address: lucy.qin@georgetown.edu

1 Introduction

Account recovery—the ability for users to regain access to their accounts after losing their password—is a near-ubiquitous feature of account-based web services. However, the typical account recovery system is not private, and relies on the web service to retain some information about the existence of an account under some identity (e.g., a username or email address). In this work, we designed an account recovery process to be compatible with a secure matching system operated by a nonprofit organization, Callisto, that uses conventional account recovery workflows but does not retain any plaintext information about a user's email address, username, security questions, or security answers. As of October 2023, this protocol has been deployed by Callisto for their matching system, which is currently available on college campuses across the United States.

We discuss the application space of secure matching systems to provide context for this work, and then elaborate on the challenge of account recovery in this setting and provide details about our protocol. While this protocol was designed to support Callisto's secure matching system, our cryptographic techniques are generic and may also be used independently by other web services that want to enable account recovery without collecting personal information about users.

Overview of secure matching systems. Secure matching systems are cryptographic tools that intend to support survivors of sexual assault by connecting them with one another. Although statistics may be difficult to capture, there is evidence that sexual assault is highly prevalent on college campuses, in particular. A 2019 study across 33 universities found that 13% of students experienced sexual assault while enrolled [30]. Due to mandatory reporting policies that limit survivors' control over their own narratives, victim-blaming, fear of retaliation, and other barriers, survivors rarely formally report incidents of sexual assault [41, 48, 68]. For those who do, survivors may experience retraumatization through interactions with the police and/or the legal system and may ulti-

mately feel failed by formal avenues of pursuing justice [67]. Inspired by the #MeToo movement, secure matching systems [8, 50, 60, 62, 75] allow survivors of sexual assault to document their experiences and securely connect with other survivors who have been harmed by the same perpetrator to seek mutual support and explore different pathways either collectively or individually, since healing and seeking justice may look different for each survivor.

Callisto is a nonprofit organization that currently deploys a free secure matching system across college campuses in the United States [18]. Callisto allows survivors of sexual assault to document details of their assault in an encrypted record (for potential future use) and participate in matching with other survivors. If two or more survivors match by providing identifying information about the same perpetrator, contact information for each survivor is revealed to a legal options counselor (a lawyer) so that rather than being cryptographically protected, the same information is then protected by attorney-client privilege [19]. Prior to a match, information about both a survivor and perpetrator is not accessible to Callisto. Once connected to a legal options counselor, survivors are then offered the opportunity to connect with one another. Upon connecting, they may choose to collectively pursue legal justice, restorative justice, or nothing at all. Matching enables survivors to seek support from a legal options counselor and one another to explore options (individually or collectively) for action, ideally providing each survivor more support when decision-making about their options for pursuing further action, should they want to.

The challenge of account recovery. In standard account recovery processes, the service provider typically stores an email address that it uses for validation and to share a link that allows a user to regain access to their account. Since contact information about a survivor may reveal sensitive information (in our setting, whether an individual is a survivor of sexual assault), privacy-respecting service providers like Callisto cannot store any contact information about its users. Maintaining a list of account-holders creates a sensitive and valuable asset that may be targeted by bad actors. It may also create vulnerability toward legal threats, through subpoenas for information about account holders. Hence, a secure matching system must protect both *data* and *metadata*—even against the service provider itself, or anyone who might attempt to compromise it. In more detail:

1. All data submitted by survivors must be encrypted with a key derived from the user's password, so that the service cannot access user-submitted information (e.g. identifying information about a user, details of their assault) until they have reached the matching threshold. This cryptographic problem has been addressed in prior works [8, 50, 60, 62, 75] with general-purpose secure multi-party computation (MPC) and specific primitives like oblivious pseudorandom functions, group signatures, and verifiable secret sharing.
2. Additionally, the service provider must not have (or even have a simple way to recover) a list of all account-holders, since even the fact of a survivor using the system is extremely sensitive. On its own, this problem can be addressed using cryptographically oblivious login mechanisms (e.g., [57]), along with network and systems security precautions supporting anonymous communications and not keeping long-term logs that could later be exfiltrated by a hacker or compelled by an authority.

When addressing these two requirements together, one runs into a practical challenge: *what happens if an account-holder forgets their password*? This issue is inevitable in a system where strong, unique passwords are encouraged but logins are infrequent, and by default it would make all of the user's encrypted data irrecoverable.

In a typical website, account recovery involves the service provider sending an email to the email address on file; however, this is incompatible with our no-metadata requirement. Even federating the email address list and the delivery of emails among multiple servers using secure multi-party computation [3, 49, 78] is insufficient for our needs because it is slow, difficult to implement, and (most importantly) is still vulnerable to the risk of all servers being compelled to disclose their secret shares of the email list.

1.1 This work

In this work, we describe the design of an account recovery system such that a web service provider (like Callisto) *does not store email addresses, and cannot easily provide a list of all email addresses even if compelled to do so (e.g., through a subpoena)*.

Protocol design. At a high level, our cryptographic protocol involves a client who (if honest) wishes to regain access to their account, along with a collection of recovery servers. The protocol uses email validation and a set of security questions/answers to determine whether the client should be given access to the account. Only the client recovers the cryptographic key that protects their account data; the recovery servers never see the security answers or key material.

In more detail, our protocol has two phases: some work is performed during *account creation* when the client knows their password and cryptographic key material, and subsequently the client can run an *account recovery* procedure if they forget the password and associated key. The recovery procedure itself contains three parts: request, verification, and restoration. First, the client requests recovery by (obliviously) providing their email address and responses to any generic questions (e.g., a phone number). Second, the servers verify that the client has control of this email address. Third, the servers fetch the client-specific security questions; the client's

answers to these questions can be used to restore access to their account.

Design requirements. Our protocol is grounded in the framework of trauma-informed design that acknowledges the impact trauma may have on the user experience and seeks to avoid retraumatization [19, 34, 77]. When designing an account recovery process that is trauma-informed in this setting, the user's interactions with the process must be familiar and easy to use: in particular, it must adhere to an expected account recovery workflow (e.g., "click on a link"), run seamlessly in a web browser, and not impose any restrictions or rate-limiting of legitimate uses.

On the back-end, our protocol had to accommodate the resource constraints of Callisto. For ease of deployment and maintenance on the web, the recovery servers must run on commodity hardware and the protocol must only use standard public and symmetric key cryptography in widely used and available software libraries.

Our protocol has been implemented and tested, and as of October 2023 is running live in production within Callisto's secure matching system. It uses only the cryptographic primitives that were already in place for Callisto's existing matching process, for ease of development and maintenance. However, we emphasize that our account recovery protocol is independent of Callisto's services; it is generic and can be used by any privacy-respecting website that desires not to keep personally-identifiable logs about their users.

Security guarantees. Our protocol provides security against three types of attackers. As with account recovery in other web services, email validation is our main security protection against external attackers. If an attacker has additionally compromised some of our recovery servers, they must still perform an online dictionary attack to obtain any email address, which we rate-limit through honest recovery servers. Finally, even if all of the recovery servers are compromised then an offline dictionary attack is still needed—which is not impossible, but onerous enough to allow Callisto to defend itself against legal compulsion of the email list.

The challenge of rate-limiting. Beyond standard symmetric key crypto primitives and a slow hash function, the only crypto primitive that we require is an *oblivious pseudorandom function* (OPRF). The client independently interacts with each server to compute an OPRF for two distinct reasons: (a) to hide the client's email address, contact information, and security answers from the recovery servers, and (b) as part of our rate-limiting mechanism, whereby an honest recovery server can detect and stop a brute-force dictionary attack of contact information or security answers.

The idea of using an OPRF for rate-limiting is not new. Several prior works have designed *partially* oblivious pseudorandom functions (pOPRFs) where the server publicly sees some information about which account is being accessed. This approach can aid in rate-limiting (e.g., [42, 81]).

However, rate-limiting in our context is challenging. This is because we need to hide the contact info and security answers during *both* account creation and recovery, but we want rate-limiting *only* in account recovery. Account creation is a delicate moment for a survivor, and we do not want to risk re-traumatizing them by denying signups due to rate-limiting.

K-pop: a new type of OPRF. As a result, in this work we design a new kind of primitive that can operate either as a pOPRF or as an OPRF. Identical outputs are produced in either mode. During account creation, we desire a pOPRF that uses a fresh public nonce as input, so that the servers are assured that this PRF query is for a new account and therefore do not need to impose rate-limiting. During account recovery, we desire a standard OPRF to hide which account is being accessed; hence, only here is rate-limiting needed.

We call this new primitive a *kaleidoscopic partially oblivious PRF*, or K-pop. In this work, we provide a formal definition of a K-pop along with a construction from any group where the Discrete Diffie-Hellman assumption is hard (e.g., elliptic curve groups), in the random oracle model.

1.2 Our contributions

In summary, this work provides three contributions.

First, we contribute a new primitive called a *kaleidoscoping partially oblivious PRF*, or K-pop. This function can be calculated either as an OPRF or pOPRF, and reaches the same result either way. We construct a K-pop without bilinear maps and provide an open-source implementation of this construction.

Second, we use the K-pop in order to construct an interactive protocol for account recovery where even the service provider does not require, and does not learn, the email addresses of its account-holders. This construction uses only standard public and symmetric key primitives that are available in many crypto libraries. This construction has been developed, tested, integrated, and deployed within Callisto's matching system.

Third, we provide two proofs of security of our account recovery protocol: a game-based security analysis against static adversaries, and a universally composable (UC) security [22] analysis against adaptive adversaries. In both analyses, a key challenge is to ensure security up to abort against malicious adversaries, but while avoiding the use of zero knowledge proofs (and therefore, also avoiding verifiable OPRFs).

1.3 Ethics and limitations

The question of whether a new technology is even needed and appropriate must be determined in consultation with subject-matter experts and in partnership with relevant communities.

Since these systems are intended to support survivors of sexual assault, designing any cryptosystem (new or otherwise) should only be done if deemed necessary and useful to survivors. *Development of this account recovery protocol was specifically requested by Callisto*, based on their users' needs.

We wish to stress the importance of developing technology through a trauma-informed lens that minimizes the risk of retraumatization. Lack of account recovery may exacerbate existing trauma as it prevents survivors from accessing prior information they had submitted. This may produce feelings of loss of control as personal details about sexual violence they had experienced are rendered inaccessible. Reactions to trauma, such as heightened anxiety and hypervigilance, can also impact survivors' interactions with technology [34]. It is therefore critical that an account recovery process is familiar and easy to use so as to not increase survivors' cognitive burden. When creating this account recovery process, we worked extensively with Callisto to make sure that the design would meet their technical constraints and also enable them to use a common and familiar user interface on the front end. As part of our ongoing work together, we have been in regular communication with Callisto about the publication of this work, which they are supportive of.

Both our problem formulation and our proposed solution have important limitations. First, our threat models are tailored to the specific needs of Callisto. Assumptions made in some of our threat models, such as honest server behavior during account creation and a non-colluding set of servers, may not be reasonable for other web services. Second, implementing our protocol necessarily increases the attack surface of the web service. A security bug in implementation could expose user information and put vulnerable users at risk. Third, providing stronger anonymity on the Internet may not always be a desirable goal, and we stress the importance of working with domain experts to determine the appropriate balance between any privacy technology and other social principles, policy objectives, or legal requirements in the context of a particular web service.

Lastly, our work does not address the structural barriers that make it difficult for survivors of sexual assault to find support and seek their own pathways toward accountability and healing. We note the significant barriers survivors face in addressing harms due to victim-blaming and institutional failures. Our protocol, and secure matching systems in general, are not solutions to this root problem, but nevertheless may be helpful to some survivors when navigating their options.

2 Technical Overview

In this section, we provide an informal summary of the technical contributions in this work. We begin by detailing the system setup, threats considered, and rationale for our design choices. Then, we describe our K-pop and account recovery constructions and explain how they build upon prior work.

2.1 Design principles and threat model

In this section, we take a deeper dive into the security threats and design considerations that influence our protocol design.

System setup. The account recovery system contains two types of actors: one or more clients who possess user accounts in the system, and a collection of N servers who participate in account recovery. The design follows the 'anytrust' paradigm in which only one server needs to be honest to provide the strongest security, and it also provides some meaningful security guarantees even if all N servers are compromised. Here, an honest server has two responsibilities: not sharing its OPRF secret key with the other servers, and properly enforcing rate limits on requests from clients and the other servers. All network communications are protected using TLS, and the adversary is presumed to have some amount of network control but not a fully global view—for instance, we presume that the adversary does not have full visibility or control of the client's email service provider.

Design principles. In this section, we expand upon the design requirements discussed in §1.1. Then, we describe some design accommodations that were deemed acceptable in our setting by our team and Callisto. While admittedly some of these principles are specific to our specific tech transition, nevertheless we believe that many of our design principles—and the motivations behind them—may generalize to other web services that want private account recovery.

As discussed in §1.3, our account recovery system is designed for survivors of sexual assault who may be using the system while having just experienced trauma. It is essential that any technology that is built is trauma-informed and is designed to account for these experiences to avoid retraumatization. Since user accounts for Callisto's matching system contain personal information that may include details about sexual assault, it is imperative that survivors are able to access their accounts quickly and seamlessly. Therefore, an account recovery process must be easy and familiar for survivors to use. As such, we designed a protocol to be compatible with a front-end user experience that is common to account recovery: a user enters their email along with additional contact information and receives security answers they must respond to. Survivors should not be burdened with learning new mechanisms that are complicated or unfamiliar. Trauma-informed design principles such as *safety* and *trust* [77] are pursued by creating an account recovery process that operates in a predictable manner while respecting the privacy of the user's personal information.

We impose three server-side requirements. First, the recovery servers must run on commodity machines without making any specific assumptions about the hardware. These constraints are common in many web applications on the cloud, and in particular we rejected the use of trusted hardware due

to concerns about cost, challenges with upgrades and maintenance, and security vulnerabilities (e.g., [17,32,43,64,65,82]). Second, to simplify software development and maintenance, the recovery servers must only use common crypto primitives with widely-used software libraries. In particular, we avoided the use of pairing-based cryptography since these libraries are less widely available in deployment settings and less understood by the general software engineering community. Third, for ease of deployment, the recovery servers must be able to communicate independently with the client (e.g., to execute separate instances of the OPRF protocol) rather than with each other; that said, they can maintain joint state like a counter acting as a nonce.

On the flip side, a non-collusion assumption in the anytrust setting was deemed acceptable if it provides defense-in-depth such that no collection of all-but-one recovery servers (or the engineers who administer them) can recover email addresses. Moreover, we deemed it to be acceptable if the recovery servers learn the client's email address at the moment of account creation for two reasons: (a) the website's onboarding process already involves sending an email at the moment of account creation, and (b) the main threat was deemed to be bulk, retrospective extraction of already-registered accounts rather than the smaller set of accounts created or recovered during the interval of server compromise.

We remark that it is possible to protect addresses even during the act of email delivery by using MPC for TLS [3,78], but even that would not prevent against a different threat: one of compulsion. Due to our focus on trauma-centered design and ensuring that the secure matching system itself could not be used to harm survivors, it was important to consider the possibility that all servers become under control of a single adversary through technical or legal means, and even then to avoid a large-scale breach of users' identity and data.

Threats and security guarantees. We require that the account recovery system provide correctness (up to abort) and privacy against three types of malicious adversaries. We emphasize upfront that this work focuses on threats and mitigations at the cryptographic level. As stated in §1.3, this work does not impose any new barriers to an adversary's ability to exploit a web service; instead, we seek to mitigate the damage of an adversary who controls some or all recovery servers.

1. An *external adversary* who interacts with the recovery system protocols (but lacks control of any recovery server) should be unable even to determine whether any email address they do not control corresponds to an account in the system. Looking ahead, our account recovery protocol achieves this goal by not providing any information to the client to determine whether an email address matches an account in the system—at least, not until the moment that an honest client receives an email in her inbox.
2. An *internal adversary* who has compromised some of the recovery servers must be restricted from performing a brute-force attack of email addresses. The threat here is that an internal adversary might conduct a probing attack in which it pretends to be a client, submits a variety of different email addresses, and uses its insider access to determine whether an account exists by observing whether an email is sent out to the victim. We address this threat in two ways: first to require some additional information beyond the email address even at the first step of recovery request, and second to have the honest server(s) perform online rate limiting of the attacker. In particular, we intentionally avoid providing any deterministic function of the client's data to an individual server, because that could be used to perform an offline dictionary attack.
3. A *legal adversary* that subpoenas all recovery servers must still need to execute a (now inevitable) offline brute-force attack to learn any account information, and furthermore they must be required to perform a brute-force attack after the last server has been compromised. Looking ahead, our recovery request and account restoration protocols provide this guarantee through their use of a slow, password-based hash function to make offline dictionary attacks slower and more costly. Moreover, the slow hash function requires the outputs of the $\mathsf{K\text{-}pop}$, which prevents so-called "pre-computation attacks" [57] and ensures that the expensive offline attack must occur after the servers are corrupted.

Finally, all of these security properties must hold even against honest clients who have performed account recovery or changed their security questions, potentially several times. Looking ahead, our account recovery protocol uses client and server nonces to ensure domain separation of the space that must be brute-forced before vs. after an account reset.

2.2 Our model of account recovery

Our account recovery protocol has 4 procedures: account creation and the 3 parts of account recovery (plus a simple initialization step for the servers to generate their cryptographic keys). We presume the client (who we call Alice) already has a web account with a key k_u (e.g., derived from her password) that is used for encryption of her account data at rest.

Account creation. The client Alice begins this procedure with her user account key k_u, and the goal is for her to prepare and upload some cryptographic material that facilitates later recovery of k_u. Specifically, she (obliviously) provides her account's email address E, responses to additional generic questions like her phone number x, a set of personalized security questions Q together with their corresponding answers A and a recovery email address e where she would like to be

contacted if recovery is needed (this can, but does not have to, be the same as the account email address E). Anyone who later proves ownership of the recovery email account e and knows the answers x and A to the generic and personalized security questions will be able to recover the account key k_u.

Recovery request. After Alice forgets her password, she (obliviously) submits her email address E and her responses x to any additional questions asked of everyone. Importantly, Alice herself receives no output from this protocol; she does not yet know whether E and x matched any previously-created accounts. If a match exists, then the servers receive (secret shares of) Alice's recovery email address e and security questions Q. This stage has two purposes: involving the servers for online rate-limiting of any dictionary attack, and providing the servers with the information they need to perform verification and ask personalized security questions to the client.

Account verification. The servers send an email to Alice's recovery address e containing a one-time link. In this work, we model the email delivery as an instance of secure message transmission $\mathcal{F}_{\mathsf{SMT}}$. This functionality can be instantiated by the servers in the clear (in which case they temporarily learn Alice's email address and must then delete it afterward) or using secure multi-party computation (so that they can jointly send the email while individually not learning e) [3,49,78].

Account restoration. This stage begins when the client clicks on the link from the email, which (among other things) contains her security questions Q. The client obliviously provides her corresponding security answers A, which—if correct—can be used to recover her key k_u. The servers receive no output from restoration. Upon restoration, Alice should immediately re-run account creation to choose a new password for her account and select (possibly new, or possibly the same) security questions and answers to protect her keys.

2.3 Related work

In this section, we describe some prior work that informs our choice of a protocol design.

Secure matching systems. Secure matching systems[1] are cryptographic protocols that aim to augment informal "whisper networks" that exist in many communities such as college campuses. They permit a collection of survivors of sexual assault to determine whether they have been harmed by the same individual—even though the survivors have never met each other, never communicate directly with each other, and may never be online at the same time as each other. To accomplish this goal, secure matching systems use special-purpose secure computation among a non-colluding set of servers.

The work of Rajan et al. [75] was the first to construct a cryptographically secure matching system; their work relies on non-collusion of two servers, and it provides confidential matching when two survivors provide identifying information about the same individual. Subsequent works achieved stronger functionality and integrity guarantees, at the expense of requiring a public key infrastructure and an honest majority among 3 or more servers. WhoToo [62] allows for each survivor to choose their own matching threshold (potentially greater than two), and adds traceability for false accusations. Arun et al. [8] contribute a constant-time matching protocol to determine, after each submission, whether any collection of reports has exceeded the threshold. WhoToo+ [50] fixes some ambiguities in WhoToo and extends it also to support matching with a constant number of online operations. Finally, Shield [60] improves upon the work of Arun et al. in two ways: hiding the threshold chosen by each survivor, adding integrity checks against fake and duplicative submissions.

MPC and TEEs. Secure multi-party computation (MPC) is a cryptographic technique for a collection of servers to perform a joint computation while not learning any of the underlying data. While the fundamental concepts of MPC have been known for four decades [13,15,33,46,84], MPC on its own is typically incapable of withstanding the threat of legal adversaries, as we describe in more detail in §2.5.

Trusted execution environments (TEEs) offer an alternative method to protect sensitive data using hardware isolation (rather than cryptography). They have been proposed in commercial processors (e.g., [6,7,35,58,80]), academic prototypes (e.g., [36,52,76,83]), and cloud-based services [5,47,70]. However, nearly all of these systems are subject to both physical and remote attacks (e.g., [17,32,43,64,65,82]), and it remains an open question as to what extent the vision of trusted hardware can be realized. For these reasons and due to our design requirement to operate on general-purpose hardware, we avoid use of TEEs in this work.

Oblivious PRFs. A pseudorandom function (PRF) is a deterministic but "random-looking" function $y = f_k(x)$ in which the mapping between the input x and output y is unpredictable without knowledge of the secret key k. An oblivious PRF is an interactive protocol to evaluate $f_k(x)$ that hides the server

[1]While some other works in this area refer to these systems as "secure allegation escrows," we have selected an alternative name based on Callisto's terminology for their service [20], which they describe as a "matching system." The use of the term "allegation" may *unintentionally* imply a lack of belief in survivors' experiences. As advocates have noted in a guide on language use for sexual assault [59], "Many people say they use the word 'alleged' to refer to sexual assault cases, because they have not reached a final resolution within the criminal justice system ... However, it is important to keep in mind that only a miniscule percentage of sexual assaults ever make their way through the entire criminal justice process. ... [A]lmost all sexual assaults remain "unresolved" by the legal system, and it would be inappropriate to refer to all such reports (or even disclosures) of sexual assault as 'alleged.' "

key k and the client input x from each other; it also allows the server to perform service-wide rate-limiting by refusing to participate after some number of queries within a time period. A *partially* oblivious pseudorandom function (pOPRF) has two inputs $f_k(x_{\mathsf{kal}}, x_{\mathsf{priv}})$, where the server provides k and x_{kal}, and the client provides x_{priv}. This additional 'nonce' or 'salt' input x_{kal} can be used to perform per-account rate-limiting while still hiding the client's input x_{priv}.

There are a wide variety of OPRF works in the literature, starting with the work of Naor and Reingold [71]. Modern constructions tend to be based on a Hashed-Diffie-Hellman PRF $f_k(x) = H(x)^k$ that is secure in the random oracle model, or the Dodis-Yampolskiy PRF $f_k(x) = g^{1/(k+x)}$ [40]; we will use some ideas from both of these constructions in this work. Both styles of OPRFs also have verifiable counterparts that add integrity checks [42,54], often via zero-knowledge proofs (which we purposely avoid in this work as discussed in §2.1). We refer readers to the SoK by Casacuberta et al. [31] for additional OPRF and pOPRF constructions.

Among their many uses, OPRFs and pOPRFs have found value in several applications involving oblivious interactions on the web with passwords or other low-entropy secrets, such as (threshold) password-protected secret sharing [11,54,56], password hardening services [42, 63], compromised password checkers [73,79], password-authenticated key exchange [54, 57, 81], and single sign-on [12]. Many of these works also provide universally composable (UC) security models of OPRFs, optionally with a threshold or verifiability requirement. Looking ahead, we use the UC modeling of Jarecki et al. [56] in this work, which is intentionally designed to avoid the verifiability requirement. We also desire security against pre-computation attacks, as defined within OPAQUE [57].

2.4 Overview of our K-pop protocol

In this section, we provide an informal overview of our construction of a *kaleidoscoping partially oblivious PRF*, or K-pop. As a reminder, this is a function that can be calculated as an OPRF or pOPRF, and reaches the same result either way. In other words, the input x_{kal} is *kaleidoscopic* in that it can be rapidly changed between being client- or server-provided.

We observe that if bilinear maps are permissible, then it is straightforward to build a K-pop. Concretely, the Pythia pOPRF of Everspaugh et al. [42] is computed as $f_k(x_{\mathsf{kal}}, x_{\mathsf{priv}}) = e(H_1(x_{\mathsf{kal}}), H_2(x_{\mathsf{priv}}))^k$, where H_1 and H_2 are random oracles. Using standard techniques for oblivious exponentiation (shown in §3), this function f can be easily computed either as an ordinary OPRF (where the client computes the bilinear map and the server obliviously exponentiates by k) or as a pOPRF (where the client blinds the left input before the server computes the bilinear map).

However, introducing a bilinear map violates our design principle only to use widely available crypto libraries. As a result, we have designed a K-pop that can be built purely from group and finite field operations, in the random oracle model. Constructing this is non-trivial, as there are no OPRFs and pOPRFs in the literature [31] that already compute the same outputs as each other.

Our starting point for this work is the recent pOPRF of Tyagi et al. [81]. It uses the pseudorandom function family $f_k(x_{\mathsf{priv}}, x_{\mathsf{kal}}) = H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, H_1(x_{\mathsf{priv}})^{1/(k+H_3(x_{\mathsf{kal}}))})$, which combines characteristics of the Hashed-DH and Dodis-Yampolskiy OPRFs described in §2.3. We extend this construction to a K-pop by providing an OPRF protocol as well, which uses additively homomorphic encryption between the server (who holds k) and the client (who holds x_{kal}) to compute the exponent $\frac{1}{k+H_3(x_{\mathsf{kal}})}$ that is used in oblivious exponentiation. While the OPRF mode requires a larger number of public-key operations, this mode of our K-pop only needs to be executed in the (less frequent) account recovery phase.

2.5 Overview of our account recovery protocol

This section provides a high-level description of our account recovery protocol. To provide some intuition about the challenges involved in this construction, we iteratively build it up by describing several ideas that look promising but ultimately fall short of meeting our objectives. We then propose changes until arriving at our final construction.

