POBA: Privacy-Preserving Operator-Side Bookkeeping and Analytics

Dennis Faut[1,3], Valerie Fetzer[1,3], Jörn Müller-Quade[1,3], Markus Raiber[1,3] and Andy Rupp[2,3]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
[2] University of Luxembourg, Esch-sur-Alzette, Luxembourg
[3] KASTEL Security Research Labs, Karlsruhe, Germany

Abstract. Many user-centric applications face a common privacy problem: the need to collect, store, and analyze sensitive user data. Examples include check-in/check-out based payment systems for public transportation, charging/discharging electric vehicle batteries in smart grids, coalition loyalty programs, behavior-based car insurance, and more. We propose and evaluate a generic solution to this problem. More specifically, we provide a formal framework integrating privacy-preserving data collection, storage, and analysis, which can be used for many different application scenarios, present an instantiation, and perform an experimental evaluation of its practicality.

We consider a setting where multiple operators (e.g., different mobility providers, different car manufacturers and insurance companies), who do not fully trust each other, intend to maintain and analyze data produced by the union of their user sets. The data is collected in an anonymous (wrt. all operators) but authenticated way and stored in so-called user logbooks. In order for the operators to be able to perform analyses at any time without requiring user interaction, the logbooks are kept on the operator's side. Consequently, this potentially sensitive data must be protected from unauthorized access. To achieve this, we combine several selected cryptographic techniques, such as threshold signatures and oblivious RAM. The latter ensures that user anonymity is protected even against memory access pattern attacks.

To the best of our knowledge, we provide and evaluate the first generic framework that combines data collection, operator-side data storage, and data analysis in a privacy-preserving manner, while providing a formal security model, a UC-secure protocol, and a full implementation. With three operators, our implementation can handle over two million new logbook entries per day.

Keywords: Analytics · Bookkeeping · Building-Block · MPC · Protocols · Privacy · Provable Security · UC · Universal Composability

1 Introduction

Privacy-friendly data analysis is now more important than ever: Companies generate vast amounts of data that they would like or need to analyze. A number of studies have shown that individuals are willing to share their data with companies if there is a potential benefit to them.[1] However, it is crucial to ensure that this data is processed in a way that protects individuals' privacy. Furthermore, the storage of large amounts of sensitive data

E-mail: dennis.faut@kit.edu (Dennis Faut), valerie.fetzer@kit.edu (Valerie Fetzer), joern.muellerquade@kit.edu (Jörn Müller-Quade), markus.raiber@kit.edu (Markus Raiber), andy.rupp@uni.lu (Andy Rupp)

[1] https://venturebeat.com/business/were-giving-away-more-personal-data-than-ever-despite-growing-risks/

for analysis is also a source of concern: There are recurrent cases where cybercriminals infiltrate company networks and steal large amounts of user-specific data.[2] These leaks have a negative impact on companies as well as the individuals whose data gets abused. To prevent these data leaks, it would make sense to not collect sensitive user data in the first place. This is not always an option, however, either because of laws/regulations (anti-money laundering, etc.), or the companies need the data for their business model (for usage-based fees, targeted ads, etc.), for behavior-dependent pricing (behavior-based car insurance, etc.), or to improve services (demand-based transport planning, etc.).

Instead, the idea of *privacy-preserving data analysis* has emerged in recent years: Here, the data is no longer stored in plaintext by the operator, but is protected in some way. One popular approach to obtain aggregate statistics is to not store individual data points but only update aggregates. This works well in practice if only aggregate statistics are needed, but does not work for general analysis, such as usage-based fees or targeted ads. Another approach is to apply differential privacy mechanisms during data collection, so the data being stored and processed is no longer personal data. But this has the drawback that a large amount of noise must be added during data collection to achieve a meaningful level of privacy, which reduces the utility too much for many use cases.

In contrast, we propose a method that collects precise data but protects the link between the data and the individual it belongs to. We can then perform desired analyses and apply differential privacy mechanisms on the output if needed. To this end, we use *secure multi-party computation (MPC)* as a central component. MPC is a cryptographic building block that allows multiple parties to evaluate a common function without parties learning the input (and output) of the other parties. Thus, with MPC, multiple operators, each owning a part of the data, can perform joint computations on the union of their data sets.

