Single-Server Verifiable PIR with Updates and Universally Composable Security

Julia Guskind[1⋆], Ariel Hamlin[2], Ryan Little[1], Daniel S. Roche[3], and Mayank Varia[1]

[1] Boston University⋆⋆, Boston MA, USA
[2] Northeastern University, Boston MA, USA
[3] United States Naval Academy⋆⋆⋆, Annapolis MD, USA

Abstract. Private information retrieval (PIR) is a fundamental cryptographic primitive that allows a client to retrieve an entry of a database from a server without revealing which entry was retrieved. PIR security is traditionally defined with a distinguishing game that ensures the clients' access patterns are kept private from a *semi-honest server*. Verifiable PIR (VPIR) adds another game-based property that holds against a *malicious server*: the server is bound to a particular database and cannot cause a client to retrieve a database entry that is inconsistent with this database. Recent work by Alon and Beimel [ITC 2025] deviated from the traditional game-based PIR definitions and contributed a definition of standalone simulation-based security for multi-server PIR. Their techniques, however, do not readily extend to single-server VPIR and do not consider concurrent protocol composition when PIR is used as a building block within a larger application.

In this work, we further the study of simulation-based VPIR security. We are the first to formalize a universally composable (UC) definition of VPIR in the *single-server* setting by giving an ideal VPIR functionality. We motivate the need for UC security by showing how game-based VPIR properties fail under sequential and concurrent protocol composition. We also demonstrate the generality of our UC VPIR functionality by providing two realizations based on a trivial PIR and VeriSimplePIR [de Castro and Lee, USENIX Security 2024].

Additionally, we introduce a new kind of VPIR, called Updatable VPIR (UVPIR), which guarantees to clients that (1) database updates are authorized by permissioned clients and (2) responses to their queries are consistent with a specific version of the PIR database. We show that UVPIR can be constructed in a black-box manner on top of any VPIR protocol.

Keywords: Private Information Retrieval · Universal Composability · Verifiability · Simulation-based Security

1 Introduction

Private Information Retrieval (PIR) is a well-studied cryptographic primitive that enables clients to fetch items efficiently from a database while hiding which record they are fetching [2, 6, 11, 13, 15, 31, 35, 37, 40, 46, 55, 60, 61, 86]. PIR serves as a critical building block in a variety of privacy-preserving applications in research and practice, including anonymous messaging [7, 29, 71], private web search [9, 54], certificate transparency logs [44, 55, 72, 78], password breach alerts [4, 67, 76, 80], private identity lookup for contact discovery or blocklisting [58, 59], and private content delivery of advertising, media, and software updates [3, 27, 52, 53].

Traditionally, PIR is defined with two game-based security properties for honest clients against a semi-honest server. First is correctness, which says if the client queries for index i and the server

⋆ This material is supported by the Clare Boothe Luce Graduate Fellowship.
⋆⋆ This material is based upon work supported by DARPA under Agreement No. HR00112020021 and by the National Science Foundation under Grants No. 2209194 and 2613424.
⋆⋆⋆ The views expressed in this article are those of the authors and do not reflect the official policy or position of the U.S. Naval Academy, Department of the Navy, the Department of Defense, or the U.S. Government.

behaves honestly, then the client will learn the database entry at index i. Second is query privacy, which guarantees that the server does not learn anything about the underlying index i.

A recent line of work starting with [33] has introduced *Verifiable PIR* (VPIR), also known as malicious PIR. VPIR is an extension of PIR that captures security against malicious servers that may arbitrarily deviate from the protocol [33]. VPIR has been defined by prior works in various ways, but generally seeks to capture two intuitive security properties [33,36,38,43,44,77,82]. First, an *integrity* guarantee that "binds" a malicious server to a particular database, ensuring that the server cannot change the database and cause the client to learn an incorrect entry. Second, *security against selective failures*, which guarantees a server cannot ascertain any information about a client's index, even when they can learn whether a client aborts during the protocol. While security against selective failures is conceptually straightforward, making this guarantee rigorous requires that we identify the "correct" database for each execution. Multi-server PIR has a straightforward answer [5]: the database is whatever the majority (or plurality) of servers think it is. Since this approach does not work in single-server VPIR, prior constructions [8,33,36,38,43,77,81,82] require all participants to agree on a commitment, or digest, of the database beforehand. All prior works on single-server VPIR build bespoke security games that are based on the specific commitment protocol used.

Composition. As with many cryptographic objects, PIR is often composed with other building blocks within a larger protocol, or with itself by running multiple instances concurrently. Concurrent self-composition readily arises in practice: a client may batch multiple queries simultaneously, or a slow network may cause a client to retransmit a query while the server is still processing it. Given this, PIR's security guarantees should be *composable*.

The bespoke game-based security definitions in prior (V)PIR papers are often designed to showcase specific properties, such as correctness, query privacy, privacy against selective failure, and integrity. However, game-based definitions that nominally provide a security property in a standalone setting can often fall apart under self-composition, much less composition with other protocols [16,69]; this is particularly true in scenarios involving secure computation with both privacy and integrity guarantees [26]. Making matters even worse, (V)PIR security games tend not to use 'standard' techniques to achieve sequential composition, like the use of auxiliary information [5, Remark 14].

Indeed, this barrier is not unique to VPIR; simulation-based definitions in the real/ideal paradigm fail in subtle yet spectacular ways when malicious parties abuse concurrent composition [1,19,39,49,51]. For these reasons, we desire a simulation-based definition for Verifiable PIR which captures the intuitive security properties even in the presence of arbitrary protocol composition.

The challenge. These issues motivate the overarching challenge of this work, which is to define a simulation-based VPIR that is simultaneously

1. General — can capture state-of-the-art game-based constructions
2. Extensible — can be used as a building block in broader protocols
3. Precise — achieves a robust, comprehensive notion of security

1.1 This work

In this paper, we provide a generic definition of Verifiable PIR as an ideal functionality $\mathcal{F}_{\mathsf{VPIR}}$ in the framework of universally composable (UC) security by Canetti [17]. We also showcase the strength and generality of our modeling in several ways: finding weaknesses in prior game-based

definitions that are addressed by $\mathcal{F}_{\mathsf{VPIR}}$, showing that existing VPIR constructions can be used to instantiate $\mathcal{F}_{\mathsf{VPIR}}$ with universal composability, and constructing a new primitive of *updatable* VPIR in a modular way on top of $\mathcal{F}_{\mathsf{VPIR}}$. We describe these four contributions in more detail below.

Defining single-server simulation-based VPIR. The cornerstone of our work is definitional. In Section 5, we provide a general simulation-secure *single-server* verifiable PIR definition ($\mathcal{F}_{\mathsf{VPIR}}$) in the UC security framework. The core motivations for verifiability are to prevent selective failure attacks and to ensure that VPIR can safely be used as a sub-protocol of a larger system without fear that error messages will breach security.

Our UC functionality $\mathcal{F}_{\mathsf{VPIR}}$ guarantees malicious security up to abort. It leverages some specific features of the UC framework (such as the session id sid and the ability to call protocol-specific "internal code" from an ideal functionality [21, 24]) in order to be agnostic to the specific choice of commitment used and the method of consensus used to disperse it. We give an overview of the definition in Section 2, with details in Section 5.

Separating our definition from prior works. In Section 3, we motivate the need for a stronger and more flexible VPIR definition. We identify tradeoffs in definitions from prior works, namely that VPIR definitions which allow client-specific preprocessing do not guarantee secure composition or adaptive security.

Realizing our definition. We show that our modeling is broad enough to capture two different constructions. In Appendix C, we construct a trivial PIR protocol with a digest given by a collision-resistant hash of the database and communication linear in the size of the database. In Appendix D, we provide a specification of the VeriSimplePIR [36] construction (which adds verifiability on top of SimplePIR [55]) with adjustments from [77], with a small modification of our own to allow for straight-line simulation. We show in Theorems 1 to 2 that both of these protocols UC-realize our ideal functionality $\mathcal{F}_{\mathsf{VPIR}}$.

Updatable VPIR. Finally, in Section 6, we contribute a new primitive, Updatable VPIR (UVPIR), which is meaningful in its own right and also demonstrates the modular design of our $\mathcal{F}_{\mathsf{VPIR}}$ functionality. This new primitive addresses a natural question: what if parties want to update the database in a VPIR? It turns out that constructing updatable VPIR is the easy part, because existing constructions tend to have linearly homomorphic commitments that are easy to update. The challenge instead lies in *defining* what UVPIR should even mean. Updatability and verifiability are in tension: a scheme that allows for arbitrary updates sacrifices verifiability, as it is not clear how to tell whether the database was tampered with by malicious updates. We resolve this issue by enforcing that updates can be made only by certain permissioned clients.

In Section 6.1 we give an ideal functionality $\mathcal{F}_{\mathsf{UVPIR}}$, which is constructed generically from any $\mathcal{F}_{\mathsf{VPIR}}$ in a single, highly flexible protocol. In Section 6.2 we then give a UVPIR protocol that UC-realizes our $\mathcal{F}_{\mathsf{UVPIR}}$. Leveraging UC features, our UVPIR protocol can treat the underlying VPIR either as a black box for full generality, or as a white box to enable efficiency improvements allowed by some VPIR constructions.

1.2 Related Work

Semi-honest PIR. PIR was first introduced by Chor et al. [31] in the multi-server setting, where multiple servers hold the same duplicated database, and a client wishes to retrieve a single indexed element without revealing anything about that index to any server. Their construction achieved

information-theoretic security and set off a line of work seeking to improve the costs and number of servers required [47].

In the single-server setting, information-theoretic security is impossible with sub-linear communication complexity, so computational assumptions are required. There is a long line of works on single-server PIR, starting with Kushilevitz and Ostrovsky [60]. A number of efficient semi-honest PIR protocols have been proposed in recent years, utilizing some combination of homomorphic encryption and client- or server-side preprocessing to improve the asymptotic complexity and practical performance [55, 70, 73–75]. We note that all of these papers assume the server is *semi-honest* and does not deviate from the protocol.

Verifiable PIR. Several works construct verifiable PIR protocols with game-based definitions. Selective failure in PIR was first addressed directly by work on *authenticated* PIR by Colombo et al. [33], who developed an efficient scheme where any lookup is guaranteed to fail, regardless of the index, if the database used by the server does not correspond to an honestly pre-computed digest. This model was further explored in [82, 85]. Subsequent work [36, 38, 43, 44, 77] strengthens malicious security by removing the assumption that the digest be honestly computed. Falk et al. [43] give a generic compiler that can transform any semi-honest PIR into a maliciously-secure one.

Alon and Biemel [5] propose a standalone simulation-based security definition for VPIR in the multi-server setting. Their definition does not securely compose under sequential or concurrent composition (c.f. Remark 3.2, [5]). A work by Ben-David et al. [12] proposes a different extension to PIR—also call verifiable PIR—that explores the general problem of proving a server responding to a PIR query uses a database satisfying some arbitrary property. Their definition is standalone and simulation-based. Our notion of verifiable PIR (used throughout this work) may be seen as a special case of Ben-David et al.'s problem in which the property being proved is consistency with a known digest.

In concurrent work, Choi et al. [30] presented a malicious-secure PIR with sublinear online client communication size. They provide a UC ideal functionality that is tailored to their specific protocol. By comparison, only our verifiable PIR functionality provides a correctness guarantee and is stress-tested in its use as a sub-routine within a larger system.

Access pattern hiding with updates. When we consider a PIR that supports updates, primitives such as Oblivious RAM (ORAM), which dates back to the seminal work of [48, 50] come to mind. The key difference between ORAM and PIR with updates is that ORAM must hide the database itself and the update operations from the server, whereas in PIR the database, and therefore the updates to it, are know to the server in the clear. Additionally, supporting multiple clients in ORAM remains a challenge, while PIR schemes (including our UVPIR scheme) are able to support multiple clients with no significant overhead.

Proofs of Retrievability. Proofs of retrievability (PoR), introduced by Juels and Kaliski [57], address the problem of auditing an untrusted server to confirm that it is storing the entire database, without actually fetching any entries.

PoR necessarily require some sort of digest to be held by the client, and must inherently be malicious-secure (the client does not need any privacy, and the problem is trivial if the server is semi-honest). Some more recent work, such as [41], looked at the problem of efficiently updating the database and corresponding digest as well. As pointed out by [30] and examined further in this paper, certain definitions of verifiable PIR inherently provide proofs of retrievability to the underlying database.

PIR with database updates. Some pre-processing PIR constructions support updates for database entries and, correspondingly, the preprocessing material and client hints [56,64,68,86]. Recent work TAPIR [44] considers updatable *and* verifiable PIR where updates can be made by a malicious server. The update sequence can be arbitrary and adversarial; updates are not authenticated by other semi-honest parties in the system and there is no strict notion of the "right" database.

In the context of universal composability, modeling secure updates to cryptographic objects has been explored by Badertscher et al. [10]. Their work captures the notion of agile cryptography in UC, where the goal is to fully replace the code of a compromised functionality with newer, updated code.

2 Technical Overview

We now present an overview of our core contributions. This section captures the main technical insights in the body of this paper, particularly Sections 5 to 6.

Challenges to defining security of single-server VPIR. Prior works have proposed various single-server VPIR definitions based on extending existing PIR defintions in a game-based framework [36,38,43,44,77,82]. Though these definitions are all attempts to capture essentially the same security guarantees, many are incompatible with each other. In Section 3, we discuss one subtle but impactful difference: whether or not a client may store long-term state. Long-term state provides significant flexibility in protocol design, allowing for VPIR constructions that follow the PIR-with-preprocessing paradigm. However, a natural way of extending some VPIR definitions to support persistent client state yields a definition that does not guarantee security against sequential composition—meaning query privacy may only hold for a single query. Moreover, we observe that no existing definition which supports persistent client state simultaneously implies security against concurrently composed queries, or adaptive queries.

In the multi-server setting, recent work by Alon and Beimel [5] diverges from the property-based approach and instead defines security of (honest majority) multi-server VPIR using a real versus ideal paradigm. Roughly speaking, in the ideal functionality [5, Section 3] the client sends its desired index i, and each server sends a database, to the trusted party. The trusted party will check if the databases are equal and if so, return the requested entry at index i to the client.

There is no obvious analogue for this functionality in the single-server setting. To see why, we first recall the helpful observation that a cryptographically binding digest is necessary to achieve single-server VPIR [43]. Thus, it is natural to expect an *ideal* VPIR process to also have a mechanism to "bind" the server to a particular database. However, [5] does not enforce this binding property in the single-server setting; nothing prevents a single malicious server from sending a different database to the trusted party for every response. Beyond this, [5] assumes that the adversary does not maintain state between executions (see Remark 3.2 [5]), causing their definition to not imply secure composition.

Also, PIR is often used as a subroutine in a larger system (e.g., [3,4,6,7,9,27,29,52–55,58,59,67,71,72,76,78,80]), so it is critical to consider a client that is corrupted within the larger system and subsequently makes PIR queries as a sub-routine.

This work: ideal single-server verifiable PIR. We take a clean-slate approach and build a new, single-server VPIR functionality $\mathcal{F}_{\mathsf{VPIR}}$ in the framework of universal composability [17]. Our functionality captures the interaction between a client that wishes to retrieve an entry from a database being stored by a (potentially) malicious server. Our $\mathcal{F}_{\mathsf{VPIR}}$ functionality also protects

Algorithm 1 Simplified version of $\mathcal{F}_{\mathsf{VPIR}}$

This ideal process is parameterized by a session-ID sid.

1: **Init.** Server runs a one-time initialization of the protocol by sending a database DB, which $\mathcal{F}_{\mathsf{VPIR}}$ stores in memory. This ensures all clients that interact with this $\mathcal{F}_{\mathsf{VPIR}}$ have a consistent view of the database.
2: **Query.** To make a query, a client will provide an index i to $\mathcal{F}_{\mathsf{VPIR}}$.
3: **Response.** To respond to a client's query, the server will provide a full database DB'. $\mathcal{F}_{\mathsf{VPIR}}$ will check equality of DB' against the DB in memory. If the check passes, $\mathcal{F}_{\mathsf{VPIR}}$ returns the entry $\mathsf{DB}[i]$ to the client and otherwise gives the client an error message.

against a malicious client that submits malformed queries. We consider an adversarial protocol environment that can arbitrarily delay or drop network messages.

Algorithm 1 shows a simplified version of the interfaces in our single-server VPIR functionality $\mathcal{F}_{\mathsf{VPIR}}$; note that the full version can be found in Algorithm 3. Our $\mathcal{F}_{\mathsf{VPIR}}$ is a single definition that captures many of the intuitive properties of game-based definitions:

1. *Correctness.* For every initialized database DB and index i queried by an honest client, if the server responds honestly, the client will learn $\mathsf{DB}[i]$.
2. *Query privacy with abort.* For every initialized database DB and index i queried by an honest client, a malicious server learns nothing about the index i, even if the server learns whether the client aborts.
3. *Integrity.* For every initialized database DB and index i queried by an honest client, if the malicious server responds then client will recover either the correct entry $\mathsf{DB}[i]$ or abort.
4. *Database binding.* The server responds to all queries using the same database.
5. *Database proof of retrievability.* A successful response convinces the client that the server is storing the full database.

We now discuss several pertinent design choices in our $\mathcal{F}_{\mathsf{VPIR}}$.

- *Sequential and concurrent client queries.* We consider the scenario of clients submitting multiple queries in parallel, yet the queries arrive out-of-order at the server. The full version of our functionality (see Algorithm 3) addresses this by having the client and server attach a unique query identifier (qid) to each corresponding query/response pair.
- *Adversarially controlled network.* We consider an adversary that can arbitrarily delay or drop network messages.
- *Malicious digest generation.* The method to calculate the digest (i.e. does a client compute it themselves initially from a given database, or is it verified with a proof), how the digest is communicated with client (i.e. is it provided in person, or during a one-time interaction), and trust assumptions of it generation vary widely between different pre-existing VPIR protocols. To allow for flexibility of different schemes, our $\mathcal{F}_{\mathsf{VPIR}}$ instead abstracts away the digest from $\mathcal{F}_{\mathsf{VPIR}}$ and instead embeds it in the session identity of the running instance. We use advanced techniques from universal-composability to protect against malicious digest generation by enforcing that the database "inside" any given $\mathcal{F}_{\mathsf{VPIR}}$ functionality is consistent with the digest encoded in its session identity. This flexibility also allows us to simulate VPIR schemes that have per-client hints.

– *Support for client-side and server-side preprocessing.* Our definition allows for protocols in which either the client, the server, or both, utilize a preprocessing stage for efficiency gains. Both client-side preprocessing (e.g. [34,45,59,62,63,65,79,86]) and server-side preprocessing (e.g. [11,68,83]) have proven to allow for highly efficient semi-honest PIR schemes. Our definition is maximally flexible, supporting either paradigm.

Realizing our ideal single-server $\mathcal{F}_{\mathsf{VPIR}}$. To demonstrate the feasibility of our new definition, we show in Appendices C to D that our $\mathcal{F}_{\mathsf{VPIR}}$ can be realized by two protocols. As a simple check of correct design, we first show that $\mathcal{F}_{\mathsf{VPIR}}$ is realized by the trivial VPIR protocol in which the server sends the client the full database and the digest is given by a collision-resistant hash of the database. We also prove UC-security of an existing protocol: VeriSimplePIR by de Castro and Lee [36] (with slight modifications). Our full proofs of UC-security hold against adversaries that make static corruptions and may arbitrarily delay or drop network messages, but cannot modify or spoof them.

Single-server VPIR with updates. To demonstrate the broader utility of our $\mathcal{F}_{\mathsf{VPIR}}$ ideal process, we define and build an extension of verifiable PIR that we call Updatable Verifiable PIR (UVPIR).

Updates have been often considered in the PIR literature, but rarely in combination with verifiability. In some sense, the goals of updatable and verifiable PIR are at odds, because integrity appears to require a predictable, static database. The recent work TAPIR [44] proposes a game-based definition that attempts to unify both of these goals (for both single-server and multi-server PIR), but their single-server definition allows the malicious server to make arbitrary updates to the database that the client will accept and treat as the new database. Hence, integrity may be guaranteed on the "local scale" of an individual update, but their definition sacrifices on verifiability *across* a sequence updates, since it is no longer clear to clients how to convince themselves that the current database is "right" or "meaningful."

We fill this definitional gap in our UVPIR by adding a mechanism for clients to convince themselves that they are interacting with a VPIR server whose database has been derived from a sequence of valid updates that are authorized by permissioned writing clients. By enforcing that database updates must be authorized by permissioned clients, our definition enjoys both a simple form of access control and a meaningful "history" of updates. Permissioned updates are a natural setting in many real-world applications of PIR like password breach alerts [4,67,76,80], private DNS [84], and transparency logs [44, 55, 72, 78]; in all of these settings, some clients should have read-only query access whereas others have permission to modify the database.