MPC-based approaches. A natural approach here is to use MPC, either generally (e.g., run account recovery as a large circuit) or using private information retrieval, encrypted database search, or other special-purpose primitives. Within our 3-step recovery request, verification, and restoration procedure, it might seem at first glance that the hardest approach to satisfy with MPC is the email verification step. But actually this part is mostly a solved problem: Abram et al. [3] showed how a coalition of MPC servers can collectively emulate TLS 1.3 communications, and MPCAuth [78] applied this technique to email delivery in such a way that the servers never learn the client's email address or contents of the email.

Instead, the primary challenge is to withstand the threats of an internal and legal adversary during the recovery request and restoration steps, since MPC-based approaches typically do not (a) provide any form of rate-limiting of queries or (b) stop a colluding coalition of all servers from instantly reconstructing all data. Here the recent TLS-OPAQUE construction of Hesse et al. [49] is more promising: it uses secure computation of TLS for email delivery along with an OPRF that could be adapted to ensure that the client's data (e.g., email address and security questions/answers) is protected by online rate limiting if even a single server is honest and requires an offline dictionary attack even after all servers are corrupted. In this work, we purposely choose not to use any form of MPC-for-TLS since (a) it is slow and non-standard to implement, (b) it is not necessary to satisfy our design principles (§2.1), and (c) even with MPC-for-TLS, it remains unclear

how to protect the client's data using an OPRF. We focus on this latter question next.

OPRF-based approaches. When using an OPRF in account recovery, an initial thought might be for recovery request to involve a standard OPRF on the email address E: that is, to have the client and servers calculate $y = f_k(E)$ and to protect the client's user account key using y. However, this approach would allow an external adversary to conduct a brute-force search of others' email addresses. A better approach is to reveal y only after verifying that the client controls the email address. Even so, an internal adversary (who maliciously also acts as a client) could perform a brute-force search of email addresses and recover user data; ultimately, email addresses do not have enough entropy on their own.

Hence, we wish to add additional questions, but here we face a chicken-and-egg issue. Ideally, we want each client to be able to choose their own custom security questions, and then construct some method where the OPRF output y is used by the servers to retrieve each client's custom questions. But since a retrieving client has forgotten all passwords and crypto keys, any brute-force attacker would also be able to read the same security questions. Hence, an internal adversary could use the mere fact that security questions have been successfully retrieved to conclude that an email address E must be registered in the system.[2] We resolve this issue by allowing for generic questions for additional information x (e.g., a phone number) and computing the OPRF as $y = f_k(E, x)$ where y is used as a key to unlock a second layer of custom security questions. In this way: the generic questions provide increased resistance against brute-force attack by an internal adversary, and the custom questions provide stronger (and industry standard) protection against external adversaries.

The next challenge is that if the same OPRF $y = f_k(E, x)$ is used during account creation and recovery, then both need to be rate-limited or else an internal adversary could use account creation requests to brute-force y. We resolve this issue by using a K-pop togther with a server-chosen nonce n that is guaranteed to be unique for all account creation requests. Then, we can run a pOPRF $y = f_k(n, (E, x))$ during account creation and the corresponding OPRF during recovery. With a K-pop, we can enforce rate-limiting of account recovery requests in such a way that we do not know which account is being recovered and need not rate-limit account creation.

Finally, any OPRF-based approach (on its own) does nothing against a legal adversary in control of all servers, since a PRF provides no security against the key-holder. We resolve this issue by feeding y into a slow cryptographic hash

[2] We remark that this leakage is not inherent from a cryptographic perspective; indeed, it is possible to calculate the security questions pseudorandomly from the email address E so that it can be retrieved whether or not E was ever registered in the system. But we rejected this approach because it went against our goal of trauma-informed care: if a survivor accidentally mistyped their email address, we did not want to subject them to try in vain to respond to an incorrect set of security questions.

Symbol	Description
$\mathbf{C}$	Client
$\mathbf{S}_i$	Recovery server i
N	Number of recovery servers (default is $N = 2$)
λ	Cryptographic security parameter, e.g., 256 bits
k_u	Client's user account key (which decrypts account data)
H	Cryptographic hash function, modeled as a random oracle
E	Client's email address associated with the account
x	Client's answers to any generic questions (e.g., contact info)
e	Client's recovery email address (can be the same as E)
Q	Security questions chosen by the client
A	Answers to the client-chosen security questions
n	Nonce chosen publicly by the recovery servers
m	Nonce chosen privately by the client
r	Recovery string, set as $r = e \parallel Q \parallel m \parallel padding$
ℓ	fixed length of the recovery string after padding is applied
D	Database held by both recovery servers (this stores all data provided by all clients during account creation)
$\hat{E}_i$	Result of the K-pop run in recovery request with server i
$\hat{A}_i$	Result of the K-pop run in recovery restoration with server i
id	Client's identifier for her record in the database D
$\mathsf{ct}_r, \mathsf{ct}_u$	Ciphertexts produced by the client to store in the server database; they are a one-time pad of r and k_u, respectively
k_E, k_A	One-time pad keys for the ciphertexts above
k	Server's secret key for an OPRF
x_{priv}	Client's private input to an OPRF or pOPRF
x_{kal}	Input provided by the server in a pOPRF, or by the client in an OPRF

Table 1: Notation used in this paper.

function H that can slow down the rate of offline brute-force attempts. Importantly, the OPRFs of all servers are performed first and only the results are fed into H, so that this offline brute-force attack must occur *after* the moment that all server keys are compromised. In this way, our construction is secure against pre-computation attacks [57]. This finally yields a secure construction, which we describe in detail in §4.

3 Kaleidoscopic Partially Oblivious PRF

In this section, we define and construct a new primitive that we call a *kaleidoscopic partially oblivious PRF*, or K-pop for short. This primitive is named for the fact that it can be quickly changed between an OPRF and a pOPRF at will.

3.1 Preliminaries

In this section, we briefly introduce the notation and OPRFs that we use in this work. We describe other cryptographic primitives (hash functions, Diffie-Hellman groups, and additively homomorphic encryption) in Appendix A.

Notation. In this work, we often use upper-case letters like S to denote finite sets, and calligraphic letters $\mathcal{D}$ to denote distributions. The notation $x \leftarrow \mathcal{D}$ means to sample x from the distribution $\mathcal{D}$, and the notation $x \leftarrow S$ means to select

Functionality $\mathcal{F}_{\mathsf{SMT}}$

- Upon invocation, with input $(\mathsf{send}, \mathsf{sessionid}, R, m)$ from a sender party S that is intended for a receiver party R, send a message $(\mathsf{sent}, \mathsf{sessionid}, S, R, |m|)$ to the adversary $\mathcal{A}^*$.
- Upon receiving message $(\mathsf{delivered}, \mathsf{sessionid})$ from $\mathcal{A}^*$: If not yet generated output, then output $(\mathsf{sent}, \mathsf{sessionid}, S, R, m)$ to R.
- Upon receiving message $(\mathsf{corrupt}, \mathsf{sessionid}, m', R')$ from the environment **Env**: record being corrupted. Additionally, if no output has been generated yet, then output $(\mathsf{sent}, \mathsf{sessionid}, S, R', m')$ to R'.

Figure 1: Secure message transmission functionality $\mathcal{F}_{\mathsf{SMT}}$, adapted from [22]. In this work, we model both network communication via TLS and the act of sending an email from the servers to the client's email provider as instantiations of $\mathcal{F}_{\mathsf{SMT}}$.

Functionality $\mathcal{F}_{\mathsf{OPRF}}$

An instance of this functionality is uniquely defined by a session id $\mathsf{sessionid} = (\mathsf{OPRF}, \mathsf{sid})$ containing the party ID of the server (sid). The functionality interacts with both the server $\mathbf{S}_{\mathsf{sid}}$ and anyone who acts in the role of a client **C**, including possibly the adversary $\mathcal{A}^*$.

- Upon receiving $(\mathsf{Init}, \mathsf{sessionid})$ from the server $\mathbf{S}_{\mathsf{sid}}$, uniformly sample $\mathsf{kid} \leftarrow\$ \{0,1\}^\lambda$, and initialize an empty table $T(\mathsf{kid}, x)$.
- Upon receiving $(\mathsf{Eval}, \mathsf{sessionid}, \mathsf{qid}, x)$ from a client **C** (which can be adversary $\mathcal{A}^*$) where qid is a unique identifier for this Eval query: end this invocation if called before Init. Otherwise, record $\langle \mathsf{qid}, \mathbf{C}, x \rangle$ and send $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid})$ to $\mathcal{A}^*$.
- Upon receiving $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \mathsf{kid}^*)$ from $\mathcal{A}^*$: end this invocation if there is no record with qid. Otherwise, find and delete record $\langle \mathsf{qid}, \mathbf{C}, x \rangle$. If $\mathbf{S}_{\mathsf{sid}}$ is honest, set $\mathsf{kid}^* \leftarrow \mathsf{kid}$ (i.e., use the correct key). Next, fetch $\rho \leftarrow T(\mathsf{kid}^*, x)$ as follows:
  - If $T(\mathsf{kid}^*, x)$ is defined, then look up $\rho \leftarrow T(\mathsf{kid}^*, x)$ and send $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, \rho)$ to **C**.
  - Otherwise pick ρ at random from $\{0,1\}^\ell$, assign $T(\mathsf{kid}^*, x) := \rho$, and send $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, \rho)$ to **C**.
- Upon receiving $(\mathsf{OfflineQuery}, \mathsf{sessionid}, x, \mathsf{kid}^*)$ from **C**: fetch $\rho \leftarrow T(\mathsf{kid}^*, x)$ as above. Send $(\mathsf{OfflineQuery}, \mathsf{sessionid}, \rho)$ to **C**.
- Upon receiving $(\mathsf{corrupt}, \mathsf{pid})$ from the environment **Env**: mark the party corresponding to pid as corrupted. Additionally:
  - For corruption of the server, send kid to the adversary.
  - For corruption of a client **C**, send all corresponding records $\langle \mathsf{qid}, \mathbf{C}, x \rangle$ to the adversary.

Figure 2: Functionality $\mathcal{F}_{\mathsf{OPRF}}$, based on Jarecki et al. [55] modified to use a unique query identifier qid like Das et al. [37] to track commands sent to $\mathcal{F}_{\mathsf{OPRF}}$ about the same evaluation query. $\mathcal{F}_{\mathsf{OPRF}}$ does not guarantee honest behavior by the server; the key identifier kid corresponds to the server's choice of key to use when responding to each query. OfflineQuery allows anyone to evaluate the OPRF on input x and key id kid^* of their choice—which is unlikely to match the honest kid unless **S** is corrupted.

x uniformly at random from the set S. We also use $[n] = \{1, \ldots, n\}$ to denote the set of integers from 1 to n, inclusive.

We denote the protocol participants using bold letters. A client is denoted as **C**, and a server is denoted as **S**. For protocols that contain multiple servers, we use subscripts to distinguish them: server 1 is denoted as $\mathbf{S}_1$, server 2 is $\mathbf{S}_2$, and so on. We also use **Adv** to denote a real-world adversary who attempts to attack our protocol—either maliciously or semi-honestly, as stated in the respective theorem statements—and **Sim** to denote the corresponding ideal-world simulator. See Table 1 for a detailed list of all variables used in this work.

Oblivious pseudorandom function. As described in §2.3, an OPRF is an interactive two-party protocol in which a client **C** has input x, a server **S** has input k, and they jointly compute $y = f_k(x)$ in such a way that neither party learns anything about the other input. However, the server can deviate from the protocol, and correctness is not guaranteed in this case. Formally, we model an OPRF as an instantiation of the UC functionality $\mathcal{F}_{\mathsf{OPRF}}$ shown in Figure 2, which we adapt from Jarecki et al. [55] with a small tweak to add support for server-side rate-limiting. If a client submits an input x and the server is honest, then the client receives the correct output—potentially after some adversarially-chosen delay since this is an interactive protocol. If the server is malicious, then it could act as though it used a different key or produce a random output, which the functionality also accounts for.

3.2 Defining K-pop

Like an OPRF or pOPRF, a K-pop is an interactive protocol between two participants who we call the *client* **C** and the *server* **S**. Concretely, a K-pop is a keyed family of pseudorandom functions with two inputs $f_k^{\mathsf{K\text{-}pop}}(x_{\mathsf{kal}}, x_{\mathsf{priv}})$. Looking ahead to our account recovery protocol: the client will execute independent instances of K-pop with each recovery server.

The K-pop can be computed either as a pOPRF or OPRF. Concretely, the input x_{kal} is *kaleidoscopic* in that it can be rapidly changed between being public to the server (in pOPRF mode) or private to the client (in OPRF mode). Importantly, the K-pop returns the same answer in either mode.

We formalize these requirements through the UC functionality $\mathcal{F}_{\mathsf{K\text{-}pop}}$ provided in Figure 3, which codifies both the functionality and security requirements of K-pop. It calls the OPRF functionality $\mathcal{F}_{\mathsf{OPRF}}$ (Fig. 2) as a subroutine. We stress that $\mathcal{F}_{\mathsf{K\text{-}pop}}$ makes the *same* call to $\mathcal{F}_{\mathsf{OPRF}}$ whether the input

Functionality $\mathcal{F}_{\text{K-pop}}$

An instance of this functionality is identified by a session id $\text{sessionid} = (\text{K-pop}, \text{sid})$ containing the party ID sid of its server. It interacts with the server **S** and any client **C** (possibly the adversary $\mathcal{A}^*$), it has a subroutine $\mathcal{F}_{\text{OPRF}}$ and a query limit L, and it operates as follows.

- Upon receiving $(\text{Init}, \text{kid})$ from server $\mathbf{S}_{\text{sid}}$: set $\text{ctr} \leftarrow 0$ and send $(\text{Init}, \text{kid})$ to $\mathcal{F}_{\text{OPRF}}$. (Other methods abort if called before Init.)
- Upon receiving (Reset) from the server $\mathbf{S}_{\text{sid}}$: set $\text{ctr} \leftarrow 0$ and send the message (ResetComplete) to $\mathcal{A}^*$.
- Upon receiving $(\text{Eval}, \text{sessionid}, \texttt{OPRF-mode}, \text{qid}, x_{\text{kal}}, x_{\text{priv}})$ or $\boxed{(\text{Eval}, \text{sessionid}, \texttt{pOPRF-mode}, \text{qid}, x_{\text{kal}}, x_{\text{priv}})}$ from a client **C**:
  - Send an error $(\text{Eval}, \text{sessionid}, \text{qid}, \perp)$ to $\mathcal{A}^*$ if there was a prior message with qid, or if **S** is honest and $\text{ctr} > L$.
  - $\boxed{\text{If in pOPRF mode: send } (\text{Eval}, \text{sessionid}, \text{qid}, x_{\text{kal}}) \text{ to } \mathcal{A}^*\text{, and wait for a response } (\text{EvalContinue}, \text{sessionid}, \text{qid}).}$
  - Increment query counter $\text{ctr} \leftarrow \text{ctr} + 1$, and send $(\text{Eval}, \text{sessionid}, \text{qid}, (x_{\text{kal}}, x_{\text{priv}}))$ to $\mathcal{F}_{\text{OPRF}}$ on behalf of **C**.
- Upon receiving $(\text{EvalComplete}, \text{sessionid}, \text{qid}, \rho)$ from $\mathcal{F}_{\text{OPRF}}$: output $(\text{EvalComplete}, \text{sessionid}, \text{qid}, (x_{\text{kal}}, x_{\text{priv}}), \rho)$ to client **C**.
- Upon receiving $(\text{OfflineQuery}, \text{sessionid}, x_{\text{kal}}, x_{\text{priv}}, \text{kid}^*)$ from **C**: send this message to $\mathcal{F}_{\text{OPRF}}$, and send its response to client **C**.
- Upon receiving $(\text{corrupt}, \text{pid})$ from the environment **Env**: mark the party with pid as corrupted, and send $(\text{corrupt}, \text{pid})$ to $\mathcal{F}_{\text{OPRF}}$.

Figure 3: Functionality $\mathcal{F}_{\text{K-pop}}$ with subroutine $\mathcal{F}_{\text{OPRF}}$. Differences between OPRF and pOPRF modes are shown in a $\boxed{\text{box}}$. $\mathcal{F}_{\text{K-pop}}$ maintains a counter ctr of the number of Eval queries, and it enforces a limit of L evaluations between Reset commands.

K-pop in pOPRF mode

Client (knows $x_{\text{kal}}, x_{\text{priv}}$)		**Server** (knows k, x_{kal})
$r \leftarrow\$ \{1,\dots,p\}$		
$\alpha = H_1(x_{\text{priv}})^r$	pOPRF $\xrightarrow{x_{\text{kal}},\ \alpha}$	stop if query limit exceeded
		$v = k + H_3(x_{\text{kal}})$
$\gamma = \beta^{1/r}$	$\xleftarrow{\beta}$	$\beta = \alpha^{1/v}$
Output $H_2(x_{\text{kal}}, x_{\text{priv}}, \gamma)$		

Figure 4: K-pop in pOPRF mode, where both the client and server know x_{kal}. The server need not be honest; it can use incorrect k^* or x^*_{kal} in its oblivious exponentiation. All communications use secure message transmission $\mathcal{F}_{\text{SMT}}$ (Fig. 1).

K-pop, in OPRF mode

Client (knows $x_{\text{kal}}, x_{\text{priv}}$)		**Server** (knows k)
		$(\text{sk}, \text{pk}) \leftarrow\$ \text{HomKeyGen}()$
	$\xleftarrow{\text{pk},\ [\![k]\!]}$	$[\![k]\!] = \text{HomEnc}_{\text{pk}}(k)$
$r, s \leftarrow\$ \{1,\dots,p\}$		
$t \leftarrow\$ \{1,\dots,N/p\}$		
$\alpha = H_1(x_{\text{priv}})^r$		
$[\![z]\!] = \text{HomEval}_{\text{pk}}(tp + s([\![k]\!] + H_3(x_{\text{kal}})))$	OPRF $\xrightarrow{\alpha,\ [\![z]\!]}$	stop if query limit exceeded
		$z = \text{HomDec}_{\text{sk}}([\![z]\!])$
		$z = z \bmod p$
$\gamma = \beta^{s/r}$	$\xleftarrow{\beta}$	$\beta = \alpha^{1/z}$
Output $H_2(x_{\text{priv}}, x_{\text{kal}}, \gamma)$		

Figure 5: K-pop in OPRF mode, where x_{kal} and x_{priv} are both private inputs supplied by the client. All communications use secure message transmission $\mathcal{F}_{\text{SMT}}$ (Fig. 1). The first message can be sent in a preprocessing step or using a PKI.

x_{kal} is provided by the server (in pOPRF mode) or by the client (in OPRF mode). Since $\mathcal{F}_{\text{OPRF}}$ does not know whether it was called in OPRF or pOPRF mode, its response must be the same in both cases; in other words, $\mathcal{F}_{\text{K-pop}}$ ensures identical outputs in the OPRF and pOPRF modes by design. Also, note that if the parties are honest, then $\mathcal{F}_{\text{K-pop}}$ is guaranteed to be deterministic and repeatable because $\mathcal{F}_{\text{OPRF}}$ is.

However, if the server **S** is malicious, then in pOPRF mode it *can* ignore the agreed-upon x_{kal} and choose a different x^*_{kal}. While one could incorporate a verifiability guarantee into a K-pop, we choose not to do so because it is not needed in our account recovery protocol in §4 and this way we do not need to use zero knowledge proofs. Instead, we leave it up to any protocol that uses K-pop to detect and abort in case of malicious server behavior.

3.3 Constructing K-pop

In this section, we construct a K-pop by combining and extending techniques from Tyagi et al. [81] and Miao et al. [69]. Specifically, our K-pop construction computes the same function as the pOPRF of Tyagi et al. [81]:

$$f_k^{\text{K-pop}}(x_{\text{kal}}, x_{\text{priv}}) = H_2\left(x_{\text{kal}}, x_{\text{priv}}, H_1(x_{\text{priv}})^{1/(k+H_3(x_{\text{kal}}))}\right).$$

Here, H_1 is a cryptographic hash function whose output is a point on an elliptic curve group, and H_2 and H_3 are random oracles that return finite field elements. We use x_{priv} and x_{kal} to

denote the inputs, where x_{priv} is always a secret input supplied by the client, and x_{kal} is the kaleidoscopic input. In the OPRF mode, x_{kal} is chosen by the client and kept secret from the server; by contrast, in the pOPRF mode, x_{kal} is known and agreed upon by the client and server.

This construction by Tyagi et al. [81] combines ideas from two prior PRF families: the Dodis-Yampolskiy [40] PRF $f_k(x) = g^{1/(k+x)}$ and the Hashed-DH PRF $f_k(x) = H(x)^k$. The outer-most hash function H_2 is useful to provide extraction for UC security, in the random oracle model [54]. Below, we reproduce the pOPRF design of Tyagi et al. [81], and then we construct an OPRF that computes the same result. In both cases, we model all network communications with a secure message transmission functionality $\mathcal{F}_{\mathsf{SMT}}$ in Fig. 1; that said, it is sufficient to use an authenticated communication channel as defined in several prior works (e.g., [10, 22, 23, 29]).

pOPRF Mode. We begin by describing the K-pop in pOPRF mode. Because the server knows x_{kal}, it can compute $(k + H_3(x_{\mathsf{kal}}))^{-1}$ and use this as the exponent in a two-message oblivious exponentiation protocol with the client, as illustrated in Figure 4. Finally, the client computes the outer hash function H_2. This mode exactly follows the work of Tyagi et al. [81] (but without a zero knowledge proof), and it is the more efficient of the two modes: it requires 3 group exponentiations along with some hash function evaluations.

OPRF Mode. It remains to compute the same function in OPRF mode, in which x_{kal} (like x_{priv}) is a secret known only to the client that the server is not supposed to learn. Rather than having the server compute $v = k + H_3(x_{\mathsf{kal}})$ directly, in this mode we have the client and server compute v together, using additively homomorphic encryption so that neither party learns any intermediate state along the way. Let p denote the order of the elliptic curve group, and let the notation $[\![x]\!]$ denote a homomorphic encryption of the plaintext value x.

First, we describe the main technique under the simplifying assumption that the plaintext space of the homomorphic encryption algorithm is also a group of order p. With this simplifying assumption, we can perform a secure computation of the required addition and multiplicative inversion as follows.

1. The server chooses an ephemeral key pair for homomorphic encryption, computes $[\![k]\!] = \text{HomEnc}(k)$, and sends this ciphertext to the client.
2. The client chooses two blinding factors $r, s \leftarrow \{1, \dots, p\}$. The client computes the group element $\alpha = H_1(x_{\mathsf{priv}})^r$ in the same way as in the pOPRF mode, and also uses homomorphic addition and scalar multiplication to compute the ciphertext $[\![z]\!]$ corresponding to the plaintext $z = s(k + H_3(x_{\mathsf{kal}}))$. The client sends α and $[\![z]\!]$ to the server.
3. The server decrypts z, inverts it, and sends $\beta = \alpha^{1/z} = H_1(x_{\mathsf{priv}})^{\frac{r}{s(k+H_3(x_{\mathsf{kal}}))}}$ to the client.
4. The client computes $\gamma = \beta^{s/r} = H_1(x_{\mathsf{priv}})^{1/(k+H_3(x_{\mathsf{kal}}))}$.

Correctness is clear by inspection, and privacy against the server follows from the fact that z is blinded using s so that the server cannot learn the client's x_{kal}.

The remaining challenge is to address the fact that the exponents and HomEnc plaintexts are *not* from the same group, which we handle using a technique from Miao et al. [69]. Concretely, let $N \gg p$ denote the order of the plaintext space within an additively homomorphic encryption scheme, like Paillier [72] or Camenisch-Shoup [21] encryption (for the latter, we omit the integrity check). The issue here is that we wanted the homomorphic evaluation to produce $z = s(k + H_3(x_{\mathsf{kal}})) \bmod p$ because the server plans to use z as an exponent, but the homomorphic evaluation performs the calculation mod N instead. If $N \gg p^2$ then the calculation $z = s(k + H_3(x_{\mathsf{kal}})) \bmod N$ does not perform any modular wrapping at all, so the blinding is ineffective and revealing z to the server would allow it to learn information about the size of the client's secrets s and $H_3(x_{\mathsf{kal}})$. Fortunately, there is a simple solution to this problem shown by Miao et al. [69]: add a random multiple of p, so that the resulting plaintext is $z = s(k + H_3(x_{\mathsf{kal}})) + tp \bmod N$, where $t \leftarrow \{1, \dots, N/p\}$ is another blinding factor chosen by the client. In this way, z still reduces to the correct exponent mod p with overwhelming probability, and its size does not reveal information about the client's secret inputs.

This protocol requires 3 group exponentiations (just like pOPRF mode) along with one homomorphic encryption, evaluation, and decryption. We illustrate the protocol in Figure 5 and prove the following theorem in Appendix E.

Theorem 1. *Assume that group G satisfies the one-more gap strong Diffie-Hellman inversion assumption and that* HomEnc *satisfies indistinguishability under a chosen plaintext attack (IND-CPA). Then, the* K-pop *protocol $\Pi_{\mathit{K\text{-}pop}}$ (in Figures 4-5) UC-realizes the ideal functionality $\mathcal{F}_{\mathit{K\text{-}pop}}$ in the presence of a programmable random oracle $\mathcal{F}_{\mathsf{pro}}$.*

4 Account Recovery

In this section, we describe our cryptographic protocol for private account recovery. Our protocol consists of the four stages described in §2.2: *account creation*, *recovery request*, *account verification*, and *account restoration*. Below we describe constructions for each protocol, with full details provided in Figures 6-8. For simplicity, our constructions are written here for the setting of $N = 2$ servers, but the protocols immediately generalize to allow for more servers. We provide a security analysis of this protocol in §4.6 and the appendices.

4.1 Initialization

Each server independently and secretly chooses a K-pop key. We denote server i's K-pop key as k_i. Additionally, a com-

Client **Server 1** **Server 2**

$m \xleftarrow{\$} \{0,1\}^{256}$

Agree on unique nonce n (Server 1 ↔ Server 2)

n (Server 1 → Client)

E (Client → Server 1)

E (Client → Server 2)

E, x → K-pop pOPRF mode ← E, k_1 (Server 1); output $\hat{E}_1$ to Client

E, x → K-pop pOPRF mode ← E, k_2 (Server 2); output $\hat{E}_2$ to Client

$n, (A \parallel m)$ → K-pop pOPRF mode ← n, k_1 (Server 1); output $\hat{A}_1$ to Client

$n, (A \parallel m)$ → K-pop pOPRF mode ← n, k_2 (Server 2); output $\hat{A}_2$ to Client

$\mathsf{id}, k_E = H(E \parallel \hat{E}_1 \parallel \hat{E}_2)$

$k_A = H(A \parallel m \parallel \hat{A}_1 \parallel \hat{A}_2)$

$r = E \parallel Q \parallel m \parallel \mathsf{padding}$

$\mathsf{ct}_r = k_E \oplus r$

$\mathsf{ct}_u = k_A \oplus k_u$

$\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u$ (Client → Server 1)

$\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u$ (Client → Server 2)

Store $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u, n)$ in a database shared between both servers

Figure 6: Account Creation

mon database DB is initialized, and each server is granted read/write access.