The use of MPC for privacy-preserving data analysis has been studied extensively, see for example [LP09]. However, in typical applications of MPC, the data owners are all directly involved in the computation, e.g., banks or hospitals wishing to obtain results based on their combined customer data, but each having full access to their own customer data. In contrast, we consider a setting *where no single operator has access to the user data to be analyzed*.

However, such an integrated framework *combining privacy-preserving data collection, storage, and analysis* has received little attention in the literature. Currently, there is only one *generic* system for privacy-preserving bookkeeping and analysis [FKM+22] and few proposals for specific application scenarios (cf. Section 1.3).

For designing a bookkeeping and analytics framework, there are two architectural options regarding data storage: either on the *user's side*, i.e., each user keeps track of its own collected data, or on the *operator's side*. We now take a closer look at both options.

If data is stored on the user's side until analysis, the user has full control over their own data at all times. This is very privacy-friendly, but it also has drawbacks: Users must explicitly cooperate and provide their data to the MPC protocol each time data analyses are to be performed. It is highly impractical to require active user participation for recurring automated tasks such as usage-based invoice generation or to obtain usage statistics. Also, users are responsible for the availability of their data, i.e., a lost smartphone on which the collected data was stored might result in the need to pay the maximum amount in a usage-based fee model. Furthermore, mechanisms to ensure the authenticity and "freshness" of the collected data during analysis are required to prevent users from selectively omitting some data points when providing their collected data. This results in the need for heavier crypto components on the user's side, which impact performance

[2] See https://www.csoonline.com/article/534628/the-biggest-data-breaches-of-the-21st-century.html and https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-2023 for examples.

for users who may only have a resource-constrained device, and can make the protocol infeasible in practice.

When data is stored on the operator's side, users must give up control over their data, but are no longer responsible for securing their data. However, it is now imperative to ensure that the data is stored securely and cannot be extracted in plaintext by an operator. This can be achieved for example through secret-sharing the data between the operators and performing the analysis with MPC, or by using fully homomorphic encryption (FHE). Since users do not have control over how their data is used in this approach, it is important that the protocol allows only functions approved by the consortium of operators to be evaluated on the data, and that this consortium has enough members that will object to the evaluation of analyses that violate user privacy (cf. Section 2.1 for further discussion).

1.1 Contribution

We introduce **POBA**, a generic building block for **P**rivacy-preserving **O**perator-side **B**ookkeeping and **A**nalytics. POBA consists of a consortium of operators and many users, where users collect data to store in *logbooks*, and operators can collectively evaluate analysis functions involving the logbooks of (subsets of) all users. POBA can be used for many different applications that want to combine privacy-preserving data collection, storage, and analysis. POBA is modeled and proven secure in the Universal Composability (UC) framework [Can00; Can20], which ensures that it can be securely integrated into arbitrary applications.

This generic building block has various applications in areas where users need to perform transactions anonymously and without being linked, while still allowing operators to analyze the collected transaction data in the background. Examples of applications include anonymous check-in/check-out multimodal transportation systems, charging/discharging electric vehicle batteries in smart grids, coalition loyalty programs, behavior-based car insurance, and more. Our running example to better explain our system will be a check-in/check-out based payment system for multimodal public transportation: Users can anonymously check in and out of various modes of transportation. Both actions create corresponding log entries, and operators cannot link multiple entries by the same user. At the end of each billing period, an invoice is generated for each user based on the collected log entries. This allows for a sophisticated pricing system that, e.g., combines multiple individual trips into a daily flat rate. Route optimization can be achieved by analyzing trip data from users of multiple transportation companies together. Such systems are already employed in a non-privacy-preserving matter,[3] where check-ins/check-outs contain a unique identifier of the traveler, allowing for the generation of comprehensive movement profiles. In Section 4 we take a closer look at how POBA can be used for anonymous check-in/check-out multimodal transportation systems.

High-Level Idea of our Approach Our system consists of three major steps: data collection, data storage and data analysis.