Designing an ideal $\mathcal{F}_{\mathsf{UVPIR}}$. In Section 6, we build an ideal functionality $\mathcal{F}_{\mathsf{UVPIR}}$, which captures the process of a server that maintains a list of identities of authorized writing clients and a "version history" of databases representing a sequence of updates. Our $\mathcal{F}_{\mathsf{UVPIR}}$ allows clients to sync to the latest version of the database while guaranteeing that this version was indeed produced from a *sequence* of updates made by authorized clients. We consider write updates that do not change the size of the database.

We now provide an overview of the main design choices in $\mathcal{F}_{\mathsf{UVPIR}}$. The functionality is parameterized by a list of authorized writing clients and a description of the initial version of the database. $\mathcal{F}_{\mathsf{UVPIR}}$ stores the global state of all database versions as a directed acyclic graph (DAG) of $\mathcal{F}_{\mathsf{VPIR}}$ instances, where directed edges indicate an update issued by an authorized client from one database version to another. $\mathcal{F}_{\mathsf{UVPIR}}$ will maintain a local state for each client and allow clients to "sync" to a more recent version by progressing them down a path in the DAG. Our $\mathcal{F}_{\mathsf{UVPIR}}$ supports forking

(i.e. clients may diverge into distinct versions of the database if they submit updates in parallel), prevents cycles (i.e. updates that "revert" the database to an older version are still treated as a new version upstream), and supports merging (i.e. clients can "hop" to pre-existing database versions on the DAG long as they do not induce a cycle).

Designing a protocol to realize $\mathcal{F}_{\mathsf{UVPIR}}$. We build a protocol Π_{UVPIR} and formally show that it UC-realizes $\mathcal{F}_{\mathsf{UVPIR}}$. This protocol exists in a hybrid model with access to ideal functionalities $\mathcal{F}_{\mathsf{VPIR}}$, $\mathcal{F}_{\mathsf{NIZK}}$ for ideal NIZK proofs, and $\mathcal{G}_{\mathsf{Sig}}$ for ideal signatures. At a high-level, the server uses $\mathcal{F}_{\mathsf{NIZK}}$ to prove that a database update was performed correctly and the authorized client uses $\mathcal{G}_{\mathsf{Sig}}$ to "sign-off" on the update. Our protocol acts as a generic compiler, turning any VPIR implementation into a UVPIR instance.

3 On the Difficulty of Defining Verifiable PIR Under Composition

In this section we discuss the challenges that arise when defining verifiable PIR under arbitrary protocol composition. Since semi-honest PIR was first defined in 1998 by Chor, Goldreich, Kushilevitz and Sudan [32], the definition of PIR has evolved into numerous sub-types with different efficiency optimizations and security guarantees (e.g. preprocessing PIR, secret-key PIR, doubly-efficient PIR, verifiable PIR). To the best of our knowledge, however, there is no systematization and formal analysis of the (in)security of PIR definitions under *protocol composition*. We address this gap below.

We begin our discussion on the definition of PIR with the seminal work of [32]. The definition put forth in this work is concise, elegant, and includes two properties, correctness and privacy. Correctness is defined in the usual cryptographic sense, and privacy is defined as distributional indistinguishability between two different (sequences of) queries.

While [32] considers a semi-honest setting, we observe that their definition also implies some notion of malicious security. This is because Chor et al. defines PIR as a single-round protocol; the server sees a single client message per query, which does not depend on any server-produced values. Therefore, a maliciously behaving server cannot learn anything beyond what a (semi-)honest server learns. We also observe that Chor et al.'s notion of sequential query privacy also automatically implies query privacy when queries are made in *parallel* or *concurrently*. This is because Chor et al. stipulate that the Query algorithm be *memoryless*—no persistent client state can be included as an input. Indeed, composed multi-query privacy is reducible to single-shot query privacy via a hybrid argument, by having the adversary simulate queries that are statistically indistinguishable from a real client's query.

A well-known drawback of Chor et al.'s definition is that it requires the server to touch every entry of the database on each query [11]. Modern PIR protocols (e.g. [34, 45, 59, 62, 63, 65, 79, 86]) often circumvent this lower bound by modifying the definition to allow some form of preprocessing, most notably in the *client preprocessing* model [34], in which the client may store a long-term hint. In the client preprocessing model, the Reconstruct algorithm is no longer memoryless—clients store a long-term hint that is used as an input to the Reconstruct algorithm. However, since the Query algorithm remains stateless, PIR in the client preprocessing model allows for the same composable security guarantees as Chor et al.'s definition.

Given the power of the client preprocessing model, it is natural to want to define *verifiable* PIR in the same model. However, the presence of a malicious server complicates the secure composition property implied by semi-honest PIR in the client preprocessing model. A malicious server who

learns if the client aborts while reconstructing their output may reveal something about the client's long-term secrets, which may break query privacy.

It is non-trivial to support stateful clients in a game-based VPIR definitions while maintaining secure composition. We elaborate on this in Appendix B.1. In Appendix B.2, we show that the only prior VPIR definition to allow the client preprocessing model (De Castro and Lee's [36]) guarantees secure composition under sequential composition, but *not* under concurrent composition.

Another consideration when defining VPIR is whether the client's sequence of queries are chosen statically at the beginning of time, or whether they are adaptively, adversarially chosen. In Appendix B.3, we show that the VPIR definition of [36], which assumes static queries, loses its sequential composition guarantee when relaxed to allow for adaptive queries.

4 Preliminaries

In this section, we provide an overview of our threat model, a short primer on universally composable (UC) security, and the description of an "internal code" library from [21, 24]. In Appendix A, we include three UC functionalities that we use as building blocks in our protocols: a common reference string ($\mathcal{G}_{\mathsf{CRS}}$), digital signatures ($\mathcal{G}_{\mathsf{Sig}}$), and non-interactive zero-knowledge proofs ($\mathcal{F}_{\mathsf{NIZK}}$).

Threat model. We consider private information retrieval in the setting with a single server and arbitrarily many clients. The adversary can statically corrupt the server and any number of clients.

In the face of this adversary, a verifiable PIR scheme should provide honest clients with malicious security up to abort. Note that honest clients never need to talk to each other, though they are also not forbidden from doing so.

Additionally, we allow the adversary to delay or drop messages arbitrarily, but not to inject new spoofed messages. We implicitly assume throughout this work that all communications between honest parties use an authenticated channel, a technique used by prior works in the UC framework (e.g. [17, Footnote 1] and [22, §2.3]))

UC primer. We provide a short overview of universally composable (UC) security here and refer interested readers to [17, 20] for more details. UC security is a special case of simulation-based security in which the real world execution of a protocol π is compared with an idealized abstraction $\mathcal{F}$. The UC security framework gives two special powers to the distinguisher, called the *environment* $\mathcal{Z}$: the ability to choose inputs and view outputs of the protocol/functionality, and the ability to interact with the real-world adversary $\mathcal{A}$ but with the connection monitored and potentially changed by the simulator. A protocol Π is deemed to be a *UC-realization* of functionality $\mathcal{F}$ if there exists an efficient simulator Sim such that no environment can tell whether it is interacting with π or with $\mathcal{F}$ mediated by Sim.

Execution within the UC model follows some distinctive rules, yet it is expressive enough to capture most realistic computational systems. Turing machines interact with each other; the receiver is dynamically created if it does not exist. Each machine, and each protocol instance, are identified by a unique identifier and the machine's program code. By convention, identities consist of a session ID (sid) and party ID (pid).

UC security allows one to "think local and act global." The UC model of execution considers only a single instance of the protocol under consideration, leading to relative simplicity of the specification and analysis. Still, the UC framework provides a generic composition theorem. Suppose that protocol Π UC-realizes $\mathcal{F}$, and there exists another "hybrid" protocol ρ that makes (perhaps many) subroutine calls to functionality $\mathcal{F}$. Now, consider the protocol $\rho^{\mathcal{F}\to\Pi}$ that replaces all instances of the ideal

Algorithm 2 Functionality $\mathcal{G}_{\mathsf{lib}}$ (adapted from [21, 24])

1: **Store:** Upon receiving an input $(\texttt{Store}, \mathsf{Identifier}, \mathsf{Code})$ from the simulator:
   1. Store $(\mathsf{Identifier}, \mathsf{Code})$, overwriting any previously stored code with the same identifier.
2: **Retrieve:** Upon receiving an input $(\texttt{Retrieve}, \mathsf{Identifier})$ from a party P:
   1. Output $(\mathsf{Identifier}, \mathsf{Code})$ to P, if such a tuple is stored.

functionality $\mathcal{F}$ with the real protocol Π. The composition guarantees that the instantiation $\rho^{\mathcal{F}\to\Pi}$ is "just as secure" as the ρ itself, in the same sense defined above. In this case we say that $\rho^{\mathcal{F}\to\Pi}$ UC-emulates ρ.

Internal code library. Looking ahead, one challenge in our modeling is that the ideal functionality $\mathcal{F}_{\mathsf{VPIR}}$ must be convinced that a database is consistent with its own digest, but the digest is based on the real-world instantiation that the ideal functionality does not know. Canetti et al. [21, 24] show how to bridge this gap with "internal" or "don't-care code"; that is, some portion of the real-world code that can be fetched and run by the ideal functionality $\mathcal{F}$ or simulator Sim. We provide the library functionality in Algorithm 2.

5 Verifiable PIR

We present the ideal UC-functionality, $\mathcal{F}_{\mathsf{VPIR}}$, for a single-server verifiable PIR in Algorithm 3. We first discuss an ideal execution of our $\mathcal{F}_{\mathsf{VPIR}}$ functionality. Then, we will briefly discuss the two instantiations of our $\mathcal{F}_{\mathsf{VPIR}}$ functionality.

5.1 Defining VPIR

The ideal functionality consists of three main interactions: the initial setup, a client initiating a query, and a server authoring a response to the query.

Parameters. The functionality is parameterized by an alphabet σ of valid database entries, and a database size N. Without loss of generality, we assume the session ID (sid) contains the party ID of the intended server, as well as the real-world digest material that corresponds to some underlying database.

Intialization. The initialization process is a one-time setup that takes in a database from the server and checks if it is consistent with the digest that is encoded in the sid. The initialization process fails if `Init` has been successfully run before. Although the digest is contained in the session ID, the *structure* of the digest is abstracted out of the ideal $\mathcal{F}_{\mathsf{VPIR}}$ functionality. Instead, we rely on *internal code* from the global code library $\mathcal{G}_{\mathsf{lib}}$ to validate digest and database connection.[4] Once the database is validated, it is leaked to the simulator. This is because we do not require the digest to provide a hiding property, and so it may reveal arbitrary information about the database. The functionality stores the database, and the server is notified of success (or failure if the internal code rejects).

Query. A query is initiated by a client who provides as inputs a query id (qid), and the index i they wish to query. We impose structure on the qid, requiring it to include the sid, the client's identity,

[4] We use a modeling technique of Canetti et al. to allow the functionality $\mathcal{F}_{\mathsf{VPIR}}$ to run a piece of adversarial internal code $\mathcal{V}$ obtained from the simulator (see [24, §2.2]).

Algorithm 3 Functionality $\mathcal{F}_{\mathsf{VPIR}}$ for verifiable PIR for database size N and alphabet Σ

This functionality has a session ID sid that is assumed to contain the party ID $\mathbf{S}$ of the intended database server. It is initialized with a variable $\mathsf{initialized} \leftarrow \mathsf{false}$.

1: **Init:** Upon receiving $(\texttt{Init}, \mathsf{sid}, \mathsf{DB} \in \Sigma^N)$ from server $\mathbf{S}$ do:
   i. If $\mathsf{initialized} = \mathsf{true}$ or $\mathbf{S}$ does not match the party ID in sid, end execution. // *Init is run once with the prescribed server*
   ii. Send $(\texttt{Retrieve}, \mathsf{DigestVerifyCode})$ to $\mathcal{G}_{\mathsf{lib}}$ to obtain the internal code $\mathcal{V}$ and compute $\mathsf{state} \leftarrow \mathcal{V}(\mathsf{sid}, \mathsf{DB})$. If $\mathsf{state} = \texttt{Reject}$, output $(\texttt{InvalidDigest}, \mathsf{sid})$ to $\mathbf{S}$.
   iii. Otherwise, send $(\texttt{ReportInit}, \mathsf{sid}, \mathsf{DB})$ to the simulator Sim. // *Leak* DB *to the simulator*
   iv. Upon receiving $(\texttt{ReportInit}, \mathsf{sid}, \texttt{OK})$ from Sim: set $\mathsf{initialized} \leftarrow \mathsf{true}$, store DB, initialize an empty dictionary Q, and return $(\texttt{Initialized}, \mathsf{sid})$ to the server $\mathbf{S}$. // *Bind the functionality to a single database*

2: **Query:** Upon receiving $(\texttt{Query}, \mathsf{qid}, i \in [N])$ from client $\mathbf{C}$ with $\mathbf{C}$, where $\mathsf{qid} = (\mathsf{sid}, \mathbf{C}, \mathsf{nonce})$, do:
   i. If the query ID qid has been used in a previous **Query** call, end execution. // *Query IDs are unique*
   ii. Otherwise, set $Q[\mathsf{qid}] \leftarrow i$ and send $(\texttt{ReportQuery}, \mathsf{qid})$ to Sim. When the simulator responds with $(\texttt{DeliverQuery}, \mathsf{qid})$, output $(\texttt{Query}, \mathsf{qid})$ to $\mathbf{S}$.

3: **Response:** Upon receiving $(\mathsf{Response}, \mathsf{qid}, \mathsf{DB}^* \in \Sigma^N \cup \{\perp\})$ from server $\mathbf{S}$ with $\mathbf{S}$ do:
   i. End execution if (a) $\mathbf{S}$ is honest and $\mathsf{initialized} = \mathsf{false}$, (b) $\mathbf{S}$ did not previously receive output $(\texttt{Query}, \mathsf{qid})$, (c) qid has been used in a previous $\texttt{Response}$ call, or (d) $\mathbf{S}$ is honest and $\mathsf{DB}^* \neq \mathsf{DB}$.
   ii. Otherwise, send $(\texttt{ReportResponse}, \mathsf{qid})$ to Sim. Wait for Sim to respond with $(\texttt{DeliverResponse}, \mathsf{qid})$.
   iii. If $\mathbf{S}$ is malicious and either $\mathsf{initialized} = \mathsf{false}$ or $\mathsf{DB}^* \neq \mathsf{DB}$, return $(\texttt{VerificationFail}, \mathsf{qid})$ to $\mathbf{C}$.
   iv. Otherwise, set $i \leftarrow Q[\mathsf{qid}]$ and return $(\texttt{Answer}, \mathsf{qid}, \mathsf{DB}[i])$ to $\mathbf{C}$.

and a nonce to ensure uniqueness. The functionality stores i in a dictionary indexed by qid, which will be used to ensure the client's eventual output from Response is correct. We allow the simulator an opportunity to prevent or delay the server's output. This allows the ideal world to mimic the the real-world adversary's ability to arbitrarily delay messages. The server's output contains the qid, which in turn contains the client's identity, but does not reveal anything about the index the client is requesting, providing query privacy (with abort).

Response. We define Response as its own method (rather than combining Query and Response into one) to mimic the way semi-honest PIR is defined, with separate query and response algorithms. However, our Query and Response methods allow much more flexible protocols: they may involve multiple rounds of communication, and may take in and store long-term state, such as client hints or server preprocessing material.

The Response functionality is triggered by the server, who inputs the qid of a previous client's query, along with a full database. At this point, the functionality performs four checks and ends execution if any hold true. The purpose and consequences of these checks are as follows:

– Check (a) ensures that, for honest servers, Init must have been successfully run before any successful responses can be run. If the server is malicious, execution is allowed to continue even if $\mathsf{initialized} = \mathsf{false}$. This is to allow the client to potentially obtain a `VerificationFail` output (step iii ensures that this is the only possible output a client have have in this case). This makes it possible to simulate a malicious server who does not know a valid DB and simply responds to queries with garbage, causing honest clients' verification to fail.
– Check (b) ensures each Response must correspond to a completed Query.
– Check (c) ensures that no Query can receive more than one Response.
– Check (d) binds an honest server to the database used during Init, capturing database binding. If the server is malicious but $\mathsf{DB}^* \neq \mathsf{DB}$, execution may continue, but step iii ensures the client will only output `VerificationFail`. Taken together, this enforces the proof of retrievability property: a server must remember the database used at initialization to successfully respond to a query.

The simulator is allowed to delay the client's output, mirroring the adversary's ability to delay message delivery. Finally, the client is given their output: either `VerificationFail`, if the server is malicious and did not initialize, or initialized with a different database (this is the malicious versions of checks (a) and (d) from step i), or $\mathsf{DB}[i]$ otherwise. This ensures correctness and integrity.

5.2 Realizing $\mathcal{F}_{\mathsf{VPIR}}$

We provide two protocols that realize our $\mathcal{F}_{\mathsf{VPIR}}$ functionality. As a proof-of-concept, we first realize it with a trivial PIR where the digest is a collision resistant hash of the database, and the server responds with the entire database on every query. The full protocol and proof is described in Appendix C.

Our functionality can also be realized by non-trivial protocols with sublinear communication. We demonstrate this by adapting the VeriSimplePIR protocol of de Castro and Lee [36] and proving its UC security. We use a simplification of the VeriSimplePIR protocol described by Rathee, Lee, and Popa [77]. We additionally modify the protocol to require the server to send the client a straightline-extractable proof of knowledge to the preimage of the database commitment. This modification is necessary, as the SIS commitment used in vanilla VeriSimplePIR is not known to be straightline extractable; therefore the typical knowledge extractor for SIS commitments cannot be used by a UC simulator. The full protocol and proof is contained in Appendix D.

6 Verifiable PIR with Updates

We present the ideal UC-functionality of a single-server VPIR with updates, $\mathcal{F}_{\mathsf{UVPIR}}$, in Section 6. We discuss the ideal functionality in this section, and describe a protocol to instantiate it in Appendix E.

The $\mathcal{F}_{\mathsf{UVPIR}}$ functionality extends $\mathcal{F}_{\mathsf{VPIR}}$ to allow the database to be updated. This goal is simple to state, however it is not obvious how to define it in a useful way. If any party can make updates to the database, then a key property of verifiable PIR—database binding—is lost: a malicious server can "update" the database to whatever they choose before a client makes a query. Separately, determining the most up-to-date database can prove complicated. If UVPIR is defined to ensure that all clients can query a single most-recent database, the protocol must solve a consensus problem amongst clients that may require heavy cryptographic hammers like public bulletin boards.

Algorithm 4 $\mathcal{F}_{\mathsf{UVPIR}}$

This functionality is parameterized by the database size N, and uses an sid which contains a unique nonce, the server's identity S, a set $\mathsf{authClients}$ of party IDs of authorized write clients, and the session ID ssid_0 of a $\mathcal{F}_{\mathsf{VPIR}}$ instance. On creation, the functionality is initialized a variable $\mathsf{initialized} \leftarrow \mathsf{false}$, empty map $\mathsf{clientView}$, and empty DAG $\mathsf{updateGraph}$. On the first input from any new client $\mathbf{C}_i$, set $\mathsf{clientView}[\mathbf{C}_i] \leftarrow \texttt{INITIAL}$.