4.2 Account creation

Account creation allows a client to register a new account in the system. We depict the complete procedure in Figure 6. The client begins by choosing the following information:

- E, the client's official email address for this account,
- e, a personal email address to use in account recovery,
- x, additional personal information provided in response to any generic questions, such as a telephone number,
- k_u, a cryptographic key used to encrypt all account data,
- Q, a set of client-chosen security questions,
- A, the client's answers to the security questions Q,
- m, a randomly generated nonce of length λ, where λ denotes the security parameter (e.g., 256 bits), and
- $r = e \parallel Q \parallel m$, a recovery string padded to fixed length ℓ.

First, the servers jointly agree on a unique nonce n for the client. This nonce is immediately sent to the client, who sends back their email string E to both servers. The servers should verify the client's email at this point by emailing them a link. Also, providing E to the servers serves a cryptographic purpose in our protocol: the client makes a K-pop query to each server in pOPRF mode, using email E as the public input and personal info x as the private input. Let $\hat{E}_1$ and $\hat{E}_2$ denote the respective K-pop outputs to the client.

Next, the client computes $H(E \parallel \hat{E}_1 \parallel \hat{E}_2)$, where H is a

Client **Server 1** **Server 2**

E, x → K-pop OPRF mode ← k_1; returns $\hat{E}_1$

E, x → K-pop OPRF mode ← k_2; returns $\hat{E}_2$

$\mathsf{id}, k_E = H(E \parallel \hat{E}_1 \parallel \hat{E}_2)$

id, k_E → Server 1

id, k_E → Server 2

Each server aborts if id is not in the database. Else:

$r = k_E \oplus \mathsf{ct}_r$

Contact client via contact information stored in r using $\mathcal{F}_{\mathsf{SMT}}$, send the message (Q, m, ct_u, n)

Figure 7: Recovery Request

slow password-based key derivation function of variable output length (modeled as a random oracle). The slowness of this function is the reason why our construction provides offline rate limiting against a legal adversary, and the fact that it uses the outputs of the K-pop means that an offline attack can only be performed *after* the adversary compromises all servers. We parse the output of H into two strings: id is the first λ bits of this output, and k_E is the next ℓ bits. This ensures that k_E is the same length as r. The client then computes a one-time-pad ciphertext ct_r of r encrypted under key k_E.

Then, the client makes another K-pop query to each server in pOPRF mode, this time using n as the public input and $A \parallel m$ as the private input, receiving back the respective outputs $\hat{A}_1$ and $\hat{A}_2$. Using the same password-based key derivation function H, they compute $k_A = H(A \parallel \hat{A}_1 \parallel \hat{A}_2)$, where k_A is of length λ. The client then XOR-s k_A with their user key to produce a one-time-pad ciphertext ct_u. Finally, the client sends $\mathsf{id}, \mathsf{ct}_r$, and ct_u to each server. The servers store these values together with the nonce n in a single row of the shared database DB.

4.3 Recovery request

In this stage, the client initiates a request to recover their account after losing their user key. We assume that the client still remembers the same email string E and personal information string x used during account creation. We depict the full protocol in Figure 7.

Recovery requests begin with the client making a K-pop query to each server in OPRF mode, using inputs E and x, and receiving back outputs $\hat{E}_1, \hat{E}_2$. Note that the inputs are the same ones the client provided to the K-pop during account creation, but this time E is a private input instead of a public input. Since both K-pop modes of operation compute the same function, the outputs are the same as in account creation.

The client computes $\mathsf{id}, k_E = H(E \parallel \hat{E}_1 \parallel \hat{E}_2)$, and sends id and k_E to each server. The servers can then look up id in their shared database and retrieve the corresponding ct_r ciphertext. Both servers can decrypt this with k_E to reveal the client's r string, which consists of their recovery email address e, security questions Q, and nonce m.

To impose rate limiting on recovery requests, each server can restrict the number of K-pop queries they respond to in OPRF mode. This limit will not restrict the server's ability to process account creations, since all K-pop operations in account creation are run in pOPRF mode.

4.4 Account verification

In this stage, the recovery servers send an email to the recovery address e. This email contains a customized link that encodes the message (Q, m, ct_u, n). The servers only agree to participate in account restoration if the client clicks on this link within a short time interval.

Our protocol supports two options for account verification. First, the verification step can be done non-cryptographically by having the recovery data r revealed to the servers at the end of the recovery phase, in which case they can send an email in the clear to the recovery address e. This approach has the advantage of being easier to implement, since all other aspects of our account recovery protocol use standard public and symmetric key primitives, but it has the disadvantage that the servers temporarily learn the contact info of clients during account recovery and must be trusted to delete this

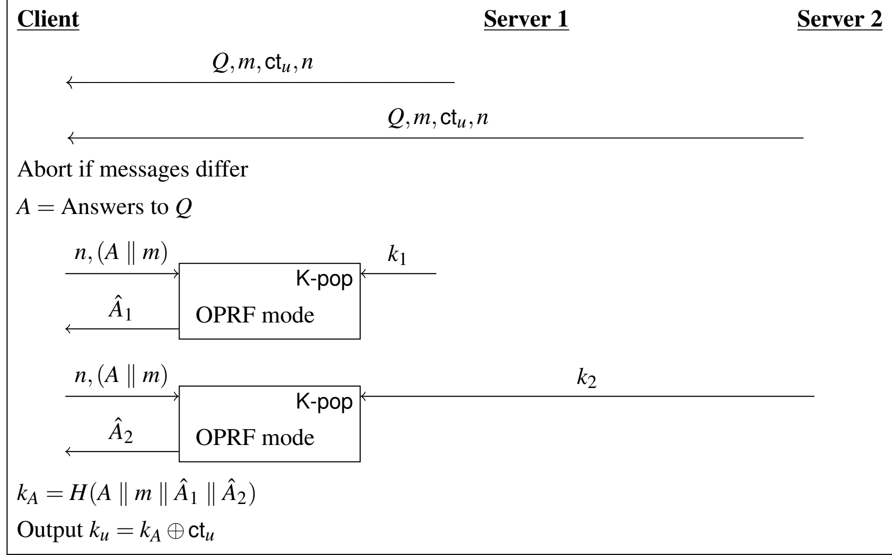

Figure 8: Account Restoration

information promptly. (Still, we stress the importance here of learning at most the email addresses of clients who perform account recovery, rather than a full list of account-holders.) Second, the servers can jointly perform account verification under secure multi-party computation, in order to prevent any server from learning the client's email address. In this case, either the Oblivious TLS protocol of Abram et al. [3] or the MPCAuth protocol [78] would suffice.

In our modeling, we consider both of these options as instantiations of the UC functionality for secure message transmission $\mathcal{F}_{\mathsf{SMT}}$, as shown in Figure 1.

4.5 Account restoration

This final stage of the protocol is shown in Figure 8. Conceptually, this stage is most similar to recovery request, in that it starts with the client submitting K-pop queries to each of the servers in OPRF mode, this time with inputs $x_{\mathsf{priv}} = (A \parallel m)$ and $x_{\mathsf{kal}} = n$. Additionally, the client computes a slow hash function on the OPRF inputs and outputs. One difference between the stages is that account restoration is intended to provide an output to the client, not the servers. Ergo, recovering the one-time pad key k_A from the OPRF outputs is nearly the end of the protocol; all that remains is to use this ephemeral key to recover the user account key k_u.

After the client's account has been restored, it is crucial that they immediately reset their account information (e.g., the password that they have forgotten) so they maintain access in the future. While most of the reset process involves properties of the service outside of our modeling, one step that is crucial is to perform another instance of account creation in order to choose a new nonce to protect the account going forward (and optionally new security questions and answers, if desired).

4.6 Security analysis

In this work, we provide game-based and simulation-secure security analyses of our account recovery construction.

First, we provide an indistinguishability game-based security analysis in Appendices B-C. Our game defines an adversarial model in which the attacker can statically corrupt one of two recovery servers, and adaptively corrupt any number of clients. The adversary can additionally control when honest clients create accounts and run account recovery. The adversary wins if they are able to distinguish between an uncorrupted client's real user key and a randomly sampled key. We prove that our account recovery scheme has the following security.

Theorem 2. *An adversary that sends at most q messages to honest parties has advantage at most $\frac{q}{|A|} + \varepsilon$ in the random oracle model, where ε is negligible in the security parameter, and A is a set of possible security question answers from which real clients' answers are uniformly sampled.*

This result demonstrates the effectiveness of rate-limiting against internal adversaries. Rate-limiting allows honest recovery servers to effectively impose a q value limiting the number of queries that other servers can make. The honest parties can therefore restrict the advantage of internal adversaries via the choice of q.

Our game-based analysis additionally shows that our account recovery construction detects adversarial deviations from the protocol during account recovery and restoration and therefore provides security up to abort, even though our

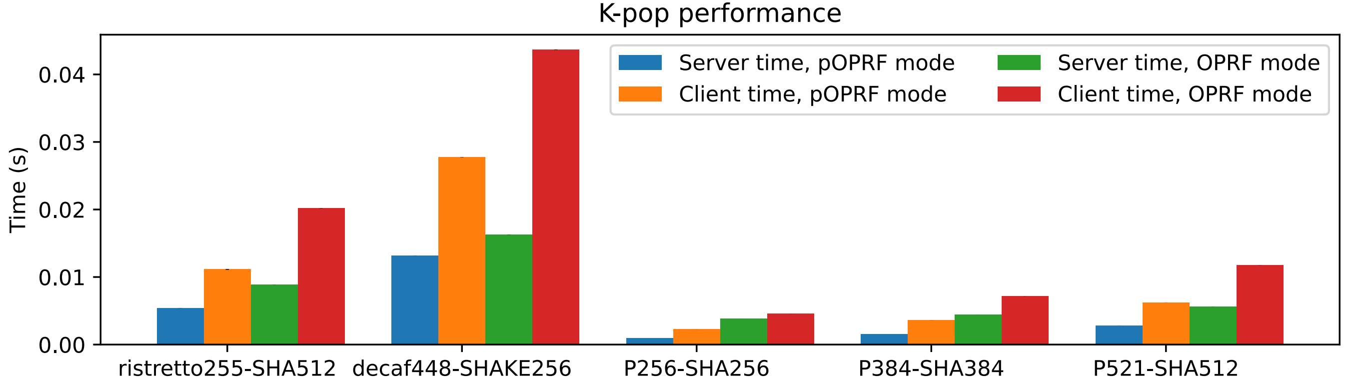

Figure 9: Performance of K-pop implementation over different choices of elliptic curve group and cryptographic hash function. Each bar represents the average of 1000 measurements.

K-pop on its own does not provide verifiability. This proof is conceptually the simpler one to understand, but it restricts the adversary to a single server corruption and only protects accounts created prior to the corruption.

Second, in Appendix F, we provide a simulation-based analysis with universally composable (UC) security in the model of Canetti [22]. We write a protocol description Π_{acc} in the UC style, contribute an idealized version of our protocol as the functionality $\mathcal{F}_{\text{acc}}$, and prove the following.

Theorem 3. *Protocol Π_{acc} UC-realizes functionality $\mathcal{F}_{\text{acc}}$ in the random oracle model.*

This analysis provides a stronger guarantee because it does not restrict the power of the adversary: it can act maliciously from the beginning and in all phases of account creation, recovery, verification, and restoration. To prove Theorem 2, we construct a simulator **Sim** and argue the environment **Env**'s view is identically distributed whether it interacts with (a) Π_{acc} and real-world adversary **Adv** in the $\mathcal{F}_{\text{K-pop}}$-hybrid world or (b) $\mathcal{F}_{\text{acc}}$ and **Sim** in the ideal world. Our formal theorem statements and proofs are provided in Appendices B-F.

5 Implementation

We provide an open-source implementation of the K-pop construction (from §3.3) in Sage.[3] We emphasize that this codebase is distinct from Callisto's own implementation in TypeScript for use on the web.

Our code is based on an implementation [53] of RFC 9497 [39], which standardizes the pOPRF construction of Tyagi et al. [81]. Our implementation of the K-pop in pOPRF mode uses their code directly, with the only modification being the removal of zero-knowledge proofs (since our construction does not require verifiability). For the OPRF mode, we instantiate the additively homomorphic encryption scheme using an open-source implementation [38] of Paillier encryption with a key length of 2048 bits.

[3] Our K-pop implementation code is available at https://github.com/ryanjlittle/kpop-oprf.

5.1 Single-threaded experimental results

We evaluated our implementation on a 1.6GHz Intel Core i5-10210U CPU with 16GB RAM. Figure 9 shows the client-side and server-side performance of the K-pop in both modes of operation for five different choices of Diffie-Hellman group and hash function. Excluding network time, the end-to-end time of a K-pop interaction takes 3.1-41.2 ms in pOPRF mode and 8.4-60.3 ms in OPRF mode, depending on the choice of group and hash function.

Our proof-of-concept implementation is not optimized for speed, and we expect that the computation time can be reduced further. Even so, already these benchmarks lend credence to the efficiency of the account recovery protocol, which requires two separate instances of the K-pop together with a small number of hash function operations, XORs, and database operations. Since all of these server-side costs are negligible compared to the K-pop, our benchmarks show that in a 2-server account recovery system, account creation and the entire recovery procedure each run in 12.4-164.8 ms, ignoring network latency. The client-side runtime would be dominated by the (tunable) cost of the slow, memory-hard hash function like argon2 [16] or scrypt [4].

5.2 Multiprocessing experimental results

In a real-world deployment of our account recovery system, the server side work can scale to a larger number of clients by running many K-pop instances in parallel. We attempted to measure the performance in this setting by evaluating our K-pop implementation locally running on multiple cores. We simulated 512 (single-process) clients simultaneously inter-

Ciphersuite	Mode	Number of cores		
		1	2	4
ristretto255-SHA512	pOPRF	6.866	3.942	2.534
	OPRF	8.829	5.364	3.671
decaf448-SHAKE256	pOPRF	13.259	8.165	5.566
	OPRF	16.357	10.209	6.779
P256-SHA256	pOPRF	1.027	0.544	0.366
	OPRF	3.823	2.289	1.416
P384-SHA384	pOPRF	1.584	0.889	0.616
	OPRF	4.478	2.648	1.639
P521-SHA512	pOPRF	2.824	1.653	1.018
	OPRF	5.547	3.368	2.326

Figure 10: Amortized evaluation time in milliseconds of K-pop server running on $P \in \{1,2,4\}$ parallel cores. Each data point is the average of 512 measurements.

acting with one server that delegates the server response for each client to one of P processes run on separate parallel cores, for $P \in \{1,2,4\}$. The work is evenly split such that each core handles $512/P$ clients.

The performance results for this experiment are given in Figure 10. Experiments were run on an 4-core 1.6 GHz Intel Core i5-10210U CPU with 16GB RAM. On $P = 4$ cores, the server performance is 2-3 times faster than a single-core implementation, depending on the mode of operation and ciphersuite. The run times for $P = 1$ core are slightly higher than the results of the experiment of §5.1 due to overhead from multiprocessing.

6 Conclusion

In summary, this work designs an account recovery protocol that works under stringent functionality and privacy requirements. First, the service provider cannot know or learn the email addresses of all clients. Second, the clients must be able to follow a 'normal' account recovery workflow and have the ability to choose their own security questions—but again, without creating a visible mapping between identities and their choice of security questions. Third, a non-collusion assumption might be broken, and even in this setting the protocol must resist the adversary's ability to recover client data. Finally, our design uses the cryptographic building blocks that our partner organization, Callisto, already understood and knew how to implement and maintain, such as an oblivious pseudorandom function.

Our design is inspired by the application to a secure matching system, and it has been deployed for use in this setting. That said, at least the first three requirements from above are generic and can apply to other account-based privacy-preserving services as well. This work shows that strong security and privacy can be compatible with usability and quality-of-life features like account recovery.

Acknowledgments

R. Little and M. Varia were supported by DARPA under Agreement No. HR00112020021 and by the National Science Foundation under Grants No. 1801564, 1915763, 2209194, 2217770, and 2228610. L. Qin was supported by the NSF Graduate Research Fellowship while at Brown University (where she was during the completion of most of this work) and the Fritz Fellowship through the Initiative for Technology and Society at Georgetown University. We thank Tracy DeTomasi, Scott MacDonald, Ari Adair, and David Archer for lending their expertise in engineering and trauma-informed design through numerous conversations. We also thank the anonymous reviewers and shepherd for their detailed, thoughtful feedback on several iterations of this work.

References

[1] Michel Abdalla, Manuel Barbosa, Peter B. Rønne, Peter Y.A. Ryan, and Petra Šala. Security characterization of J-PAKE and its variants. Cryptology ePrint Archive, Report 2021/824, 2021. https://eprint.iacr.org/2021/824.

[2] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 191–208. Springer, Heidelberg, February 2005.

[3] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious TLS via multi-party computation. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 51–74. Springer, Heidelberg, May 2021.

[4] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 33–62. Springer, Heidelberg, April / May 2017.

[5] Amazon Web Services. AWS nitro system. https://aws.amazon.com/ec2/nitro/, 2023.

[6] Apple Platform Security. Secure enclave. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web, 2021.

[7] ARM. TrustZone technology for the ARMv8-M architecture version 2.0. https://developer.arm.com/documentation/100690/0200/ARM-TrustZone-technology, 2019.

[8] Venkat Arun, Aniket Kate, Deepak Garg, Peter Druschel, and Bobby Bhattacharjee. Finding safety in numbers with secure allegation escrows. In *NDSS 2020*. The Internet Society, February 2020.

[9] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, 2007.

[10] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 1–30. Springer, Heidelberg, November 2020.

[11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 433–444. ACM Press, October 2011.

[12] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. PESTO: proactively secure distributed single sign-on, or how to trust a hacked server. In *EuroS&P*, pages 587–606. IEEE, 2020.

[13] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[14] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000.

[15] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *EuroS&P*, pages 292–302. IEEE, 2016.

[17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*. USENIX Association, 2017.

[18] Callisto. Mission + Vision.

[19] Callisto. Designing trauma-informed and inclusive technology for survivors of sexual assault, November 2017.

[20] Callisto. Callisto Vault, June 2024.

[21] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, Heidelberg, August 2003.

[22] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[23] Ran Canetti. Universally composable signature, certification, and authentication. In *CSFW*, page 219. IEEE Computer Society, 2004.

[24] Ran Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.

[25] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 3–22. Springer, Heidelberg, August 2015.

[26] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

[27] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2022.

[28] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, August 2003.

[29] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global PKI. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 265–296. Springer, Heidelberg, March 2016.

[30] David Cantor, Bonnie Fisher, Susan Chibnall, Shauna Harps, Reanne Townsend, Gail Thomas, Hyunshik Lee, Vanessa Kranz, Randy Herbison, and Kristin Madden. Report on the AAU Campus Climate Survey on Sexual Assault and Misconduct, 2020.

[31] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: Oblivious pseudorandom functions. In *EuroS&P*, pages 625–646. IEEE, 2022.

[32] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy*, pages 1416–1432. IEEE Computer Society Press, May 2020.

[33] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.

[34] Janet X. Chen, Allison McDonald, Yixin Zou, Emily Tseng, Kevin A Roundy, Acar Tamersoy, Florian Schaub, Thomas Ristenpart, and Nicola Dell. Trauma-informed computing: Towards safer technology experiences for all. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[35] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[36] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 857–874. USENIX Association, August 2016.

[37] Poulami Das, Julia Hesse, and Anja Lehmann. DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In Yuji Suga,

Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 682–696. ACM Press, May / June 2022.

[38] CSIRO's Data61. Python paillier library. https://github.com/data61/python-paillier, 2013.

[39] Alex Davidson, Armando Faz-Hernandez, Nick Sullivan, and Christopher A. Wood. Oblivious pseudorandom functions (OPRFs) using prime-order groups. Internet Requests for Comments, December 2023.

[40] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, Heidelberg, January 2005.

[41] Marla E. Eisenberg, Katherine Lust, Michelle A. Mathiason, and Carolyn M. Porta. Sexual Assault, Sexual Orientation, and Reporting Among College Students. *Journal of Interpersonal Violence*, 36(1-2):62–82, 2021.

[42] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015.

[43] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Comput. Surv.*, 54(6):126:1–126:36, 2022.

[44] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

[45] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, Cambridge, UK, 2001.

[46] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[47] Google Cloud. Confidential computing concepts. https://cloud.google.com/confidential-computing/confidential-vm/docs/about-cvm, 2023.

[48] Christine L. Hackman, Sarah E. Pember, Amanda H. Wilkerson, Wanda Burton, and Stuart L. Usdan. Slut-shaming and victim-blaming: a qualitative investigation of undergraduate students' perceptions of sexual violence. *Sex Education*, 17(6):697–711, November 2017.

[49] Julia Hesse, Stanislaw Jarecki, Hugo Krawczyk, and Christopher Wood. Password-authenticated TLS via OPAQUE and post-handshake authentication. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 98–127. Springer, Heidelberg, April 2023.

[50] Alejandro Hevia and Ilana Mergudich-Thal. Implementing secure reporting of sexual misconduct - revisiting WhoToo. In Patrick Longa and Carla Ràfols, editors, *LATINCRYPT 2021*, volume 12912 of *LNCS*, pages 341–362. Springer, Heidelberg, October 2021.

[51] Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, July 2015.

[52] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *NSDI*, pages 817–833. USENIX Association, 2020.

[53] Internet Engineering Task Force. Oblivious pseudorandom functions (OPRFs) using prime-order groups. https://github.com/cfrg/draft-irtf-cfrg-voprf, 2023.

[54] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014.

[55] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, pages 276–291. IEEE, 2016.

[56] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017.

[57] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018.

[58] David Kaplan, Jeremy Powell, and Tom Woller. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[59] Kimberly A. Lonsway and Sergeant Joanne Archambault. Suggested Guidelines on Language Use for Sexual Assault. Technical report, End Violence Against Women International, June 2013.

[60] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Shield: Secure allegation escrow system with stronger guarantees. In *WWW*, pages 2252–2262. ACM, 2023.

[61] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. The IITM model: A simple and expressive model for universal composability. *Journal of Cryptology*, 33(4):1461–1584, October 2020.

[62] Benjamin Kuykendall, Hugo Krawczyk, and Tal Rabin. Cryptography for #MeToo. *PoPETs*, 2019(3):409–429, July 2019.

[63] Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 899–916. USENIX Association, August 2017.

[64] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 487–504. USENIX Association, August 2020.

[65] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 717–732. USENIX Association, August 2021.

[66] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://eprint.iacr.org/2016/046.

[67] Katherine Lorenz, Anne Kirkner, and Sarah E. Ullman. A Qualitative Study Of Sexual Assault Survivors' Post-Assault Legal System Experiences. *Journal of Trauma & Dissociation*, 20(3):263–287, May 2019.

[68] Mara Dolan. These Students Are Bringing Transformative Justice to Their Campus. *The Nation*, January 2020.

[69] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2020.

[70] Microsoft Ignite. Azure confidential computing. https://learn.microsoft.com/en-us/azure/confidential-computing/, 2023.

[71] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467. IEEE Computer Society Press, October 1997.

[72] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[73] Bijeeta Pal, Mazharul Islam, Marina Sanusi Bohuk, Nick Sullivan, Luke Valenta, Tara Whalen, Christopher A. Wood, Thomas Ristenpart, and Rahul Chatterjee. Might I get pwned: A second generation compromised credential checking service. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 1831–1848. USENIX Association, August 2022.

[74] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *2001 IEEE Symposium on Security and Privacy*, pages 184–200. IEEE Computer Society Press, May 2001.

[75] Anjana Rajan, Lucy Qin, David W. Archer, Dan Boneh, Tancrède Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *COMPASS*, pages 49:1–49:4. ACM, 2018.

[76] Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Design and implementation of the ascend secure processor. *IEEE Trans. Dependable Secur. Comput.*, 16(2):204–216, 2019.

[77] SAMHSA's Trauma and Justice Strategic Initiative. SAMHSA's Concept of Trauma and Guidance for a Trauma-Informed Approach. Technical report, Substance Abuse and Mental Health Administration, July 2014.

[78] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. MPCAuth: Multi-factor authentication for distributed-trust systems. In *2023 IEEE Symposium on Security and Privacy*, pages 829–847. IEEE, 2023.

[79] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1556–1571. USENIX Association, August 2019.

[80] Trusted Computing Group. Architecture overview. *Specification Revision*, 1, 2007.

[81] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious PRF, with applications. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 674–705. Springer, Heidelberg, May / June 2022.

[82] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 991–1008. USENIX Association, August 2018.

[83] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *OSDI*, pages 681–696. USENIX Association, 2018.

[84] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

A Additional Cryptographic Background

In this section, we expand upon §3.1 and provide definitions for more cryptographic primitives that we use in this work.

Hash functions. We rely on a cryptographic hash function $H : X \to Y$ in the random oracle model, which asserts that H can only be computed via an oracle query, and furthermore that this function is chosen uniformly at random from the space of all such functions from its finite domain X to its finite codomain Y. In this work, we consider random oracles where the input and output spaces can be either group elements or finite field elements. Our security reductions apply in the programmable random oracle model, which we define via the UC functionality $\mathcal{F}_{\mathsf{pro}}$ in Figure 16 (which itself is reproduced from Canetti et al. [27] with minor changes for our setting).

Because we will use multiple random oracles in this work, for notational convenience we choose to write the oracles as functions H_1, H_2, H_3 and indicate a query to the first oracle as "$h = H_1(m)$," rather than the more cumbersome phrase "send $(\texttt{HashQuery}, m)$ to the instance of $\mathcal{F}_{\mathsf{pro}}$ with session id $\mathsf{sessionid}_1$ and denote its response as h." Additionally, some of our random oracles are assumed to be instantiated with slow, memory-hard hash functions (e.g., `argon2` [16] or `scrypt` [4]) to hinder offline brute-force attacks.