Data collection: This is done by an (anonymous) user agreeing with one of the operators what data entry should be added to their logbook. In this step, the content of the entry is visible to both the user and the operator, but the identity of the user is encrypted. The fact that user and operator agree on the content of each entry prevents malicious users from sabotaging the analysis by submitting invalid or malformed data.

Data storage: This is handled by secret-sharing the entry between the operators and securely recovering the identity of the user through MPC to obtain the correct logbook

[3] E.g., in transport for NSW https://transportnsw.info/tickets-opal/opal/contactless-payments or transport for London https://tfl.gov.uk/fares/how-to-pay-and-where-to-buy-tickets-and-oyster/pay-as-you-go

to insert it. To prevent the user identity from being leaked in this step, oblivious RAM (ORAM) is used to protect the logbook address, and thus the user identity associated with it. We choose to store data on the operator's side to avoid the need for active user interaction for analysis evaluation and the problem of users losing data, and to reduce the workload on user devices.

Data analysis: Finally, data analysis is performed by evaluating functions approved by the consortium of operators through MPC on (a subset of) the user logbooks. As the logbooks are secret-shared between the operators, no single operator (or collusion of a minority of operators) can directly access the data or evaluate unapproved functions.

Having the operator agree on the content of each entry is important to ensure correctness and authenticity of the collected data. While in some use cases it would be possible for users to prove that their entry is well-formed, it is generally not possible to ensure the authenticity of the data in this way. This means that our system is not suitable for applications in which the content of an entry alone (without the connection to the user's identity) is sensitive, e.g., because the user's identity can directly be inferred from it. However, our system is well suited if the sensitive information is the link between data and user identity, particularly when the entry contains data inevitably observable by the operator (e.g., the presence of some user/vehicle at a specific position, the items on a shopping receipt). A consequence of this is the importance of preventing linkage between the creation of an entry and its usage during analytics, which might involve only a single user's logbook. We make use of oblivious RAM to break this link, ensuring that the information which memory was accessed during a particular analytics evaluation provides no information about when the entries in the involved logbooks were created, without needing to access *all memory* to do so. Otherwise, an operator could for example track the memory address accessed when storing each check-in entry and then later correlate the addresses accessed to create the invoice of a user with this list. We note, however, that even when hiding the content of each entry from the operator during creation, this linkage still needs to be prevented, as it would for example still allow the creation of pseudonymous movement profiles from the interactions with stationary terminals.

Formal Security Model We formalize security (and functionality) for POBA through an ideal functionality $\mathcal{F}_{\text{POBA}}$ modeled in the UC framework. This functionality captures the properties we want a POBA system to have in an ideal world: Users can create an account with one of the system operators, who becomes that user's *home operator*. Users can also create logbook entries with any operator (not just their home operator). The functionality ensures that only registered users can do this, that users can only add entries to their own logbook, and that the user and the operator agree on the content of the new logbook entry. Logbook entries are also anonymous and unlinkable, i.e., the identity of the user remains hidden from the operator during entry creation, and no operator can link different entries from the same user. The functionality also enforces that analyses can only be performed if all operators participate and agree on the analysis function. It further guarantees that the analysis function does not reveal additional private data beyond the intended output.

Provably secure protocol We present a protocol Π_{POBA} that UC-realizes the functionality $\mathcal{F}_{\text{POBA}}$ in the presence of a static adversary that may corrupt any number of users and up to $t \leq \frac{n-1}{2}$ of the n system operators, where corrupted users may behave maliciously, while corrupted system operators are assumed to behave semi-honestly. It uses cryptographic building blocks such as threshold signatures, threshold encryption, non-interactive zero-knowledge (NIZK) proofs, secret-sharing, and MPC that supports ORAM. One of the challenges has been to carefully combine the different primitives and

techniques into a UC-secure protocol, while also providing practical efficiency. To this end, it is important to only prove statements in zero-knowledge for which efficient proofs are possible, to take care of what needs to be evaluated inside MPC, and to consider the limited computational power of users. In particular, we assume that users run constrained devices such as smartphones or smartwatches, and minimize the computation and communication required from them.