1: **Init:** Upon receiving $(\texttt{Init}, \mathsf{sid}, \mathsf{DB}_0)$ from $\mathbf{S}$:
   1. If $\mathsf{initialized} = \mathsf{true}$, end execution. Otherwise, parse ssid_0 from sid and send $(\texttt{Init}, \mathsf{ssid}_0, \mathsf{DB}_0)$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}_0}$ on behalf of $\mathbf{S}$. If $(\texttt{Initialized}, \mathsf{ssid})$ is not outputted, end execution. // *Initialize the first $\mathcal{F}_{\mathsf{VPIR}}$ instance*
   2. Send $(\texttt{ReportInit}, \mathsf{sid}, \mathsf{DB}_0)$ to the simulator, and await response $(\texttt{ReportInit}, \mathsf{sid})$. // *Leak initial DB to the simulator*
   3. Add a node $(\texttt{INITIAL}, \mathsf{ssid}_0, \mathsf{DB}_0)$ to $\mathsf{updateGraph}$. // *$\mathsf{updateGraph}$ represents the update history. Elements contain a unique label, the sid of a $\mathcal{F}_{\mathsf{VPIR}}$ functionality, and the corresponding full database.*
   4. Set $\mathsf{initialized} \leftarrow \mathsf{true}$ and output $(\mathsf{initialized}, \mathsf{sid})$ to $\mathbf{S}$.
2: **Query:** Upon receiving $(\texttt{Query}, \mathsf{sid}, \mathsf{qid}, i \in [N])$ input from a client $\mathbf{C}$:
   1. Look up the client's current state $\mathsf{label} \leftarrow \mathsf{clientView}[\mathbf{C}]$ and find node $(\mathsf{label}, \mathsf{ssid}, \cdot)$ in $\mathsf{updateGraph}$
   2. Send $(\texttt{Query}, \mathsf{ssid}, \mathsf{qid}, i)$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}}$ on behalf of $\mathbf{C}$, and forward any output to the intended recipient
3: **Response:** Upon receiving $(\texttt{Response}, \mathsf{sid}, \mathsf{qid}, \mathsf{DB} \in \Sigma^N \cup \{\perp\})$ from $\mathbf{S}$, where $\mathsf{qid} = (\mathsf{sid}, \mathbf{C}, \mathsf{nonce})$:
   1. Parse the client ID $\mathbf{C}$ from qid, look up $\mathsf{label} \leftarrow \mathsf{clientView}[\mathbf{C}]$, and the node $(\mathsf{label}, \mathsf{ssid}, \cdot)$ in $\mathsf{updateGraph}$
   2. Forward the message $(\texttt{Response}, \mathsf{ssid}, \mathsf{qid}, \mathsf{DB})$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}}$, and forward any outputs to the intended recipients.
4: **Update:** Upon receiving $(\texttt{Update}, \mathsf{sid}, \mathsf{uid}, \Delta \in \Sigma^N)$ from a client $\mathbf{C}$: // *Note, the update vector Δ will typically be sparse, corresponding to how much of the database the client is changing.*
   1. End execution if: (a) $\mathsf{initialized} = \mathsf{false}$, (b) $\mathbf{C} \notin \mathsf{authClients}$, or (c) uid has been used in a previous update.
   2. If $\mathbf{C}$ is honest, look up $\mathsf{label} \leftarrow \mathsf{clientView}[\mathbf{C}]$. If $\mathbf{C}$ is corrupted, send $(\texttt{MaliciousUpdateState}, \mathsf{uid}, \mathbf{C})$ to the simulator and await response $(\texttt{MaliciousUpdateState}, \mathsf{uid}, \mathbf{C}, \mathsf{label})$. // *Honest clients can only update off of their current state, malicious clients may attempt to update off of any state.*
   3. Find node $(\mathsf{label}, \mathsf{ssid}, \mathsf{DB})$ in $\mathsf{updateGraph}$, or end execution if none exists.
   4. Compute the updated database $\mathsf{DB}' \leftarrow \mathsf{DB} + \Delta$
   5. Send $(\texttt{Retrieve}, \mathsf{SIDUpdateCode})$ to $\mathcal{G}_{\mathsf{lib}}$ to obtain code $\mathcal{U}$. Compute $\mathsf{ssid}' \leftarrow \mathcal{U}(\mathsf{ssid}, \mathsf{DB}, \Delta)$ // *ssid' is a $\mathcal{F}_{\mathsf{VPIR}}$ session ID that corresponds to the updated database.*
   6. If ssid' is not part of in any node in $\mathsf{updateGraph}$, send $(\texttt{Init}, \mathsf{ssid}', \mathsf{DB}')$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}'}$ on behalf of $\mathbf{S}$. If $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}'}$ outputs $(\texttt{InvalidDigest}, \mathsf{ssid}')$, end execution. // *Initialize a new $\mathcal{F}_{\mathsf{VPIR}}$ instance. Inside, $\mathcal{F}_{\mathsf{VPIR}}$ checks that the digest is consistent with the database.*
   7. If a node $v = (\cdot, \mathsf{ssid}', \mathsf{DB}'')$ already exists in $\mathsf{updateGraph}$, check that DB' is identical to DB''. If not, end execution. // *For initialized $\mathcal{F}_{\mathsf{VPIR}}$ instances, ensure DB' matches the database that previously passed the digest check.*
   8. Send $(\texttt{UpdateRequest}, \mathsf{uid}, \mathbf{C}, \mathsf{ssid}', \Delta)$ to the simulator and await response $(\texttt{UpdateStatus}, \mathsf{uid}, \mathsf{label}')$. Give output $(\texttt{UpdateReject}, \mathsf{sid}, \mathsf{uid})$ to $\mathbf{C}$ and end execution if: (a) $\mathsf{label}' = \perp$, (b) a node $(\mathsf{label}', \mathsf{ssid}'', \cdot)$ exists in $\mathsf{updateGraph}$ with $\mathsf{ssid}' \neq \mathsf{ssid}''$, or (c) a path exists in $\mathsf{updateGraph}$ from $(\mathsf{label}', \mathsf{ssid}', \mathsf{DB}')$ to $(\mathsf{label}, \mathsf{ssid}, \mathsf{DB})$.
   9. Add a node $(\mathsf{label}', \mathsf{ssid}', \mathsf{DB}')$ if one does not exist, add a directed edge from $(\mathsf{label}, \mathsf{ssid}, \mathsf{DB})$ to $(\mathsf{label}', \mathsf{ssid}', \mathsf{DB}')$ to $\mathsf{updateGraph}$, and set $\mathsf{clientView}[\mathbf{C}] \leftarrow \mathsf{label}'$.
   10. Output $(\texttt{UpdateComplete}, \mathsf{sid}, \mathsf{uid}, \mathbf{C}, \Delta)$ to $\mathbf{S}$, and give delayed output $(\texttt{UpdateComplete}, \mathsf{sid}, \mathsf{uid})$ to $\mathbf{C}$. // *Delayed outputs allow for adversarial network delay*
5: **Sync:** Upon receiving $(\texttt{Sync}, \mathsf{sid})$ from a client $\mathbf{C}$:
   1. Send $(\texttt{ClientSync}, \mathbf{C})$ to the simulator and await response $(\texttt{SyncClientState}, \mathbf{C}, \mathsf{label}')$.
   2. Let $\mathsf{label} \leftarrow \mathsf{clientView}[\mathbf{C}]$. If there is a directed path (possibly empty) in $\mathsf{updateGraph}$ from label to label', set $\mathsf{clientView}[\mathbf{C}] \leftarrow \mathsf{label}'$ and output $(\texttt{Synced}, \mathsf{sid})$ to $\mathbf{C}$. Otherwise, end execution.

Our approach is to only allow updates from a publicly known subset of authorized clients. A malicious server must corrupt one of the authorized clients to make malicious changes to the database. We also do not guarantee a single linear chain of updates, which avoids having to solve the consensus problem. Instead, our definition allows authorized client to create forks in the path of updates. When a fork is created, the server is able to lead honest clients down the branch of their choosing. In some applications, e.g. privately fetching a Git repository, allowing forks is a desirable property. When forks are not desired, protocols can be written to allow an honest server to enforce a linear update chain (we provide such a protocol in Appendix E). We describe $\mathcal{F}_{\mathsf{UVPIR}}$ at a high level below, and give low-level details in Section 6.1.

Internally, $\mathcal{F}_{\mathsf{UVPIR}}$ represents the different update states of the database as a collection of 'version controlled' $\mathcal{F}_{\mathsf{VPIR}}$ instances, stored as a directed acyclic graph (DAG). This technique is inspired by Badertscher, Ciampi, and Kiayias' [10] approach to defining "agile cryptography," in which cryptographic primitives used in a larger protocol can be updated via software updates.[5] All $\mathcal{F}_{\mathsf{VPIR}}$ instances are run locally inside $\mathcal{F}_{\mathsf{UVPIR}}$, and cannot be directly interacted with by the $\mathcal{F}_{\mathsf{UVPIR}}$ parties or the environment. Inside $\mathcal{F}_{\mathsf{UVPIR}}$, each client is associated with a pointer to a node in the DAG, corresponding to the most recent version of the database they know of. We denote $\mathcal{F}_{\mathsf{VPIR}}$ session IDs as ssid, as sid is used for the unique session ID of $\mathcal{F}_{\mathsf{UVPIR}}$. Authorized clients may create updates, which are represented as new leaf nodes in the DAG that contain the ssid of a new $\mathcal{F}_{\mathsf{VPIR}}$ instance. Any client may "sync" to a more recent version of the database. Internally, this is represented by moving the client's pointer to another node in the DAG downstream of their previous node.

The same logical database may be represented in multiple nodes in the DAG. Each node is required to have a unique label along with a $\mathcal{F}_{\mathsf{VPIR}}$ ssid, but two nodes may contain the same ssid, or two ssid's may correspond to the same underlying database. This allows updates to revert the database to a previous state without creating a cycle in the DAG. Cycles would be problematic; the topological ordering of a DAG is used to determine which updates are more recent.

6.1 $\mathcal{F}_{\mathsf{UVPIR}}$ details

We are now ready to describe the $\mathcal{F}_{\mathsf{UVPIR}}$ functionality, which is written in full detail in Section 6.

Parameters. $\mathcal{F}_{\mathsf{UVPIR}}$ is parameterized by a database size N and a type Σ for each database element. We assume that Σ forms a group, and denote the group operation as addition. As with $\mathcal{F}_{\mathsf{VPIR}}$, a $\mathcal{F}_{\mathsf{UVPIR}}$ instance is identified with an sid that is assumed to contain certain information—in particular, the set of authorized updating clients and the session ID ssid_0 of some initial VPIR instance.

Activation. When the $\mathcal{F}_{\mathsf{UVPIR}}$ functionality is first activated, it initializes state that will used throughout execution: a flag to indicate if initialization has been completed, a directed acyclic graph (DAG) $\mathsf{updateGraph}$ to track the connections between VPIR instances as updates are executed, and a map $\mathsf{clientView}$ which stores for each client a pointer to a node of the DAG, representing what each client believes is the most recent state.

[5] Badertscher et al. [10] capture the idea of updating the code of a cryptographic module from a trusted source when security is compromised. While both our $\mathcal{F}_{\mathsf{UVPIR}}$ and the ideal functionality from [10] represent a series of updates using a DAG, $\mathcal{F}_{\mathsf{UVPIR}}$ considers updating only a part of the data rather than the entire file. We also consider a set of authorized writing clients rather than a single trusted source. Regardless, we view this relationship to [10] as a positive; it speaks to the usefulness of designing a "version-control-like" functionality that supports updates over modular ideal functionalities.

Initialization. Initialization is a one-time setup procedure invoked by the server, who inputs the initial (root) database. A root VPIR is initialized with the full database by running $\mathcal{F}_{\mathsf{UVPIR}}$ `Init` protocol. As in $\mathcal{F}_{\mathsf{VPIR}}$, the database is leaked to the simulator, as the digest may not hide the contents of the database.

Query and Response. Query and Response are mainly wrappers for the Query and Response methods in $\mathcal{F}_{\mathsf{VPIR}}$. However, since each $\mathcal{F}_{\mathsf{UVPIR}}$ contains many $\mathcal{F}_{\mathsf{VPIR}}$ instances, we have $\mathcal{F}_{\mathsf{UVPIR}}$ determine which $\mathcal{F}_{\mathsf{VPIR}}$ instance the client and server's inputs should be routed to. This is done based on $\mathsf{clientView}$, which stores the most recent $\mathcal{F}_{\mathsf{VPIR}}$ ssid each client has seen. The result is that clients can only make queries to the most recent $VPIR$ instance in their view. $\mathcal{F}_{\mathsf{UVPIR}}$ does not explicitly check that $\mathsf{initialized} = \mathsf{true}$ during Query and Response. That is handled at the level of $\mathcal{F}_{\mathsf{VPIR}}$; the query or response will fail if the $\mathcal{F}_{\mathsf{VPIR}}$ instance has not been initialized.

Update. Updates allow authorized clients to make changes to the database. Updates are invoked by a client who inputs an update ID uid and a vector of changes Δ it wishes to make to the database. This allows updating many rows of the database, or rewriting it entirely.

Internally, $\mathcal{F}_{\mathsf{UVPIR}}$ represents updates as adding a new node to the DAG, and an edge from the existing graph to the new node. The functionality ensures that any honest client $\mathbf{C}$ may only add an edge originating from the node contained in $\mathsf{clientView}[\mathbf{C}]$; i.e. honest clients can only update off of what they believe to be the most recent database. Malicious clients, however, may create edges originating from any node in the DAG. This allows real-world malicious authorized clients to "sign off" on an update from any state.

The updated database, DB', is computed by adding Δ to the DB from the client's current $\mathcal{F}_{\mathsf{VPIR}}$ ssid, as defined by the pointer in $\mathsf{clientView}$ (or, for malicious clients, the DB at a DAG node of the simulator's choosing). The $\mathcal{F}_{\mathsf{VPIR}}$ ssid of the updated database is computed by calling a piece of internal code that takes in DB and Δ. Similar to the internal code in $\mathcal{F}_{\mathsf{VPIR}}$, this trick allows us to abstract away real-world details of the structure of the digest. If the new ssid' is not in $\mathsf{updateGraph}$, a new $\mathcal{F}_{\mathsf{VPIR}}$ instance is initialized with ssid' and database DB'. Otherwise, the functionality checks that DB' matches the one stored in the existing $\mathcal{F}_{\mathsf{VPIR}}$ with session if ssid'. In either case, the functionality asks the simulator to provide a label for the updated node. Labels are used to differentiate nodes that may have the same ssid and database. If the label corresponds to a node that does not exist in $\mathsf{updateGraph}$, the ideal functionality creates a new node containing ssid', the updated database DB', and the label. It is possible for the simulator to choose a label that already exists in the graph (allowing merges), however the label cannot be upstream of the client's current node (preventing cycles). Finally, an directed edge is created between the old node and the new node.

Sync. Sync allows the functionality to move a client's view to a more up-to-date version of the database. The simulator is allowed to choose the new node for the client's pointer; however it must be downstream of the client's current node. This enables simulating a malicious server who chooses to hide the existence of updates from the client, which we cannot prevent without solving a conensus problem. Internally, the functionality updates the $\mathsf{clientView}$ pointer, and then simply outputs the message `Synced` to the client.

6.2 Π_{UVPIR}: realizing $\mathcal{F}_{\mathsf{UVPIR}}$ with $\mathcal{F}_{\mathsf{VPIR}}$

Our Π_{UVPIR} protocol, described in Appendix E, exists in a hybrid model with access to local functionalities $\mathcal{F}_{\mathsf{VPIR}}$ and $\mathcal{F}_{\mathsf{NIZK}}$, and a global functionality $\mathcal{G}_{\mathsf{Sig}}$.

The Π_{UVPIR} protocol is parameterized by a database size N, a list of authorized clients `AuthClients`, and the sub-session ID of a $\mathcal{F}_{\mathsf{VPIR}}$ instance that contains the initial version of the database. The server stores the update history of the database as a linked list. This ensures that all updates are linear, and no branching or merging can occur. Internally, each node in the linked list contains the session id of a $\mathcal{F}_{\mathsf{VPIR}}$ instance, a *version number*, which is simply the depth from the root node, and for all non-root nodes, a signature from an authorized client. We will describe the signature later.

To initialize the protocol, the server provides an initial database that is consistent with the $\mathcal{F}_{\mathsf{VPIR}}$ root functionality. The server also maintains a list of each client's current ssid, which implicitly corresponds to the database version. To perform VPIR queries/responses, the client/server forwards the appropriate messages to the VPIR instance corresponding to the ssid of the client's current state.

When a client is activated, they initialize a local state by storing the publicly known subsession ID of the root $\mathcal{F}_{\mathsf{VPIR}}$ functionality. To handle updates, an authorized writing client and server engage in a multi-round protocol. Similar to $\mathcal{F}_{\mathsf{VPIR}}$, we use unique update identifiers uids to ensure that each query request/response is appropriately matched. Upon the client issuing an update request to the server, the server applies the requested update to the client's current version of the database, and produces a new ssid using internal code. Using internal code allows the protocol to abstract away the implementation-specific details of the digest. Furthermore, it allows the new VPIR digest to be computed more efficiently in certain cases. In many VPIR protocols (e.g. [36, 38, 77]), the digest has a homomorphic property that allows new digests to be more efficiently computed given Δ. Using internal code allows these implementation-specific optimizations to be optionally used by the protocol, while still supporting $\mathcal{F}_{\mathsf{VPIR}}$ implementations that do not have such optimizations. The server initializes the new $\mathcal{F}_{\mathsf{VPIR}}$ instance with the updated database. Finally, the server uses $\mathcal{F}_{\mathsf{NIZK}}$ to send a proof to the client that proves the new digest and updated database was computed correctly, according to Δ. The client, upon receiving and verifying the server's proof, uses $\mathcal{G}_{\mathsf{Sig}}$ to produce a signature σ on a tuple m containing the old ssid, the new ssid, the client's own identity, and the new version number (which increments by one on each update). The version number allows differentiating between two nodes that have the same ssid. The server appends the client's (m, σ) pair to the linked linked list and updates their view of the client's state.

When a client requests to sync, the server locates the ssid corresponding to the client's current version within the linked-list and sends the client the chain of (m, σ) pairs in the linked list between the client's current version and the most up-to-date version. The client verifies that each of the signatures was signed by an authorized client, and checks that the ssids form a consistent path starting from their stored ssid. Upon verifying the validity of the path, the client updates their local state to the most recent ssid.

We remark that it is possible to avoid sending all (m, σ) pairs in the linked list by replacing the linked list with a skiplist-like data structure. In this version, on each update an authorized client would sign multiple m tuples, linking the new ssid to multiple previous ssids. This creates "fast lanes" in the list. A syncing client could then be given a shorter path from their current node to the most recent node. More precisely, if a syncing client is k versions out of date, the skiplist structure would allow them to sync while only downloading and verifying $O(\log k)$ signatures, instead of k.

7 Open Problems

We identify two open problems inspired by this work:

Connection to proofs of retrievability. Our VPIR definition is written to immediately imply a UC proof of retrievability (PoR) protocol (a property we share with [30]). Other VPIR definitions (e.g. [36, 38]) require related properties about database extractability, though they allow for rewindable extractors rather than the straight-line simulation required by UC. This gap is the reason our VPIR protocol in Appendix D requires a generic proof of knowledge that is not present in the original construction of [36]. It would be interesting to find other ways to adapt non-UC VPIR constructions to satisfy UC security. Further work exploring the connection between VPIR and PoR is also an interesting direction.

Verifiable Doubly-Efficient PIR in UC. *Doubly-efficient PIR* (DEPIR) is an restriction of PIR that requires $o(N)$ communication and computation per query for both client and server, with the help of a super-linear preprocessing stage [14, 23, 66, 68]. Constructing a verifiable DEPIR is an interesting problem that has been addressed by only one work ([42]), which applies a generic compiler to transform semi-honest (DE)PIR schemes to verifiable (DE)PIR, in a non-UC model. It is not obvious how to construct a doubly-efficient protocol that satisfies our UC VPIR definition, even if we had a generic compiler from semi-honest PIR to UC VPIR. The issue arises from having the environment provide the database for the server's response—a design choice we made to guarantee the protocol is a proof of retrievability. An honest server must reject invalid databases during the response protocol, which seems to inherently require some linear-time check, precluding double-efficiency. It would be interesting to find a UC VPIR definition that allows for double-efficiency while also enforcing a proof of retrievability.

References

1. Shweta Agrawal, Vipul Goyal, Abhishek Jain, Manoj Prabhakaran, and Amit Sahai. New impossibility results for concurrent composition and a non-interactive completeness theorem for secure computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 443–460. Springer, Berlin, Heidelberg, August 2012.
2. Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, April 2016.
3. Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3327–3344. USENIX Association, August 2022.
4. Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1811–1828. USENIX Association, August 2021.
5. Bar Alon and Amos Beimel. On the definition of malicious private information retrieval. In Niv Gilboa, editor, *ITC 2025*, volume 343 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl, August 2025.
6. Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.
7. Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569. USENIX Association, 2016.
8. Pranav Shriram Arunachalaramanan and Ling Ren. Single-server stateful pir with verifiability and balanced efficiency. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 1385–1402. IEEE, 2026.
9. Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Guy N. Rothblum, Kunal Talwar, Karl Tarbe, Ruiyu Zhu, and Marco Zuliani. Scalable private search with wally. *CoRR*, abs/2406.06761, 2024.
10. Christian Badertscher, Michele Ciampi, and Aggelos Kiayias. Agile cryptography: A universally composable approach. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023, Part IV*, volume 14372 of *LNCS*, pages 480–509. Springer, Cham, November / December 2023.

11. Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Berlin, Heidelberg, August 2000.
12. Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part III*, volume 13749 of *LNCS*, pages 3–32. Springer, Cham, November 2022.
13. Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith, III. Public key encryption that allows PIR queries. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 50–67. Springer, Berlin, Heidelberg, August 2007.
14. Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 662–693. Springer, Cham, November 2017.
15. Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Berlin, Heidelberg, May 1999.
16. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.
17. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Paper 2000/067, 2000.
18. Ran Canetti. Obtaining universally compoable security: Towards the bare bones of trust (invited talk). In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 88–112. Springer, Berlin, Heidelberg, December 2007.
19. Ran Canetti. Security and composition of cryptographic protocols: A tutorial. In *Secure Multi-Party Computation*, Cryptology and Information Security Series, pages 61–119. IOS Press, 2013.
20. Ran Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.
21. Ran Canetti and Megan Chen. Universally composable succinct vector commitments and applications. In Benny Applebaum and Huijia (Rachel) Lin, editors, *TCC 2025, Part IV*, volume 16271 of *LNCS*, pages 642–674. Springer, Cham, December 2025.
22. 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, Berlin, Heidelberg, August 2015.
23. Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 694–726. Springer, Cham, November 2017.
24. 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, Cham, August 2022.
25. 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, Berlin, Heidelberg, March 2016.
26. Ran Canetti, Alley Stoughton, and Mayank Varia. EasyUC: Using EasyCrypt to mechanize proofs of universally composable security. In Stephanie Delaune and Limin Jia, editors, *CSF 2019 Computer Security Foundations Symposium*, pages 167–183. IEEE Computer Society Press, 2019.
27. Justin Cappos. Avoiding theoretical optimality to efficiently and privately retrieve security updates. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 386–394. Springer, Berlin, Heidelberg, April 2013.
28. Megan Chen, Pousali Dey, Chaya Ganesh, Pratyay Mukherjee, Pratik Sarkar, and Swagata Sasmal. Universally composable non-interactive zero-knowledge from sigma protocols via a new straight-line compiler. In Tibor Jager and Jiaxin Pan, editors, *PKC 2025, Part I*, volume 15674 of *LNCS*, pages 381–417. Springer, Cham, May 2025.
29. Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas E. Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns. In *ACSAC*, pages 84–99. ACM, 2020.
30. Seung Geol Choi, Alexis Galan, Daniel S. Roche, and Mayank Varia. CVPIR: Verifiable PIR with efficient online communication complexity. https://www.roche.work/research/papers/cvpir.pdf, 2026.
31. Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995.
32. Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

33. Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3835–3851. USENIX Association, August 2023.
34. Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Cham, May 2020.
35. Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. *PoPETs*, 2023(1):365–383, January 2023.
36. Leo de Castro and Keewoo Lee. VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5931–5948, 2024.
37. Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 122–138. Springer, Berlin, Heidelberg, May 2000.
38. Marian Dietz and Stefano Tessaro. Fully malicious authenticated PIR. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part IX*, volume 14928 of *LNCS*, pages 113–147. Springer, Cham, August 2024.
39. Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.
40. Changyu Dong and Liqun Chen. A fast single server private information retrieval protocol with low communication cost. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014, Part I*, volume 8712 of *LNCS*, pages 380–399. Springer, Cham, September 2014.
41. Jean-Guillaume Dumas, Aude Maignan, Clément Pernet, and Daniel S. Roche. VESPo: Verified evaluation of secret polynomials (with application to dynamic proofs of retrievability). *PoPETs*, 2023(3):354–374, July 2023.
42. Brett Falk, Pratyush Mishra, and Matan Shtepel. Malicious security for PIR (almost) for free. Cryptology ePrint Archive, Paper 2024/964, 2024.
43. Brett Falk, Pratyush Mishra, and Matan Shtepel. Malicious security for PIR (almost) for free. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part VIII*, volume 16007 of *LNCS*, pages 175–210. Springer, Cham, August 2025.
44. Francesca Falzon, Laura Hetz, and Annamira O'Toole. TAPIR: A two-server authenticated PIR scheme with preprocessing, 2026.
45. Ben Fisch, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. ThorPIR: Single server PIR via homomorphic thorp shuffles. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 1448–1462. ACM Press, October 2024.
46. Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 438–464. Springer, Cham, December 2019.
47. Fatemeh Ghasemi, Swastik Kopparty, and Madhu Sudan. Improved PIR schemes using matching vectors and derivatives. In Michal Koucký and Nikhil Bansal, editors, *57th ACM STOC*, pages 1648–1656. ACM Press, June 2025.
48. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.
49. Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM J. Comput.*, 25(1):169–192, 1996.
50. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
51. Vipul Goyal, Divya Gupta, and Abhishek Jain. What information is leaked under concurrent composition? In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 220–238. Springer, Berlin, Heidelberg, August 2013.
52. Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1591–1601. ACM Press, October 2016.
53. Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath T. V. Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *NSDI*, pages 91–107. USENIX Association, 2016.
54. Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with Tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 396–416, New York, NY, USA, 2023. Association for Computing Machinery.
55. Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3889–3905. USENIX Association, August 2023.

56. Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: Single-server PIR with efficient updates via invertible PRFs. Cryptology ePrint Archive, Paper 2024/318, 2024.
57. Ari Juels and Burton S. Kaliski, Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 584–597. ACM Press, October 2007.
58. Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1447–1464. USENIX Association, August 2019.
59. Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 875–892. USENIX Association, August 2021.
60. Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997.
61. Eyal Kushilevitz and Rafail Ostrovsky. One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 104–121. Springer, Berlin, Heidelberg, May 2000.
62. Arthur Lazzaretti and Charalampos Papamanthou. Near-optimal private information retrieval with preprocessing. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023, Part II*, volume 14370 of *LNCS*, pages 406–435. Springer, Cham, November / December 2023.
63. Arthur Lazzaretti and Charalampos Papamanthou. TreePIR: Sublinear-time and polylog-bandwidth private information retrieval from DDH. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 284–314. Springer, Cham, August 2023.
64. Arthur Lazzaretti and Charalampos Papamanthou. Single pass client-preprocessing private information retrieval. Cryptology ePrint Archive, Paper 2024/303, 2024.
65. Arthur Lazzaretti and Charalampos Papamanthou. Single pass client-preprocessing private information retrieval. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.
66. Keewoo Lee. Barely doubly-efficient simplepir. 2026. To appear. Preprint available at Cryptology ePrint Archive, Report 2025/1305.
67. Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1387–1403. ACM Press, November 2019.
68. Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 595–608, 2023.
69. Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. *Electron. Colloquium Comput. Complex.*, TR17, 2017.
70. Chenyang Liu, Xukun Wang, and Zhifang Zhang. VIA: Communication-efficient single-server private information retrieval. In *IEEE Symposium on Security and Privacy (S&P)*, 2026. To appear. Preprint available at Cryptology ePrint Archive, Report 2025/2074.
71. Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: Oblivious message retrieval with reduced communication and computation. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.
72. Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Berlin, Heidelberg, January 2015.
73. Rasoul Akhavan Mahdavi, Ahmed Diaa, and Florian Kerschbaum. ZipPIR: High-throughput single-server PIR without client-side storage. In *Proceedings of the 35th USENIX Security Symposium*, 2026. To appear. Preprint available at arXiv:2603.09190.
74. Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*, pages 930–947. IEEE Computer Society Press, May 2022.
75. Samir Jordan Menon and David J. Wu. YPIR: High-throughput single-server PIR with silent preprocessing. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.
76. 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.
77. Mayank Rathee, Keewoo Lee, and Raluca Ada Popa. Verifiable PIR with small client storage. In *SP*. IEEE, 2026.

78. Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS 2014*. The Internet Society, February 2014.
79. Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 641–669, Virtual Event, August 2021. Springer, Cham.
80. 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.
81. Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In *Information and Communications Security: 20th International Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings*, page 478–493, Berlin, Heidelberg, 2018. Springer-Verlag.
82. Yinghao Wang, Xuanming Liu, Jiawen Zhang, Jian Liu, and Xiaohu Yang. Crust: Verifiable and efficient private information retrieval with sublinear online time. Cryptology ePrint Archive, Paper 2023/1607, 2023.
83. David Woodruff and Sergey Yekhanin. A geometric approach to information-theoretic private information retrieval. In *20th Annual IEEE Conference on Computational Complexity (CCC'05)*, pages 275–284. IEEE, 2005.
84. Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. Two-servers pir based dns query scheme with privacy-preserving. In *The 2007 International Conference on Intelligent Pervasive Computing (IPC 2007)*, pages 299–302, 2007.
85. Liang Zhao, Xingfeng Wang, and Xinyi Huang. Verifiable single-server private information retrieval from lwe with binary errors. *Information Sciences*, 546:897–923, 2021.
86. Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: Extremely simple, single-server PIR with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy*, pages 4296–4314. IEEE Computer Society Press, May 2024.

A Additional Preliminaries

We provide UC functionalities for three common cryptographic primitives: a common reference string (Algorithm 5), digital signature (Algorithm 6), and non-interactive zero knowledge proof (Algorithm 7).

Algorithm 5 Functionality $\mathcal{G}_{\mathsf{CRS}}$ (adapted from [18])

This functionality is parameterized by a distribution $\mathcal{D}$.

1: Upon receiving an input $(\mathsf{CRS}, \mathsf{sid})$ from a party P:
   1. If there is no crs value stored, sample $\mathsf{crs} \leftarrow \mathcal{D}$.
   2. Send $(\mathsf{CRS}, \mathsf{sid}, \mathsf{crs}, P)$ to Sim.
   3. When Sim responds with $(\mathsf{CRS}, \mathsf{sid}, \mathtt{OK})$, output $(\mathsf{CRS}, \mathsf{sid}, \mathsf{crs})$ to P.

Algorithm 6 Functionality $\mathcal{G}_{\mathsf{Sig}}$ (reproduced from [25])

Parameterized by a party identity *pid*, functionality $\mathcal{G}_{\mathsf{Sig}}$ proceeds as follows:

1: **Key Generation:** Upon receiving a value $(\mathsf{KeyGen}, \mathsf{sid})$ from some party P_{pid}, verify that this is the first request and $\mathsf{sid} = (\mathsf{pid}, \mathsf{sid}')$ for some sid'. If not, then ignore the request. Else, hand $(\mathsf{KeyGen}, \mathsf{sid})$ to the adversary. Upon receiving $(\mathsf{Verification\ Key}, \mathsf{sid}, v)$ from the adversary, output $(\mathsf{Verification\ Key}, \mathsf{sid}, v)$ to P_{pid}.

2: **Signature Generation:** Upon receiving a value $(\mathsf{Sign}, \mathsf{sid}, m)$ from P_{pid}, verify that $\mathsf{sid} = (\mathsf{pid}, \mathsf{sid}')$ for some sid'. If not, then ignore the request. Else, send $(\mathsf{Sign}, \mathsf{sid}, m)$ to the adversary. Upon receiving $(\mathsf{Signature}, \mathsf{sid}, m, \sigma)$ from the adversary, verify that no entry $(m, \sigma, v, 0)$ is recorded. If it is, then output an error message to P_{pid} and halt. Else, output $(\mathsf{Signature}, \mathsf{sid}, m, \sigma)$ to P_{pid}, and record the entry $(m, \sigma, v, 1)$.

3: **Signature Verification:** Upon receiving a value $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, v')$ from party P, where $\mathsf{sid} = (\mathsf{pid}, \mathsf{sid}')$ for some sid', verify that a pair (pid, v) is recorded. If not, output $(\mathsf{Verified}, \mathsf{sid}, m, 0)$ to P. Else, hand $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, \mathsf{sid}, m, \phi)$ from the adversary do:

(a) If $v' = v$ and the entry $(m, \sigma, v, 1)$ is recorded, then set $f = 1$. *// This condition guarantees completeness: if the verification key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.*

(b) Else, if $v' = v$, the signer is not corrupted, and no entry $(m, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, v, 0)$. *// This condition guarantees unforgeability: if v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.*

(c) Else, if there is an entry (m, σ, v', f') recorded, then let $f = f'$. *// This condition guarantees consistency: all verification requests with identical parameters will result in the same answer.*

(d) Else, let $f = \phi$ and record the entry (m, σ, v', ϕ).

Output $(\mathsf{Verified}, \mathsf{sid}, m, f)$ to P.

Algorithm 7 Functionality $\mathcal{F}_{\mathsf{NIZK}}$ (reproduced from [28])

This functionality is parameterized by a relation R.

1: **Prove:** Upon receiving an input $(\mathtt{Prove}, \mathsf{sid}, x, w)$ from a party P:

1. If $R(x, w) = 1$, send $(\mathtt{Prove}, \mathsf{sid}, P, x)$ to the simulator
2. Upon receiving $(\mathtt{Prove}, \mathsf{sid}, \Pi)$ from the simulator, store (sid, x, Π) and output $(\mathtt{Proof}, \mathsf{sid}, \Pi)$ to P.

2: **Verify:** Upon receiving an input $(\mathtt{Verify}, \mathsf{sid}, x, \Pi)$ from V:

1. If a tuple (sid, x, Π) is stored, send $(\mathtt{Verify}, \mathsf{sid}, x, \Pi)$ to the simulator.
2. Wait to receive a message $(\mathtt{Witness}, w)$ from the simulator. If $R(x, w) = 1$, store the tuple (sid, x, w). Output $(\mathtt{Verification}, \mathsf{sid}, x, \Pi, R(x, w))$ to V.

B Definitional attacks on verifiable PIR

B.1 Stateful clients

Consider the VPIR definition of Dietz and Tessaro [38]. This definition composes securely under sequential and concurrent composition, since clients are memoryless. Now, we describe a straw-man attempt to extend this definition to support stateful clients which appears straightforward, but has no security under composition. Simply modify Dietz and Tessaro's definition to allow the Reconstruct algorithm to take auxiliary state state_C, which is produced by the client (potentially interactively) in a preprocessing phase.

More formally, let ($\mathsf{Query}, \mathsf{Answer}, \mathsf{Rec}$, etc.) be algorithms of a secure VPIR scheme under Dietz and Tessaro's original defintion. Define a new scheme ($\mathsf{Query}', \mathsf{Answer}', \mathsf{Rec}', \ldots$) as follows, in which Rec' takes in auxillary state state_C as an additional input. For simplicity, we omit public parameter inputs.

- During preprocessing, the client secretly samples long-term state $\mathsf{state}_C \leftarrow \{0,1\}^\lambda$.
- $\mathsf{Query}'(i)$: return $(\mathsf{state}_q, q) \leftarrow \mathsf{Query}(i)$.
- $\mathsf{Answer}'(\mathsf{DB}, q)$: Let $a \leftarrow \mathsf{Answer}(\mathsf{DB}, q)$. Optionally, append values $(j \in [\lambda], x \in \{0,1\}^\lambda)$ to a. An honest server will never do this, but a malicious server may append either or both values.
- $\mathsf{Rec}'(\mathsf{state}_C, \mathsf{state}_q, \mathsf{dig}, a)$: Parse a to potentially recover j and x. If x is included and $x == \mathsf{state}_C$, abort iff the first bit of the querying index i is 1. Otherwise if j is included, abort iff the j-th bit of state_C is 1. In all other cases, return $\mathsf{Rec}(\mathsf{state}_q, \mathsf{dig}, a)$.

All other algorithms of the new scheme (Setup, VQuery, etc.) are unchanged from the base scheme.

When multiple queries are composed sequentially, this construction fails spectacularly against a malicious server. A malicious server can attach a different index j to the responses of λ different queries. By observing if the client aborts or not, each query reveals a bit of state_C to the server. After fully reconstructing state_C, the server can attach this as the x value to another query, causing the client to abort based on their secret index.

B.2 Concurrent composition

Let Π be a secure VPIR scheme according to the definition of [36], with algorithms $\mathsf{PrQuery}$, Query, Answer, Verify, Rec, etc. Additionally, let $\mathcal{F}$ be a family of trapdoor one-way functions $\{f_k : \mathcal{X} \to \mathcal{Y}\}$, and suppose elements in $\mathcal{X}$ can be represented as a length-ℓ bit string. We define a new scheme as follows. Public parameter inputs are omitted for simplicity.

- $\mathsf{PrQuery}'(\mathsf{dig})$: Run $\mathsf{PrQuery}(\mathsf{dig})$. Then, sample a trapdoor key t for a function f in $\mathcal{F}$. Append t to the secret state.
- $\mathsf{Query}'(i)$: Let $q \leftarrow \mathsf{Query}(i)$. Then, uniformly sample $y \leftarrow\!\!\$\, \mathcal{Y}$ and append y to the query q.
- $\mathsf{Answer}'(\mathsf{DB}, q')$: Parse q' as (q, y) Let $a \leftarrow \mathsf{Answer}(\mathsf{DB}, q)$. Optionally, append values $(y' \in \mathcal{Y}, j \in [\ell])$ and/or $(x \in \mathcal{X})$ to a. An honest server will never do this, and instead output $(a, (\perp, \perp), \perp)$.
- $\mathsf{Verify}'(\mathsf{dig}, (\pi, t), (q, y), (a, (y', j), x))$: If $x \neq \perp$ and $f(x) == y$ (where y is the value sampled in Query'), output Reject iff the first bit of i is 1. Otherwise, if $(y', j) \neq (\perp, \perp)$, use t to compute $x' \leftarrow f^{-1}(y')$ and output Reject iff the j-th bit of x' is 1. In all other cases, return $\mathsf{Verify}(\mathsf{dig}, \pi, q, a)$.
- $\mathsf{Rec}'(\mathsf{State}, (a, (y', j), x)), \mathsf{dig})$: Return $\mathsf{Rec}'(\mathsf{State}, a, \mathsf{dig})$.

All other algorithms are unchanged from Π.

Our modified scheme achieves query privacy according to [36, Definition A.4]. The simulator samples a random trapdoor key t' for a function f' on its first execution. Verify' is simulated as follows: ignore any x value the adversary attaches. If $(y', j) \neq (\perp, \perp)$, use t' to compute $x' \leftarrow f^{-1}(y')$ and output Reject iff the j-th bit of x' is 1. Otherwise, run the simulator for Π. The simulator can ignore the x value because, with overwhelming probability, the adversary cannot guess the preimage of the client's freshly-sampled y value; therefore the real-world client will never reject on the first check of Verify'.

We now present an attack under concurrent composition. The idea is that a malicious server can get a y value from the client in one query, use concurrent queries to learn information about the preimage of y, then return to the original query and cause the client to selectively abort based on their index. Suppose a client submits $\ell + 1$ queries $(q_k, y_k)_{k \in [\ell+1]}$ in parallel to the server where $q_k \leftarrow \mathsf{Query}(i_k)$ for index i_k. Let y_1 be the y value of the first query in the list. For queries $j \in \{2, \ldots, \ell + 1\}$, the server will compute the corresponding responses as $(a_j, (y_1, j - 2), \perp)$. Each of these responses contains the same y_1 value, and a different bit index in $[\ell]$. Since the server does not include the optional x value to the tuple, the client's abort behavior will reveal the preimage of y_1 to the server bit-by-bit. Once the server has learned the full pre-image x_1, it responds to the client's first query with $(a_1, (\perp, \perp), x_1)$. The client will check that $f(x_1) = y_1$, and abort based on the first bit of their queried index i_1—breaking query privacy.

We stress that de Castro and Lee's *protocol* is not susceptible to attacks under concurrent composition (in fact, we prove this in Appendix D). The above counterexample simply illuminates a weakness in their *definition*.

B.3 Adaptive queries

VPIR definitions in prior works have different approaches to handling multiple queries. VeriSimplePIR [36] enforces that for every statically fixed index sequence $i_1, \ldots, i_L$, the distributions of real and ideal sequential queries are indistinguishable. TAPIR [44] captures query-privacy with a bit-guessing game. The challenge bit b is chosen up front, and the malicious server must guess from which sequence of queries it is receiving. mPIR [42] fixes two distinct sequences of indices and says that the distributions are indistinguishable. CRUST [82] considers adaptively, adversarially chosen query indices. Balanced PIR [8] gives definitions for both static and adaptive query privacy.

In general, it is understood that security against an adaptive adversary is at least as strong as security against a static adversary.

In this section, we formalize the gap between static and adaptively chosen sequential queries when defining query privacy in VPIR. We present a counterexample VPIR protocol that is secure under the static query definition from [36], but *insecure* when queries can be adaptively chosen by the adversary.

At a high-level, the idea is that when queries are statically chosen up-front, then with overwhelming probability the client's abort behavior is independent of their query index. However, when chosen adaptively, query indices can strategically chosen make the client's abort behavior leak information about the query index. We give a counterexample below.