Elliptic curve Diffie-Hellman group. An elliptic curve group is a specific finite, cyclic group G generated by a generator element g in which discrete logarithms are hard. We denote the (known) order of this finite group as $p = |G|$, and hence $\mathbb{F}_p$ is the field of exponents for this group. We assume that G satisfies a strong variant of the Diffie-Hellman assumption introduced by Tyagi et al. [81]—namely, the (m,n) one-more gap strong Diffie-Hellman inversion (SDHI) assumption, which we define formally in §E.1.

Additively homomorphic encryption. An additively homomorphic encryption scheme is a tuple of four algorithms (HomKeyGen, HomEnc, HomEval, HomDec). We use double-bracket notation to denote ciphertexts, in order to remember the plaintext contained within it; for instance, $[\![x]\!] = \text{HomEnc}_{\mathsf{pk}}(x)$. The scheme must satisfy the usual encryption guarantees of correctness (decryption always returns the previously-encrypted plaintext) and semantic security (encryptions of two chosen messages. Additionally, the evaluation algorithm HomEval can compute a new ciphertext that corresponds to addition and/or scalar multiplication of known ciphertexts or plaintexts. We assume that the evaluated ciphertext looks indistinguishable from a freshly-encrypted ciphertext of the same message, even to the holder of the secret decryption key sk.

B Game-Based Security against a Semi-Honest Adversary

This section contains our first security analysis of the account recovery construction. This analysis is perhaps the easiest to understand, albeit with the weakest security model. Specifically, in this section we consider an adversary that is restricted to semi-honest actions, and that can only statically corrupt one of the two servers. Additionally, the adversary is permitted to corrupt as many clients as it wants, and to do so adaptively.

We prove security in this setting via a game-based indistinguishability definition. In this section, we first define our game and then show our proof of security.

B.1 Game definition

Our game-based definition models an internal attacker who has access to one of the two recovery servers and can potentially compromise arbitrary clients. In the game, the adversary can statically corrupt one of the two recovery servers and adaptively corrupt any number of clients. They may prompt uncorrupted clients to honestly perform the various protocols of account recovery: account creation, recovery request, and account restoration. We do not model account verification since it is a non-cryptographic procedure, and instead simply assume that verification always succeeds. The adversary can also instruct a corrupted client to perform the protocols semi-honestly, but with adversary-specified inputs. This captures the idea of an attacker who may lie about user-specified information (e.g. email string and security questions/answers), but cannot modify the code used in the account recovery system. We model a fully malicious adversary who can modify the code in Appendix C.

Our game-based definition is inspired by the game-based definitions of PAKE schemes described in [1, 2, 14]. Our game operates in the real-or-random paradigm, and the adversary is tasked with distinguishing between real user keys and randomly sampled keys. At the onset of the game, the challenger generates servers S_1 and S_2, each initialized with a random $\mathsf{K}\text{-}\mathsf{pop}$ key, and instantiates an empty database shared between the two servers. The challenger also generates a set of N clients. Each client is initialized with an email string e, a personal contact information string x, a set of security questions q, and their respective answers a. These values are sampled from sets E, X, Q, and A respectively. Additionally, each client is instantiated with a randomly sampled user key k_u. The initialization procedures for servers and clients are described in Figure 11.

Finally, the game master samples a random bit b, which determines if the game will be played in the real world or the random world. The adversary wins if they output a bit that matches b.

The adversary is allowed to interact with the game through four oracles. The first is a programmable random oracle H,

InitServer()
$k \xleftarrow{\$} \mathcal{K}^{\text{K-pop}}$ state $= \bot$

InitClient()
$e \xleftarrow{\$} E$ $x \xleftarrow{\$} X$ $q \xleftarrow{\$} Q$ $a \xleftarrow{\$} A$ $k_u \xleftarrow{\$} \mathcal{K}$ state $= \bot$ corrupted $=$ False tested = False

Figure 11: Server and Client initialization procedures

which we use to model the slow, password-based key derivation function in our protocol. We also allow the adversary to make queries to the oracles Execute(), Corrupt(), and Test(). All oracle queries are routed through the game master, who takes the adversary's query and returns the oracle response. The oracles are described in detail in Figure 12. At a high level, the Execute() oracle allows the adversary to specify a client and protocol and have that client run the protocol. If the client is corrupted, the adversary may supply inputs for the client to use in the protocol. The Corrupt() oracle allows the adversary to reveal a specified client's entire internal state. The Test() oracle allows the adversary to specify a client and reveal either their user key or a random key, depending on the bit b. The adversary is not allowed to use the Corrupt() and Test() oracles on the same client–once a client is corrupted it cannot be tested, and vice-versa.

Let WIN be the event that an adversary $\mathcal{A}$ wins the account recovery game (by outputting a bit that equals b). We define the advantage of $\mathcal{A}$ in the semi-honest account recovery game as $\mathbf{Adv}^{\mathsf{SH-acc-rec}}(\mathcal{A}) = 2 \cdot \mathbb{P}[\mathsf{WIN}] - 1$, and say an account recovery protocol is secure against semi-honest adversaries if the advantage is negligible for all possible adversaries.

B.2 Security analysis

Theorem 4. *For all semi-honest polynomial time adversaries $\mathcal{A}$ attacking the account recovery protocol in section 4 that make at most q_{ex} calls to the* Execute() *oracle and q_{ro} calls to the random oracle, there exist the following adversaries:*

- *$\mathcal{B}_1$ attacking the pseudorandomness of the K-pop scheme*
- *$\mathcal{B}_2$ attacking the obliviousness of inputs in the K-pop scheme*

Execute($\mathbf{C}_i, \Pi, x$)
Models an honest client executing part of the protocol. Π is the subprotocol to run: either recovery part one or part two. The input x is an optional argument specifying the client's state and internal variables. If $\mathbf{C}_i$ is corrupted, they run the protocol with the specified state and variables in x. If $\mathbf{C}_i$ is uncorrupted, the input is ignored and the protocol is run with their internal state and variables. The oracle returns the transcripts of the interactions and the view of the corrupted server. If $\mathbf{C}_i$ is corrupted, the oracle additionally outputs their updated view after executing the protocol.

Corrupt($\mathbf{C}_i$)
If $\mathbf{C}_i$ has previously been tested, return **Fail**. Otherwise, set $\mathbf{C}_i$.corrupted $=$ True and return all of $\mathbf{C}_i$'s private attributes and state.

Test($\mathbf{C}_i$)

If $\mathbf{C}_i$ has previously been corrupted, return **Fail**.
Otherwise, set $\mathbf{C}_i$.tested $=$ True and do the following:
If $b = 0$:
  Return $\mathbf{C}_i$'s user key and any past user keys
Else:
  Let n be the number of user keys $\mathbf{C}_i$ has ever used.
  Return n randomly sampled keys.

Figure 12: Oracles available to semi-honest adversary

such that

$$\mathbf{Adv}^{\mathsf{SH-acc-rec}}(\mathcal{A}) \leq \frac{q_{ex}}{|A|} + \frac{N^2 + q_{ro}}{2^\lambda} + \mathbf{Adv}^{\mathsf{PRF}}(\mathcal{B}_1) + \mathbf{Adv}^{\textit{K-pop}}(\mathcal{B}_2),$$

where A is the dictionary of security question answers, N is the number of clients, and λ is the security parameter.

Proof. The proof is a series of hybrid experiments. For each experiment i we define the event WIN_i representing the event that the adversary succeeds in the game of experiment i.

Experiment 0: The real protocol. By definition, we have

$$\mathbf{Adv}^{\mathsf{SH-acc-rec}}(\mathcal{A}) = 2 \cdot \mathbb{P}[\mathsf{WIN}_0] - 1.$$

Experiment 1: In the Execute oracle, replace all outputs of the honest server's K-pop (in both public and private input modes) with outputs from a random function. Distinguishing between this experiment and the previous one is at least as hard as breaking the pseudorandomness of the underlying PRF.

Lemma 1. *There exists an adversary $\mathcal{B}_1$ such that* $|\mathbb{P}[\mathsf{WIN}_0] - \mathbb{P}[\mathsf{WIN}_1]| \leq \mathbf{Adv}^{\mathsf{PRF}}(\mathcal{B}_1)$.

Proof. Suppose an adversary $\mathcal{A}_1$ can distinguish between experiments 0 and 1 with non-negligible advantage. Then let $\mathcal{B}_1$ be an adversary attacking the pseudorandomness of the honest server's K-pop function. $\mathcal{B}_1$ acts as the challenger in the account recovery game, but replaces all honest server K-pop calls with calls to the real-or-random PRF oracle. If we are in the real world of the PRF game, the account recovery game matches experiment 0, while if we are in the ideal world, the game matches experiment 1. $\mathcal{B}_1$ runs $\mathcal{A}_1$ on this game and outputs its result. Then $\mathcal{B}_1$ has non-negligible advantage in breaking the pseudorandomness of the underlying PRF of the K-pop. □

Experiment 2: In the execute oracle, replace all K-pop protocol interactions between an honest client and the corrupted server with a simulation of the interactions. Further, any time an honest client is corrupted with the Corrupt query, re-program the random oracle such that the client's output in all K-pop interactions is consistent with the simulated transcript.

Lemma 2. *There exists an adversary $\mathcal{B}_2$ such that* $|\mathbb{P}[\mathsf{WIN}_1] - \mathbb{P}[\mathsf{WIN}_2]| \leq \mathbf{Adv}^{\textsf{K-pop}}(\mathcal{B}_2)$.

Proof. Suppose the adversary $\mathcal{A}_2$ can distinguish between experiments 1 and 2 with non-negligible advantage. Then let $\mathcal{B}_2$ be an adversary against the obliviousness of the corrupted server K-pop. Have $\mathcal{B}_2$ act as the challenger in the account recovery game of experiment 1, but replace all K-pop communications between an honest client and the corrupted server with messages from either the real or ideal distribution of the K-pop. These respective worlds determine if the account recovery game matches experiment 1 or experiment 2. $\mathcal{B}_2$ runs $\mathcal{A}_2$ on this game and returns the output. Then $\mathcal{B}_2$ has non-negligible advantage in breaking the obliviousness of the K-pop function. □

Experiment 3: During client creation, ensure all nonce values m are unique by sampling without replacement.

Lemma 3. $|\mathbb{P}[\mathsf{WIN}_2] - \mathbb{P}[\mathsf{WIN}_3]| \leq \frac{N^2}{2^\lambda}$.

Proof. Experiments 2 and 3 can only be distinguished if a collision occurs among the nonces in experiment 2. The probability of this occurring is bounded by the birthday bound, which gives an upper bound of $\frac{N^2}{2^\lambda}$. □

Experiment 4: Let B be the event that the adversary uses the Execute oracle on a corrupted client to run account creation or restoration with specified inputs A and m that match the security answer and nonce string of some uncorrupted client. Suppose that B has NOT occurred, and the adversary make a random oracle query with input $(A, m, \hat{A}_1, \hat{A}_2)$, where A and m match the security answer and nonce string of an uncorrupted client, and $\hat{A}_1$ and $\hat{A}_2$ match the respective K-pop outputs of (A, m). If this occurs, have the game output "SUCCESS" and abort.

Lemma 4. $|\mathbb{P}[\mathsf{WIN}_3] - \mathbb{P}[\mathsf{WIN}_4]| \leq q_{ro} \cdot \frac{1}{2^\lambda}$.

Proof. The honest server's K-pop was replaced with a random function in experiment 1, so the only way an adversary can learn the honest server's PRF output on input (A, m) is by either evaluating it via the Execute oracle or randomly guessing the output. The event B corresponds to the case in which the adversary uses the Execute oracle to evaluate the K-pop. So assuming B has not occurred, the adversary's success probability at guessing $\hat{A}_1$ or $\hat{A}_2$ is no better than randomly guessing. Moreover, the adversary can only check a random guess for a single client at a time, since the random oracle is only reprogrammed if the adversary provides the correct nonce m as well, which is a unique value by experiment 3. The K-pop output length is λ, so by the union bound the probability of randomly guessing correctly is $\leq q_{ro} \cdot \frac{1}{2^\lambda}$. □

Experiment 5: In all honest client account creations, replace the ciphertext ct_u with a random string of the same length, ct'_u. Suppose B, as defined in experiment 3, has not occurred. Then upon corruption of a client i, reprogram H such that $H(A_i, m_i, \hat{A}_1^i, \hat{A}_2^i)$ returns a key k'_A such that k'_A decrypts ct'_u to k_u. In other words, set $k'_A = \mathsf{ct}'_u \oplus k_u$.

Lemma 5. $\mathbb{P}[\mathsf{WIN}_4] = \mathbb{P}[\mathsf{WIN}_5]$

Proof. Note that the value of k'_A is uniformly distributed (due to the randomness of k_u), so the reprogrammed random oracle maintains a uniformly random distribution. Further, experiment 4 forbids the adversary from using the random oracle to find $H(A_i, m_i, \hat{A}_1^i, \hat{A}_2^i)$. And since B has not occurred, the adversary cannot have used the Execute oracle to learn $H(A_i, m_i, \hat{A}_1^i, \hat{A}_2^i)$. Therefore reprogramming the random oracle incurs no security loss.

Furthermore, due to the perfect secrecy of one-time-pad encryption, replacing ct_u with a random ciphertext incurs no security loss.

□

Lemma 6. $\mathbb{P}[WIN_5] \leq \frac{q_{ex}}{|A|}$

Proof. Suppose B, as defined in experiment 4, never occurs. The only oracle queries that are affected by an honest client's k_u value is the generation of z during account creation. But this is replaced with a random ciphertext in experiment 4. Therefore all honest client's user keys are completely indpendent from all oracle outputs, so the adversary's advantage of distinguishing between the real key and a random key is zero.

Finally, we bound the probability that B occurs. This event corresponds to the event that the adversary initiates a malicous execute request that correctly guesses an honest client's A and m values together. Since the adversary can learn an honest client's m value by having them honestly run account recovery, we only consider the difficulty of guessing A. Note that the adversary can only make a guess of this for for a single client at a time, since their guess must contain m, which is unique

to each client by experiment 3. Since each client's A value is uniformly sampled from the set A, the union bound gives us that the probability of the adversary guessing an honest client's key is $\leq \frac{q_{ex}}{|\mathcal{A}|}$. □

Finally, Lemmas 1-6 together prove Theorem 4. □

C Game-Based Security against a Malicious Adversary

This section extends the analysis from Appendix B to consider a malicious adversary.

C.1 Game definition

The malicious security game is a modification of the semi-honest security game defined in §B.1. The threat model that the game captures is a malicious adversary who corrupts one of the two recovery servers, and is attempting to learn information about clients who created their account prior to the corruption.

The beginning of the game is identical to the semi-honest game: the game master initializes all honest clients and servers, the adversary chooses a server to corrupt, and can then make as many Corrupt(), Test(), and Execute() queries as it wants. At some point though, the adversary may tell the game it wishes to switch to malicious security. At this point, the adversary loses access to the Execute oracle but gains access to a new oracle: Send. The Send oracle, described in Figure 13, allows the adversary to send arbitrary messages *within protocols* from a corrupted client or corrupted server. The Send oracle can only send messages in the recovery request and account restoration protocols, not account creation. Thus once the adversary switches to malicious security, they lose the ability to create new accounts.

The adversary's win condition is the same as the semi-honest game: they win if they output a bit that matches b. We define the advantage of an adversary $\mathcal{A}$ as $\mathbf{Adv}^{\mathsf{mal-acc-rec}}(\mathcal{A}) = 2\cdot\mathbb{P}[\mathsf{WIN}]-1$, and say an account recovery protocol is secure against malicious adversaries if the advantage is negligible for all possible adversaries.

To prove that our protocol has malicious security, we treat the K-pop as a white box. In particular, we use the K-pop construction of §3.3, and model the outer hash function H_2 as a random oracle.

C.2 Security analysis

Theorem 5. *For all polynomial time adversaries $\mathcal{A}$ attacking the account recovery protocol of section 4 that make at most q_s calls to the* Send *oracle, q_{ex} queries to the* Execute *oracle, q_{ro_1} calls to the random oracle H (used for key derivation), and q_{ro_2} calls to the random oracle H_2 (used in the K-pop), there exist the following adversaries:*

- *$\mathcal{B}_1$ against the pseudorandomness of the PRF underling the K-pop*
- *$\mathcal{B}_2$ against the obliviousness of the K-pop scheme*

such that

$$\mathbf{Adv}^{\mathsf{mal-acc-rec}}(\mathcal{A}) \leq \frac{q_{ex}+q_s}{|A|} + \frac{N^2+q_{ro_2}^2+q_{ro_1}}{2^\lambda} + \mathbf{Adv}^{\mathsf{PRF}}(\mathcal{B}_1) + \mathbf{Adv}^{\textsf{K-pop}}(\mathcal{B}_2),$$

Send(*Sender*, *Recipient*, m)
If *Sender* is an uncorrupted party or *Recipient* is a corrupted party, return **Fail**. Otherwise, have *Recipient* and all other honest parties process the message m honestly, update their state, and return their response messages (if any).

Figure 13: The Send oracle

Proof. The security analysis proceeds through a series of hybrid experiments. For each experiment i, we define an event WIN_i representing the event that the adversary succeeds in the game of experiment i.

Experiment 0: The real protocol. By definition, we have

$$\mathbf{Adv}^{\mathsf{mal-acc-rec}}(\mathcal{A}) = 2\cdot\mathbb{P}[\mathsf{WIN}_0]-1.$$

Experiment 1: Disallow collisions in H_2, the outer random oracle used in the K-pop. If the adversary makes a random oracle query that collides with a previous different random oracle input, output **Success** and end the game.

Lemma 7. $\mathbb{P}[|WIN_0 - WIN_1|] \leq \frac{q_{ro_2}^2}{2^\lambda}$

Proof. Distinguishing between experiment 0 and 1 requires finding a collision in the random oracle. By the birthday bound, the probability of finding a collision in q_{ro_2} queries is $\leq \frac{q_{ro_2}^2}{2^\lambda}$. □

Experiment 2: If the adversary uses the Send oracle during a K-pop procedure to send a server message inconsistent with their K-pop key, have the honest server abort on this execution. This event is detectable to the game master, since the game master knows the corrupted server's K-pop key.

Lemma 8. $\mathbb{P}[WIN_1] = \mathbb{P}[WIN_2]$

Proof. Suppose (for contradiction) the adversary $\mathcal{A}_1$ can distinguish between experiments 0 and 1 with nonzero advantage. Then $\mathcal{A}_1$ must have sent an incorrect K-pop output that causes the client to construct the correct output. Since we have banned collisions in the H_2 oracle, this means a collision

occurred in the inner PRF: i.e., the adversary used some K-pop key $k' \neq k$ such that for the corresponding inputs $(x_{\mathsf{kal}}, x_{\mathsf{priv}})$, we have $H_1(x_{\mathsf{priv}})^{1/(k+H_3(x_{\mathsf{kal}}))} = H_1(x_{\mathsf{priv}})^{1/(k'+H_3(x_{\mathsf{kal}}))}$. But this is impossible, since we are working over a cyclic group. □

Experiment 3: If the adversary uses the Send oracle to send value id, k_E from a malicious client to the honest server, where id, k_E do not match the values of any corrupted client, have the honest server abort.

Lemma 9. $|\mathbb{P}[WIN_2] - \mathbb{P}[WIN_3]| \leq \frac{q_{ex}}{2^\lambda}$

Proof. We demonstrate this claim through a security reduction: given any adversary $\mathcal{A}$ against Experiment 2, we construct a new adversary $\mathcal{B}$ against Experiment 3 that is almost as likely to succeed. In more detail, $\mathcal{B}$ acts as the game master in an emulation of Experiment 2 that it plays with $\mathcal{A}$. It forwards along every query from $\mathcal{A}$ to its own Experiment 3 game, except for the queries that cause aborts (which its own game will not respond to, by definition) so $\mathcal{B}$ needs to find a different way to respond to these queries.

Note that if the malicious client provides an id that does not match the identifier for any client (honest or corrupted), then aborting is the correct behavior by the honest server. The new wrinkle introduced by Experiment 3 is that the honest server aborts even if id happens to correspond to an honest client.

If the adversary $\mathcal{A}$ submits a maliciously-chosen id, k_E whose id happens to match an honest client, then $\mathcal{B}$ can take on the role of the honest server in Experiment 2 by fetching the corresponding ciphertext ct_r and calculating $r = \mathsf{ct}_r \oplus k_E$. In particular, we focus on the client nonce m inside of the recovery string r. Due to the properties of the one-time pad, there is a bijection between the adversary's choice of k_E and the subsequent nonce m that results. This nonce can fall into one of two options:

- The adversary $\mathcal{B}$ has already observed the nonce m, because it corresponds to an honest client that has already run account recovery and provided m to the malicious server. In this case, $\mathcal{B}$ has already seen this honest client's contact info, so it can submit a new account recovery to Experiment 3 acting as the honest client. Experiment 3 will not abort on this query, and will allow $\mathcal{B}$ to proceed to account restoration. At this point, $\mathcal{B}$ will forward along any message that $\mathcal{A}$ sent in restoration for the (non-aborting) Experiment 2.
- The adversary $\mathcal{B}$ has not previously observed this nonce m. In this case, $\mathcal{B}$ pretends to be the honest server and, when adversary $\mathcal{A}$ submits an K-pop query during account restoration, $\mathcal{B}$ returns a random value in response.

Because $\mathcal{B}$'s emulation is perfect in the first case, it remains to analyze the probability that $\mathcal{A}$ detects $\mathcal{B}$'s forgery in the second case. This is precisely the probability that the nonce m is subsequently used by an honest client for account recovery and restoration, in which case both Experiments 2 and 3 will return the PRF output of $A \parallel m$ with the honest server's actual key rather than the forgery that $\mathcal{B}$ has made. But because clients choose m uniformly at random from the space $\{0,1\}^\lambda$ and keep them secret until recovery, the probability of adversary $\mathcal{A}$ guessing the not-yet-revealed nonce of an honest client is at most $\frac{q_{ex}}{2^\lambda}$.

□

Experiment 4: Replace the Send oracle with the following oracle.

ExecuteWithAbort(*Sender*, *Receiver*, *m*)
If the adversary has previously queried the Corrupt() oracle, return **Fail**. Otherwise, Have $\mathbf{C}_i$ run protocol Π honestly. Return the state of the malicious server and a transcript of all messages sent to and from the malicious server. If *Sender* is an uncorrupted party or *Recipient* is a corrupted party, return **Fail**. If m does not match an expected, honest message that would be sent by the Sender party, return **Abort**. Otherwise, send the message and return the responses of all honest parties.

Lemma 10. $\mathbb{P}[WIN_3] = \mathbb{P}[WIN_2]$

Proof. After the changes made in Experiments 2 and 3, the protocol is guaranteed to abort if the adversary ever deviates from the prescribed protocol. Thus, the Send oracle from Experiment 2 has now become exactly the same as the ExecuteWithAbort oracle. □

Experiment 5: Replace the ExecuteWithAbort oracle with the following oracle:

Execute'(*Sender*, *Receiver*, *x*)
Let Π be the protocol that message m belongs to. run Execute(*Sender*, Π), ignoring whether or not *Sender* is corrupted, and return the output.

Lemma 11. $\mathbb{P}[WIN_3] \leq \mathbb{P}[WIN_4]$.

Proof. Suppose $\mathcal{A}_4$ is an adversary attacking the game in experiment 4. We can construct an adversary $\mathcal{B}$ that attacks the game in experiment 3 with the same success probability. Have $\mathcal{B}$ act as the challenger in the game of experiment 4. Run $\mathcal{A}_4$ on this game. Any time $\mathcal{A}_4$ makes an ExecuteWithAbort query, first check that if the query should cause an abort given the state of the game. If so, return an **Abort** to $\mathcal{A}_4$. Otherwise, let Π be the protocol associated with the message m specified by $\mathcal{A}_4$, and make an Execute' query for that protocol. Return just the responses to message m in that protocol to $\mathcal{A}$. Have $\mathcal{B}$ output the result of $\mathcal{A}_4$. Since the game running inside $\mathcal{B}$ is exactly experiment 3, $\mathcal{B}$ and $\mathcal{A}_4$ have equal win probabilities.
□

Lemma 12.

$$\mathbb{P}[WIN_4] \leq \frac{q_{ex}+q_s}{|A|} + \mathbf{Adv}(\mathcal{B}_{PRF}) + \mathbf{Adv}(\mathcal{B}_{K\text{-}pop}) + \frac{N^2}{2^{\lambda+1}} + \frac{q_{ro}}{2^\lambda}.$$

Proof. Experiment 4 corresponds to the semi-honest account recovery game with $q_{ex}+q_s$ possible queries to the Execute oracle. Lemma 12 then follows from Theorem 4. □

Finally, Theorem 5 follows from Lemmas 7- 12. □

D Universally Composable Security Primer

In the remainder of this work, we provide a simulation-based security analysis of our K-pop and account recovery protocols in the UC security framework of Canetti [22]. To provide context for these security analyses, in this section we provide a short (and thus necessarily incomplete) primer on the concepts involved in UC security.

Overview of simulation-based security. By way of contrast to game-based security analyses such as the ones in §B-C, simulation-based security analyses compare a real-world cryptographic protocol to an idealized abstraction of the functionality that one would want a trusted party to execute on everyone's behalf, if such a trusted party were to exist. Simulation-based proofs show that the real-world protocol is no "worse" or "leakier" than the ideal functionality, in the sense that any real-world artifacts of the protocol can be *simulated* from the relevant ideal-world inputs and outputs. Hence, simulation-based security analyses involve three parties: the adversary **Adv**, simulator **Sim**, and distinguisher.

One benefit of simulation-based security (beyond game-based notions) is to facilitate security analyses involving composition—that is, when combining either multiple instances of a single cryptographic protocol, combining many cryptographic protocols together, or integrating cryptographic protocols into a larger system. Simulation-based security has been shown to facilitate the analysis of sequential composition in many domains such as zero-knowledge proofs and secure multi-party computation (see e.g., Goldreich [45, §4.3.4] and Lindell [66, §6.3] for details).

Overview of UC security. Even simulation-based security sometimes struggles when protocols are arbitrarily interleaved, or to consider security when composed against arbitrary, unspecified other protocols. Several frameworks for *universally composable security* have been developed to address these concerns (e.g., [9, 22, 25, 51, 61, 74]). In this work, we follow the formalism of Canetti [22] with several of its enhancements over time (e.g., [10, 24, 26, 28]).