Evaluation and Benchmarks We implemented Π_{POBA} to evaluate its practicality and to determine which cryptographic building blocks are the bottlenecks. Our results show that the performance of ORAM within secure computation is currently a significant bottleneck, limiting practicality to a small number of operators and users, especially since this step does not parallelize across more hardware. All other parts of our protocol are efficient enough for practical use, and the operator computation besides ORAM accesses can be parallelized across additional hardware to scale with the number of users. There have been many efforts in recent years to develop protocols for secure computation on ORAM for the special cases of $n = 3$ (or 4) and $t = 1$, but none of these generalize to a setting with $t > 1$ as n increases. For a setting with $t \leq \frac{n-1}{2}$ corruptions, the state of the art is still a variant of PathORAM [SvS+13]. Possible future optimizations in this setting will directly translate to a respective performance increase of our protocol.

Using the MP-SPDZ framework [Kel20] with its implementation of PathORAM [KS14] for $n = 9$, $t = 4$, up to 2^{15} registered users and logbooks with up to 100 entries, our protocol has a throughput of ≈40k entries per day. Regarding our transportation payment example, this would translate to a small city of 20k people with an average of one trip per person per day and a limit of 50 trips per person per week when one week is used as the period duration.

In the restricted $n = 3$, $t = 1$ setting, GigaDORAM [FOSZ23] reports 700 queries/s even with memories of size 2^{30}, indicating an access time of less than 2 ms. Using GigaDORAM, we achieve a throughput of over two million inserts per day with one million registered users, corresponding to a medium-to-large European city.

Overall, our benchmarks already suggest that when restricting to a small number of operators and moderate number of users, POBA could be practical in a check-in/check-out transportation scenario for a city up to a few hundred thousand people. Since our protocol uses MPC with ORAM as a black-box building block, any future improvements there can be directly applied to our protocol.

1.2 Overview of Technical Challenges

Designing and implementing a cryptographic protocol with the complexity of POBA comes with a number of challenges. First, secure computation protocols are comparatively slow, and making users wait for or even participate in them reduces user satisfaction or might simply be impossible. But in many use cases it is sufficient for user and operator to agree on a new log entry, and the actual writing to the logbook can be delayed until a later time. This is especially useful in situations where timely feedback to the user is important, but the operator's device used for the interaction has spotty connectivity, which may be the case for check-ins/check-outs in mobile vehicles such as trains. Therefore, we split the writing of new entries into two tasks: In the first task, the user and an operator agree on the new entry, and in the second task, the operator interacts with the other operators to actually write the entry into the correct logbook. In this scenario, the user is no longer needed for the second task.

Second, when implementing an oblivious logbook, letting its size grow arbitrarily leads to (some information about) the current size of the logbook being leaked when accessing it. In addition, large logbooks have a large performance impact. To deal with this problem, we divide logbooks into periods, with a fixed size for each period. When creating new

entries, user and operator now also agree on the period in which the entry should land. This allows new entries to be inserted efficiently, and analysis calculations can specify the relevant periods and thus benefit from better efficiency when only a subset of periods is needed.

Third, it may be necessary for a user to convince an operator that an entry was made previously. Again, proof of check-in is important for spot checks when considering check-in/check-out-based public transportation payment systems.

Fourth, some way of mapping users to logbooks (i.e., locations in memory) is required. This mapping must be done in a way that prevents impersonation of honest users, even by an adversary corrupting several operators. An intuitive solution would be to use a user's public key as the address of the logbook, and to require a proof-of-knowledge of the secret key when creating new entries. But for this to be secure, the public key has to be large, i.e., λ bit, or $\geq$250 bit when using elliptic curves, while the number of users is typically much smaller, i.e., 2^{20} to 2^{30}. It is not feasible to allocate memory for 2^{λ} potential logbooks. Hashing the public key into a smaller address requires a collision-free hash function, since collisions would allow impersonation of honest users. But using a collision-resistant hash function still results in addresses of λ bit length. Another method is to create a fixed mapping from a user's public key to their logbook address. This mapping could then be stored in a hash table, where collision-freeness can be guaranteed by choosing a new random universal hash function whenever inserting a new entry would result in a collision. But this approach requires that the hash table is queried in an oblivious way, and thus requires another layer of oblivious RAM. Since ORAM is the bottleneck of our protocol, we decided against this approach and instead chose the following solution: The mapping is given to the user, along with a signature on the public key/address pair. The user then provides a zero-knowledge proof of knowledge of the secret key along with the signature on the public key/address pair, and an encryption of the address. This introduces another challenge: Even semi-honest corruption of an operator leaks their secret key to the adversary. This secret key can then be abused by malicious users to generate signatures that map their public key to the address of an honest user's logbook. To prevent this, we use a threshold signature scheme, which requires secret keys of at least $t + 1$ operators to generate a valid signature.