Our new scheme is given by $(\mathtt{Query}^\dagger, \mathtt{Response}^\dagger, \mathtt{Reconstruct}^\dagger)$ below and makes black-box use of an underlying VPIR. Upon initialization, the client chooses λ query indices at random and defines the sequence as $T := (i_1, \ldots, i_\lambda)$. Let N be the length of the database, let $n = \log N$ be the number

of bits used to describe a location in the database, and assume that the client has successfully obtained the correct digest dig. The client stores this sequence as secret, long-term state.

- $\texttt{Query}^\dagger$: The client, on input an index i, will generate a query $q \leftarrow \texttt{Query}(i)$ and send q to the server.
- $\texttt{Response}^\dagger$: The server, upon receiving the query q, will first run $a \leftarrow \texttt{Response}(\mathsf{dig}, q)$. Then choose indices $s \in [n]$ and $k \in [\lambda]$, a flag bit $b \in \{0, 1\}$, and sends the tuple (a, s, k, b) to the client.
- $\texttt{Reconstruct}^\dagger$: The client, upon receiving the response (a, s, k) will run $v \leftarrow \texttt{Reconstruct}(\mathsf{dig}, a)$ to obtain the database value.
  If the previous λ queries (up to but *not* including the current query) are exactly the sequence T and x and the flag bit $b = 0$, the client checks if the least-significant bit of the index i is 0 and if so, the client aborts. Otherwise, output the reconstructed database value v.
  Else, if the flag bit $b = 1$ the client checks if the sth bit of i_k is 0 and aborts if it is. Otherwise, output the reconstructed database value v.

We first discuss why this scheme satisfies the static-query privacy definition from VeriSimplePIR. At a high-level, the client's abort behavior reveals something about its query index only if the previous λ queries are exactly the special query sequence T. Since the static query sequence is fixed before the start of the protocol, the probability that the client actually makes these queries is negligible in λ and thus query privacy follows from the underlying VPIR scheme. Otherwise, the client's abort behavior reveals information about the query sequence T, which is independent of the client's query index.

However, we give an attack carried out by the malicious server that shows this protocol is insecure when queries are selected adaptively and adversarially. First, the adversary will make the client submit arbitrary queries q_j for $j \in [\lambda \cdot n]$. For each query in this sequence, the server will respond with $(a_j, j \bmod n, j \bmod \lambda, 1)$. During reconstruction, the client will see that the flag bit $b = 1$ and abort based on the value of the indices in secret query sequence. This allows the server to learn every index in the client's secret sequence $T = (i_1, \ldots, i_\lambda)$. Next, the adversary will make the client submit λ queries, exactly given by the sequence T learned above. For the subsequent query, the server will set its flag bit $b = 0$. During reconstruction, the client will detect that the previous λ queries match the sequence T and abort based on the LSB of the current query index, thus breaking query privacy.

C Trivial PIR UC-realizes $\mathcal{F}_{\mathsf{VPIR}}$

In this section, we construct a modified version of Trivial PIR with verifiability provided by a simple hash-based commitment to the database contents. We then provide a security analysis to show that our trivial VPIR protocol UC-realizes $\mathcal{F}_{\mathsf{VPIR}}$.

To start, we specify our trivial VPIR protocol in Appendix C and provide the corresponding simulator in Algorithm 9.

Algorithm 8 Trivial Verifiable PIR Protocol

This protocol has a session ID sid that is assumed to contain the party id $\mathbf{S}$ of the intended database server, and a database digest dig. Upon first activation, each party calls $\mathcal{G}_{\mathsf{CRS}}$ to obtain a random string and uses this to deterministically select a function H from a collision-resistant hash function family $\mathscr{H}$.

1: **Init:** Upon receiving $(\texttt{Init}, \mathsf{sid}, \mathsf{DB} \in \{0,1\}^N)$ from $\mathcal{Z}$, the server $\mathbf{S}$ does:
   a. If the $\mathsf{initialized}$ variable has not been set before, let $\mathsf{initialized} = \mathsf{false}$.
   b. If $\mathsf{initialized} = \mathsf{true}$ or $\mathbf{S} \neq \mathbf{S}'$, then end this activation.
   c. Let dig be the digest contained in sid, and $\mathsf{dig}' \leftarrow H(\mathsf{DB})$. If $\mathsf{dig} \neq \mathsf{dig}'$, output $(\texttt{InvalidDigest}, \mathsf{sid})$.
   d. Otherwise, set $\mathsf{initialized} \leftarrow \mathsf{true}$, store $(\mathsf{sid}, \mathsf{DB})$, and output $(\mathsf{initialized}, \mathsf{sid})$.

2: **Query:** Upon receiving $(\texttt{Query}, \mathsf{qid}, i \in [N])$ from $\mathcal{Z}$, the client $\mathbf{C}$ does:
   a. If this query id qid has been used in a previous $\texttt{Query}$ call, output $(\texttt{QueryInvalid}, \mathsf{qid})$.
   b. Otherwise: store $(\texttt{Query}, \mathsf{qid}, i)$ and send $(\texttt{Query}, \mathsf{qid})$ to $\mathbf{S}$.

3: *ReceiveQuery:* Upon receiving a message $(\texttt{Query}, \mathsf{qid})$ from a client $\mathbf{C}$, the server $\mathbf{S}$ does:
   1. If this is the first received $\texttt{Query}$ message containing qid, store $(\texttt{Query}, \mathsf{qid})$ and output $(\texttt{Query}, \mathsf{qid})$.

4: **Response:**Upon receiving $(\texttt{Response}, \mathsf{qid}, \mathsf{DB} \in \{0,1\}^N)$ from $\mathcal{Z}$, the server $\mathbf{S}$ does:
   a. End execution if any of the following holds: (a) $\mathsf{initialized} = \mathsf{false}$, (b) the server did not receive a query message matching qid, (c) the query ID qid has previously been responded to, or (d) $H(\mathsf{DB}) \neq \mathsf{dig}$, where dig is the digest contained in sid.
   b. Otherwise, send $(\texttt{Response}, \mathsf{qid}, \mathsf{DB})$ to $\mathbf{C}$.

5: *ReceiveResponse:* Upon receiving a message $(\texttt{Response}, \mathsf{qid}, \mathsf{DB})$ from $\mathbf{S}$, a client $\mathbf{C}$ does:
   1. Look up the stored message of the form $(\texttt{Query}, \mathsf{qid}, i)$, or end execution if no stored message exists.
   2. If $\mathsf{dig} = H(\mathsf{DB})$, output $(\texttt{Answer}, \mathsf{qid}, \mathsf{DB}[i])$. Otherwise, output $(\texttt{VerificationFail}, \mathsf{qid})$.

Algorithm 9 Trivial Verifiable PIR Simulator

The simulator has a session ID sid that is assumed to contain the party id $\mathbf{S}$ of the intended database server, and a database digest dig.

1: Upon first activation:
   1. Call $\mathcal{G}_{\mathsf{CRS}}$ to obtain a random string and uses this to deterministically select a function H from a collision-resistant hash function family $\mathscr{H}$.
   2. Send $(\texttt{Store}, \mathsf{DigestVerifyCode}, \mathcal{V})$ to $\mathcal{G}_{\mathsf{lib}}$, where $\mathcal{V}(\mathsf{sid}, \mathsf{DB})$ is code that returns OK if $H(\mathsf{DB})$ matches the digest in sid, and otherwise returns Reject.

2: Upon receiving $(\texttt{ReportInit}, \mathsf{state}, \mathsf{DB})$ from $\mathcal{F}_{\mathsf{VPIR}}$:
   a. Store DB and send $(\texttt{ReportInit}, \texttt{OK})$ to $\mathcal{F}_{\mathsf{VPIR}}$.

3: Upon delivery of a message $(\mathsf{Query}, \mathsf{qid})$ from a client $\mathbf{C}$ to honest $\mathbf{S}$:
   a. If $\mathbf{C}$ is honest, send $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$.
   b. If $\mathbf{C}$ is malicious, send $(\texttt{Query}, \mathsf{qid}, 0)$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{C}$ // *Dummy query. This will trigger line 4a..*

4: Upon receiving $(\texttt{ReportQuery}, \mathsf{qid})$ from $\mathcal{F}_{\mathsf{VPIR}}$:
   a. Parse the client id $\mathbf{C}$ from qid. If $\mathbf{C}$ is malicious, send $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$ and end execution.
   b. If $\mathbf{C}$ is honest, send a message $(\texttt{Query}, \mathsf{qid})$ to $\mathbf{S}$ on behalf of $\mathbf{C}$. Upon delivery, send $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$.

5: Upon delivery of a message $(\mathsf{Response}, \mathsf{qid}, \widetilde{\mathsf{DB}})$ from $\mathbf{S}$ to an honest $\mathbf{C}$:
   a. If $\mathbf{S}$ is honest, send $(\texttt{ReportResponse}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$ and end execution.
   b. If $\mathbf{S}$ is malicious:
      (a) If there is no stored DB, send $(\texttt{Init}, \mathsf{sid}, \widetilde{\mathsf{DB}})$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$. // *In response, $\mathcal{F}_{\mathsf{VPIR}}$ will trigger step 2 of the simulator, after which $\mathcal{F}_{\mathsf{VPIR}}$ will return execution to Sim to run the next step.*
      (b) Send $(\mathsf{Response}, \mathsf{qid}, \widetilde{\mathsf{DB}})$ to $\mathcal{F}_{\mathsf{VPIR}}$.

6: Upon receiving $(\texttt{ReportResponse}, \mathsf{qid})$ from $\mathcal{F}_{\mathsf{VPIR}}$:
   a. If $\mathbf{S}$ is malicious, send $(\texttt{DeliverResponse}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$ and end execution.
   b. If $\mathbf{S}$ is honest, fetch DB in storage. // *Leaked to us during init, which must have succeeded for $\mathcal{F}_{\mathsf{VPIR}}$ to trigger this step*
   c. Parse the client ID $\mathbf{C}$ from qid, and send message $(\mathsf{Response}, \mathsf{qid}, \mathsf{DB})$ to $\mathbf{C}$ on behalf of $\mathbf{S}$. // *If $\mathbf{C}$ is malicious, then Sim's work for this query is done. Otherwise, Sim might later execute step 5.*

C.1 Security Analysis

Theorem 1. *Let $\mathscr{H}$ be a collision-resistant hash function family. The protocol $\Pi_{\mathsf{Triv\text{-}VPIR}}$ UC-realizes $\mathcal{F}_{\mathsf{VPIR}}$ in the presence of global functionality $\mathcal{G}_{\mathsf{lib}}$ and $\mathcal{G}_{\mathsf{CRS}}$, with static corruptions, and against an adverary who cannot forge or modify network traffic.*

Proof. We present a simulator in Algorithm 9. The idea of our proof is to inductively show that each successive action an environment may perform gives the adversary at most a negligible advantage in distinguishing the real and ideal worlds. For the inductive step, we assume that if all previous actions performed by the environment are indistinguishable in the real vs ideal world, then the next Setup, Query, Response, or Reconstruct action will preserve indistinguishability of the two worlds.

We will consider the four different actions separately.

Init. We observe that the behavior of `Init` in the ideal functionality and real protocol is mostly identical, except for two differences:

1. The ideal functionality fetches and runs internal code $\mathcal{V}$, while the protocol computes a hash $H(\mathsf{DB})$ and compares it against the digest contained in sid.
2. The ideal functionality interacts with the simulator and awaits a $(\texttt{ReportInit}, \texttt{OK})$ response to continue.

Neither of these differences affect the environment's view. The first difference is addressed by having the simulator begin by loading code that computes the hash function H into the internal code, causing the check done in the ideal functionality to identically match the protocol. The second difference is addressed by having the simulator always respond $(\texttt{ReportInit}, \texttt{OK})$ with no delay to the functionality when the database is leaked. Therefore, the simulator perfectly simulates the real world Init protocol.

Query. Next, we consider the `Query` functionality and its real-world counterpart. We will examine separately the cases in which $\mathbf{C}$ is honest and malicious.

When an honest $\mathbf{C}$ makes a query, both the real-world protocol and ideal functionality first check that the qid is unused. Then, in the ideal world, the simulator produces an identical message as the real-world client and sends it to the server. To simulate network delay, the simulator delays responding to $\mathcal{F}_{\mathsf{VPIR}}$ until the query message is delivered, ensuring that $\mathbf{S}$ gets its output at the same time in both real and ideal worlds.

Next, we consider the case where $\mathbf{C}$ is corrupted by the adversary. In this case, we will only consider instances where $\mathbf{S}$ is honest. If both $\mathbf{C}$ and $\mathbf{S}$ are corrupted, simulation is trivial: the simulator does not need to interact with the adversary at all, since there is nothing to be hidden from the adversary.

When an adversary sends a Query message from a corrupted client to an honest server, the simulator runs the `Query` ideal functionality on behalf of the client, using 0 as a dummy index. In running the Query functionality, the simulator is triggered again on line 4, and immediately informs the functionality that the message was delivered. In both cases, simulation is perfect.

Response. We consider the cases where $\mathbf{S}$ is honest and $\mathbf{S}$ is corrupted separately.

First, suppose $\mathbf{S}$ is honest. In the real world, the server returns a `ResponseInvalid` error if any of four conditions are met:

1. $\mathsf{initialized} == \mathsf{false}$

2. The server did not previously receive a query message matching qid
3. The query qid has previously been responded to
4. $H(\mathsf{DB}) \neq \mathsf{dig}$

The first three conditions are identically checked in the ideal functionality. The only difference is in the fourth condition, where the ideal functionality checks if the provided DB matches the one originally used in setup. Let $\mathcal{B}$ be the event that the environment has given the server an input $\widetilde{DB}$ that does not match the input DB used in setup, but $H(\mathsf{DB}) = H(\widetilde{\mathsf{DB}}) = \mathsf{dig}$, for the dig contained in sid. If $\mathcal{B}$ does not occur, the checks in the ideal world and real world behave identically. But by the collision resistance of H, $\mathcal{B}$ occurs with at most negligible probability, making the real and ideal world checks computationally indistinguishable.

Assuming all checks pass, the real-world response message is easily simulated, since the simulator knows the entire database—it was leaked to the simulator during Init. Note that Init must have been successfully run, or else the first check in the list above would have failed.

In the case that $\mathbf{C}$ is honest, the simulator waits until the adversary delivers a response message to the client to send $(\texttt{ReportResponse}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$. This ensures that the client obtains their output at the same time in both real and ideal worlds. Moreover, the client's output is the same in both worlds, supposing event $\mathcal{B}$ above does not occur.

Next, we consider the case in which $\mathbf{S}$ is corrupted. We further assume that $\mathbf{C}$ is honest — the case where both $\mathbf{C}$ and $\mathbf{S}$ are corrupted can be trivially simulated without the involvement of the simulator.

When the adversary sends a $(\mathsf{Response}, \mathsf{qid}, \widetilde{\mathsf{DB}})$ message on behalf of the server, the simulator uses the provided $\widetilde{\mathsf{DB}}$ to run the $\texttt{Init}$ ideal functionality, if $\texttt{Init}$ has not yet been run. If $H(\mathsf{DB}) = \mathsf{dig}$, the ideal functionality stores $\widetilde{\mathsf{DB}}$ as its database. Otherwise, the ideal functionality remains uninitialized.

In any case, the simulator sends $(\mathsf{Response}, \mathsf{qid}, \widetilde{\mathsf{DB}})$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$. The simulator immediately sends $(\texttt{ReportResponse}, \mathsf{qid})$ back to $\mathcal{F}_{\mathsf{VPIR}}$. At this point, the functionality ensures that if $\widetilde{\mathsf{DB}} = \mathsf{DB}$, the client gets a successful output, and if not, they get a VerificationFail error. In the real world, the client gets a successful output if and only if $H(\widetilde{DB}) = \mathsf{dig}$. Assuming event $\mathcal{B}$ above does not occur, these conditions are equivalentmaking the real and ideal world outputs computationally indistinguishable.

□

D VeriSimplePIR [36] UC-realizes $\mathcal{F}_{\mathsf{VPIR}}$

We subsequently prove that a slight modification to the VeriSimplePIR protocol of de Castro and Lee [36] (with additional refinements from Rathee, Lee, and Popa [77]) UC-realizes our functionality.

We define our functionality in Algorithm 3. Protocol D describes the VeriSimplePIR protocol. Our protocol relies on a common reference string functionality $\mathcal{G}_{\mathsf{CRS}}$ (Algorithm 5) and a NIZK functionality $\mathcal{F}_{\mathsf{NIZK}}$ (Algorithm 7). We replace the preprocessing scheme of VeriSimplePIR with the simplified preprocessing step described by Rathee, Lee and Popa [77]. Additionally, we modify the protocol to require the server to send the client a simulation-extractable proof of knowledge of a preimage to the public SIS commitment during preprocessing.

We give a high-level overview of some key points below.

– **Correctness.** Correctness of simulation when the client and server are honest follows from correctness of the VeriSimplePIR protocol.

– **Query privacy with abort.** To emulate the server's view when a client makes a query, our simulator makes a "dummy query" at index 0 of the database, which is computationally indistinguishable by query-privacy of VeriSimplePIR. If the server's response is invalid, the simulator will inform the functionality that the client should abort.
– **Integrity.** The simulator, who acts as a "translator" between the real and ideal world, can use the VeriSimplePIR digest to verify the malicious server's responses and relays the outcome to the functionality to ensure that the client either learns the correct entry or aborts.
– **Proof of retrievability.** To realize the proof of retrievability property in the presence of a malicious server, the ideal simulator must be able to "extract" the full database from the VeriSimplePIR server's response. This is especially crucial since a malicious server may not even initialize the protocol (by sending a database to $\mathcal{F}_{\mathsf{VPIR}}$) to begin with, leaving the simulator with the task of figuring out what the "real" database is. However, the VeriSimplePIR extractor is based on rewinding, which is inherently incompatible with UC. To side-step this issue, we slightly modify the server's response message in the VeriSimplePIR real-world protocol to include a straight-line extractable proof of knowledge of the database.
– **Simulating real-world pre-processing.** VeriSimplePIR uses a pre-processing phase to generate and verify a digest. Since there is no explicit pre-processing API in our ideal $\mathcal{F}_{\mathsf{VPIR}}$ functionality, we rely on the simulator to fully emulate the view of the preprocessing phase to a malicious party. To ensure that malicious clients or malicious servers do not "speak out of turn" (e.g. make a query before completing processing), the simulator also stores which clients have successfully completed pre-processing, along with their respective preprocessing material.

D.1 UC Security proof for VeriSimplePIR

Theorem 2. *Let* LHE *be a semantically secure linearly homomorphic encryption scheme. Assuming hardness of* $\mathsf{SIS}_{\lambda,m,q,4B}$*, the protocol* $\Pi_{\mathsf{VeriSimplePIR}}$ *UC-realizes* $\mathcal{F}_{\mathsf{VPIR}}$ *in the presence of global functionalities* $\mathcal{G}_{\mathsf{lib}}$ *and* $\mathcal{G}_{\mathsf{CRS}}$ *and local functionality* $\mathcal{F}_{\mathsf{NIZK}}$*. We additionally assume the adversary may statically corrupt all parties, and may arbitrarily delay or drop network messages, but cannot modify or spoof them.*

Proof. We present a simulator in Algorithm 11. The idea of our proof is to inductively show that each successive action an environment may perform gives the adversary at most a negligible advantage in distinguishing the real and ideal worlds. For the inductive step, we assume that if all previous actions performed by the environment are indistinguishable in the real vs ideal world, then the next Init, Query, or Response action will preserve indistinguishability of the two worlds.

We will consider the three different events separately.

Init. We observe that the behavior of `Init` in the ideal functionality and real protocol is mostly identical, with two differences:

1. The ideal functionality checks that the database matches the digest by running internal code $\mathcal{V}$, while the protocol computes a SIS commitment $D \cdot A$ and compares it against dig.
2. The ideal functionality sends the simulator a $(\texttt{ReportInit}, \mathsf{sid}, \mathsf{DB})$ message and awaits a $(\texttt{ReportInit}, \mathsf{sid}, \texttt{OK})$ response to continue.

Neither of these differences affect the environment's view. The first difference is addressed by having the simulator begin by loading internal code $\mathcal{I}$ that deterministically encodes the database

Algorithm 10 VeriSimplePIR++ Protocol

This protocol has a session ID sid that is assumed to contain the party ID $\mathbf{S}$ of the intended database server, and a digest $\mathsf{dig} \in \mathbb{Z}_q^{n \times \lambda}$. Let $\mathsf{Encode} : \{0,1\}^{n^2} \to \mathbb{Z}_q^{n \times n}$ be a function that reshapes vectors to a $n \times n$ matrix and lifts the elements to $\mathbb{Z}_q$. The client is initialized with a bit $\mathsf{preprocessed} = \mathsf{false}$. This value is stored by each party and used throughout the protocol. Let $R : (\mathbb{Z}_q^{m \times \lambda} \times \mathbb{Z}_q^{n \times \lambda}) \times \mathbb{Z}_q^{n \times n} \to \{0,1\}$ be a relation such that $R((A, x), w) = 1$ iff x is a valid SIS commitment to w with respect to the matrix A. Let $\mathcal{D}$ be the uniform distribution over $\mathbb{Z}_q^{m \times \lambda}$. Each party, upon being activated with an input for the first time, makes a call to $\mathcal{G}_{\mathsf{CRS}}^{\mathcal{D}}$ to fetch the matrix A and stores it.