Canetti's UC security is a special case of simulation-based security that embodies the idea that other protocols (whether other instances of the same protocol, or different protocols entirely) may be run at the same time and interleaved arbitrarily with the protocol under consideration. To allow for such generality and to enable modular analysis of a single cryptographic protocol at a time, the UC security model folds all other protocol executions into a new entity called the *environment* **Env**. In Canetti's model, the environment also takes on the role played by the distinguisher: attempting to tell apart the real and ideal worlds. (It turns out that the environment can also take over the role of the real-world adversary, although for the sake of clarity we choose in this work to keep the roles of **Env** and **Adv** separate.)

UC formalism. A UC analysis begins with the specification of a real-world protocol Π and an ideal functionality $\mathcal{F}$. This idealized abstraction should have identical input-output behavior as the real-world protocol, and it should provide at least as much "leakage" to the simulator **Sim** as the real-world protocol provides to the adversary **Adv**.

Additionally, the environment **Env** has a few powers: it starts each protocol execution by providing the inputs of all honest parties, it receives the outputs of all honest parties, and it can interact arbitrarily with the adversary. The environment can use all of this information when performing its job of distinguishing the real and ideal worlds.

Concretely, a protocol Π is deemed to be a *UC-realization* of the functionality $\mathcal{F}$ if there exists an efficient simulator **Sim** such that no environment can tell whether it is interacting with Π together with **Adv**, or with $\mathcal{F}$ together with **Sim**. Formally, we write this by saying that $\text{exec}_{\mathbf{Env},\Pi,\mathbf{Adv}} \approx \text{exec}_{\mathbf{Env},\mathcal{F},\mathbf{Sim}}$ for all probabilistic polynomial time (PPT) environments **Env**.

We emphasize that only the view of the environment must be indistinguishable between the real and ideal worlds. Other protocol participants might have internal state that differs in the two executions; for instance, the simulator has local state in the ideal world but does not even exist in the real world.

UC execution model. Implicitly, the UC model allows the environment to run other cryptographic protocols in tandem with the one under considration. To model this behavior, the environment is in charge of the scheduling of protocols and the delivery of network messages.

In more detail, Canetti's UC framework models concurrent behavior in a non-standard but generic and flexible manner: by actually forbidding concurrency in the model itself so that only one protocol participant can be active to perform computation and/or communication at a time, and then giving

the environment control over which protocol participant gets activated next. In fact, in this model, control flow proceeds together with communication: that is, if party A sends a message to party B, then that is the end of party A's activation and the start of party B's activation.

Canetti's UC model formalizes this approach using interactive Turing machines (ITMs). Each machine has tapes for local computation, and tapes for sending messages to other machines. Messages can take the form of direct input/output messages that are transmitted to the entity that invoked the protocol (typically the environment, unless composition is involved) or indirect messages that are sent through the adversary (typically for all communication between parties within the execution of a single protocol).

Since a single protocol can be executed multiple times in sequence, parallel, or concurrently: by convention the UC framework assigns a unique identifier sessionid to each session of a single protocol. Individual machines are identified by (a) the session id of the protocol in which they participate and (b) an identifier of one particular party within the protocol.

Composition and subroutines. A protocol or functionality in the UC model is allowed to invoke another protocol or functionality as a *subroutine* (e.g., a zero knowledge proof system is allowed to invoke a message commitment protocol). Direct messages are used for communication between a subroutine and its caller.

When analyzing a protocol that contains one or more subroutines, it would suffice to "inline" all instance of the subroutine protocol (e.g., pick a single instantiation of message commitment and analyze the resulting zero knowledge proof protocol). However, the UC security framework provides a powerful abstraction through its *universal composition theorem*. The UC theorem states that if:

- real-world protocol Π UC-realizes ideal functionality $\mathcal{F}$,
- we form a larger "hybrid" protocol ρ that uses the ideal functionality $\mathcal{F}$ as a subroutine, and
- this hybrid protocol ρ UC-realizes ideal functionality $\mathcal{G}$,

then the instantiation $\rho^{\mathcal{F}\to\Pi}$ (which replaces every invocation of $\mathcal{F}$ with its real-world counterpart Π) also UC-realizes $\mathcal{G}$.

In other words, it suffices to analyze building blocks separately in the UC framework, after which they can be composed arbitrarily. Formally, in this situation we say that $\rho^{\mathcal{F}\to\Pi}$ UC-emulates ρ.

UC with global subroutines. The original UC framework by Canetti [22] insisted that $\mathcal{F}$ be *subroutine respecting*, meaning that it only interacts with a single caller. In our zero knowledge example: while many cryptographic protocols could use message commitment protocols, the model would insist that the specific message commitments used within the ZK protocol are *only* used by the ZK protocol and never invoked by any other protocol or functionality. Many cryptographic protocols can naturally be decomposed in a subroutine-respecting manner.

However, this limitation is difficult to impose in scenarios that involve global setup, such as a random oracle or common reference string that is available to everyone and that all protocols can use. Subsequent refinements of the UC framework [10, 24, 26] allow for the existence of such *global subroutines* that exist in both the real and ideal worlds. We say that protocol Π UC-realizes functionality $\mathcal{F}$ *in the presence of global subroutine* $\mathcal{H}$ if there exists an efficient simulator **Sim** such that no environment can tell whether it is interacting with Π, **Adv**, and $\mathcal{H}$, or if it is interacting with $\mathcal{F}$, **Sim**, and $\mathcal{H}$. Looking ahead, in this work we consider UC analyses with a single global subroutine: the programmable random oracle functionality $\mathcal{F}_{\mathsf{pro}}$.

E UC Security Analysis of K-pop

In this section, we provide a UC-secure analysis of the account recovery protocol $\Pi_{\text{K-pop}}$ (from Figures 4-5) in order to prove Theorem 1 that it UC-realizes $\mathcal{F}_{\text{K-pop}}$.

Theorem 1. *Assume that group G satisfies the one-more gap strong Diffie-Hellman inversion assumption and that* HomEnc *satisfies indistinguishability under a chosen plaintext attack (IND-CPA). Then, the* K-pop *protocol $\Pi_{\text{K-pop}}$ (in Figures 4-5) UC-realizes the ideal functionality $\mathcal{F}_{\text{K-pop}}$ in the presence of a programmable random oracle $\mathcal{F}_{\mathsf{pro}}$.*

As a reminder of UC terminology, the claim that "A UC-realizes B in the presence of C" means that the environment cannot distinguish between the real world A and adversary **Adv** or the ideal world B and simulator **Sim**, where both worlds have access to a global subroutine C. In this case, the global subroutine $\mathcal{F}_{\mathsf{pro}}$ is used to instantiate all three random oracles H_1, H_2, and H_3. We refer readers to §D for a broader overview of UC security.

By contrast, the functionality $\mathcal{F}_{\mathsf{OPRF}}$ is not mentioned in the theorem above because it is subroutine-respecting: each instance of $\mathcal{F}_{\mathsf{OPRF}}$ only interacts with its own corresponding instance of $\mathcal{F}_{\text{K-pop}}$ and is not otherwise accessible to the outside world. (In other words: once an OPRF is used to construct a K-pop, then clients can only call the pOPRF and OPRF modes within the K-pop. It is not permitted also to expose the underlying $\mathcal{F}_{\mathsf{OPRF}}$ for clients to call it directly.)

Overview of proof strategy. To prove this theorem, we describe the construction of a simulator **Sim** that emulates the execution of the real protocols in both pOPRF and and OPRF modes. As with any UC security analysis, our simulator **Sim** interacts with the ideal functionality $\mathcal{F}_{\text{K-pop}}$ and also has oracle access to the real-world adversary **Adv**.

The adversary itself is expecting to interact with an instance of $\Pi_{\text{K-pop}}$, so the simulator acts as the environment, the programmable random oracle $\mathcal{F}_{\text{pro}}$, and all honest parties in this emulation of the real world. The simulator must take its messages from $\mathcal{F}_{\text{K-pop}}$ and construct a view of the honest parties in its interaction with **Adv** that is computationally indistinguishable from a real-world execution on the same inputs. Then, it must interpret the resulting actions of the adversary **Adv** and use this to inform its subsequent commands to the ideal functionalities $\mathcal{F}_{\text{K-pop}}$ and $\mathcal{F}_{\text{OPRF}}$.

In this security analysis, we show that any environment (and associated dummy adversary [22]) that can distinguish the simulation from reality must also be able to break one of three security assumptions. The first two are standard in cryptography: the birthday bound on finding collisions in a random oracle, and semantic security of the (additively homomorphic) public key encryption scheme. The last one is a strong variant of the Diffie-Hellman assumption on the group G; while non-standard, we remark that it is known to hold in the algebraic group model [44, 81].) We describe this assumption in more detail below.

Differences from prior work. Our pOPRF construction in Figure 4 is nearly identical to the construction of Tyagi et al. [81], so our security analysis follows the same high-level approach as theirs. However, there are three important distinctions between our work and theirs.

First, we generalize their security analysis to allow for the possibility that the server chooses different OPRF key k and (in the pOPRF case) public input x_{kal}. By contrast, they construct a verifiable OPRF by using zero-knowledge proof to check the honesty of the server's actions. To accomplish this goal, our proof uses techniques from non-verifiable OPRFs like the work of Jarecki et al. [55]. As a consequence, we carefully design and reason about the simulator **Sim**'s actions for all queries when both the client and server are either honest or corrupted.

Second, we strengthen their security guarantee from a standalone game-based security analysis to a simulation-style analysis in the setting of universally composable (UC) security. In particular, we provide security against adaptive corruptions of the server and clients. As a consequence, we carefully design **Sim** at the moment of client or server corruption to ensure that any newly-discovered secrets are consistent with the prior view of the adversary (using the SDHI assumption).

Third, our K-pop adds an OPRF mode of operation on top of the pOPRF mode from Tyagi et al. [81]. This adds a few new challenges in our simulator design and security analysis. Between the pOPRF and OPRF modes, we must ensure that **Sim**'s emulation of evaluations and oracle queries yields consistent answers. Within the OPRF mode, we need to simulate additional data held by the client and server (e.g., the variables s, z, sk, pk) and handle server corruption in a different way due to the different way that the variable β is calculated.

E.1 SDHI assumption

In this section, we specify the (m,n) one-more gap strong Diffie-Hellman inversion (SDHI) assumption introduced by Tyagi et al. [81]. We provide a formal statement of the assumption itself for completeness, but we note upfront that we only use it for the purpose of proving Lemma 13, so readers should feel free to skip ahead to that if desired.

First stage. This assumption proceeds via an interactive game between a game-coordinator $\mathcal{G}$ and a challenger $\mathcal{A}^*$, and it contains two stages. The first stage is simple: the game-coordinator sends the description of a Diffie-Hellman group $\mathbb{G}$, and the challenger $\mathcal{A}^*$ responds with a set of n distinct exponents $C = \{c_1, c_2, \ldots, c_n\}$.

Section stage. The second stage is much more involved. At the start, the game-coordinator $\mathcal{G}$ secretly samples a uniformly-random generator h, secret exponent k, and m more secret exponents $y_1, y_2, \ldots y_m$. The game-coordinator then sends the challenger the group elements $h, h^k, h^{y_1}, g^{y_2}, \ldots, h^{y_m}$; for notational convenience we write $Y_i = h^{y_i}$. We emphasize that not even the generator h was known to $\mathcal{A}^*$ in the first stage. Next, the challenger can interact with two oracles: SDH and SDDH.

SDH(E, c_i) : This oracle computes selected instances of the Dodis-Yampolskiy PRF [40]. It receives as input a group element E and an exponent $c_i \in C$ contained in the set from the first stage. The oracle outputs $E^{1/(k+c_i)}$.

SDDH(E, F, c_i) : This oracle solves selected instances of the decisional Diffie-Hellman problem. It receives as input two group elements E, F and an exponent $c_i \in C$ contained in the set from the first stage. The oracle outputs a boolean value about whether $F \stackrel{?}{=} E^{1/(k+c_i)}$.

At the end of the second stage, the challenger wins if it outputs an exponent $c^* \in C$ and a set of d pairs (Z_i, j_i) such that:

- $Z_i = Y_{j_i}^{1/(k+c^*)}$ for all $i \in \{1, 2, \ldots, d\}$, and
- d is greater than the number of SDH oracle queries whose second input is c^*. (Hence the name "one more".)

In our proof, we rely on the SDHI assumption to show the following lemma about adversaries who interact with $\Pi_{\text{K-pop}}$.

Lemma 13. *Let* **Adv** *be any real-world adversary for protocol* $\Pi_{\text{K-pop}}$. *At any point during the execution of the protocol, suppose that* **Adv** *has:*

- *not (yet) corrupted the server,*
- *made at most J pOPRF-mode* Eval *queries via* $\Pi_{\text{K-pop}}$.
- *made at most m queries to its H_1 oracle,*

- *and chose at most n values of x_{kal}, combined, in its H_3 queries and the first messages in its $\Pi_{K\text{-}pop}$ evaluations.*

Then, **Adv** *only has negligible probability to make $J+1$ oracle queries $H_2(x_{kal}, x_{priv}, Z)$ such that $Z = H_1(x_{priv})^{1/(k+H_3(x_{kal}))}$, where k denotes the honest server's key.*

Proof. This lemma follows directly from the (m,n) one-more strong gap Diffie-Hellman inversion assumption, and the claim is implied by the analysis in Appendix B, Theorem 4, Claim 4 of Tyagi et al. [81]. We provide a direct proof of the lemma here so that this work is complete and self-contained.

From any $\Pi_{\text{K-pop}}$ adversary **Adv**, we construct an adversary $\mathcal{A}^*$ for the SDHI game as follows.

- $\mathcal{A}^*$ initially samples n values $C = \{c_1, c_2, \ldots, c_n\}$ that it will use as its responses to the H_3 queries made by **Adv**.
- $\mathcal{A}^*$ receives a collection of m group elements $Y_1, Y_2, \ldots, Y_m$ that it will use as its responses to the H_1 queries made by **Adv**.
- Whenever **Adv** wants to run the K-pop protocol and sends a first message (x_{kal}, α), then $\mathcal{A}^*$:
  - queries $H_3(x_{\text{kal}})$, if it has not already done so, and
  - uses its SDH oracle to compute $\beta = \alpha^{1/(k+H_3(x_{\text{kal}}))}$. (After sending this response, **Adv** could query H_2 to finish the $\Pi_{\text{K-pop}}$ protocol, if it wants.)
- Whenever **Adv** makes an H_2 query (whether as part of a $\Pi_{\text{K-pop}}$ execution or not), then $\mathcal{A}^*$ creates this oracle on the fly in the honest way: it chooses responses uniformly at random subject to consistency with previous queries. Also, $\mathcal{A}^*$ inspects each $H_2(x_{\text{kal}}, x_{\text{priv}}, Z)$ query as follows.
  1. $\mathcal{A}^*$ first checks to see if **Adv** had previously made oracle queries $Y = H_1(x_{\text{priv}})$ and $c = H_3(x_{\text{kal}})$. If not, the inspection ends here.
  2. Next, $\mathcal{A}^*$ uses its SDDH oracle to determine whether $Z \stackrel{?}{=} Y^{1/(k+c)}$. If so, $\mathcal{A}^*$ records (Y, Z).

We observe that $\mathcal{A}^*$'s responses to each individual oracle and evaluation query is distributed like a real-world execution, with only a few distinctions. Every H_1 and H_3 query is distributed identically to the real oracles, except that in our experiment H_3 query outputs are always distinct whereas in the real world they can repeat; this leads to a negligible birthday bound difference between this experiment and the real world. Additionally, the SDH oracle ensures perfect emulation of the honest server in each K-pop execution, and the H_2 oracle is emulated perfectly.

By induction using an identical-until-bad argument across all oracle and evaluation queries, it follows that **Adv**'s actions in this experiment are computationally indistinguishable from its actions in the real world. Ergo, if **Adv** has a non-negligible probability of making $J+1$ distinct H_2 oracle queries that result in recording a pair (Y, Z) in case 2 above in the real world, then the same is true in this experiment as well.

Finally, if there is ever a point where $\mathcal{A}^*$ records at least $J+1$ distinct pairs (Y, Z) but has only executed J or fewer $\Pi_{\text{K-pop}}$ instances, then $\mathcal{A}^*$ immediately stops the experiment. For each x_{kal}, it counts:

- how many recorded pairs (Y, Z) involve the corresponding $c = H_3(x_{\text{kal}})$, and
- how many $\Pi_{\text{K-pop}}$ first messages involve this x_{kal}.

By the pigeonhole principle, there must exist at least one c^* such that the former is larger than the latter. Then, $\mathcal{A}^*$ outputs all pairs (Y, Z) corresponding to c^* as a winning output of the SDHI game. Since we have assumed that the SDHI game can only be won with negligible probability, then so too can the above outcome only happen with negligible probability. ☐

E.2 Initialization and random oracle responses

We describe some initial steps undertaken by **Sim**, and the method by which **Sim** responds to oracle queries.

Initialization. Upon initialization, the simulator **Sim** samples the honest server's OPRF key k uniformly at random (i.e., following the honest server procedure) and chooses a group generator g. It also initializes three empty key-value stores:

- $\mathcal{H}$, which will be used to store data about H_1 queries
- $\mathcal{Q}$, which will be used to store data about Eval queries.
- $\mathcal{V}$, which will be used to store data about key ids kid*.

Oracle queries to H_1. For any query $H_1(x_{\text{priv}})$ that **Adv** or **Sim** make to the H_1 oracle, the simulator acts as follows.

- If this is a new value that has not previously been queried, then **Sim** samples a uniformly-random exponent b, programs $H_1(x_{\text{priv}}) := g^b$, and records $\mathcal{H}[x_{\text{priv}}] = b$ in its key-value store $\mathcal{H}$.
- If $H_1(x_{\text{priv}})$ has been previously queried, then the simulator searches its key-value store for $b = \mathcal{H}[x_{\text{priv}}]$ and returns g^b for consistency. We stress that either **Adv** or **Sim** might be the first entity to query H_1 on this particular input x_{priv}; if the other entity later makes the same query, then this procedure ensures a consistent response.

Oracle queries to H_3. Whenever **Adv** or **Sim** make a query $H_3(x_{\text{kal}})$ to the H_3 oracle: if x_{kal} has been queried before then **Sim** returns the same answer as before. Otherwise, if this is a new x_{kal} input value, then **Sim** programs $c := H_3(x_{\text{kal}})$ subject to the constraint that it is different than all previous H_3 oracle responses. Note that **Adv** can only distinguish this behavior from a real-world oracle (in which collisions are permitted) via a birthday bound in n, the number of queries to H_3.

Oracle queries to H_2. This is the most complicated of the three oracles to simulate. Whenever **Adv** or **Sim** makes a query $H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, \gamma)$, then **Sim** must ensure consistency between K-pop evaluations and H_2 queries. **Sim** does so in one of four ways. First, if $H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, \gamma)$ has already been queried before, then **Sim** returns the same response as before.

Second, if $H_1(x_{\mathsf{priv}})$ has not already been queried by **Adv**, then **Sim** does not program the H_2 oracle. In this case, it is computationally infeasible for the adversary **Adv** to find the discrete logarithm v^* that makes $\gamma^{v^*} = H_1(x_{\mathsf{priv}})$, so H_2 is not required to be consistent with any K-pop evaluation.

Third, we check whether the adversary's inputs to a H_2 query are consistent with an honest execution of $\mathcal{F}_{\text{K-pop}}$. If the inputs satisfy the equation $\gamma = H_1(x_{\mathsf{priv}})^{1/(k+H_3(x_{\mathsf{kal}}))}$, then the simulator reprograms the H_2 oracle to produce the K-pop result with client input $(x_{\mathsf{priv}}, x_{\mathsf{kal}})$ and server key k. One potential concern is that this algorithm seems incompatible with our query rate-limiting system: if the adversary's queries to H_2 (which is not rate-limited) potentially require the simulator to query $\mathcal{F}_{\text{K-pop}}$ (which is rate-limited) to determine how to reprogram the H_2 oracle, then **Adv** might exhaust **Sim**'s budget. Fortunately, Lemma 13 resolves this concern; it shows that the number of reprogrammings is less than or equal to the number of executions of $\Pi_{\text{K-pop}}$ in the real world. Hence, **Sim** can make a corresponding number of evaluation queries to $\mathcal{F}_{\text{K-pop}}$ in the ideal world, and stay within the query limit.

Otherwise, we assume that the adversary's inputs are consistent with an execution of $\mathcal{F}_{\text{K-pop}}$ with an *incorrect* key k^*, and **Sim** responds accordingly. That is, we assume that **Adv** has sampled some key k^* such that $\gamma = H_1(x_{\mathsf{priv}})^{1/v^*}$, where $v^* = k^* + H_3(x_{\mathsf{kal}})$. We emphasize that this adversarial behavior can occur even while the server is honest: **Adv** could make an H_2 query, then corrupt the server, then execute $\mathcal{F}_{\text{K-pop}}$ with the key k^* to see if the result matches the previously-made oracle query. **Sim** responds to such queries as follows.

- **Sim** fetches $b = \mathcal{H}[x_{\mathsf{priv}}]$. Recall that $H_1(x_{\mathsf{priv}}) = g^b$.
- **Sim** calculates $w = \gamma^{1/b} = g^{1/v^*}$.
- **Sim** fetches the key id $\mathsf{kid}^* = \mathcal{V}[w]$. If no such mapping exists in the key-value store, then **Sim** samples a new kid^* uniformly at random and stores $\mathcal{V}[w] := \mathsf{kid}^*$.
- Finally, **Sim** runs $(\mathsf{OfflineQuery}, \mathsf{sessionid}, x, \mathsf{kid}^*)$, receives back $(\mathsf{OfflineQuery}, \mathsf{sessionid}, \rho)$, and programs $H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, \gamma) := \rho$.

We postpone the explanation and analysis of this last part of the H_2 oracle until we describe more details about the simulation in the corrupted server setting.

E.3 Simulating Eval queries in pOPRF mode

Next, we describe the behavior of **Sim** to emulate the K-pop in pOPRF mode. We explain **Sim**'s actions during Eval queries in all four settings when the client and server are each honest or corrupted. Since error handling (e.g., when a malicious client or server provides a clearly-malformed input or exceeds the query rate limit) is trivial to perform, we focus in this text on the simulator's response to valid, well-formed messages. We also explain **Sim**'s actions at the moment of client or server corruption to ensure that the adversary **Adv**'s prior view remains consistent with its new information about the client **C**'s inputs or the server **S**'s key.

Eval query with honest parties. As long as the environment **Env** calls the functionality with an honest server and one or more honest clients, the simulator **Sim** emulates the protocol execution of the pOPRF protocol in Figure 4 by choosing x_{kal} correctly and sampling α, β uniformly. Concretely, **Sim** uses this 3-step procedure to respond to a pOPRF query:

1. When **Sim** receives $(\mathsf{Eval}, \mathsf{sessionid}, \mathsf{qid}, x_{\mathsf{kal}})$ from $\mathcal{F}_{\text{K-pop}}$, then **Sim** creates the first message of the pOPRF protocol as follows. It samples a uniformly-random exponent a and records $Q[\mathsf{qid}] = a$ in its key-value store Q. Then, it transmits the message $(x_{\mathsf{kal}}, \alpha = g^a)$ via $\mathcal{F}_{\mathsf{SMT}}$ on behalf of the honest client intended for the server.
2. Once the real-world adversary allows this transmission to proceed to the honest server, then **Sim** calculates $v = k + H_3(x_{\mathsf{kal}})$ and $\beta = \alpha^{1/v}$, and sends the message β to $\mathcal{F}_{\mathsf{SMT}}$ on behalf of the server and destined for the client.
3. Once the real-world adversary allows this network transmission to proceed to the honest client, then **Sim** allows the ideal-world execution to complete. It sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, x_{\mathsf{kal}})$ to $\mathcal{F}_{\text{K-pop}}$. After this, $\mathcal{F}_{\mathsf{OPRF}}$ immediately sends a message of the form $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid})$, and **Sim** responds back with the message $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \perp)$.

We stress that this 3-step process is at the core of our simulation. While the exact steps will change once the client and/or server become corrupted (as described below), we will use this procedure as a starting point in those cases as well.

Analysis. We now argue that the simulation in this 3-step procedure is computationally indistinguishable from the real-world evaluation. Actually this is a simple claim to prove, since $\mathcal{F}_{\mathsf{SMT}}$ only reveals to the adversary **Adv** the *lengths* of x_{kal}, α, and β, and those are clearly of the right size. But looking ahead, let's consider what happens if the adversary *could* see these transmissions in the clear; after all, it will be able to do so as soon as either party is corrupted. Since x_{kal} is correct and α, β are uniformly sampled group elements from the correct group, the adversary **Adv**'s view in this emulation is distributed identically to a real-world execution of $\Pi_{\text{K-pop}}$.

The only way for the adversary **Adv** to distinguish the real world from the emulation is if it somehow queried $H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, \gamma)$ *before* making the corresponding evaluation query. At the time of the H_2 query, **Sim** does not know the output of $\mathcal{F}_{\text{K-pop}}$ and would therefore not program the oracle. Later when the evaluation query is made, the adversary

and environment would detect the difference. However, **Adv** can only make such a query with negligible probability by the computational Diffie-Hellman assumption (which is implied by SDHI). This is because $\langle \alpha, H_1(x_{\mathsf{priv}}), \beta, \gamma \rangle$ forms a Diffie-Hellman tuple, so even if we assume without loss of generality that **Adv** knows the x_{priv} value corresponding to this query and has submitted it to the H_1 oracle, it is still computationally infeasible for **Adv** to find the correct choice of γ that completes the CDH tuple and submit it an H_2 oracle query.

Corruption of a client. Next, let's consider what happens if a client **C** is corrupted while the server remains honest.

At the moment of corruption, the simulator **Sim** receives a list of all records $R = \langle \mathsf{qid}, \mathbf{C}, (x_{\mathsf{kal}}, x_{\mathsf{priv}}) \rangle$ corresponding to pOPRF evaluations that have been started by the client but that are not yet complete. In the emulation, **Adv** also requests corruption of the client, and **Sim** responds as follows.