The next challenge is to encrypt and sign the address in a way that is compatible with both efficient zero-knowledge proofs and decryption within a secure computation protocol. Groth–Sahai proofs are very efficient when working with structure-preserving primitives. Thus, using a structure-preserving threshold signature scheme and encoding the address as a point on the elliptic curve is the natural answer. The latter is typically done by multiplying it by for example 2^{10}, interpreting this as the x-coordinate, and then adding 1 to this value until one ends up with a valid point on the curve. But this only works if the subgroup used has a small cofactor. The popular BLS curves use subgroups with very large cofactors, in the order of 2^{100}, which results in having to do around 2^{100} iterations before finding a valid point in the subgroup. Fortunately, Barreto–Naehrig curves, which have fallen out of favor due to new attacks that mandated larger parameters to maintain 128 bit security, use the whole curve as $\mathbb{G}_1$ resulting in cofactor 1 and thus allow this type of encoding.

Fifth, secure computation works over a finite field or ring, the size of which has a significant impact on performance. To allow decryption of a ciphertext over a Barreto–Naehrig curve with $\approx$128 bit security, it is necessary to work over a finite field of size $\approx 2^{382}$. Performing oblivious RAM operations on this field has a prohibitive performance cost. As a solution, we split our secure computation into two parts: First, the ciphertext is decrypted over the large field. Then, using the share conversion method of [DT08], the resulting plaintext (which is known to be an address much smaller than 382 bit) is converted to a smaller field, and the secure computation involving oblivious RAM is performed over this

smaller field. Specifically, we operate on a 128 bit field for this.

1.3 Related Work

Privacy-preserving data analysis is not only a topic discussed in society, but it will also become an important economic factor in the future: The global market size for privacy-enhancing computation was already estimated at $2.2 billion for 2023 and is projected to reach approximately $24.7 billion by 2033.[4] As a consequence, we should make sure that there are cryptographically sound methods to protect our personal data during the full data analysis life cycle. For the setting where a small number of data-holders want to evaluate functions on their combined data, i.e., banks performing clustering on their combined transaction graphs for money-laundering detection, standard MPC approaches can easily be employed. However, when involving the data of a large number of users without huge computational resources, mechanisms to collect the data in a privacy-preserving manner have to be combined with mechanisms to securely evaluate analysis functions.

When reviewing the prior work on the combination of privacy-preserving data collection, storage and analysis, we distinguish between solutions targeting specific applications and generic solutions. We defer our review of application-specific solutions for the application scenarios privacy-preserving ticketing for public transport and privacy-preserving charging/discharging of electric vehicles to Appendix A, since almost all of them focus on billing and do not allow other analyses, and focus on generic solutions in the following.

Generic solutions for user data collection and analysis mainly fall into two categories: applying noise to the data during collection to achieve differential privacy, or being limited to aggregation functions (e.g., vector sum, heavy hitters) over the collected data.

Differential Privacy during Data Collection There are multiple systems employing differential privacy during data collection to protect user privacy. Examples include RAPPOR [EPK14; FPE16], which is used in Chromium. The downside of this approach is that when adding little noise, the system provides weak privacy guarantees, whereas when adding a lot of noise, the system provides low utility.

Privacy-Preserving Aggregate Statistics Several protocols to generate aggregate statistics in a privacy-preserving way have been proposed [MDD16; CB17; BBC+21; BGL+22; AGJ+22; MST23]. These protocols allow computation of aggregation functions, such as the sum of input vectors (e.g., Prio, Prio+ [CB17; AGJ+22]) or heavy hitters and subset-histograms (e.g., Poplar, Doplar [BBC+21; DPRS23]). To this end, one input per client is aggregated over an epoch and then the output of the function is revealed. The drawback of these approaches is that the class of analysis functions that can be evaluated is constrained, in particular no analyses regarding multiple inputs (from different points in time) of one user can be obtained, as would be required for scenarios such as usage-based fee calculation or targeted advertisements.