1: **Init:** Upon receiving $(\texttt{Init}, \mathsf{sid}, \mathsf{DB} \in \{0,1\}^N)$ from $\mathcal{Z}$, the server $\mathbf{S}$ does:
   a. If the $\mathsf{initialized}$ variable has not been set before, let $\mathsf{initialized} = \mathsf{false}$.
   b. If $\mathsf{initialized} = \mathsf{true}$ or $\mathbf{S} \neq \mathbf{S}'$, then end this activation.
   c. Let dig be the digest contained in sid.
   d. Let $D = \mathsf{Encode}(\mathsf{DB})$. If $D \cdot A \neq \mathsf{dig}$, output $(\texttt{InvalidDigest}, \mathsf{sid})$.
   e. Otherwise, set $\mathsf{initialized} \leftarrow \mathsf{true}$ and store $(\mathsf{sid}, \mathsf{DB})$.
   f. Return $(\texttt{Initialized}, \mathsf{sid})$.

2: **Query:** Upon receiving $(\texttt{Query}, \mathsf{qid}, i \in [N])$ from $\mathcal{Z}$, the client $\mathbf{C}$ does:
   a. If this query ID qid has been used in a previous $\texttt{Query}$ call, end execution.
   b. If $\mathsf{preprocessed} = \mathsf{false}$: *// Obtain a proof from the server*
      (a) Sample $C \leftarrow\!\!\$\ \{0,1\}^{\kappa \times n}$ and $sk \leftarrow \mathsf{LHE.KeyGen}(1^\lambda)$.
      (b) Let $U = \mathsf{LHE.Encrypt}(sk, C^T)$. *// Element-wise encryption of C^T. $U \in Z_q^{n \times \kappa}$.*
      (c) Send $(\texttt{ProofGen}, \mathsf{qid}, U)$ to $\mathbf{S}$.
      (d) Wait to receive back $(\texttt{ProofGen}, \mathsf{qid}, V \in \mathbb{Z}_q^{m \times \kappa}, \Pi)$ from $\mathbf{S}$.
      (e) Store $(\mathsf{qid}, sk, C, V, \Pi)$. *// Verification will happen later*
   c. Write i as $(i_r, i_c) \in [n] \times [m]$, and let $\mathbf{e}_{i_c} \in \mathbb{Z}_q^m$ be a one-hot vector with "1" in the i_c^{th} position.
   d. Sample $sk \leftarrow \mathsf{LHE.KeyGen}(1^\lambda)$, and let $\mathbf{u} \leftarrow \mathsf{LHE.Encrypt}(sk, \mathbf{e}_{i_c})$ *// Element-wise encryption.* $\mathbf{u} \in \mathbb{Z}_q^m$
   e. Store $(\mathsf{qid}, i, sk, \mathbf{u})$
   f. Send $(\texttt{Query}, \mathsf{qid}, \mathbf{u})$ to $\mathbf{S}$.

3: *ReceiveQuery:* Upon receiving $(\texttt{Query}, \mathsf{qid}, \mathbf{u} \in \mathbb{Z}_q^m)$ from a client $\mathbf{C}$, the server $\mathbf{S}$ does:
   1. If this is the first received $\texttt{Query}$ message containing qid, store $(\mathsf{qid}, \mathbf{u})$ and output $(\texttt{Query}, \mathsf{qid})$.

4: **Response:** Upon receiving $(\texttt{Response}, \mathsf{qid}, \mathsf{DB} \in \{0,1\}^N)$ from $\mathcal{Z}$, the server $\mathbf{S}$ does:
   a. Let $D = \mathsf{Encode}(\mathsf{DB})$
   b. End execution if any of the following holds:
      (a) $\mathsf{initialized} = \mathsf{false}$
      (b) The server did not previously output $(\texttt{Query}, \mathsf{qid})$
      (c) The query ID qid has previously been used in a previous $\mathsf{Response}$ call
      (d) $D \cdot A \neq \mathsf{dig}$, where dig is the digest contained in sid.
   c. Attempt to retrieve the client's proof generation message $(\texttt{ProofGen}, \mathsf{qid}, U \in \mathbb{Z}_q^{n \times \kappa})$. If one exists:
      (a) Let $V = D^T \cdot U$. *// Performed using LHE homomorphism*
      (b) Send $(\texttt{Prove}, (A, \mathsf{dig}), \mathsf{DB})$ to $\mathcal{F}_{\mathsf{NIZK}}^R$ and receive back $(\texttt{Proof}, \Pi)$.
      (c) Send $(\texttt{ProofGen}, \mathsf{qid}, V, \Pi)$ to client $\mathbf{C}$.
      (d) Wait to receive a query message $(\texttt{Query}, \mathsf{qid}, \mathbf{u})$ from client $\mathbf{C}$ for the same qid. Run *ReceiveQuery* before continuing.
   d. Retrieve $(\mathsf{qid}, \mathbf{u} \in \mathbb{Z}_q^m)$ from storage. *// Stored in step 3.*
   e. Compute $\mathbf{v} = D \cdot \mathbf{u} \in \mathbb{Z}_q^n$, using the homomorphism of the LHE scheme.
   f. Send $(\texttt{Response}, \mathsf{qid}, \mathbf{v})$ to $\mathbf{C}$.

5: *ReceiveResponse:* Upon receiving $(\texttt{Response}, \mathsf{qid}, \mathbf{v} \in \mathbb{Z}_q^n)$ from $\mathbf{S}$, the client $\mathbf{C}$ does the following:
   1. If no $\texttt{Query}$ message containing qid was previously sent to $\mathbf{S}$, or a response was already received for this qid, end execution.
   2. If $\mathsf{preprocessed} = \mathsf{false}$:
      (a) Look up stored tuple $(\mathsf{qid}, sk, C, V, \Pi)$. If no such tuple exists, output $(\texttt{VerificationFail}, \mathsf{qid})$.
      (b) Let $Z^T \leftarrow \mathsf{LHE.Decrypt}(sk, V)$ *// Element-wise decryption. $Z \in \mathbb{Z}_p^{\kappa \times m}$*
      (c) Send $(\texttt{Verify}, (A, \mathsf{dig}), \Pi)$ to $\mathcal{F}_{\mathsf{NIZK}}^R$ and receive back $(\texttt{Verification}, \mathsf{dig}, \Pi, b)$.
      (d) Output $(\texttt{VerificationFail}, \mathsf{qid})$ if any of the following are true: (a) $b = 0$, (b) $Z \cdot A \neq C \cdot \mathsf{dig}$, or (c) $\|Z\|_\infty \geq B$.
      (e) Otherwise, set $\mathsf{preprocessed} \leftarrow \mathsf{true}$, store C and Z, and continue.
   3. Look up preprocessing material C and Z, the query tuple of the form $(\mathsf{qid}, i, sk, \mathbf{u})$ and the server's response message $(\texttt{Response}, \mathsf{qid}, \mathbf{v})$. If any of these do not exist, end execution. *// Client needs to have sent a query and received a response to reconstruct an output*
   4. If $Z\mathbf{u} \neq C\mathbf{v}$, client outputs $(\texttt{VerificationFail}, \mathsf{qid})$
   5. Compute $\mathbf{y} \leftarrow \mathsf{LHE.Decrypt}(sk, \mathbf{v})$ *// Client reconstructs the response immediately*
   6. Output $(\texttt{Answer}, \mathsf{qid}, \mathbf{y}[i_r])$

Algorithm 11 VeriSimplePIR++ Simulator

1: At the beginning of activation:
   1. Set $P \leftarrow \emptyset$ // *P is the set of clients who have completed preprocessing*
   2. Call $\mathcal{G}_{\mathsf{CRS}}^{\mathcal{D}}$ to obtain the matrix A and store it.
   3. Send $(\texttt{Store}, \mathsf{DigestVerifyCode}, \mathcal{V})$ to $\mathcal{G}_{\mathsf{lib}}$, where $\mathcal{V}(\mathsf{sid}, \mathsf{DB})$ is code that does the following: Let dig be the digest contained in sid. Let $D = \mathsf{Encode}(\mathsf{DB})$. If $D \cdot A = \mathsf{dig}$, return OK. Otherwise, return Reject.
2: Upon receiving $(\texttt{ReportInit}, \mathsf{sid}, \mathsf{DB})$ from $\mathcal{F}_{\mathsf{VPIR}}$: // *Leak DB to simulator when server is honest. "Extraction" for free! :)*
   a. Store DB and send $(\texttt{ReportInit}, \mathsf{sid}, \texttt{OK})$ to $\mathcal{F}_{\mathsf{VPIR}}$.
3: Upon delivery of a message of the form $(\mathsf{Query}, \mathsf{qid}, \mathbf{u} \in \mathbb{Z}_q^n)$ from a client $\mathbf{C}$ to honest $\mathbf{S}$:
   a. If $\mathbf{C}$ is honest, send $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$.
   b. If $\mathbf{C}$ is malicious, send $(\mathsf{Query}, \mathsf{qid}, 0)$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{C}$.
4: Upon receiving $(\texttt{ReportQuery}, \mathsf{qid})$ from $\mathcal{F}_{\mathsf{VPIR}}$:
   a. Parse the client $\mathbf{C}$ from qid. If $\mathbf{C}$ is malicious, send $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$ and end execution.
   b. Otherwise, parse the client id $\mathsf{pid}_{\mathbf{C}}$ from qid. If $\mathsf{pid}_{\mathbf{C}} \notin P$:
      (a) Sample $C \leftarrow\!\!\$\ \{0,1\}^{\kappa \times n}$ and $\mathsf{sk}_P \leftarrow\!\!\$\ \mathsf{LHE.KeyGen}(1^\lambda)$.
      (b) Let $U = \mathsf{LHE.Encrypt}(\mathsf{sk}_P, C^T)$. // *Element-wise encryption of C^T. $U \in Z_q^{n \times \kappa}$.*
      (c) Send $(\texttt{ProofGen}, \mathsf{qid}, U)$ to $\mathbf{S}$ on behalf of $\mathbf{C}$.
      (d) Wait for $\mathbf{C}$ to receive a message from $\mathbf{S}$ of the form $(\texttt{ProofGen}, \mathsf{qid}, V, \Pi)$. Store $(\mathsf{qid}, \mathsf{sk}_P, C, V, \Pi)$.
   c. Let $\mathbf{e}_0 \in \mathbb{Z}_q^m$ be a one-hot vector with "1" in the 0-th position. // *Dummy query*
   d. Sample $\mathsf{sk}_Q \leftarrow \mathsf{LHE.KeyGen}(1^\lambda)$, and let $\mathbf{u} \leftarrow \mathsf{LHE.Encrypt}(\mathsf{sk}_Q, \mathbf{e}_{i_c})$ // *Element-wise encryption. $\mathbf{u} \in \mathbb{Z}_q^m$*
   e. Store $(\mathsf{qid}, \mathbf{u})$, and send $(\mathsf{Query}, \mathsf{qid}, \mathbf{u})$ to $\mathbf{S}$ on behalf of $\mathbf{C}$.
5: Upon delivery of a message $(\mathsf{Response}, \mathsf{qid}, \mathbf{v})$ from $\mathbf{S}$ to an honest $\mathbf{C}$ with ID $\mathbf{C}$:
   a. If $\mathbf{S}$ is honest, set $P \leftarrow P \cup \{\mathbf{C}\}$, send $(\texttt{DeliverResponse}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$ and end execution.
   b. Otherwise, if $\mathbf{C} \notin P$:
      (a) Retrieve the client's preprocessing material $(\mathsf{qid}, \mathsf{sk}_P, C, V, \Pi)$. If none exists, end execution. // *Mirrors real-world client ignoring an unexpected response message*
      (b) Let dig be the digest contained in sid, and $Z^T \leftarrow \mathsf{LHE.Decrypt}(\mathsf{sk}_P, V)$ // *Element-wise decryption. $Z \in \mathbb{Z}_p^{\kappa \times m}$*
      (c) Send $(\texttt{Verify}, (A, \mathsf{dig}), \Pi)$ to $\mathcal{F}_{\mathsf{NIZK}}^R$ on behalf of $\mathbf{C}$. Wait for $\mathcal{A}$ to send a message $(\texttt{Witness}, \widetilde{\mathsf{DB}})$ to $\mathcal{F}_{\mathsf{NIZK}}^R$ (acting in the role of simulator for $\mathcal{F}_{\mathsf{NIZK}}^R$), and the client to receive output $(\texttt{Verification}, (A, \mathsf{dig}), \Pi, b)$.
      (d) If no database is stored, send $(\texttt{Init}, \widetilde{\mathsf{DB}})$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$.
      (e) If any of the following are true: (a) $b = 0$, (b) $Z \cdot A \neq C \cdot \mathsf{dig}$, or (c) $\|Z\|_\infty \geq B$: send $(\texttt{Response}, \mathsf{qid}, \perp)$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$ and end execution.
      (f) Otherwise, set $P \leftarrow P \cup \{\mathsf{pid}_{\mathbf{C}}\}$, store $(\mathsf{pid}_{\mathbf{C}}, C, Z)$ and continue.// *Store client's hint*
   c. Fetch the client's verification material $(\mathsf{pid}_{\mathbf{C}}, C, Z)$ and the client's query material $(\mathsf{qid}, \mathbf{u})$.
   d. If $Z\mathbf{u} \neq C\mathbf{v}$, send $(\texttt{Response}, \mathsf{qid}, \perp)$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$. // *This will cause the client to output* `VerificationFail`
   e. Otherwise, fetch the stored database DB and send $(\texttt{Response}, \mathsf{qid}, \mathsf{DB})$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$. // *If we reach this point, DB is guaranteed to be stored (it was extracted from a valid NIZK proof)*
6: Upon receiving $(\texttt{ReportResponse}, \mathsf{qid})$ from $\mathcal{F}_{\mathsf{VPIR}}$:
   a. If $\mathbf{S}$ is malicious, send $(\texttt{DeliverResponse}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$ and end execution. // *Response message has already been delivered*
   b. Fetch DB in storage. // DB *must have been leaked previously during init.*
   c. If a preprocessing message of the form $(\texttt{ProofGen}, \mathsf{qid}, U)$ was delivered to $\mathbf{S}$ from the client, and the server has not yet responded to it:
      (a) Let $V \leftarrow D^T \cdot U$. Have $\mathbf{S}$ send input $(\texttt{Prove}, \mathbf{S}, (A, \mathsf{dig}))$ to $\mathcal{F}_{\mathsf{NIZK}}$ to obtain a proof Π.
      (b) Send $(\texttt{ProofGen}, \mathsf{qid}, V, \Pi)$ on behalf of $\mathbf{S}$ to $\mathbf{C}$.
      (c) Wait for delivery of a message of the form $(\mathsf{Query}, \mathsf{qid}, \mathbf{u})$, sent from $\mathbf{C}$ to $\mathbf{S}$, then continue. // *Delivery will trigger line 3 first.*
   d. Fetch the tuple $(\mathsf{qid}, \mathbf{u})$ matching qid. // *Stored in line 4e.*
   e. Compute $\mathbf{v} \leftarrow \mathsf{DB} \cdot \mathbf{u}$, using the homomorphism of the LHE scheme.
   f. Send $(\mathsf{Response}, \mathsf{qid}, \mathbf{v})$ on behalf of $\mathbf{S}$ to $\mathbf{C}$.

(that is passed as input to the internal code) and computes the same SIS commitment as the real protocol. This produces identical behavior in the real and ideal world: if the commitment matches the digest dig contained in sid, the database is stored and Init returns $(\texttt{Initialized}, \mathsf{sid})$ to the server, and if not, ends execution. The second difference is addressed by having the simulator always respond $(\texttt{ReportInit}, \texttt{OK})$ with no delay to the functionality when the database is leaked. Therefore, the simulator perfectly simulates the real world Init protocol.

Query. Next, we consider the `Query` functionality and its real-world counterpart. We will examine separately the cases in which $\mathbf{C}$ is honest and corrupted by the adversary $\mathcal{A}$. In both cases, we show that the server $\mathbf{S}$ gets the same output at the same time in both real and ideal worlds.

In the honest $\mathbf{C}$ case, the real-world client ends execution if qid has been previously used; the ideal functionality performs an identical check. Supposing the check passes, in the ideal world execution is passed to the simulator, beginning at step 4.

In the real world, a client who has not completed preprocessing will send a `ProofGen` query to the server. This occurs if and only if the client has not yet successfully verified a preprocessing message upon receiving a response message. In the ideal world, the simulator maintains a set P of clients who have completed preprocessing, and if the client is not in P, it generates an identically distributed `ProofGen` message on behalf of the client. We will argue later that the simulator's set P is identical to the list of clients who have completed preprocessing in the real world.

Subsequently, the real-world client and simulator always generate a query message. The simulator's message is an LHE encryption of $\mathbf{e}_0$, while the real client's message is an LHE encryption of $\mathbf{e}_{i_c}$. By semantic security, these are indistinguishable.

In the real world, the adversary may delay or drop the query message over the network. To simulate this, the simulator waits for the adversary to deliver the query message (step 3), at which point it sends $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$. Hence, the simulator ensures that $\mathbf{S}$ gets its output at the same time in the ideal world as an honest server $\mathbf{S}$ does in the real world.

We now consider the case where $\mathbf{C}$ is corrupted by the adversary. In this case, we will only consider instances where $\mathbf{S}$ is honest. If both $\mathbf{C}$ and $\mathbf{S}$ are corrupted, simulation is trivial: the simulator does not need to interact with the adversary at all, since there is nothing to be hidden from the adversary.

Any time the adversary delivers a query $(\texttt{Query}, \mathsf{qid}, \mathbf{u})$ message from a corrupted client to an honest server, the real world server outputs $(\texttt{Query}, \mathsf{qid})$. It is the simulator's task to produce the same behavior in the ideal world. When a query message from a malicious client is delivered, the simulator is invoked on line 3. At this point, the simulator calls `Query` ideal functionality on behalf of the client, using 0 as a dummy index. This passes control to the functionality, which immediately hands control back to the simulator (on line 3) by sending a $(\texttt{ReportQuery}, \mathsf{qid})$ message. The simulator then immediately sends $(\texttt{DeliverQuery}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$. This causes the server to output $(\texttt{Query}, \mathsf{qid})$, identically to the real-world behavior.

Response. We consider the cases where $\mathbf{S}$ is honest and $\mathbf{S}$ is corrupted separately.

First, suppose $\mathbf{S}$ is honest. When the environment induces an honest real-world server to run the $\mathsf{Response}$ protocol with inputs $(\mathsf{qid}, \mathsf{DB})$, the real-world client first checks four conditions (lines b.a-b.d) and ends execution if any are not satisfied.

The first three conditions are identically checked in the ideal functionality. Note that the if the first condition passes (in both worlds), then $\mathbf{S}$ has successfully run Init. In the ideal world, this means that a database DB was leaked to the simulator, such that the commitment of DB matches dig.

For the fourth condition, the real world protocol checks that $\mathsf{Encode}(\mathsf{DB}) \cdot A = \mathsf{dig}$, while the ideal functionality instead checks that the provided database $\widetilde{\mathsf{DB}}$ matches the database DB used in Init. Suppose the environment has produced inputs that cause the checks to behave differently in the real and ideal world: i.e., $\widetilde{DB} \neq \mathsf{DB}$, but $\mathsf{Encode}(\mathsf{DB}) \cdot A = \mathsf{Encode}(\widetilde{\mathsf{DB}}) \cdot A = \mathsf{dig}$. Call this event $\mathcal{B}$. Denote $D = \mathsf{Encode}(\mathsf{DB})$ and $\widetilde{D} = \mathsf{Encode}(\widetilde{\mathsf{DB}})$. Then, $(D - \widetilde{D}) \cdot A = 0$. We have that $D - \widetilde{D} \neq 0$ (since Encode is injective), and $D - \widetilde{D}$ is a short vector (since D and $\widetilde{D}$ are both short). This yields a SIS solution for A, which is hard to find by assumption. Therefore, event $\mathcal{B}$ happens with negligible probability.