- For each $x_{\mathsf{priv}} \in R$ that has not already been queried to H_1, the simulator performs the query using its usual procedure: assign $H_1(x_{\mathsf{priv}}) := g^b$ as a random group element whose discrete log is known, and record $\mathcal{H}[x_{\mathsf{priv}}] = b$.
- For each record in R, **Sim** fetches the corresponding $a = Q[\mathsf{qid}]$ and $b = \mathcal{H}[x_{\mathsf{kal}}]$ from its key-value stores, and **Sim** sets $r = a/b$. As a result, $H_1(x_{\mathsf{priv}})^r = (g^b)^r = \alpha$.
- **Sim**, in its role as the environment in the emulation, responds to **Adv**'s corruption request by sending the list of exponents r; this is the only state that a real client maintains between the two messages of the pOPRF protocol.

Analysis. This emulation of the real world is perfect: every r is sampled uniformly at random since that was also true of a. Note that we assume secure erasures of client randomness for any pOPRF executions that have already completed; this assumption could be removed, but then $\mathcal{F}_{\mathsf{K\text{-}pop}}$ must reveal the entire history of client inputs to **Sim** at the time of corruption. Additionally, the H_2 oracle is designed so that if **Adv** makes a subsequent query to $H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, \gamma = \beta^{1/r})$ now that it knows r, then the response of the H_2 oracle will agree with the pOPRF evaluation.

Also, we claim that it makes no difference whether the server **S** has already been corrupted before the moment of client corruption; **Sim** can follow procedure above either way. This claim follows from three facts:

- We only emulate evaluations that are still in progress at the time of client corruption.
- $\Pi_{\mathsf{K\text{-}pop}}$ is a two-round protocol, so the corrupted server has not had an opportunity to send any messages.
- The client produces its first message in a manner that is independent of the server's state (i.e., its key k).

Ergo, even if **Adv** knows the server key k, then **Sim**'s emulation still perfectly matches the real world.

Eval query with corrupted client and honest server. For any new pOPRF invocations by a corrupted client, there are only three minor differences of note relative to the 3-step process in the honest-client setting. First, all secure message transmissions are readable to **Sim**. Fortunately, we have already shown that the simulation is perfect even in this case.

Second, the adversary **Adv** now has the power to choose the client's inputs and first message $(x_{\mathsf{kal}}, \alpha)$ of any invocation. As a result, **Sim** no longer needs to produce this message itself, so it skips step 1 and goes directly to steps 2-3 of the 3-step procedure from the honest client setting. Concretely, **Sim** calculates the server's response $\beta = \alpha^{1/v}$. Once the adversary **Adv** lets this network communication proceed, then **Sim** sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, x_{\mathsf{kal}})$ to $\mathcal{F}_{\mathsf{K\text{-}pop}}$ and then $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \perp)$ to $\mathcal{F}_{\mathsf{OPRF}}$.

Third, the adversary **Adv** now has full control to choose the output of the corrupted client—that is, independent of whether the adversarial client performs the protocol honestly, **Adv** can simply change the output to the environment to be whatever it wants. In our emulation, **Adv**'s output message actually goes to the simulator **Sim**. Then, **Sim** passes this output along to the real environment **Env**. (We remark that this is a standard operation in UC security analyses, and it occurs independently of **Sim**'s interactions with the functionality $\mathcal{F}_{\mathsf{K\text{-}pop}}$.)

Corruption of the server. Next, we consider what happens at the moment that the server **S** is corrupted. Since there are multiple clients, we must consider the possibility that some clients are still honest (at least for now) whereas other clients have already been corrupted. The simulator strategy described below ensures consistency with real-world views of all prior and future interactions with (honest and corrupted) clients.

At the moment of corruption, the simulator **Sim** performs two actions: one with $\mathcal{F}_{\mathsf{K\text{-}pop}}$ and one with **Adv**.

- **Sim** receives from $\mathcal{F}_{\mathsf{K\text{-}pop}}$ the honest key identifier kid. In response, it stores a mapping $\mathcal{V}[k] := \mathsf{kid}$ from the OPRF key to its corresponding key id in $\mathcal{V}$.
- **Sim** (acting as though it is the environment within **Adv**'s emulation of the real world) provides **Adv** with the OPRF key k that it has been using to act as the honest server up to this point; this is the only state that the real-world server maintains between invocations of the pOPRF protocol.

Analysis. Because **Sim** sampled and used the key k honestly in all (completed and in-progress) invocations up to this point, the emulation of this step is perfect: the environment has no way (yet) to distinguish it from the real world.

For any subsequent pOPRF mode evaluations involving a corrupted server, the biggest challenge is that now the adversary **Adv** can deviate from the protocol and use an incorrect key k^*—or can even choose β in a way that does not clearly correspond to any key—and the simulator must determine

how to choose its key id kid^* to use in its $\mathsf{EvalContinue}$ message. The simulator's response depends on whether the client is honest or corrupted. We describe these actions next.

Eval query with honest client and corrupted server. In this case, **Sim** begins the query by acting as the honest client in step 1 of our 3-step procedure. That is, it samples a uniformly-random exponent a, records $Q[\mathsf{qid}] = a$ in its key-value store Q, and transmits the message $(x_{\mathsf{kal}}, \alpha = g^a)$ via $\mathcal{F}_{\mathsf{SMT}}$ on behalf of the honest client intended for the server.

Since the server is corrupted, it is no longer the simulator's job to perform step 2 of our 3-step procedure. Instead, **Sim** waits for **Adv** to send a response β from the corrupted server to the honest client. We emphasize here that the adversary does not need to use the honest key k.

Finally, the simulator sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{K\text{-}pop}}$, and all that remains is for **Sim** to determine the key identity kid^* to send in its $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \mathsf{kid}^*)$ message to $\mathcal{F}_{\mathsf{OPRF}}$ so that the protocol instance can complete. We remark that the adversarial server can choose to deviate from the protocol by using an incorrect key k^* or kaleidoscopic input x^*_{kal}, but since these terms are added together, for the purposes of the simulation it suffices to assume that a dishonest adversary always mauls the key since subsequent calls to the H_2 oracle must only be consistent with the honest client's input x_{kal}.

Unfortunately, **Sim** cannot directly extract the key k^* that **Adv** used. Nevertheless, we make the following observations:

- There must be some field element v^* such that $\beta = \alpha^{1/v^*}$. (If **Adv** acted honestly then this field element would be $v := k + H_3(x_{\mathsf{kal}})$ with the honest key and input, but otherwise v^* could be something else.)
- Our K-pop evaluation only depends on v^*, not on k directly. For instance, if the adversary deviated from the protocol by subtracting 1 from k and adding 1 to $H_3(x_{\mathsf{kal}})$, it would not change the output of the K-pop.
- Even though **Sim** cannot calculate v^*, it can calculate $g^{v/v^*} = \beta^{(k+H_3(x_{\mathsf{kal}}))/a}$ since it knows the honest key k, kaleidoscopic input x_{kal}, and exponent a. Also, this group element uniquely identifies v^* (modulo the group order, which is all that matters in the K-pop construction).

Ergo, **Sim** can calculate the group element $w := g^{v/v^*}$ and use it to select a key identifier as follows:

1. If $w = g$, i.e., if **Adv** used the honest key k and input x_{kal}, then **Sim** fetches the honest key identifier $\mathsf{kid} \leftarrow \mathcal{V}[k]$.
2. Otherwise, if w is already in the key-value store, then **Sim** fetches the corresponding identifier $\mathsf{kid}^* \leftarrow \mathcal{V}[w]$.
3. Otherwise, **Sim** randomly generates a new, distinct kid^* and records it in the key-value store $\mathcal{V}[w] := \mathsf{kid}^*$.

In the first case, **Sim** sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \mathsf{kid})$ to $\mathcal{F}_{\mathsf{OPRF}}$ containing the honest key identifier. In the other two cases, **Sim** sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \mathsf{kid})$ containing a dishonest key identifier kid^* instead.

Analysis. We now analyze why this strategy provides perfect emulation of the real world. The first case ensures that if the adversarial server acted honestly, then the responses will be consistent with any other queries using the real key (even ones produced prior to server corruption). The remaining two cases ensure that $\mathcal{F}_{\mathsf{K\text{-}pop}}$ performs the same table lookup $T(\mathsf{kid}^*, -)$ for any two queries such that α and β are related by the same v^*; in particular, if the client makes the same query twice and the server exponentiates both α's by the same exponent, then the response will be consistent between them.

The only wrinkle here is that our simulation strategy happens to produce different kid^* if a corrupted server uses the same, incorrect k^* with two different x_{kal} values; fortunately, this is allowable and produces the same output distribution. That is: it is inconsequential whether one builds a new PRF table $T(\mathsf{kid}^*, -)$ in either of the following two scenarios:

- Each key id kid^* corresponds to a key k^* chosen by the adversary in the real world, and the inputs to this table are tuples of the form $(x_{\mathsf{kal}}, x_{\mathsf{priv}})$.
- Each key id kid^* corresponds to a (k^*, x_{kal}) tuple, and the inputs to this table are tuples of the form $(x_{\mathsf{kal}}, x_{\mathsf{priv}})$.

In both cases, all inputs to the ideal functionality result in uniformly-distributed outputs, with the consistency rule that if two executions of the protocol have the same server-provided k^* and x_{kal} and client-provided x_{priv} then the results are identical. Our simulator opts for the latter approach.

Analysis of H_2 oracle queries. Returning back to the discussion of H_2 oracle queries from Section E.2, we now reason about the correctness of our H_2 reprogramming strategy in the fourth and final case where the adversary's inputs are consistent with an execution of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ with an *incorrect* key k^*. As a reminder: for all $H_2(x_{\mathsf{kal}}, x_{\mathsf{priv}}, \gamma)$ queries in which the inputs are not related by the honest OPRF key, they are treated as if they are connected via a different key $\gamma = H_1(x_{\mathsf{priv}})^{1/v^*}$ (where $v^* = k^* + H_3(x_{\mathsf{kal}})$) chosen by the adversary, and **Sim** responds accordingly by querying $\mathcal{F}_{\mathsf{K\text{-}pop}}$ with $\mathsf{kid}^* = \mathcal{V}[w]$.

By construction, it is clear that this strategy results in the same output for the H_2 oracle and the corresponding K-pop protocol in which the malicious server uses key k^*: in both cases, v^* is uniquely determined from the group element $w = g^{1/v^*}$, and the key identity kid^* is specified as a function of w.

There is one notable difference between the simulator's actions to calculate H_2 and the corrupted server evaluation: in the former case, the simulator runs $\mathsf{OfflineQuery}$ to discover the K-pop result, whereas now there is an interactive evaluation. Fortunately, the code of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ ensures that the result is the same in both cases. Additionally, the query rate limit only applies to honest server execution; a malicious server is not

subject to the query limit so it is now acceptable for **Sim** to run as many K-pop evaluations as **Env** and **Adv** specify.

Eval query with corrupted client and server. Finally, we describe the setting in which the adversary **Adv** controls both the client and server. In this case, **Sim** does not have to construct any network transmissions; instead, it is **Adv**'s job to produce both messages $(x_{\mathsf{kal}}, \alpha)$ and β that are sent via $\mathcal{F}_{\mathsf{SMT}}$. Moreover, the adversary **Adv** produces the output of the corrupted client, which it sends to the simulator (who is acting as the environment in this emulation). In response, **Sim** provides the same output to the actual environment **Env**.

Analysis. Correctness of just this one evaluation query on its own is trivial, as are most UC security analyses when all parties are corrupted. More importantly: the adversary's view here is consistent with all prior invocations of H_2, or of Eval when one or both parties was honest. This concludes the analysis of the K-pop in pOPRF mode.

E.4 Simulating Eval queries in OPRF mode

Next, we consider uses of the $\mathcal{F}_{\mathsf{K\text{-}pop}}$ functionality in OPRF mode. While we analyze the two modes separately for clarity, we emphasize that the same K-pop is being used in both modes concurrently with the same key and potentially even the same client inputs. In other words, responses to evaluation and oracle queries must be consistent between the two modes, corruption of a client or server is a single event that impacts both modes at the same time, and the reactions of **Sim** must be consistent between the modes. We discuss this further in the analysis below. As before: we consider what happens in the honest setting and what happens during and after corruptions.

In this work, we conduct our analysis in the setting that the first round of the OPRF protocol in Figure 5 (in which the server homomorphically encrypts its OPRF key) runs in a pre-processing step so that the online execution requires only two rounds. We also adopt the simplifying technique from §3.3 that the plaintext space of HomEnc is a group of the same order as G so that the secondary blinding factor t is not needed; we stress that this simplification is not essential for the security analysis, and the ideas below can be generalized.

Additionally, in the text below we focus on differences between the simulator **Sim**'s actions in the OPRF and pOPRF modes. The OPRF evaluation in Figure 5 introduces several features that pOPRF mode lacks, such as a new crypto primitive (additively homomorphic encryption) and additional variables that parties hold (s for the client and sk, pk, and z for the server). Furthermore, the (honest) server's response β is produced in a different way: it is now $\beta = \alpha^{1/z}$ rather than $\beta = \alpha^{1/v}$. Our analysis below must therefore change to account for these differences.

Initialization. To support OPRF queries with an honest server, we make only two additions to the initialization procedure from §E.2. First, we initialize one more key-value store $\mathcal{Z}$ (in addition to $\mathcal{H}$, $\mathcal{Q}$, and $\mathcal{V}$ described above). Second, in addition to sampling an OPRF key k, the simulator **Sim** must also run the homomorphic encryption scheme's key generation routine $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathrm{HomKeyGen}()$ and then encrypt the OPRF key homomorphically as $[\![k]\!] = \mathrm{HomEnc}_{\mathsf{pk}}(k)$. This action exactly matches the server's actions in the real world (cf. Figure 5).

As stated above, we presume that the simulated server then sends this ciphertext $[\![k]\!]$ to all parties (including **Adv**) in a preprocessing step. We rely on the IND-CPA property of the encryption scheme to protect the value k; that is, for any adversary **Adv** who has not yet corrupted the server, these ciphertexts look computationally indistinguishable from ciphertexts of the message 0, and hence **Adv** does not learn any information about k. We also rely upon the fact that the honest server performs this pre-computation step before it can be compromised; this assumption can be relaxed, but then we would need some method for the simulator to extract the homomorphic encryption scheme's secret key (e.g., a ZK proof of knowledge only during pre-processing).

Additionally, we use the same procedures for **Sim** to respond to random oracle queries as described in §E.2. In fact this is required, since the random oracle queries are independent of evaluation queries, and thus independent of evaluation type (pOPRF or OPRF). In other words, we must ensure that our procedures below for the simulator to respond to OPRF queries is consistent with the same random oracle procedures and pOPRF evaluation responses described in §E.2 and E.3, respectively. In the remainder of this section, we describe the operation of **Sim** during OPRF evaluation queries and at the moment that either a client or server is corrupted.

Eval query with honest parties. We begin with the scenario in which both the client and server are honest at the start of an OPRF Eval query. In response to a query in OPRF mode, **Sim** performs a similar 3-step procedure as it did in pOPRF mode. At a high level, the main changes are that: **Sim** no longer knows x_{kal}, it needs to produce a ciphertext $[\![z]\!]$ in the first message, and there is a different procedure for generating β for the second message. Also, as a procedural matter, **Sim** can distinguish OPRF and pOPRF evaluations because in the OPRF case, its first message comes from $\mathcal{F}_{\mathsf{OPRF}}$ rather than $\mathcal{F}_{\mathsf{K\text{-}pop}}$ (see Figs. 2-3). In more detail:

1. When **Sim** receives $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid})$ from $\mathcal{F}_{\mathsf{OPRF}}$, then **Sim** creates the first message of the OPRF protocol as follows. It samples a uniformly-random exponent a and records $\mathcal{Q}[\mathsf{qid}] = a$ in its key-value store $\mathcal{Q}$. It also samples a uniformly-random exponent z and records $\mathcal{Z}[\mathsf{qid}] = z$. Then, it transmits the message $(\alpha = g^a, [\![z]\!])$ via $\mathcal{F}_{\mathsf{SMT}}$ on behalf of the honest client to the server.

2. Once the real-world adversary allows this transmission to proceed to the honest server, then **Sim** sends $\beta = \alpha^{1/z}$ to $\mathcal{F}_{\mathsf{SMT}}$ on behalf of the server intended for the client.
3. Once the real-world adversary allows this network transmission to proceed to the honest client, then **Sim** sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \bot)$ to $\mathcal{F}_{\mathsf{OPRF}}$ to allow the ideal-world execution of the Eval query to complete.

This 3-step process is at the core of our simulation of Eval queries in OPRF mode; our analysis in other cases will build from this starting point.

Analysis. As before, this emulation is computationally indistinguishable from the real-world protocol, up a negligible probability that the adversary queries the H_2 oracle beforehand that we discussed in the pOPRF setting. Once again, even if the adversary could see the contents of all network transmissions and decrypt the ciphertext: we see that z, α, and β have the distribution of uniformly-random exponents and group elements, respectively.

Corruption of the server. The simulator's actions here are very similar to our prior description in the pOPRF setting. First, **Sim** receives from $\mathcal{F}_{\mathsf{K\text{-}pop}}$ the honest key identifier kid that it stores in the key-value store $\mathcal{V}$. Then, **Sim** provides **Adv** with the OPRF key k. The only new task is that **Sim** must also provide **Adv** with the encryption scheme's secret key sk, since that is now part of the server's state as well.

Analysis. Because **Sim** has sampled and used both the OPRF and encryption keys honestly in all (completed and in-progress) evaluations up to this point, the emulation of this step is perfect: the environment has no way (yet) to distinguish it from the real world.

Corruption of a client. Next, we consider what happens if a client **C** is corrupted and the simulator **Sim** receives a list of all records $R = \langle \mathsf{qid}, \mathbf{C}, (x_{\mathsf{kal}}, x_{\mathsf{priv}}) \rangle$ corresponding to OPRF evaluations that have been started by the client but are not yet complete. Since α is produced in the same way in pOPRF and OPRF modes, the simulator **Sim** uses the same strategy as before to calculate an exponent r that is consistent with the existing network communication. (Recall that this involves fetching $H_1(x_{\mathsf{priv}}) = g^b$ and $\alpha = g^a$, and then setting $r = a/b$.)

Additionally, in OPRF mode the simulator must also produce the exponent s and send that to **Adv**, since that is now also part of the client's state. Recall that in the OPRF protocol, s is sampled uniformly at random and then z is derived from it via the equation $z = s(k + H_3(x_{\mathsf{kal}}))$. Our simulation proceeds in the opposite direction: it fetches $z = \mathcal{Z}[\mathsf{qid}]$ and sets $s = z \cdot (k + H_3(x_{\mathsf{kal}}))^{-1}$.

Analysis. Once again, this emulation of the real world is perfect: every s has the uniform distribution since this is also true of z, and r is sampled uniformly at random since that was also true of a. Once again, we assume secure erasures so that **Sim** only needs to emulate in-progress OPRF evaluations.

Eval query with corrupted client and honest server. If the client is corrupted, then **Adv** has the power to choose the client's inputs and first message $(\alpha, [\![z]\!])$ of the evaluation. In response, **Sim** decrypts $z = \mathrm{HomDec}_{\mathsf{sk}}([\![z]\!])$ and then performs steps 2-3 of the 3-step procedure. That is, **Sim** calculates and sends $\beta^{1/z}$ to the client via $\mathcal{F}_{\mathsf{SMT}}$; once the adversary allows this network transmission to reach the server, **Sim** sends $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \bot)$ to $\mathcal{F}_{\mathsf{OPRF}}$ to complete the evaluation. This action perfectly emulates the honest server's actions in the real-world protocol, whether or not z was honestly chosen by the corrupted client. Finally, **Adv** chooses the output of the corrupted client in the emulation, which **Sim** then forwards along to the environment **Env**.

Eval query with honest client and corrupted server. This case proceeds analogously as above. This time, **Sim** runs step 1 of the 3-step procedure and transmits $(\alpha, [\![z]\!])$ via $\mathcal{F}_{\mathsf{SMT}}$. Then, **Sim** waits for **Adv** (on behalf of the corrupted server) to send a response β—which need not depend on the honest key k, or indeed on any key at all. Once **Adv** allows this response message to reach the client, then **Sim** sends $\mathcal{F}_{\mathsf{OPRF}}$ the message $(\mathsf{EvalContinue}, \mathsf{sessionid}, \mathsf{qid}, \mathsf{kid}^*)$ to complete the evaluation. As before, the main challenge is to determine the key identifier kid^* to transmit.

First, **Sim** checks whether **Adv** has acted honestly. To do so, **Sim** fetches $z = \mathcal{Z}[\mathsf{qid}]$ and tests whether $\beta \stackrel{?}{=} \alpha^{1/z}$. If this is the case, then **Sim** fetches the honest key identifier $\mathsf{kid} = \mathcal{V}[k]$ and uses it in its final message to $\mathcal{F}_{\mathsf{OPRF}}$.

Otherwise, without loss of generality, we presume that **Adv** knows x_{kal} (e.g., because it was told by the environment) and therefore can calculate s such that $z = s(k + H_3(x_{\mathsf{kal}}))$ and can maul this value to a different $z^* = s(k^* + H_3(x_{\mathsf{kal}}))$. We remark that (just as with pOPRF mode) a malicious adversary could maul any of s, k, or x_{kal}, yet it suffices for the simulator **Sim** to consider only a tampering of k. This is because it suffices for the simulation to be consistent with the honest client's actions after it unblinds with the honest s and makes an H_2 oracle query with the honest x_{kal}.

The challenge here is that **Sim**'s actions must be consistent with this mauling $z^* = s(k^* + H_3(x_{\mathsf{kal}}))$, even though **Sim** does not know the client's input x_{kal} or the blinding factor s (these would only be revealed later upon client compromise). Fortunately, the quotient $\frac{z}{z^*} = \frac{k + H_3(x_{\mathsf{kal}})}{k^* + H_3(x_{\mathsf{kal}})}$ no longer depends on the blinding factor s. Moreover, this quotient corresponds to the same quotient v/v^* that we calculated in the exponent in the pOPRF setting.

Ergo, **Sim** can calculate the group element $w := g^{z/z^*} = g^{v/v^*}$ and then follow exactly the same process as in the pOPRF setting to select a key identifier.

1. If $w = g$, i.e., if **Adv** used the honest key k and input x_{kal}, then **Sim** fetches the honest key identifier $\mathsf{kid} \leftarrow \mathcal{V}[k]$.
2. Otherwise, if w is already in the key-value store, then **Sim** fetches the corresponding identifier $\mathsf{kid}^* \leftarrow \mathcal{V}[w]$.
3. Otherwise, **Sim** randomly generates a new, distinct kid^* and records it in the key-value store $\mathcal{V}[w] := \mathsf{kid}^*$.

Analysis. This strategy results in a perfect emulation of the real world: the first case ensures that if **Adv** acted honestly then the evaluation response is consistent with the real key, and the remaining cases ensure that if **Adv** makes two queries with the same mauled key k^* then **Sim** queries the ideal PRF table $T(\mathsf{kid}^*, -)$ with the same key identifier. Moreover, this strategy is consistent with **Sim**'s actions during pOPRF evaluations in terms of how the key identifier kid^* is selected as a function of the tuple (k^*, x_{kal}).

Eval query with corrupted client and server. Finally, if **Adv** has corrupted both parties, then it produces both messages $(\alpha, [\![z]\!])$ and β that are sent via $\mathcal{F}_{\mathsf{SMT}}$. Additionally, **Adv** produces the corrupted client's output, which **Sim** forwards along to **Env**.

F UC Security Analysis of Account Recovery

In this section, we provide a UC-secure analysis of the account recovery protocol Π_{acc} from Figures 6-8 in the UC security framework of Canetti [22] (and we refer readers to §D for an overview of this framework). Note that even this protocol is a hybrid, as it contains idealized subroutines like $\mathcal{F}_{\mathsf{K\text{-}pop}}$. For this reason: even though our security reduction is perfect in the theorem that follows, we emphasize that an instantiation of the K-pop itself does require computational assumptions. We provide the corresponding ideal functionality $\mathcal{F}_{\mathsf{acc}}$, and we claim the following.

Theorem 6. *Protocol Π_{acc} (perfectly) UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{acc}}$ in the presence of two global functionalities: the programmable random oracle $\mathcal{F}_{\mathsf{pro}}$, and $\mathcal{F}_{\mathsf{SMT}}$ as our modeling of email delivery.*

Proof. In order to prove Theorem 6, we construct a simulator **Sim** and argue that whether the environment **Env** interacts with Π_{acc} and in the $\mathcal{F}_{\mathsf{K\text{-}pop}}$-hybrid world or $\mathcal{F}_{\mathsf{acc}}$ and **Sim** in the ideal world, its view is identically distributed. The detailed description of **Sim** is captured in Figures 19- 20.

F.1 Description of $\mathcal{F}_{\mathsf{acc}}$

In this section, we provide an overview of the actions of $\mathcal{F}_{\mathsf{acc}}$ and its interactions with **Sim** (See Figure 17 for a more detailed description).

(init): $\mathcal{F}_{\mathsf{acc}}$ receives an initialization request from a server $\mathbf{S}_{\mathsf{sid}}$ and passes the message to **Sim**.

(create_account): $\mathcal{F}_{\mathsf{acc}}$ receives a new account creation request from **C** and assigns a contact identifier to the creation of an account using (E,x) by using contact_ctr and incrementing contact_ctr. $\mathcal{F}_{\mathsf{acc}}$ then sends an initial message to **Sim**. If **C** has been corrupted, $\mathcal{F}_{\mathsf{acc}}$ shares **C**'s account creation data (Q,E,e,x,A,k_u) along with its account tag stored in $\mathsf{DX_contact}[(E,x)]$, with **Sim** in the message $(\mathsf{create_account}, \mathbf{C}_i, (\mathsf{DX_contact}[(E,x)], Q, E, e, x, A, k_u))$. Otherwise, $\mathcal{F}_{\mathsf{acc}}$ only shares the account tag with **Sim** in the message $(\mathsf{create_account}, \mathbf{C}_i, \mathsf{DX_contact}[(E,x)])$. After the initial message is sent, if $\mathcal{F}_{\mathsf{acc}}$ receives a response $(\mathsf{create_account}, \mathsf{error})$, it indicates that a prior nonce was used again. The functionality $\mathcal{F}_{\mathsf{acc}}$ then notifies the client $\mathbf{C}_i$ by returning error and exiting. Otherwise, $\mathcal{F}_{\mathsf{acc}}$ stores the account creation data in its recovery dictionary $\mathsf{DX}[(E,x)] := (Q,e,A,k_u)$. The functionality returns $\perp$ to $\mathbf{C}_i$.