Generic Privacy-Preserving Data Collection and Analytics The only solution for privacy-preserving data collection and analytics that allows generic analysis functions we are aware of is PUBA [FKM+22], which also comes with a formal security model and proof in the UC framework. In PUBA, logbooks, which are called "user histories" there, are stored on each individual user's device. Simple single-user analysis functions can be evaluated by the user itself and correctness of the result proven in zero-knowledge, similar to the smart-metering setting of [RD11; RDK18], but since user devices are assumed to be smartphones the class of functions is even more limited. For more complex functions or functions that rely on data from multiple users, PUBA has each user secret-share their

[4] https://www.precedenceresearch.com/privacy-enhancing-computation-market

logbook between the system operator and a non-colluding "proxy" party, who then evaluate the function using MPC.

While our setting is similar to theirs, we address several shortcomings: Firstly, we allow multiple operators to collectively operate the system (PUBA does not generalize to multiple distrusting operators, as each operator could arbitrarily modify the content of user logbooks), removing the need for an external party doing heavy computations. Second, we store logbooks securely between the operators instead of on the user's device, providing robustness against data loss from lost/stolen/broken user devices and reducing the computation and communication required from users. Third, *analysis computations no longer require user interaction*, making a deployment much more practical. This reduces the delay between start and completion thereof and ensures that data from the involved users covers the same time period (in PUBA, this task needs to wait for input from each user, and users input their current logbook, causing users that provide input at a later point in time to cover a later time period). Fourth, with our concept of time periods, we support logbooks that grow over time, whereas PUBA has one fixed logbook size and old entries get overwritten over time.

2 An Idealized POBA model

In this section, we provide an overview of our system and the parties involved. Figure 1 gives an overview of how the involved parties could interact with our system in the context of an anonymous check-in/check-out based multimodal transportation system.

2.1 Parties and Trust Assumptions

Users (U) interact with the system by collecting data in their individual logbooks. Our system allows for an arbitrary number of users who are untrusted (malicious) and may collude with other corrupted parties. Users are assumed to use comparatively weaker hardware, and thus their computational burden is kept as low as possible.

System operators (O) run the system. They usually provide some kind of service that users are interested in, and as part of that service they want to collect data from users. Each user is registered with one system operator, which is called the user's *home operator (HO)*. Note that the set of operators can also be heterogeneous, since different kind of operators may have different interests in the collected data. For example, in the charging and discharging electric vehicle batteries in the smart grid scenario, operators could not only be electricity providers but also car manufacturers who may want to analyze the lifetime of batteries in their vehicles.

Since one of our goals is to evaluate the practicality of POBA, we assume the most favorable corruption model: An adversary may only corrupt a minority of operators, and corrupted operators are semi-honest rather than malicious.[5] While this is a rather strong assumption, we argue that this assumption is appropriate in many cases: It protects against accidental data leaks, curious or disgruntled employees, and data breaches. Cybercriminals are increasingly threatening to release private customer data unless a ransom is paid, as was for example the case with a Finnish mental health care provider,[6] and not having such data in the first place prevents this. Similarly, it is not uncommon for curious employees to violate the privacy of customers.[7] Active manipulation of installed software is a much

[5] While malicious parties can arbitrarily deviate from the protocol, semi-honest parties are assumed to honestly follow the protocol. The goal of a semi-honest party is to extract more information than intended from the protocol. Several semi-honest parties can also collaborate and try to gain additional knowledge from their combined sets of information.