Assuming all aforementioned checks pass, the ideal functionality hands off execution to the simulator, who begins at line 6. If a `ProofGen` message matching qid was sent from $\mathbf{C}$ to $\mathbf{S}$, the simulator generates a `ProofGen` response message. This message is generated identically to the real protocol, provided event $\mathcal{B}$ did not occur. The simulator then waits for $\mathbf{S}$ to receive a `Query` message from $\mathbf{C}$.

Next, the simulator computes a response message in an identical manner as the real protocol and sends it over the network (again, provided $\mathcal{B}$ did not occur).

At this point, if the client $\mathbf{C}$ is corrupted, simulation for `Response` is complete. If $\mathbf{C}$ is honest, we must additionally simulate their response. This is done when the response message is delivered (step 5) — the simulator simply sends $(\texttt{DeliverResponse}, \mathsf{qid})$ to $\mathcal{F}_{\mathsf{VPIR}}$, which gives $\mathbf{C}$ the correct output. At the same time, they mark the client as having completed preprocessing by adding $\mathbf{C}$ to the set P. Note that we do not need to recompute the client's verification checks in this case, since both $\mathbf{C}$ and $\mathbf{S}$ are honest, so verification will always pass.

Now we turn to the case where $\mathbf{S}$ is corrupted. In this case, we assume that $\mathbf{C}$ is honest — if both $\mathbf{C}$ and $\mathbf{S}$ are corrupted, simulation is trivial.

In this case, the database will not be leaked to the simulator through the Init functionality, so we must extract it by other means. When the adversary delivers a $(\texttt{Response}, \mathsf{qid}, \mathbf{v})$ message on behalf of the server, the simulator is invoked on line 5. The simulator at this point attempts to perform the client's preprocessing verification (if the client has not yet completed preprocessing). In doing so, the simulator verifies the NIZK proof using $\mathcal{F}^R_{\mathsf{NIZK}}$. For this to verify, the adversary must send a witness $(\widetilde{DB})$ to $\mathcal{F}_{\mathsf{NIZK}}$ satisfying the relation $R((A, \mathsf{dig}), \widetilde{DB}) = 1$; in other words, dig must be a valid SIS commitment to $\widetilde{DB}$. The simulator will intercept this message, allowing them to learn $\widetilde{\mathsf{DB}}$ and extract the database.

Subsequently, the simulator mirrors the rest of the real world client's checks of the preprocessing data. If the checks fail, the simulator sends `Response` to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$, using $\perp$ as the database. This sets up the functionality to output `VerificationFail` to the client: execution is passed to the functionality, who sends $(\texttt{ReportResponse}, \mathsf{qid})$ to the simulator, who runs line 6 and immediately sends $(\texttt{DeliverResponse}, \mathsf{qid})$ back to $\mathcal{F}_{\mathsf{VPIR}}$. Subsequently the functionality outputs `VerificationFail` to $\mathbf{C}$. Note that this matches the behavior of the real world client when the preprocessing checks fail. Assuming the preprocessing check succeeds, the simulator marks the client as having completed preprocessing by adding their pid to the set P. In the real world, the client sets their internal variable $\mathsf{preprocessed} \leftarrow \mathsf{true}$. This ensures that the set of preprocessed clients in the real world always exactly matches the set P in the ideal world.

Next, the simulator replicates the honest client's verification check of the response message. If this check fails, the simulator sends $(\texttt{Response}, \mathsf{qid}, \perp)$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$. This causes the ideal world client to receive output $(\texttt{VerificationFail}, \mathsf{qid})$, as discussed above in the case where the preprocessing checks fail. Otherwise, the simulator recovers the database DB leaked during

Init and sends $(\texttt{Response}, \mathsf{qid}, \mathsf{DB})$ to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of $\mathbf{S}$. We observe that DB must have been successfully extracted from the NIZK proof of preprocessing (if not, no valid NIZK was sent, so the preprocessing checks would have failed). Now, the ideal world client will receive $(\texttt{Answer}, \mathsf{qid}, \mathsf{DB}[i])$ as an output ($\mathcal{F}_{\mathsf{VPIR}}$ sends $(\texttt{ReportResponse}, \mathsf{qid})$ to the simulator, who immediately responds with $(\texttt{DeliverResponse}, \mathsf{qid})$ on line 6a., causing $\mathcal{F}_{\mathsf{VPIR}}$ to give $\mathbf{C}$ their output).

In the real world, the client sets $\mathbf{y} \leftarrow \mathsf{LHE.Decrypt}(sk, \mathbf{v})$ and outputs $(\texttt{Answer}, \mathsf{qid}, \mathbf{y}[i_r])$. By Lemma 4.1 in [36], we have that $\mathbf{v} = D\mathbf{u}$ with overwhelming probability. Then by correctness of the LHE scheme, we have that $y[i] = \mathsf{DB}[i]$. Therefore, the client's output in the real and ideal world is computationally indistinguishable.

We now inductively apply the three arguments above (corresponding to `Init`, `Query`, and `Response`) to each action done by the environment, to obtain that the adversary's advantage in distinguishing between the ideal and real worlds is always negligible. We conclude that $\Pi_{\mathsf{VeriSimplePIR}}$ UC-realizes $\mathcal{F}_{\mathsf{VPIR}}$.

□

E Π_{UVPIR} UC-realizes $\mathcal{F}_{\mathsf{UVPIR}}$

The protocol Π_{UVPIR} for our updatable VPIR scheme is given in algorithm 12 and algorithm 13. The simulator for our proof is given in algorithm 14 and algorithm 15. We prove our main theorem below.

E.1 Security Proof of Π_{UVPIR}

Theorem 3. *The protocol Π_{UVPIR} UC-realizes $\mathcal{F}_{\mathsf{UVPIR}}$ in the presence of global functionalities $\mathcal{G}_{\mathsf{lib}}$ and $\mathcal{G}_{\mathsf{Sig}}$, and local functionalities $\mathcal{F}_{\mathsf{NIZK}}$. We assume an adversary that can statically corrupt any subset of parties, and may delay or drop network messages, but cannot modify or spoof them.*

Proof. We present a simulator in algorithm 14 and algorithm 15. Our proof consists of showing that each action the environment takes produces indistinguishable views in the real and ideal world. Then, we conclude by induction that for any sequence of environment actions, the adversary cannot distinguish between real and ideal worlds.

Init. The behavior of Init in the ideal functionality and protocol differs only in the type of local state initialized, and the fact that the functionality leaks the DB to the server and waits for an `OK` response. The difference in local state does not affect the adversary's view, and the simulator immediately provides the `OK` response to $\mathcal{F}_{\mathsf{UVPIR}}$. Therefore, simulation is perfect.

Update. First, we consider the case where the client is honest. In this case, the simulator waits for the client to receive an `UpdateResponse` message from the server, and then performs the same checks as the client in the real protocol. The simulator ensures that the client's output in the ideal world is released upon delivery of a `AckUpdate` message from the server, mirroring the behavior of the real world. In the real world, the client verifies a proof from the server that the $\mathsf{ssid}_{\mathsf{new}}$ obtained from the server is consistent with their update. By the security of $\mathcal{F}_{\mathsf{NIZK}}$, this prevents a malicious server from cheating by sending an invalid $\mathsf{ssid}_{\mathsf{new}}$. Therefore, the real-world client updates their local variables $\mathsf{currentSSID}$ and $\mathsf{version}$ only for correct updates. The simulator tracks the state of these variables for all clients in the $\mathsf{clientView}$ map. This map is updated only when the real-world client would update their local state, ensuring that the simulator maintains a consistent representation of

all real-world clients' state. If the server is honest, their ideal-world output is released upon delivery of the client's `AuthUpdate` message, as in the real world. Finally, the outputs of all parties are identical in both worlds. Therefore, simulation in this case is perfect.

When the client is malicious and the server is honest, the simulator waits for delivery of a `UpdateRequest` message to the server, and then translates this into the ideal world by sending an `Update` input to $\mathcal{F}_{\mathsf{VPIR}}$ on behalf of the client. The functionality then performs a series of checks that mirror the honest server's checks in the real protocol. Upon the functionality returning control to the simulator, the simulator checks that the client is updating from a leaf node in updateGraph, as an analogue to the real-world protocol checking that the client is updating from the most recent version in updateList. In fact, when the server is honest updateGraph is a linked list, since the server's protocol and simulator will both enforce that new nodes can only be added off of leaf nodes. Inductively, this ensures a single leaf node at all times. (The simulator represents updateGraph as a DAG to handle the case in which the adversary corrupts the server and a writing client, in which case forks and merges can occur). Subsequently, the real-world server adds a new entry to updateList corresponding to the client's update, and the simulator mirrors this by adding a new leaf node to updateGraph, ensuring that updateList and updateGraph represent the same sequence of updates. After the simulator simulating the `AckUpdate` message and sends (`UpdateStatus`, uid, label) to $\mathcal{F}_{\mathsf{UVPIR}}$, the ideal-world client obtains their output, identically and at the same time as the real-world server. We do not need to simulate the client's output since they are malicious. Therefore, simulation is perfect.

Sync. We separately consider the cases where the syncing client is honest and the syncing client is malicious. First, suppose the client is malicious and the server is honest. In this case there are no output to honest parties, and the only goal of the simulator is to ensure that the internal state of $\mathcal{F}_{\mathsf{UVPIR}}$ is consistent with the view of the honest server. This is straightforward; the simulator simply sends a `ClientSync` input to $\mathcal{F}_{\mathsf{UVPIR}}$ on behalf of the malicious client, and updates the simulated server's view serverView according to what an honest server would do.

Now suppose the syncing client is honest. In this case, real-world client's view of their current state, before the sync (stored in currentSSID and version) and the simulated client's view of their current state (stored in clientView) are identical, as argued previously. Next, the simulator replicates the real-world client's verification of the update signature chain. We now argue that following a successful sync, the real-world client's updated view of their internal state is consistent with clientView in the ideal functionality. For each update in the real-world update chain, we show that it corresponds to an update in the ideal world. If an update was signed by an uncorrupted client, then it must have correspond to a valid update made by that client, by security of $\mathcal{G}_{\mathsf{Sig}}$. If an update is signed by a corrupted writing client, then a malicious server and writing client may have colluded to produce a signed update that does not exist in $\mathcal{F}_{\mathsf{UVPIR}}$. In this case, the simulator induces the corresponding update in $\mathcal{F}_{\mathsf{UVPIR}}$ by sending an Update input on behalf of the corrupted client and allowing a new vertex and edge to be added to updateGraph in $\mathcal{F}_{\mathsf{UVPIR}}$. Finally, we have that both the real-world and ideal-world client's output is delivered at the same time, since the simulator allows $\mathcal{F}_{\mathsf{UVPIR}}$ to release the ideal-world output upon the real-world delivery of the `SyncPath` message from the server. Therefore, simulation is perfect.

Finally, by inductively applying the arguments above, we can conclude that Π_{UVPIR} perfectly UC-realizes $\mathcal{F}_{\mathsf{UVPIR}}$. □

Algorithm 12 Protocol Π_{UVPIR}, Part 1

This protocol is parameterized by the database size N and uses an sid which contains the pid of a server $\mathbf{S}$ and a session ID ssid_0 of a $\mathcal{F}_{\mathsf{VPIR}}$ instance, corresponding to the initial database. All clients upon first activation will set and store variables $\mathsf{currentSSID} \leftarrow \mathsf{ssid}_0$, and send $(\texttt{KeyGen})$ to $\mathcal{G}_{\mathsf{Sig}}$ to generate a signing key. Let R' be the relation such that $R'((\mathsf{ssid}, \mathsf{ssid}', \Delta), \mathsf{DB}, \mathsf{DB}') = 1$ if and only if $\mathsf{DB}' = \mathsf{DB} + \Delta$, and for the internal code function $\mathsf{DigestVerifyCode}$ $\mathcal{V}$ stored in $\mathcal{G}_{\mathsf{lib}}$, both $\mathcal{V}(\mathsf{ssid}, \mathsf{DB})$ and $\mathcal{V}(\mathsf{ssid}', \mathsf{DB}')$ both accept.

1: **Init:** Upon receiving $(\texttt{Init}, \mathsf{sid}, \mathsf{DB})$, the server does:
   1. If $\mathsf{initialized} = \mathsf{true}$, end execution. Otherwise, parse ssid_0 from sid and send $(\texttt{Init}, \mathsf{ssid}, \mathsf{DB})$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}_0}$ on behalf of $\mathbf{S}$. If $(\texttt{Initialized}, \mathsf{ssid})$ is not outputted, end execution.
   2. Initialize a map $\mathsf{serverView}$. For all clients $\mathbf{C}_i$, in their first interaction with the server, the server will set $\mathsf{serverView}[\mathbf{C}_i] \leftarrow \mathsf{ssid}_0$. // *Maps each client to a $\mathcal{F}_{\mathsf{VPIR}}$ instance consistent with what the server thinks is their current version.*
   3. Initialize an empty list $\mathsf{updateList}$. // *Represents the update history of the DB. Server will enforce a single, sequential ordering of database updates.*
   4. Set $\mathsf{initialized} \leftarrow \mathsf{true}$, store the database DB, and output $(\mathsf{initialized}, \mathsf{sid})$.

2: **Query:** Upon receiving $(\mathsf{Query}, \mathsf{sid}, \mathsf{qid}, i \in [N])$, a client $\mathbf{C}$ does:
   1. Let $\mathsf{ssid} \leftarrow \mathsf{currentSSID}$. Forward $(\mathsf{Query}, \mathsf{ssid}, \mathsf{qid}, i)$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}}$.

3: **Response:** Upon receiving $(\mathsf{Response}, \mathsf{sid}, \mathsf{qid}, \mathsf{DB})$ the server $\mathbf{S}$ does
   1. If $\mathsf{initialized} = \mathsf{false}$, end execution. Otherwise, parse the client ID pid from the inputted qid, and let $\mathsf{ssid} \leftarrow \mathsf{serverView}[\mathbf{C}]$. Forward the message $(\mathsf{Response}, \mathsf{ssid}, \mathsf{qid}, \mathsf{DB})$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}}$.

4: **Update:** Upon receiving $(\texttt{Update}, \mathsf{sid}, \mathsf{uid}, \Delta \in \Sigma^N)$, a client $\mathbf{C}$ does:
   1. If $\mathbf{C} \notin \mathsf{authClients}$ or uid has been used in a previous $\texttt{Update}$ call, end execution.
   2. Let $\mathsf{ssid}_{\mathsf{old}} \leftarrow \mathsf{currentSSID}$ and send $(\texttt{UpdateRequest}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{old}}, \mathsf{version}, \Delta)$ to the server $\mathbf{S}$.
   3. If the server responds with $(\texttt{InvalidDigest}, \mathsf{uid})$, output $(\texttt{UpdateReject}, \mathsf{sid}, \mathsf{uid})$. // *Client should sync to the latest version*
   4. Wait to receive $(\texttt{UpdateResponse}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{new}}, \pi_{\mathsf{Update}})$ from the server $\mathbf{S}$.
   5. Send $(\mathsf{Verify}, (\mathsf{ssid}_{\mathsf{old}}, \mathsf{ssid}_{\mathsf{new}}, \Delta), \pi_{\mathsf{Update}})$ to $\mathcal{F}_{\mathsf{NIZK}}^{R'}$. If the output indicates verification failed, end execution. // *Proof that $\mathsf{ssid}_{\mathsf{new}}$ is consistent with a new database according to the update Δ.*
   6. Otherwise, if $\mathcal{F}_{\mathsf{NIZK}}^{R'}$ indicates successful verification, let $m = (\mathsf{ssid}_{\mathsf{old}} \parallel \mathsf{ssid}_{\mathsf{new}} \parallel (\mathsf{version} + 1) \parallel \mathbf{C})$ and send $(\texttt{Sign}, m)$ to $\mathcal{G}_{\mathsf{Sig}}$ and wait to receive $(\texttt{Signature}, m, \sigma,)$.
   7. Send $(\texttt{AuthUpdate}, \mathsf{uid}, m, \sigma)$ to the server $\mathbf{S}$. // *Client authorizes the new ssid by giving the server a signature on the update*
   8. Upon receiving $(\texttt{AckUpdate}, \mathsf{uid})$ from $\mathbf{S}$, update local variables $\mathsf{currentSSID} \leftarrow \mathsf{ssid}_{\mathsf{new}}$ and $\mathsf{version} \leftarrow \mathsf{version} + 1$, then output $(\texttt{UpdateComplete}, \mathsf{sid}, \mathsf{uid})$.

5: *UpdateResponse:* Upon receiving $(\texttt{UpdateRequest}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{old}}, \mathsf{version}, \Delta \in \Sigma^N)$ from a client $\mathbf{C}$, the server $\mathbf{S}$ does:
   1. End execution if any of the following are true: (a) $\mathsf{initialized} = \mathsf{false}$, (b) $\mathbf{C} \notin \mathsf{authClients}$, (c) uid has been used in a previous update.
   2. Look up the client's current SSID $\mathsf{ssid} \leftarrow \mathsf{serverView}[\mathbf{C}]$. If either $\mathsf{ssid} \neq \mathsf{ssid}_{\mathsf{old}}$, or $\mathsf{ssid}_{\mathsf{old}}$ and $\mathsf{version}$ are not the most recent session ID and version number contained in the most recent m value in $\mathsf{updateList}$, send $(\texttt{InvalidDigest}, \mathsf{uid})$ to $\mathbf{C}$ and end execution. // *Reject updates from clients who are not synced to the latest update*
   3. Let $\mathsf{DB}' \leftarrow \mathsf{DB} + \Delta$. // *Compute updated database from the most recent DB*
   4. Send $(\texttt{Retrieve}, \mathsf{SIDUpdateCode})$ to $\mathcal{G}_{\mathsf{lib}}$ to obtain code $\mathcal{U}$ and compute $\mathsf{ssid}_{\mathsf{new}} \leftarrow \mathcal{U}(\mathsf{ssid}_{\mathsf{old}}, \mathsf{DB}', \Delta)$. // *Internal code to compute the new $\mathsf{ssid}_{\mathsf{new}}$, derived from the old ssid and the update diff.*
   5. If $\mathsf{ssid}_{\mathsf{new}}$ is not part of any existing element in $\mathsf{updateList}$, send $(\texttt{Init}, \mathsf{ssid}_{\mathsf{new}}, \mathsf{DB}')$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}_{\mathsf{new}}}$ and record DB'. If $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}_{\mathsf{new}}}$ returns $(\texttt{InvalidDigest}, \mathsf{ssid}_{\mathsf{new}})$, end execution.// *Initialize a new $\mathcal{F}_{\mathsf{VPIR}}$ instance. Inside $\mathcal{F}_{\mathsf{VPIR}}$, this checks that the digest is consistent with the database.*
   6. If $\mathsf{ssid}_{\mathsf{new}}$ already exists in $\mathsf{updateList}$, check that DB' is equal to the database used to initialize $\mathsf{ssid}_{\mathsf{new}}$. If not, end execution.
   7. Send $(\texttt{Prove}, (\mathsf{ssid}_{\mathsf{old}}, \mathsf{ssid}_{\mathsf{new}}, \Delta), (\mathsf{DB}, \mathsf{DB}'))$ to $\mathcal{F}_{\mathsf{NIZK}}^{R}$ and receive back $(\texttt{Proof}, \pi_{\mathsf{Update}})$.
   8. Send the message $(\texttt{UpdateResponse}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{new}}, \pi_{\mathsf{Update}})$ to the client $\mathbf{C}$.
   9. Upon receiving $(\texttt{AuthUpdate}, \mathsf{uid}, \sigma, m)$ from $\mathbf{C}$, check that $m = (\mathsf{ssid}_{\mathsf{old}} \parallel \mathsf{ssid}_{\mathsf{new}} \parallel \mathsf{version} + 1 \parallel \mathbf{C})$. Then send $(\mathsf{Verify}, m, \sigma, \mathsf{vk})$ to $\mathcal{G}_{\mathsf{Sig}}$, where vk is the client's verification key, previously obtained from $\mathcal{G}_{\mathsf{Sig}}$. End execution if the check fails or verification is unsuccessful.
   10. Append an element (m, σ) to $\mathsf{updateList}$ and set $\mathsf{serverView}[\mathbf{C}] \leftarrow \mathsf{ssid}_{\mathsf{new}}$.
   11. Send $(\texttt{AckUpdate}, \mathsf{qid})$ to $\mathbf{C}$, and output $(\texttt{UpdateComplete}, \mathsf{sid}, \mathsf{uid})$

Algorithm 13 Protocol Π_{UVPIR}, Part 2

1: **Sync:** Upon receiving $(\texttt{Sync}, \mathsf{sid})$ from the environment, the client $\mathbf{C}$ does:
   1. Let $\mathsf{ssid} \leftarrow \mathsf{currentSSID}$ and look up the local variable $\mathsf{version}$. Send $(\texttt{ClientSync}, \mathsf{version}, \mathsf{ssid})$ to the server.
   2. Wait to receive response $(\texttt{SyncPath}, [(m_i, \sigma_i)]_{i \in [k]})$ from the server. Parse each m_j as $(\mathsf{ssid}_{\mathsf{old},j} \parallel \mathsf{ssid}_{\mathsf{new},j} \parallel \mathsf{version}_j \parallel \mathbf{C}_j)$. For notational ease, denote $\mathsf{ssid}_{\mathsf{new},0} \leftarrow \mathsf{ssid}$ and $\mathsf{version}_0 \leftarrow \mathsf{version}$. Then, for $j \in \{1, \ldots, k\}$, check the following:
      (a) $\mathsf{ssid}_{\mathsf{old},j} = \mathsf{ssid}_{\mathsf{new},j-1}$ // *Enforce a consistency chain between each update*
      (b) $\mathbf{C}_j \in \mathsf{authClients}$ // *Enforce that each update is made by an authorized writing client*
      (c) $\mathsf{version}_j = \mathsf{version}_{j-1} + 1$ // *Enforce that version number increments by one at each update*
      (d) Send $(\mathsf{Verify}, m_j, \sigma_j, \mathsf{vk}_j,)$ to $\mathcal{G}_{\mathsf{Sig}}$, where vk_j is the verification key of $\mathbf{C}_j$. Check that the output verifies. // *Verify the signature itself*
      (e) Generate a fresh qid and send $(\mathsf{Query}, \mathsf{ssid}_{\mathsf{new},j}, \mathsf{qid}, 0)$ to $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}_{\mathsf{new},j}}$ and wait for successful response $(\texttt{Answer}, \mathsf{qid}, \cdot)$ or failure $(\texttt{VerificationFail}, \mathsf{qid})$
   3. If any of the previous check fails, end execution. Otherwise, set $\mathsf{version} \leftarrow \mathsf{version} + k$, $\mathsf{currentSSID} \leftarrow \mathsf{ssid}_{\mathsf{new},k}$, and output $(\texttt{Synced}, \mathsf{sid})$.