(recover_account): $\mathcal{F}_{\mathsf{acc}}$ receives an account recovery request from **C** for a potential account associated with (E,x). $\mathcal{F}_{\mathsf{acc}}$ begins by checking if it has previously received (E,x) either through account creation or a prior account recovery attempt. If $\mathcal{F}_{\mathsf{acc}}$ has not previously received (E,x), $\mathcal{F}_{\mathsf{acc}}$ assigns (E,x) a contact identifier by storing $\mathsf{DX_contact}[(E,x)] :=$ contact_ctr and incrementing contact_ctr.

$\mathcal{F}_{\mathsf{acc}}$ then sends an initial message to **Sim**. If the client has been corrupted, $\mathcal{F}_{\mathsf{acc}}$ sends (E,x) along with the contact identifier, stored as $\mathsf{DX_contact}[(E,x)]$, in the message $(\mathsf{recover_account}, \mathbf{C}_i, (\mathsf{DX_contact}[(E,x)], E, x))$ to **Sim**. Otherwise, $\mathcal{F}_{\mathsf{acc}}$ sends the contact identifier in the message $(\mathsf{recover_account}, \mathbf{C}_i, \mathsf{DX_contact}[(E,x)])$.

After the initial message is sent, if $\mathcal{F}_{\mathsf{acc}}$ receives a response $(\mathsf{recover_account}, \mathsf{error})$, it indicates that at least one of the servers attempted to cheat by using a different key than it had previously in a $\mathcal{F}_{\mathsf{K\text{-}pop}}$ evaluation on the same value (E,x). In this case, $\mathcal{F}_{\mathsf{acc}}$ returns error to $\mathbf{C}_i$ and exits.

Otherwise, $\mathcal{F}_{\mathsf{acc}}$ checks if an account is associated with (E,x) by checking if (E,x) is a label within its account recovery dictionary DX. If so, $\mathcal{F}_{\mathsf{acc}}$ retrieves the previously stored account information (Q,e,A,k_u) from $\mathsf{DX}[(E,x)]$ and sends the security questions Q and recovery email e along with the contact identifier to **Sim** in the message $(\mathsf{recover_account}, \mathbf{C}_i, (\mathsf{DX_contact}[(E,x)]), Q, e)$. $\mathcal{F}_{\mathsf{acc}}$ also sends (Q,e) to each honest server. If (E,x) does not correspond to an existing account, $\mathcal{F}_{\mathsf{acc}}$ sends $\perp$ to each honest server. Regardless of whether an account exists, $\mathcal{F}_{\mathsf{acc}}$ returns $\perp$ to $\mathbf{C}_i$.

(restore_account): $\mathcal{F}_{\mathsf{acc}}$ receives an account restoration request from **C** for the account associated with (E,x) using the security answers A and nonce m. $\mathcal{F}_{\mathsf{acc}}$ sends an

initial message to **Sim**. If $\mathbf{C}_i$ has been corrupted, $\mathcal{F}_{\mathsf{acc}}$ sends the security answers and nonce (A,m) along with the contact identifier contact_identifier in the message $(\mathsf{restore_account}, \mathbf{C}_i, (\mathsf{DX_contact}[(E,x)], A, m))$ to **Sim**. Otherwise, $\mathcal{F}_{\mathsf{acc}}$ sends the message $(\mathsf{restore_account}, \mathbf{C}_i, \perp)$ to **Sim**.

After the initial message is sent, if $\mathcal{F}_{\mathsf{acc}}$ receives a response $(\mathsf{restore_account}, \mathsf{error})$, it indicates that at least one of the servers attempted to cheat by using a different key than it had previously in a $\mathcal{F}_{\mathsf{K\text{-}pop}}$ evaluation on the same value (A,m). In this case, $\mathcal{F}_{\mathsf{acc}}$ returns error to $\mathbf{C}_i$ and exits.

Otherwise, $\mathcal{F}_{\mathsf{acc}}$ checks if an account associated with (E,x) exists by checking if (E,x) is a label within its account recovery dictionary DX. If so, $\mathcal{F}_{\mathsf{acc}}$ retrieves the previously stored account information (Q,e,A',k_u) from $\mathsf{DX}[(E,x)]$. $\mathcal{F}_{\mathsf{acc}}$ then uses the previously stored security answers from account creation A' to check if the security answers A it received from the client $\mathbf{C}_i$ matches. If so, account restoration succeeded. $\mathcal{F}_{\mathsf{acc}}$ then shares the user key k_u with **Sim** through message $(\mathsf{restore_account}, k_u)$ if the client has been corrupted and directly with the client $\mathbf{C}_i$ if it is honest by returning k_u. In the case that the client is honest, $\mathcal{F}_{\mathsf{acc}}$ also returns $\perp$ to both servers. In the event that there is no account associated with (E,x), $\mathcal{F}_{\mathsf{acc}}$ returns $\perp$ to the client $\mathbf{C}_i$ and both servers.

(offline_recover): $\mathcal{F}_{\mathsf{acc}}$ receives an offline recover message from **Sim** after the adversary **Adv** sends an offline_attack message to **Sim**. In offline recovery, **Sim** provides (E,x) to $\mathcal{F}_{\mathsf{acc}}$ to check if there is an associated account. $\mathcal{F}_{\mathsf{acc}}$ checks is the label (E,x) exists in its dictionary DX. If it does, $\mathcal{F}_{\mathsf{acc}}$ sends back $(\mathsf{offline_recover}, (\mathsf{DX_contact}[(E,x)], Q, e))$ to **Sim**.

(offline_restore): $\mathcal{F}_{\mathsf{acc}}$ receives an offline restoration message from **Sim** as a continuation of **Adv**'s offline attack. This only occurs if **Adv** successfully recovered security questions Q and a recovery email e in offline recovery. $\mathcal{F}_{\mathsf{acc}}$ now checks if the provided security answers A matches previously stored security answers A'. If s0, $\mathcal{F}_{\mathsf{acc}}$ returns the user key in the message $(\mathsf{offline_restore}, k_u)$ to **Sim**.

(corrupt): Upon receiving a message from the environment **Env** that a party has been corrupted, $\mathcal{F}_{\mathsf{acc}}$ simply forwards this message to **Sim**.

F.2 Description of Sim

For a full description of **Sim**, see Figures 19 to 21. **Sim** stores the dictionaries $\mathsf{DX}_{\mathbf{Sim}}, \mathsf{DX}_*$, and a list of corrupted clients c.

- $\mathsf{DX}_{\mathbf{Sim}}$ maps a contact identifier contact_identifier to the tuple $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$
- DX_* maps a contact identifier contact_identifier to the tuple (id, k_E), where (id, k_E) are simulated values created by **Sim**

When **Sim** receives an $(\mathsf{init}, \mathsf{sid}))$ from $\mathcal{F}_{\mathsf{acc}}$, it emulates both $\mathcal{F}_{\mathsf{K\text{-}pop}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ (conducting the initialization steps exactly as described by both functionalities). Therefore **Env**'s view is unaffected.

When **Sim** receives the message $(\mathsf{corrupt}, \mathsf{cid})$ from **Env**, it tracks this corruption in its list of corruptions c and sends the message $(\mathsf{corrupt}, \mathsf{cid})$ to its own internal emulated instances of $\mathcal{F}_{\mathsf{K\text{-}pop}}$. This is consistent with the hybrid world in which **Env** sends the message $(\mathsf{corrupt}, \mathbf{C}_i)$ to both instances of $\mathcal{F}_{\mathsf{K\text{-}pop}}$.

Similarly, when **Sim** receives the message $(\mathsf{corrupt}, \mathbf{S}_{\mathsf{sid}})$ from **Env**, it forwards the message to its emulated instance of $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$. In the hybrid world, **Env** sends the message $(\mathsf{corrupt}, \mathbf{S}_{\mathsf{sid}})$ to $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$.

We now describe how **Sim**'s behavior ensures that **Env**'s view in the $\mathcal{F}_{\mathsf{K\text{-}pop}}$-hybrid world is identically distributed with its view in the ideal world. For any request to interact with $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ where sid is some server id, **Sim** receives inputs (when client is corrupted) or chooses inputs (when client is honest). It then simulates the interactions between $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ and corrupted parties as described in Figure 22.

We first discuss the case in which the client **C** is corrupted (and either server or both servers may be corrupted). We then analyze the case in which the client **C** is honest but either or both of the servers are corrupted. We end on an analysis of the case in which both servers are corrupted and conduct an offline attack to restore an account.

Case 1: corrupted client and corrupted server(s). *Account Creation.* When **Sim** receives a message $(\mathsf{create_account}, \mathsf{cid}, (Q, E, e, A, k_u))$, which indicates that a new account creation request, **Sim** first checks if a nonce n needs to be simulated, depending on whether $\mathbf{S}_1$ is corrupted. If $\mathbf{S}_1$ is corrupted, **Sim** receives a nonce from the corrupted server. In the event that $\mathbf{S}_2$ is honest, **Sim** checks if the corrupted server, $\mathbf{S}_1$ is cheating by repeating a nonce. If so, it sends an error message to $\mathcal{F}_{\mathsf{acc}}$, which then exits and returns error to **C**. This is consistent with **Env**'s view in the hybrid world. If $\mathbf{S}_1$ selects a previously used nonce, $\mathbf{S}_2$ sends back error to the client and the client exits. In both the hybrid and ideal world, if the $\mathbf{S}_2$ is corrupted, it will neglect the step of catching repeated nonces.

For each server, **Sim** emulates each instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ as described upon receiving the message $(\mathsf{Eval}, \mathsf{sessionid}, \texttt{pOPRF-mode}, \mathsf{qid}, E, x)$. If an error (such as rate-limiting) is encountered in the hybrid world during this process, $\mathcal{F}_{\mathsf{K\text{-}pop}}$ will send an error to $\mathcal{A}^*$. In the hybrid world, the same error is sent through **Sim**'s emulation of $\mathcal{F}_{\mathsf{K\text{-}pop}}$. In addition, an error message $(\mathcal{F}_{\mathsf{K\text{-}pop}}, \mathsf{error})$ is sent to $\mathcal{F}_{\mathsf{acc}}$, which returns error to the client $\mathbf{C}_i$ and exits. Otherwise, if an

$\mathcal{F}_{\mathsf{K\text{-}pop}}$ evaluation is successful, **Sim** returns a value $\hat{E}$ through its emulation of $\mathcal{F}_{\mathsf{K\text{-}pop}}$. Since this value is generated through directly emulation $\mathcal{F}_{\mathsf{K\text{-}pop}}$ as it is described in Figure 3, it is distributed identically to a value $\hat{E}$ that is returned directly through an instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$. The environment **Env**'s view is not affected. The corrupted client $\mathbf{C}_i$ then selects the values $\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u$. If the client chooses to follow the protocol, id, m are sampled uniformly at random and $\mathsf{ct}_r, \mathsf{ct}_u$ are ciphertexts created as described in Π_{acc}. Otherwise, the corrupted client may select any value of their choosing. Upon receiving the message $(\mathsf{create_account}, \mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u)$ from $\mathbf{C}_i$, **Sim** repeats the same process described above but for the emulation of the evaluation of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ in response to the message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{pOPRF\text{-}mode}, \mathsf{qid}, A \| m)$. **Sim** then stores the client chosen values along with the server chosen nonce $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}] := (\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ where $\mathsf{contact_identifier}$ is a unique identifier for the client's provided (E, x). **Sim** then forwards the client selected values $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ to each corrupted server $\mathbf{S}_{\mathsf{sid}}$. **Env**'s view is not affected since at the conclusion of account creation in the hybrid world, corrupted servers also receive $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ selected by the client.

Account Recovery. Upon receiving $(\mathsf{recover_account}, \mathsf{cid}, (\mathsf{contact_identifier}, E, x))$ from $\mathcal{F}_{\mathsf{acc}}$, **Sim** begins by emulating each instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ as described. If any errors emerge in this process, **Sim** exits. After **C** receives $\hat{E}_1, \hat{E}_2$ as a result of the $\mathcal{F}_{\mathsf{K\text{-}pop}}$ emulations, if **Sim** receives a message $(\mathsf{recover_account}, (\mathsf{contact_identifier}, Q, e))$ from $\mathcal{F}_{\mathsf{acc}}$, it signifies that the client's provided (E, x) corresponds to an existing account. **Sim** then retrieves the previously stored values $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$. If $\mathbf{C}_i$ was corrupted during account creation, these values are the same ones previously selected by the client $\mathbf{C}_i$. If the $\mathbf{C}_i$ was honest during account creation, these values were selected uniformly at random by **Sim**. Otherwise, $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ were selected by the corrupted client. In the case that the client was previously honest, a fresh corruption of the client occurred between account creation and the current account recovery request ($\mathbf{C}_i \in c$). **Sim** must then program $\mathcal{F}_{\mathsf{pro}}$ such that a hash query with the input $(E, \hat{E}_1, \hat{E}_2)$ results in an output (id, k_E) where key k_E properly decrypts to the recovery string $(e \| Q \| m \| padding)$. **Sim** can then remove $\mathbf{C}_i$ from the list of corruptions c. **Sim** then proceeds to send (Q, m, ct_u) to $\mathcal{F}_{\mathsf{SMT}}$ to forward for the client's recovery email e, where Q, e were provided by $\mathcal{F}_{\mathsf{acc}}$ and m, ct_u were stored by the simulator during account creation. **Env**'s view is not affected. In the hybrid world, the servers construct this same message and send it using $\mathcal{F}_{\mathsf{SMT}}$.

In the case that the client's provided (E, x) values do not correspond to an account, **Sim** yet again first checks if the client was freshly corrupted and previously made the same failed account recovery attempt using (E, x). If so, **Sim** retrieves the previous (id, k_E) values it sampled uniformly at random during the failed recovery attempt and programs the random oracle using the $\hat{E}_1, \hat{E}_2$ values that **Sim** just observed through the $\mathcal{F}_{\mathsf{K\text{-}pop}}$ emulations such that a hash query using the input $(E \| \hat{E}_1 \| \hat{E}_2)$ would result in the previously sampled (id, k_E).

In all cases, regardless of whether (E, x) corresponds to an existing account, the client $\mathbf{C}_i$ selects (id, k_E), which the simulator **Sim** forwards to each corrupted server, consistent with **Env**'s view in the hybrid world. If $\mathbf{C}_i$ follows the protocol, (id, k_E) will be outputs from a hash query to $\mathcal{F}_{\mathsf{pro}}$.

Account Restoration. Upon receiving the message $(\mathsf{restore_account}, \mathsf{cid}, (\mathsf{contact_identifier}, E, A, m, n))$ from $\mathcal{F}_{\mathsf{acc}}$, **Sim** begins y emulating each instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ as described. In the case that account restoration was successful (the client provided the correct security answers A), **Sim** receives the message $(\mathsf{restore_account}, k_u)$ from $\mathcal{F}_{\mathsf{acc}}$. **Sim** then retrieves the previously stored values $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ from $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}]$. **Sim** must then compute k_A using the previously stored ciphertext ct_u and the user key k_u. **Sim** then programs $\mathcal{F}_{\mathsf{pro}}$ using the values $\hat{A}_1, \hat{A}_2$ that resulted from emulation each $\mathcal{F}_{\mathsf{K\text{-}pop}}$ evaluation such that future hash queries with the input $(A \| m \} \hat{A}_1 \| \hat{A}_2)$ result in the output k_A. **Env**'s view is unaffected. In the hybrid world, the client receives k_A as the output of a hash query such that k_A is used to decrypt ct_u to k_u. This is identical to how a corrupted client $\mathbf{C}_i$ receives k_u in the ideal world.

Case 2: honest client and corrupted server(s). Upon receiving $(\mathsf{create_account}, \mathsf{cid}, (\mathsf{contact_identifier}, E))$ to indicate a new account creation request, **Sim** first checks if a nonce n needs to be simulated, depending on whether $\mathbf{S}_1$ is corrupted. If $\mathbf{S}_1$ is corrupted, **Sim** receives a nonce from the corrupted server. In the event that $\mathbf{S}_2$ is honest, **Sim** checks if the corrupted server, $\mathbf{S}_1$ is cheating by repeating a nonce. If so, it sends an error message to $\mathcal{F}_{\mathsf{acc}}$, which then exits and returns error to **C**. This is consistent with **Env**'s view in the hybrid world. If $\mathbf{S}_1$ selects a previously used nonce, $\mathbf{S}_2$ sends back error to the client and the client exits. In both the hybrid and ideal world, if the $\mathbf{S}_2$ is corrupted, it will neglect the step of catching repeated nonces.

For each server, **Sim** emulates each instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ as described through which appropriate messages are generated for any corrupted servers. Afterwards, **Sim** randomly samples: $\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u$ and stores these values in $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}]$. **Sim** then sends $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ to each corrupted server. These values are identically distributed to their counterparts in the hybrid world, therefore not affecting **Env**'s view.

Account Recovery. Upon receiving the messsage $(\mathsf{recover_account}, \mathsf{cid}, \mathsf{contact_identifier})$ from $\mathcal{F}_{\mathsf{acc}}$, **Sim** first begins by emulating each instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ as described. If the account recovery attempt was successful (the values (E, x) provided to $\mathcal{F}_{\mathsf{acc}}$ by the client correspond to an existing account), **Sim** receives

$(\text{recover_account}, (\text{contact_identifier}, Q, e))$ from $\mathcal{F}_{\text{acc}}$. **Sim** then retrieves the values it randomly sampled during account creation and the server selected nonce $(\text{id}, m, \text{ct}_r, \text{ct}_u, n)$ from $\text{DX}_{\textbf{Sim}}[\text{contact_identifier}]$. **Sim** then randomly samples $k_E \xleftarrow{\$} \{0,1\}^{\text{rlen}}$. k_E is thus identically distributed to a k_E from the hybrid world. Since a successful account recovery is not repeated, k_E is generated and never reused. **Sim** sends (id, k_E) to each corrupted server and does not store k_E. At the end of a successful account recovery request, **Sim** uses $\mathcal{F}_{\text{SMT}}$ to send (Q, m, ct_u) to the client's recovery email e, where (Q, e) came from $\mathcal{F}_{\text{acc}}$.

In the case that account recovery was unsuccessful, **Sim** checks if an honest accidentally made this same unsuccessful account recovery attempt previously using the same (E, x) values $(\text{contact_identifier} \in \text{DX}_*)$. If so, **Sim** retrieves the (id, k_E) values it previously generated from $\text{DX}_*[\text{contact_identifier}]$. Otherwise, **Sim** randomly samples $\text{id} \xleftarrow{\$} \{0,1\}^k, k_E \xleftarrow{\$} \{0,1\}^{\text{rlen}}$ and stores them in $\text{DX}_*[\text{contact_identifier}]$. **Sim** then sends (id, k_E) to each server. Since they are randomly generated values, they are distributed identically to their counterparts in the hybrid world, therefore not affecting **Env**'s view.

Account Restoration. Upon receiving $(\text{restore_account}, \text{cid}, \perp)$ from $\mathcal{F}_{\text{acc}}$, **Sim**'s only role in account restoration is to emulate each instance of $\mathcal{F}_{\text{K-pop}}$ as described. During this emulation, **Sim** will catch any potential cheating in which the key a server used in account creation is inconsistent with the key it is using during restoration. **Sim** flags this behavior to $\mathcal{F}_{\text{acc}}$ who sends an error message if appropriate. This is consistent with how the client catches server cheating in the hybrid world. Regardless of whether account restoration is successful, the servers do not learn anything in the process. If the recovery attempt is successful, the honest client will receive k_u from $\mathcal{F}_{\text{acc}}$.

Case 3: offline attack conducted by two corrupted servers. Upon receiving $(\text{offline_attack}, E, x, A)$ from **Adv**, **Sim** first checks if either server is honest. If so, the offline attack fails and **Sim***returns*$\perp$. To begin an offline recovery request, **Sim** first emulates both instances of $\mathcal{F}_{\text{K-pop}}$ as described. At the end of the emulation, **Sim** observes the values $\hat{E}_1, \hat{E}_2$.

Sim sends the message $(\text{offline_recover}, E, x)$ to $\mathcal{F}_{\text{acc}}$ to first attempt to retrieve the security questions and recovery email. If the account recovery attempt fails, **Sim** obtains (id, k_E) by sending $(\texttt{HashQuery}, (E \| \hat{E}_1 \| \hat{E}_2))$. Since there is no corresponding account, there will be no records within each server's database under id. **Sim** then exits. This is consistent with the hybrid world in which **Adv** exits the protocol.

If the provided (E, x) values correspond to an existing account, $\mathcal{F}_{\text{acc}}$ returns the message $(\text{offline_recover}, (\text{contact_identifier}, Q, e))$ to indicate a successful offline recovery request. **Sim** then retrieves the values it previously stored in correspondence to creation of the account associated with (E, x). It retrieves $(\text{id}, m, \text{ct}_r, \text{ct}_u, n)$ from $\text{DX}_{\textbf{Sim}}[\text{contact_identifier}]$. **Sim** computes the $k_E = (e \| Q \| m \| padding) \oplus \text{ct}_r$ and programs $\mathcal{F}_{\text{pro}}$ through sending the message $(\texttt{Program}, (E \| \hat{E}_1 \| \hat{E}_2), (\text{id}, k_E))$ to ensure correctness if a future client attempts account recovery. **Sim** then sends $(\text{id}, k_E, e, Q, m, \text{ct}_u)$ to **Adv**. This is identical to the outputs **Adv** receives in the hybrid world and **Env**'s view is not affected.

A successful offline recovery attempt is followed by an attempted account restoration. This begins with an emulation of both instances $\mathcal{F}_{\text{K-pop}}$ as described. At the conclusion of the emulation, **Sim** observes the values $\hat{A}_1, \hat{A}_2$. **Sim** then proceeds to send the message $(\text{offline_restore}, (\text{contact_identifier}, k_u))$ to $\mathcal{F}_{\text{acc}}$. If **Sim** receives the message $(\text{offline_restore}, (\text{contact_identifier}, k_u))$ indicating a successful account restoration attempt using A, **Sim** computes $k_A := k_u \oplus \text{ct}_u$ and programs $\mathcal{F}_{\text{pro}}$ through sending the message $(\texttt{Program}, (A \| m \| \hat{A}_1 \| \hat{A}_2), k_A)$ to ensure correctness in the event of a future restoration by a client. **Sim** then returns k_u to **Adv**. In both worlds, **Adv** receives k_u at the end of restoration. Therefore, **Env**'s view is not affected.

Protocol Π_{acc}

This is an interactive protocol involving two specific servers $\mathbf{S}_1$ and $\mathbf{S}_2$, along with an arbitrary number of clients $\mathbf{C}$. It involves several interactive methods, each of which is instantiated when the environment **Env** gives an input to a particular entity (client or server). Clients in this protocol are stateless; each server maintains a database and a rate limit counter. On first activation, both servers $\mathbf{S_1}$ and $\mathbf{S_2}$ follow the (Reset) command to set their rate limit counter to 0. This protocol is initialized with a predetermined rate limit threshold, thres. Server $\mathbf{S}_2$ maintains the set of prior nonces, nonces. In each new message that a query id qid is used, $\mathbf{C}_i$ samples a fresh random string. This protocol makes use of a few UC subroutines: $\mathcal{F}_{\mathsf{pro}}$ and two instances of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ (one per server).

- When the environment sends input (Init) to $\mathbf{S}_{\mathsf{sid}}$:
  - $\mathbf{S}_{\mathsf{sid}}$ samples $\mathsf{kid} \xleftarrow{\$} \{0,1\}^k$ and sends message $(\mathsf{Init}, \mathsf{kid})$ to $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$
- When the environment sends input $(\mathsf{create_account}, (Q, E, e, x, A, k_u))$ to $\mathbf{C}_i$:
  - $\mathbf{C}_i$ randomly samples a client nonce $m \xleftarrow{\$} \{0,1\}^{256}$.
  - $\mathbf{C}_i$ requests a server nonce n.
    - $\mathbf{S}_1$ samples a random string $n \xleftarrow{\$} \{0,1\}^k$ and sends n to $\mathbf{S}_2$ and $\mathbf{C}_i$.
    - If $n \in \mathsf{nonces}, \mathbf{S}_2$ returns error to $\mathbf{C}_i$ and $\mathbf{C}_i$ exits. Else, $\mathbf{S}_2$ adds n to nonces and returns n to $\mathbf{C}_i$.
  - Client $\mathbf{C}_i$ sends the email address E to the two servers.
  - In response, each server $\mathbf{S}_j$ runs two instances of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ in parallel with the client.
    - In the first instance, the client sends message $(\mathsf{Eval}, \mathsf{sessionid}, \texttt{pOPRF-mode}, \mathsf{qid}, E, x)$ and receives the message $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, (E, x), \hat{E}_j)$.
    - In the second instance, the client sends message $(\mathsf{Eval}, \mathsf{sessionid}, \texttt{pOPRF-mode}, \mathsf{qid}, n, A \| m)$ and receives the message $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, (n, A \| m), \hat{A}_j)$.
  - $\mathbf{C}_i$ stores the pairs $(\hat{E}_1, \hat{E}_2)$ and $(\hat{A}_1, \hat{A}_2)$.
  - The client sends two messages to $\mathcal{F}_{\mathsf{pro}}$.
    - $\mathbf{C}_i$ sends message $(\texttt{HashQuery}, E \parallel \hat{E}_1 \parallel \hat{E}_2)$ to $\mathcal{F}_{\mathsf{pro}}$ and receives message $(\texttt{HashQuery}, (\mathsf{id}, k_E))$ in response.
    - $\mathbf{C}_i$ sends message $(\texttt{HashQuery}, A \parallel m \parallel \hat{A}_1 \parallel \hat{A}_2)$ to $\mathcal{F}_{\mathsf{pro}}$ and receives message $(\texttt{HashQuery}, (\mathsf{id}, k_E))$ in response.
  - Client $\mathbf{C}_i$ sends message $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ to both servers, where $\mathsf{ct}_r = k_E \oplus (e \parallel Q \parallel m \parallel \mathit{padding})$ and $\mathsf{ct}_u = k_A \oplus k_u$. Each server appends this record to their local database, as long as id is a unique key that is distinct from all prior records.
  - Finally, client $\mathbf{C}_i$ outputs $\perp$.
- When the environment sends $(\mathsf{recover_account}, E)$ to $\mathbf{C}_i$:
  - The client $\mathbf{C}_i$ independently invokes each instance of $\mathcal{F}_{\mathsf{K\text{-}pop}}$, one with each server $\mathbf{S}_j$.
    - $\mathbf{C}_i$ sends message $(\mathsf{Eval}, \mathsf{sessionid}, \texttt{OPRF-mode}, \mathsf{qid}, E, x)$ and receives $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, (E, x), \hat{E}_j)$.
  - $\mathbf{C}_i$ retrieves the pair $(\hat{E}'_1, \hat{E}'_2)$ stored during account creation. If $(\hat{E}_1 \neq \hat{E}'_1)$ or $(\hat{E}_2 \neq \hat{E}'_2)$, then exit.
  - $\mathbf{C}_i$ sends message $(\texttt{HashQuery}, E \parallel \hat{E}_1 \parallel \hat{E}_2)$ to $\mathcal{F}_{\mathsf{pro}}$ and receives message $(\texttt{HashQuery}, (\mathsf{id}, k_E))$ in response.
  - Client $\mathbf{C}_i$ sends (id, k_E) to both servers.
  - In response, each server queries their local database for the record $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$.
    - If a record exists, they compute $r = k_E \oplus \mathsf{ct}_r$ and contact the client via the client information stored in $r = e \parallel Q \parallel m \parallel \mathit{padding}$ using $\mathcal{F}_{\mathsf{SMT}}$ with the message (Q, m, ct_u, n).
    - If no record with identifier id exists, then the servers abort this protocol.