2: *SyncResponse:* Upon receiving message $(\texttt{ClientSync}, \mathsf{version}, \mathsf{ssid})$ from a client $\mathbf{C}$, the server does:
   1. Let ℓ be the size of $\mathsf{updateList}$. If $\mathsf{version} \geq \ell$, end execution. // *Prevent malicious client from "syncing ahead" of the server*
   2. Otherwise, let $(m, \sigma) \leftarrow \mathsf{updateList}[\mathsf{version}]$, and parse m as $(\mathsf{ssid}'_{\mathsf{old}} \parallel \mathsf{ssid}'_{\mathsf{new}} \parallel \mathsf{version}' \parallel \mathbf{C}')$. If $(\mathsf{ssid}'_{\mathsf{new}}, \mathsf{version}') \neq (\mathsf{ssid}, \mathsf{version})$, end execution. // *Prevent client from syncing from an inconsistent state*
   3. Let $\mathsf{path} = [\mathsf{updateList}[\mathsf{version}], \mathsf{updateList}[\mathsf{version}+1], \ldots, \mathsf{updateList}[\ell - 1]]$. Send $(\texttt{SyncPath}, \mathsf{path})$ to $\mathbf{C}$. Set $\mathsf{serverView}[\mathbf{C}]$ to the value of $\mathsf{ssid}_{\mathsf{new}}$ in the last entry of $\mathsf{updateList}$, and end execution.

Algorithm 14 UVPIR Simulator, part 1

1: On receiving any message to/from $\mathcal{F}_{\mathsf{VPIR}}$ and the corrupted parties or real adversary, the simulator forwards the message along without modification, recording any $(\mathsf{ssid}, \mathsf{DB})$ pairs for later extraction.
2: On activation:
   1. Initialize an empty map clientView. On the first activation of any client $\mathbf{C}_i$, set $\mathsf{clientView}[\mathbf{C}_i] \leftarrow (\mathsf{ssid}_0, 0)$, where ssid_0 is parsed from the sid. // *This tracks the client's perspective of which $\mathcal{F}_{\mathsf{VPIR}}$ instance, real-world version, and ideal-world node label they are on.*
   2. Initialize an empty map serverView. On the first activation of any client $\mathbf{C}_i$, set $\mathsf{serverView}[\mathbf{C}_i] \leftarrow (\mathsf{ssid}_0, 0)$, where ssid_0 is parsed from the sid. // *This tracks the server's perspective of each client's view.*
   3. Initialize an empty DAG updateGraph.
   4. Let $\mathcal{U}$ be code such that $\mathcal{U}(\mathsf{ssid}, \mathsf{DB}, \Delta)$ computes $\mathsf{DB}' \leftarrow \mathsf{DB} + \Delta$, and returns an $\mathcal{F}_{\mathsf{VPIR}}$ session ID ssid' consistent with DB'. Send $(\texttt{Store}, \mathsf{SIDUpdateCode}, \mathcal{U})$ to $\mathcal{G}_{\mathsf{lib}}$.
3: On receiving $(\texttt{ReportInit}, \mathsf{sid}, \mathsf{DB})$ from $\mathcal{F}_{\mathsf{UVPIR}}$:
   1. Add a note $(\mathsf{ssid}_0, 0, \mathsf{DB})$ to updateGraph, then send $(\texttt{ReportInit}, \mathsf{sid}, \texttt{OK})$ to $\mathcal{F}_{\mathsf{UVPIR}}$. // *By learning the initial DB, the simulator can compute all future updated DBs.*
4: On receiving $(\texttt{UpdateRequest}, \mathsf{uid}, \mathbf{C}, \mathsf{ssid}', \Delta)$ from $\mathcal{F}_{\mathsf{UVPIR}}$:
   1. End execution early if $\mathbf{C}$ is malicious. // *The malicious client case is handled in step 5 of algorithm 15.*
   2. If this is the first update request made by this client, send $(\texttt{KeyGen})$ to $\mathcal{G}_{\mathsf{Sig}}$ on behalf of $\mathbf{C}$, and store the signing key.
   3. Let $(\mathsf{ssid}_{\mathsf{old}}, \mathsf{version}) \leftarrow \mathsf{clientView}[\mathbf{C}]$. Send message $(\texttt{UpdateRequest}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{old}}, \mathsf{version}, \Delta)$ to $\mathbf{S}$ on behalf of $\mathbf{C}$.
   4. If a message of the form $(\texttt{InvalidDigest}, \mathsf{uid})$ is delivered to $\mathbf{C}$, sent by $\mathbf{S}$, send $(\texttt{UpdateStatus}, \mathsf{uid}, \bot)$ to $\mathcal{F}_{\mathsf{UVPIR}}$ and end execution.
   5. Wait for delivery of a message of the form $(\texttt{UpdateResponse}, \mathsf{sid}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{new}}, \pi_{\mathsf{Update}})$, sent from $\mathbf{S}$ to $\mathbf{C}$.
   6. Send $(\mathsf{Verify}, (\mathsf{ssid}_{\mathsf{old}}, \mathsf{ssid}_{\mathsf{new}}, \Delta), \pi_{\mathsf{Update}})$ to $\mathcal{F}_{\mathsf{NIZK}}^{R'}$. If the output indicates verification failed, end execution.
   7. Otherwise, if $\mathcal{F}_{\mathsf{NIZK}}^{R'}$ indicates successful verification, let $m = (\mathsf{ssid}_{\mathsf{old}} \parallel \mathsf{ssid}_{\mathsf{new}} \parallel (\mathsf{version} + 1) \parallel \mathbf{C})$ and send $(\texttt{Sign}, m)$ to $\mathcal{G}_{\mathsf{Sig}}$ on behalf of $\mathbf{C}$ and wait to receive $(\texttt{Signature}, m, \sigma)$.
   8. Send a message $(\texttt{AuthUpdate}, \mathsf{uid}, m, \sigma)$ to the server $\mathbf{S}$ on behalf of $\mathbf{C}$.
   9. When either the previous message is delivered, or upon delivery of a message $(\texttt{AckUpdate}, \mathsf{uid})$ from $\mathbf{S}$ to $\mathbf{C}$ (whichever comes first), do the following:
      (a) Find the node $(\mathsf{ssid}_{\mathsf{old}}, \mathsf{version}, \mathsf{DB})$ in updateGraph. Let $\mathsf{DB}' = \mathsf{DB} + \Delta$. // *This node is guaranteed to exist because the simulator adds any label that exists in clientView to a node in updateGraph.*
      (b) Add a node $(\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1, \mathsf{DB}')$ and an edge from $(\mathsf{ssid}_{\mathsf{old}}, \mathsf{version}, \mathsf{DB})$ to $(\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1, \mathsf{DB}')$ into updateGraph.
      (c) Set $\mathsf{clientView}[\mathbf{C}] \leftarrow (\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1)$.
      (d) Send $(\texttt{UpdateStatus}, \mathsf{uid}, (\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1))$ to $\mathcal{F}_{\mathsf{UVPIR}}$. // *Ideal-world node labels have the form (ssid, version)*
   10. If the $(\texttt{AckUpdate}, \mathsf{uid})$ message was delivered to $\mathbf{C}$, release the output of $\mathcal{F}_{\mathsf{UVPIR}}$ to $\mathbf{C}$. Otherwise, delay this output until delivery of the AckUpdate message.
5: On receiving $(\texttt{ClientSync}, \mathbf{C})$ from $\mathcal{F}_{\mathsf{UVPIR}}$:
   1. If $\mathbf{C}$ is malicious, end execution. // *Malicious clients handled in step 6 of algorithm 15.*
   2. Let $(\mathsf{ssid}, \mathsf{version}) \leftarrow \mathsf{clientView}[\mathbf{C}]$. Send a message $(\texttt{ClientSync}, \mathsf{version}, \mathsf{ssid})$ to $\mathbf{S}$ on behalf of $\mathbf{C}$.
   3. Wait for delivery of a message $(\texttt{SyncPath}, [(m_i, \sigma_i)]_{i \in [k]})$, sent from $\mathbf{S}$ to $\mathbf{C}$. Parse each m_j as $(\mathsf{ssid}_{\mathsf{old},j} \parallel \mathsf{ssid}_{\mathsf{new},j} \parallel \mathsf{version}_j \parallel \mathbf{C}_j)$. For notational ease, denote $\mathsf{ssid}_{\mathsf{new},0} \leftarrow \mathsf{ssid}$ and $\mathsf{version}_0 \leftarrow \mathsf{version}$. Then, for $j \in \{1, \ldots, k\}$, check the following:
      (a) $\mathsf{ssid}_{\mathsf{old},j} = \mathsf{ssid}_{\mathsf{new},j-1}$ // *Enforce a consistency chain between each update*
      (b) $\mathbf{C}_j \in \mathsf{authClients}$
      (c) $\mathsf{version}_j = \mathsf{version}_{j-1} + 1$
      (d) Send $(\mathsf{Verify}, m_j, \sigma_j, \mathsf{vk}_j)$ to $\mathcal{G}_{\mathsf{Sig}}$, where vk_j is the verification key of $\mathbf{C}_j$. Check that the output verifies.
      (e) Generate a fresh qid and send $(\mathsf{Query}, \mathsf{ssid}_{\mathsf{new},j}, \mathsf{qid}, 0)$ to the (simulated) $\mathcal{F}_{\mathsf{VPIR}}^{\mathsf{ssid}_{\mathsf{new},j}}$ on behalf of $\mathbf{C}$ and make sure the (simulated) response has the form $(\texttt{Answer}, \mathsf{qid}, \cdot)$.
   4. If any of the previous check fails, end execution.
   5. For $j \in \{1, \ldots, k\}$:
      (a) If there does not exist a node $(\mathsf{ssid}_{\mathsf{new},j}, \mathsf{version}_j)$ in updateGraph:
         i. Lookup $\mathsf{DB}_{\mathsf{old}}, \mathsf{DB}_{\mathsf{new}}$ corresponding to $\mathsf{ssid}_{\mathsf{old},j}, \mathsf{ssid}_{\mathsf{new},j}$ from intercepted $\mathcal{F}_{\mathsf{VPIR}}$ messages and calculate $\Delta_j \leftarrow \mathsf{DB}_{\mathsf{new}} - \mathsf{DB}_{\mathsf{old}}$
         ii. Generate a unique uid and send $(\texttt{Update}, \mathsf{sid}, \mathsf{uid}, \mathbf{C}_j, \Delta_j)$ to $\mathcal{F}_{\mathsf{UVPIR}}$ on behalf of $\mathbf{C}_j$. When the functionality responds with $(\texttt{MaliciousUpdateStatus}, \mathsf{uid}, \mathbf{C}_j)$, let $\mathsf{label} \leftarrow (\mathsf{ssid}_{\mathsf{old},j}, \mathsf{version}_{j-1})$ and send back $(\texttt{MaliciousUpdateStatus}, \mathsf{uid}, \mathbf{C}_j, \mathsf{ssid}_{\mathsf{old},j}, \mathsf{label})$. Then when $\mathcal{F}_{\mathsf{UVPIR}}$ sends $(\texttt{UpdateRequest}, \mathsf{uid}, \mathbf{C}_j, \mathsf{ssid}_{\mathsf{new},j}, \Delta_j)$, let $\mathsf{label}' \leftarrow (\mathsf{ssid}_{\mathsf{new},j}, \mathsf{version}_j)$ and respond with $(\texttt{UpdateStatus}, \mathsf{uid}, \mathsf{label}')$. // *Make a malicious update. This will only occur when a malicious server colludes with a writing client.*
      (b) Add an edge in updateGraph from $(\mathsf{ssid}_{\mathsf{old},j}, \mathsf{version}_{j-1})$ to $(\mathsf{ssid}_{\mathsf{new},j}, \mathsf{version}_j)$, if one does not yet exist.
   6. Set $\mathsf{clientView}[\mathbf{C}] \leftarrow (\mathsf{ssid}_{\mathsf{new},k}, \mathsf{version} + k)$.
   7. Send $(\texttt{SyncClientState}, \mathbf{C}, \mathsf{ssid}_{\mathsf{new},k})$ to $\mathcal{F}_{\mathsf{UVPIR}}$.

Algorithm 15 UVPIR Simulator, part 2

5: On delivery of a message $(\texttt{UpdateRequest}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{old}}, \mathsf{version}, i, \Delta)$ from a malicious client $\mathbf{C}$ to honest $\mathbf{S}$:
   1. Send $(\texttt{Update}, \mathsf{sid}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{old}}, i, \Delta)$ to $\mathcal{F}_{\mathsf{UVPIR}}$ on behalf of $\mathbf{C}$. Wait to receive $(\texttt{UpdateRequest}, \mathsf{uid}, \mathbf{C}, \mathsf{ssid}_{\mathsf{new}}, i, \Delta)$ from $\mathcal{F}_{\mathsf{UVPIR}}$.
   2. Let $(\mathsf{ssid}', \mathsf{version}', \mathsf{label}) \leftarrow \mathsf{serverView}[\mathbf{C}]$. If either $(\mathsf{ssid}', \mathsf{version}') \neq (\mathsf{ssid}_{\mathsf{old}}, \mathsf{version})$ or there does not exist a leaf node in $\mathsf{updateGraph}$ of the form $(\mathsf{label}, \mathsf{ssid}_{\mathsf{old}}, \cdot)$, send $(\texttt{InvalidDigest}, \mathsf{uid})$ to $\mathbf{C}$ and end execution.
   3. Let $\mathsf{DB}' \leftarrow \mathsf{DB}_{\mathsf{old}}$, where $\mathsf{DB}_{\mathsf{old}}$ is the database associated with $\mathsf{ssid}_{\mathsf{old}}$. Then set $\mathsf{DB}'[i] \leftarrow \mathsf{DB}_{\mathsf{old}}[i] + \Delta$.
   4. Send $(\texttt{Prove}, (\mathsf{ssid}_{\mathsf{old}}, \mathsf{ssid}_{\mathsf{new}}, i, \Delta), (\mathsf{DB}, \mathsf{DB}'))$ to $\mathcal{F}_{\mathsf{NIZK}}^{R}$ on behalf of $\mathbf{S}$ and receive back $(\texttt{Proof}, \pi_{\mathsf{Update}})$.
   5. Send the message $(\texttt{UpdateResponse}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{new}}, \pi_{\mathsf{Update}})$ to $\mathbf{C}$ on behalf of $\mathbf{S}$.
   6. Upon delivery of a message $(\texttt{AuthUpdate}, \mathsf{uid}, \sigma, m)$ from $\mathbf{C}$ to $\mathbf{S}$, check that $m = (\mathsf{ssid}_{\mathsf{old}} \parallel \mathsf{ssid}_{\mathsf{new}} \parallel \mathsf{version}{+}1 \parallel \mathbf{C})$.
   7. Send $(\mathsf{Verify}, m, \sigma, \mathsf{vk})$ to $\mathcal{G}_{\mathsf{Sig}}$ on behalf of $\mathbf{S}$, where vk is the client's verification key. End execution if the check fails or verification is unsuccessful.
   8. If $(\mathsf{ssid}_{\mathsf{old}}, \mathsf{version})$ is no longer a leaf node in $\mathsf{updateGraph}$, end execution. *// Stop concurrent updates from producing forks*
   9. Add a vertex $(\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1)$ to $\mathsf{updateGraph}$, and an edge from $(\mathsf{ssid}_{\mathsf{old}}, \mathsf{version})$ to $(\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1)$. Set $\mathsf{serverView}[\mathbf{C}] \leftarrow (\mathsf{ssid}_{\mathsf{new}}, \mathsf{version} + 1)$
   10. Upon receiving $(\texttt{Initialized}, \mathsf{ssid}_{\mathsf{new}})$ from $\mathcal{F}_{\mathsf{VPIR}}$, continue. If anything else is returned, end execution.
   11. Send $(\texttt{Prove}, (\mathsf{ssid}_{\mathsf{old}}, \mathsf{ssid}_{\mathsf{new}}, i, \Delta), (\mathsf{DB}, \mathsf{DB}'))$ to $\mathcal{F}_{\mathsf{NIZK}}^{R}$ on behalf of $\mathbf{S}$ and receive back $(\texttt{Proof}, \pi_{\mathsf{Update}})$.
   12. Send message $(\texttt{UpdateResponse}, \mathsf{uid}, \mathsf{ssid}_{\mathsf{new}}, \pi_{\mathsf{Update}})$ to $\mathbf{C}$ on behalf of $\mathbf{S}$.
   13. Upon delivery of a message $(\texttt{AuthUpdate}, \mathsf{uid}, \sigma, m)$ from $\mathbf{C}$ to $\mathbf{S}$, check that $m = (\mathsf{ssid}_{\mathsf{old}} \parallel \mathsf{ssid}_{\mathsf{new}} \parallel \mathsf{version}{+}1 \parallel \mathbf{C})$. Then send $(\mathsf{Verify}, m, \sigma, \mathsf{vk})$ to $\mathcal{G}_{\mathsf{Sig}}$, where vk is the client's verification key. End execution if the check fails or verification is unsuccessful.
   14. Send message $(\texttt{AckUpdate}, \mathsf{qid})$ to $\mathbf{C}$ from $\mathbf{S}$.
   15. Send $(\texttt{UpdateStatus}, \mathsf{uid}, 1)$ to $\mathcal{F}_{\mathsf{UVPIR}}$.

6: On delivery of a message $(\texttt{ClientSync}, \mathsf{version}, \mathsf{ssid})$ from a malicious client $\mathbf{C}$ to an honest server:
   1. Send $(\texttt{Sync}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{UVPIR}}$ on behalf of $\mathbf{C}$.
   2. When $\mathcal{F}_{\mathsf{UVPIR}}$ sends $(\texttt{ClientSync}, \mathbf{C})$, wait for a message of the form $(\texttt{SyncPath}, [(m_i, \sigma_i)_{i \in [k]}])$ to be delivered from $\mathbf{S}$ to $\mathbf{C}$.
   3. Parse the last m in the list as $(\mathsf{ssid}_{\mathsf{old}} \parallel \mathsf{ssid}_{\mathsf{new}} \parallel \mathsf{version} \parallel \mathbf{C})$, set $\mathsf{serverView}[\mathbf{C}] \leftarrow (\mathsf{ssid}_{\mathsf{new}}, \mathsf{version})$, and send $(\texttt{SyncClientState}, \mathbf{C}, \mathsf{ssid}_{\mathsf{new}}, \mathsf{version})$ to $\mathcal{F}_{\mathsf{UVPIR}}$.