Figure 14: Protocol Π_{acc} for account recovery, part 1 of 2

Protocol Π_{acc} (continued)

- when the environment sends $(\mathsf{restore_account}, (E,A,m))$ to $\mathbf{C}_i$:
  - $\mathbf{C}_i$ queries the servers, who send back (Q, m, ct_u, n) in response. The client aborts if the messages from the two servers differ.
  - The client $\mathbf{C}_i$ invokes $\mathcal{F}_{\mathsf{K\text{-}pop}}$ with each server $\mathbf{S}_j$.
    - $\mathbf{C}_i$ sends message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{OPRF\text{-}mode}, \mathsf{qid}, n, A\|m)$ and receives $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, (E,x), \hat{E}_j)$.
  - $\mathbf{C}_i$ retrieves the pair $(\hat{A}'_1, \hat{A}'_2)$ stored during account creation. If $(\hat{A}_1 \neq \hat{A}'_1)$ or $(\hat{A}_2 \neq \hat{A}'_2)$, then exit.
  - Client $\mathbf{C}_i$ sends message $(\mathtt{HashQuery}, A \parallel m \parallel \hat{A}_1 \parallel \hat{A}_2)$ to $\mathcal{F}_{\mathsf{pro}}$ and receives message $(\mathtt{HashQuery}, k_A)$ in response.
  - The client $\mathbf{C}_i$ computes $k_u = k_A \oplus \mathsf{ct}_u$ and tests whether this key is correct by attempting to decrypt their account data. (If so, $\mathbf{C}_i$ then invokes account creation so that k_u can be recovered again, possibly with new security questions and answers.)
- upon receiving $(\mathsf{offline_attack}, E, A)$ from **Adv**:
  - **Adv** invokes $\mathcal{F}_{\mathsf{K\text{-}pop}}$ with itself, playing the role of both the client and the server. For each server $\mathbf{S}_j$:
    - **Adv** sends message $(\mathsf{Eval}, \mathsf{sessionid}, \mathsf{qid}, E, x)$ and receives the message $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, (E,x), \hat{E}_j)$
  - **Adv** sends message $(\mathtt{HashQuery}, E \parallel \hat{E}_1 \parallel \hat{E}_2)$ to $\mathcal{F}_{\mathsf{pro}}$ and receives message $(\mathtt{HashQuery}, (\mathsf{id}, k_E))$ in response.
  - In response, **Adv** queries each database for the record $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ and the corresponding nonce n
    - If no record with identifier id exists, **Adv** aborts this protocol.
  - Compute $r = k_E \oplus \mathsf{ct}_r$, where $r = e \parallel Q \parallel m \parallel padding$
  - **Adv** invokes $\mathcal{F}_{\mathsf{K\text{-}pop}}$ with itself, playing the role of both the client and the server. For each server $\mathbf{S}_j$
    - **Adv** sends message $(\mathsf{Eval}, \mathsf{sessionid}, \mathsf{qid}, n, A\|m)$ and receives the message $(\mathsf{EvalComplete}, \mathsf{sessionid}, \mathsf{qid}, (n, A\|m), \hat{A}_j)$
  - **Adv** sends message $(\mathtt{HashQuery}, A\|m\|\hat{A}_1\|\hat{A}_2)$ to $\mathcal{F}_{\mathsf{pro}}$ and receives message $(\mathtt{HashQuery}, k_A)$ in response
  - **Adv** computes $k_u = k_A \oplus \mathsf{ct}_u$
- upon receiving $(\mathsf{corrupt}, \mathbf{C}_i)$ from **Env**:
  - Send $(\mathsf{corrupt}, \mathbf{C}_i)$ to both instances of $\mathcal{F}_{\mathsf{K\text{-}pop}}$.
- upon receiving $(\mathsf{corrupt}, \mathbf{S}_j)$ from **Env**:
  - Send $(\mathsf{corrupt}, \mathbf{S}_j)$ to instance j of $\mathcal{F}_{\mathsf{K\text{-}pop}}$.
  - Output the server's local database to **Env**.

Figure 15: Protocol Π_{acc} for account recovery, part 2 of 2

Functionality $\mathcal{F}_{\mathsf{pro}}$

Upon receiving $(\mathtt{HashQuery}, m)$:

1. If there is a record (m, h), then output $(\mathtt{HashQuery}, h)$ to the caller.
2. Else if $m \in X$, then choose $h' \leftarrow Y$, record (m, h'), and output $(\mathtt{HashQuery}, h')$ to the caller.
3. Else (that is, if $m \notin X$), output $(\mathtt{HashQuery}, \perp)$ to the caller.

Upon receiving $(\mathtt{Program}, m, h)$ from the adversary:

1. If there is no record (m, h'), then record (m, h). (But if m has already been queried, then programming fails silently.)
2. Send $(\mathtt{Program})$ to the adversary.

Figure 16: Programmable random oracle with input domain X and output range Y. Adapted from [27].

Functionality $\mathcal{F}_{\mathsf{acc}}$

The functionality stores and manages an account recovery dictionary DX and dictionary DX_contact. The functionality sets contact_ctr $:= 0$. Let sid be a server id. It then interacts with a client $\mathbf{C}$, servers $\mathbf{S}_1$, $\mathbf{S}_2$, and the environment **Env** as follows:

- upon receiving (init) from $\mathbf{S}_{\mathsf{sid}}$:
  - If $\mathbf{S}_{\mathsf{sid}}$ is corrupted, send message $(\mathsf{init}, \mathsf{sid})$ to **Sim**
- upon receiving (create_account, (Q, E, e, x, A, k_u)) from $\mathbf{C}_i$:
  - Set DX_contact$[(E,x)]$:= contact_ctr and increment contact_ctr = contact_ctr + 1
  - Send an initial message to **Sim**:
    - If $\mathbf{C}_i$ is corrupted, send message (create_account, $\mathbf{C}_i$, (DX_contact$[(E,x)]$, Q, E, e, x, A, k_u)) to **Sim**
    - Else, send message (create_account, $\mathbf{C}_i$, DX_contact$[(E,x)]$) to **Sim**
  - If **Sim** responds with message (create_account, error) or (K-pop, error), return error to $\mathbf{C}_i$ and exit
  - Store DX$[(E,x)] := (Q, e, A, k_u)$
  - Return $\perp$ to $\mathbf{C}_i$
- upon receiving (recover_account, E, x) from $\mathbf{C}_i$:
  - If $(E,x) \notin$ DX_contact, set DX_contact$[(E,x)]$:= contact_ctr and increment contact_ctr = contact_ctr + 1
  - Send an initial message to **Sim**:
    - If $\mathbf{C}_i$ is corrupted, send message (recover_account, $\mathbf{C}_i$, (DX_contact$[E,x]$, E, x)) to **Sim**
    - Else, send message (recover_account, $\mathbf{C}_i$, DX_contact$[E,x]$) to **Sim**
  - If **Sim** responds with message (K-pop, error), return error to $\mathbf{C}_i$ and exit
  - If $(E,x) \in$ DX,
    - Retrieve (Q, e, A, k_u) from DX$[(E,x)]$
    - Send message (recover_account, (DX_contact$[(E,x)]$, Q, e)) to **Sim** to notify of successful recovery
    - Send (Q, e) to each honest server
  - Else, send $\perp$ to each honest server
  - Return $\perp$ to $\mathbf{C}_i$
- upon receiving (restore_account, (E, x, A, m)) from $\mathbf{C}_i$:
  - Send an initial message to **Sim**:
    - If $\mathbf{C}_i$ is corrupted, send message (restore_account, $\mathbf{C}_i$, (DX_contact$[E,x]$, A, m)) to **Sim**
    - Else, send message (restore_account, $\mathbf{C}_i$, $\perp$) to **Sim**
  - If **Sim** responds with message (K-pop, error),
    - If $(E, x \in$ DX), retrieve (Q, e, A, k_u) from DX$[(E,x)]$ and if $(A == A')$, return error to $\mathbf{C}_i$ and exit
  - If $(E,x) \in$ DX,
    - Retrieve (Q, e, A, k_u) from DX$[(E,x)]$
    - If $A == A'$,
      - If $\mathbf{C}_i$ is corrupted, send message (restore_account, k_u) to **Sim** to notify of successful restoration
      - Else, return k_u to $\mathbf{C}_i$ and $\perp$ to $\mathbf{S}_1, \mathbf{S}_2$
  - Return $\perp$ to $\mathbf{C}_i, \mathbf{S}_1, \mathbf{S}_2$
- upon receiving (offline_recover, E, x) from **Sim**:
  - If $(E,x) \in$ DX,
    - Retrieve (Q, e, A, k_u) from DX$[(E,x)]$
    - Send message (offline_recover, (DX_contact$[(E,x)]$, Q, e)) to **Sim**
- upon receiving (offline_restore, (E, x, A)) from **Sim**:
  - If $(E,x) \in$ DX,
    - Retrieve (Q, e, A', k_u) from DX$[(E,x)]$
    - If $A == A'$,
      - Send message (offline_restore, DX_contact$[(E,x)]$, k_u)

Figure 17: Functionality $\mathcal{F}_{\mathsf{acc}}$ for account recovery, part 1 of 2

Functionality $\mathcal{F}_{\mathsf{acc}}$

- upon receiving $(\mathsf{corrupt}, \mathbf{C}_i)$ from **Env**:
  - Send message $(\mathsf{corrupt}, \mathbf{C}_i)$ to **Sim**
- upon receiving $(\mathsf{corrupt}, \mathbf{S}_{\mathsf{sid}})$ from **Env**:
  - Send message $(\mathsf{corrupt}, \mathbf{S}_{\mathsf{sid}})$ to **Sim**

Figure 18: Functionality for account recovery, part 2 of 2 (continued from Fig. 17).

Simulator Sim

The simulator **Sim** stores dictionaries $\mathsf{DX}_{\mathbf{Sim}}$, DX_*. It also stores a list of corruptions, c. In each new message that a query id qid is used, **Sim** samples a fresh random string. **Sim** works as follows:

- upon receiving $(\mathsf{init}, \mathsf{sid})$:
  - Emulate $\mathcal{F}_{\mathsf{K\text{-}pop}}$ by internally running a copy of $\mathcal{F}_{\mathsf{K\text{-}pop}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ as described in Figures 3 and 2.
- upon receiving $(\mathsf{create_account}, \mathsf{cid}, (\mathsf{contact_identifier}, Q, E, e, x, A, k_u))$: //Corrupted client
  - If $\mathbf{S}_1$ is corrupted,
    - Receive message $(\mathsf{create_account}, n)$ from $\mathbf{S}_1$
    - If $\mathbf{S}_2$ is honest and $n \in \mathsf{nonces}$, send message $(\mathsf{create_account}, \mathsf{error})$ to $\mathcal{F}_{\mathsf{acc}}$ and exit
    - Else, add n to nonces
  - Else,
    - Sample $n \xleftarrow{\$} \{0,1\}^k$ and add n to nonces
    - Send n to $\mathbf{C}_i$ and $\mathbf{S}_2$
  - For each server $\mathsf{sid} \in \{1,2\}$, upon receiving message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{pOPRF\text{-}mode}, \mathsf{qid}, E, x)$ from $\mathbf{C}_i$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit. At the conclusion of the emulation, set $\hat{E}_{\mathsf{sid}} = \rho$.
  - upon receiving message $(\mathsf{create_account}, \mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u)$ from $\mathbf{C}_i$,
    - For each $\mathsf{sid} \in \{0,1\}$, upon receiving message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{pOPRF\text{-}mode}, \mathsf{qid}, n, A\|m)$ from $\mathbf{C}_i$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit. At the conclusion of the emulation, set $\hat{A}_{\mathsf{sid}} = \rho$.
    - Store $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}] := (\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$
    - For each corrupted server $\mathbf{S}_1$, $\mathbf{S}_2$, send $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ to $\mathbf{S}_{\mathsf{sid}}$
- upon receiving $(\mathsf{create_account}, \mathsf{cid}, (\mathsf{contact_identifier}, E))$: //Honest client
  - If $\mathbf{S}_1$ is corrupted,
    - Receive message $(\mathsf{create_account}, n)$ from $\mathbf{S}_1$
    - If $\mathbf{S}_2$ is honest and $n \in \mathsf{nonces}$, send message $(\mathsf{create_account}, \mathsf{error})$ to $\mathcal{F}_{\mathsf{acc}}$ and exit
    - Else, add n to nonces
  - Else,
    - Sample $n \xleftarrow{\$} \{0,1\}^k$ and add n to nonces
    - Send n to $\mathbf{C}_i$ and $\mathbf{S}_2$
  - For each $\mathsf{sid} \in \{1,2\}$,
    - Emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22 by first creating the message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{pOPRF\text{-}mode}, \mathsf{qid}, E, \mathsf{contact_identifier})$ to send on behalf of the client $\mathbf{C}$. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit.
    - Emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22 by first creating the message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{pOPRF\text{-}mode}, \mathsf{qid}, \perp, \perp)$ on behalf of the client $\mathbf{C}$. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit.
  - Sample $\mathsf{id} \xleftarrow{\$} \{0,1\}^k, m \xleftarrow{\$} \{0,1\}^k, \mathsf{ct}_r \xleftarrow{\$} \{0,1\}^{\mathsf{rlen}}, \mathsf{ct}_u \xleftarrow{\$} \{0,1\}^{\mathsf{ulen}}$
  - Store $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}] := (\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$
  - For each corrupted server $\mathbf{S}_1$, $\mathbf{S}_2$, send $(\mathsf{id}, \mathsf{ct}_r, \mathsf{ct}_u)$ to $\mathbf{S}_{\mathsf{sid}}$

Figure 19: Description of the simulator **Sim**, part 1 of 3

Simulator Sim

- upon receiving $(\mathsf{recover_account}, \mathsf{cid}, (\mathsf{contact_identifier}, E, x))$: //Corrupted client
  - For each server $\mathsf{sid} \in \{1,2\}$,
    - Upon receiving message $(\mathsf{Eval}, \mathsf{sessionid}, \texttt{OPRF-mode}, \mathsf{qid}, E, x)$ from $\mathbf{C}_i$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit. At the conclusion of the emulation, set $\hat{E}_{\mathsf{sid}} = \rho$.
  - If **Sim** receives message $(\mathsf{recover_account}, (\mathsf{contact_identifier}, Q, e))$ from $\mathcal{F}_{\mathsf{acc}}$,
    - Retrieve $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ from $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}]$
    - If $\mathbf{C}_i \in c$,
      - Set $k_E = e \| Q \| m \| padding \oplus \mathsf{ct}_r$
      - Send message $(\texttt{Program}, (E, \hat{E}_1, \hat{E}_2), (\mathsf{id}, k_E))$ to $\mathcal{F}_{\mathsf{pro}}$ and remove $\mathbf{C}_i$ from c
    - Send message $(\mathsf{send}, (\mathsf{sessionid}, \texttt{smt}), e, (Q, m, \mathsf{ct}_u))$ to $\mathcal{F}_{\mathsf{SMT}}$
  - Else,
    - If $\mathbf{C}_i \in c$ and $\mathsf{contact_identifier} \in \mathsf{DX}_*$, //Incorrect (E,x) used by previously honest client
      - Retrieve (id, k_E) from $\mathsf{DX}_*[\mathsf{contact_identifier}]$
      - Send message $(\texttt{Program}, (E, \hat{E}_1, \hat{E}_2), (\mathsf{id}, k_E))$
      - Delete $\mathsf{DX}_*[\mathsf{contact_identifier}]$
  - Upon receiving (id, k_E) from $\mathbf{C}_i$, send message (id, k_E) to each corrupted server
- upon receiving $(\mathsf{recover_account}, \mathsf{cid}, \mathsf{contact_identifier})$: //Honest client
  - For each $\mathsf{sid} \in \{1,2\}$,
    - Emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22 by first creating the message $(\mathsf{Eval}, \mathsf{sessionid}, \texttt{OPRF-mode}, \mathsf{qid}, E, \mathsf{contact_identifier})$. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit.
  - If **Sim** receives message $(\mathsf{recover_account}, (\mathsf{contact_identifier}, Q, e))$ from $\mathcal{F}_{\mathsf{acc}}$,
    - Retrieve $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ from $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}]$
    - Samples $k_E \xleftarrow{\$} \{0,1\}^{\mathsf{rlen}}$
    - Send (id, k_E) to each server
    - Send message $(\mathsf{send}, (\mathsf{sessionid}, \texttt{smt}), e, (Q, m, \mathsf{ct}_u))$ to $\mathcal{F}_{\mathsf{SMT}}$
  - Else, //Incorrect (E,x) from honest client
    - If $\mathsf{contact_identifier} \in \mathsf{DX}_*$,
      - Retrieve (id, k_E) from $\mathsf{DX}_*[\mathsf{contact_identifier}]$
    - Else,
      - Sample $\mathsf{id} \xleftarrow{\$} \{0,1\}^k, k_E \xleftarrow{\$} \{0,1\}^{\mathsf{rlen}}$
      - $\mathsf{DX}_*[\mathsf{contact_identifier}] := (\mathsf{id}, k_E)$
    - For each server $\mathbf{S}_1, \mathbf{S}_2$, send (id, k_E)

Figure 20: Description of the simulator **Sim**, part 2 of 3

Simulator Sim

- upon receiving $(\mathsf{restore_account}, \mathsf{cid}, \bot)$: //Honest client, restoration failure OR account exists
  - For each $\mathsf{sid} \in \{1,2\}$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22 by first creating the message $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{OPRF\text{-}mode}, \mathsf{qid}, \bot, \bot)$. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit.
- upon receiving $(\mathsf{restore_account}, \mathsf{cid}, (\mathsf{contact_identifier}, E, A, m, n))$: //Corrupted client
  - For each $\mathsf{sid} \in \{1,2\}$,
    - ∗ upon receiving $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{OPRF\text{-}mode}, \mathsf{qid}, n, A\|m)$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22. If an error message was sent to $\mathcal{F}_{\mathsf{acc}}$ during this emulation, exit. At the conclusion of the emulation, set $\hat{A}_{\mathsf{sid}} = \rho$.
  - If **Sim** receives message $(\mathsf{restore_account}, k_u)$ from $\mathcal{F}_{\mathsf{acc}}$,
    - ∗ If $\mathbf{C}_i \in c$,
      - · Retrieve $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ from $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}]$
      - · Set $k_A = k_u \oplus \mathsf{ct}_u$
      - · Send message $(\mathtt{Program}, A\|m\|\hat{A}_1\|\hat{A}_2, k_A)$ to $\mathcal{F}_{\mathsf{pro}}$
- upon receiving $(\mathsf{corrupt}, \mathbf{S}_{\mathsf{sid}})$ from **Env**:
  - Send $(\mathsf{corrupt}, \mathbf{S}_{\mathsf{sid}})$ to emulated instance of $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$.
- upon receiving $(\mathsf{corrupt}, \mathsf{cid})$ from **Env**:
  - Append cid to c
  - Send $(\mathsf{corrupt}, \mathbf{C}_i)$ to both internal emulated instances of $\mathcal{F}_{\mathsf{K\text{-}pop}}$.
- upon receiving $(\mathsf{offline_attack}, E, x, A)$ from **Adv**:
  - If either server is honest, return $\bot$ and exit
  - For each $\mathsf{sid} \in \{1,2\}$, upon receiving $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{OPRF\text{-}mode}, \mathsf{qid}, E, x)$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22. At the conclusion of the emulation, set $\hat{E}_{\mathsf{sid}} = \rho$.
  - Send message $(\mathsf{offline_recover}, E, x)$ to $\mathcal{F}_{\mathsf{acc}}$
  - If $\mathcal{F}_{\mathsf{acc}}$ responds with message $(\mathsf{offline_recover}, (\mathsf{contact_identifier}, Q, e))$,
    - ∗ Retrieve $(\mathsf{id}, m, \mathsf{ct}_r, \mathsf{ct}_u, n)$ from $\mathsf{DX}_{\mathbf{Sim}}[\mathsf{contact_identifier}]$
    - ∗ Upon receiving $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{OPRF\text{-}mode}, \mathsf{qid}, E, x)$
    - ∗ Set $k_E = e\|Q\|m\|padding \oplus \mathsf{ct}_r$
    - ∗ Send message $(\mathtt{Program}, (E\|\hat{E}_1\|\hat{E}_2), (\mathsf{id}, k_E))$ to $\mathcal{F}_{\mathsf{pro}}$
    - ∗ Send message $(\mathsf{id}, k_E, e, Q, m, ct_u)$ to **Adv**
  - Else,
    - ∗ Send message $(\mathtt{HashQuery}, (E\|\hat{E}_1\|\hat{E}_2))$ to $\mathcal{F}_{\mathsf{pro}}$ and receive (id, k_E)
    - ∗ Send (id, k_E) to each corrupted server and exit
  - For each $\mathsf{sid} \in \{1,2\}$, upon receiving $(\mathsf{Eval}, \mathsf{sessionid}, \mathtt{OPRF\text{-}mode}, \mathsf{qid}, n, A\|m)$, emulate $\mathcal{F}^{\mathsf{sid}}_{\mathsf{K\text{-}pop}}$ according to Figure 22. At the conclusion of the emulation, set $\hat{A}_{\mathsf{sid}} = \rho$.
  - Send message $(\mathsf{offline_restore}, E, A)$ to $\mathcal{F}_{\mathsf{acc}}$
  - If $\mathcal{F}_{\mathsf{acc}}$ responds with message $(\mathsf{offline_restore}, (\mathsf{contact_identifier}, k_U))$,
    - ∗ Set $k_A := k_U \oplus ct_u$
    - ∗ Send message $(\mathtt{Program}, ((A\|m\|\hat{A}_1, \hat{A}_2), k_A))$ to $\mathcal{F}_{\mathsf{pro}}$ and return k_u to **Adv**
  - Return $\bot$ to **Adv**

Figure 21: Description of the simulator **Sim**, part 3 of 3

Simulator Sim & $\mathcal{F}_{\textsf{K-pop}}$ Emulation

The Simulator **Sim** stores a dictionary $\textsf{DX}_{\textsf{kid}}$ that maps the tuple $(\mathbf{C}, x)$ to a key id $\textsf{kid}$. **Sim** emulates $\mathcal{F}_{\textsf{OPRF}}$ during the evaluation of some inputs $x_{\textsf{kal}}, x_{\textsf{priv}}$ as described:

If the client **C** is honest and **S** is corrupted,

- Upon creating message $(\textsf{Eval}, \textsf{sessionid}, \texttt{OPRF-mode}, \textsf{qid}, x_{\textsf{kal}}, x_{\textsf{priv}})$ or $(\textsf{Eval}, \textsf{sessionid}, \texttt{pOPRF-mode}, \textsf{qid}, x_{\textsf{kal}}, x_{\textsf{priv}})$ on behalf of client **C**:
  - If in pOPRF mode: send $(\textsf{Eval}, \textsf{sessionid}, \textsf{qid}, x_{\textsf{kal}})$ to $\mathcal{A}^*$, and wait for a response $(\textsf{EvalContinue}, \textsf{sessionid}, \textsf{qid})$.
  - Emulate $\mathcal{F}_{\textsf{OPRF}}$ as follows:
    - Send message $(\textsf{EvalContinue}, \textsf{sessionid}, \textsf{qid})$ to $\mathcal{A}^*$
    - Upon receiving $(\textsf{EvalContinue}, \textsf{sessionid}, \textsf{qid}^*, \textsf{kid}^*)$ from $\mathcal{A}^*$, if $(\mathbf{C}, (x^*_{\textsf{kal}}, x_{\textsf{priv}})) \notin \textsf{DX}_{\textsf{kid}}$, store $\textsf{DX}_{\textsf{kid}}[(\mathbf{C}, (x_{\textsf{kal}}, x_{\textsf{priv}}))] := \textsf{kid}^*$. Otherwise:
      - If $\textsf{DX}_{\textsf{kid}}[(\mathbf{C}, (x_{\textsf{kal}}{}^*, x_{\textsf{priv}}))] \neq \textsf{kid}^*$, send error message $(\textsf{K-pop}, \textsf{error})$ to $\mathcal{F}_{\textsf{acc}}$ and exit

Else,

- The Simulator **Sim** internally runs a copy of $\mathcal{F}_{\textsf{K-pop}}$ and $\mathcal{F}_{\textsf{OPRF}}$ as described in Figures 3 and 2. In this emulation, **Sim** interacts with the client **C**, server **S**, and adversary $\mathcal{A}^*$ on behalf of $\mathcal{F}_{\textsf{K-pop}}$ and $\mathcal{F}_{\textsf{OPRF}}$ by keeping track of their internal states and sending messages to the appropriate parties. This includes outputting a final message to **C** (when appropriate) containing ρ where ρ is selected exactly as described in Figure 2. If at any point during this emulation an error message of the form $(\textsf{error}, \textsf{sessionid}, \textsf{qid}, \perp)$ is sent to $\mathcal{A}^*$, send the error message $(\textsf{K-pop}, \textsf{error})$ to the functionality $\mathcal{F}_{\textsf{acc}}$.

Figure 22: Description of the simulator **Sim** & $\mathcal{F}_{\textsf{K-pop}}$ Emulation