[6] https://www.wired.com/story/hacker-threaten-release-therapy-notes-patients/

[7] https://interestingengineering.com/culture/google-fires-80-employees-for-exploiting-user-data

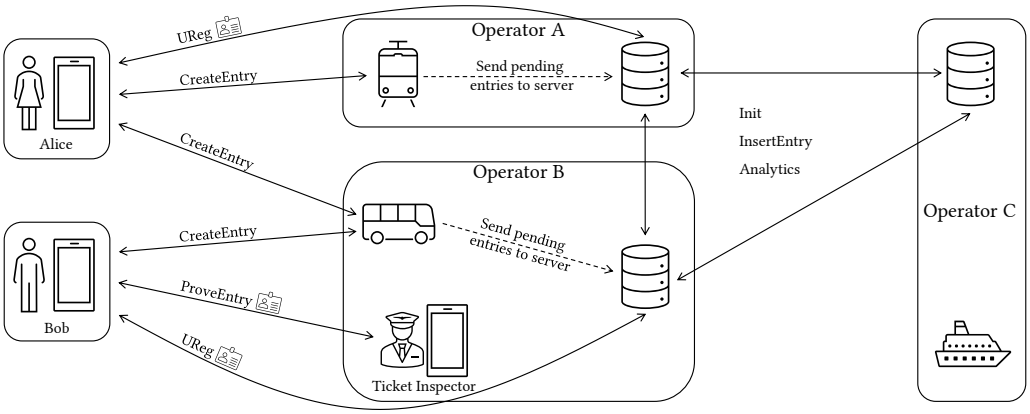

Figure 1: Intended use of the system using the application example check-in/check-out transportation systems with different mobility providers. Arrows with a 🪪 represent identifying interactions and dotted arrows are assumed to happen outside of the protocol.

higher barrier than abusing legitimate access mechanisms, and tampered software could also be detected by remote attestation mechanisms.

When performing analytics, we assume that operators agree among themselves beforehand which analytics they will run on which data. Since the majority of operators are assumed to be honest, they will not agree to perform analyses that violate privacy rules. If desired, privacy advocates could be allowed to participate in the MPC as observers to ensure that the analysis function protects the privacy of user data.

2.2 Logbooks and User Data

Logbooks are at the heart of our system: All the data collected by a user is stored in them. As mentioned above, there is a separate logbook for each user. Each logbook consists of a list of entries, where each entry has a fixed (maximum) length. For efficiency purposes, logbooks are further divided into time periods and have a maximum size (i.e., number of entries). The length of a period and the size of a logbook should be chosen so that the logbooks don't overflow. The logbook for user U and period p is referred to as Log_U^p. When operators want to perform analysis, these logbooks are used as a data basis.

To prevent malicious users from sabotaging analyses by submitting invalid or malformed data, we require the user to agree with one of the operators on the content of each entry. Thus, the system is well suited for purposes where the data in each entry itself is not personal information, but the combination of data and user identity is relevant and sensitive.

2.3 Desired Security Properties

Informally, we want to achieve the following guarantees:

- Users can only create logbook entries for their own logbook, and only after creating an account
- Logbook entries can only be generated if both user and operator agree on their content
- Operators can not identify users during entry creation
- Operators can not link different entries to the same user
- Users can prove that they created a specific logbook entry if and only if they actually did so
- Computation of an analytic function does not reveal additional private data besides the intended output

- Computation of an analytic function returns the correct output

It is easy to see that our ideal functionality satisfies these.

2.4 Our Ideal Functionality

$\mathcal{F}_{\text{POBA}}$ internally manages all logbooks and pending entries, and provides system operators with the output of analytic functions they all agree to evaluate. In Fig. 1 we illustrate the interplay of the different tasks using the check-in/check-out based transportation system of Section 4 as an example. We now provide both a high-level description of these tasks and a formal description in Fig. 2.

Notation Our ideal functionality is described as a set of tasks that can be called by the parties. Each task has inputs of the form "Input P: $(\texttt{cmd}, \mathsf{ssid}, msg)$" (where P is the party that should provide this input, `cmd` specifies the task, ssid is the subsession id, and msg is the input of that party for the given task), a behavior, and outputs of the form "Output P: (ssid, msg)" (where ssid is the same subsession id and msg the output for that party). This means that after the functionality receives messages from all parties P specified as having input with the same ssid, it runs the instructions listed under "Behavior". If doing so does not cause it to ignore the inputs, it makes private delayed outputs to the parties P with content (ssid, msg) as listed under "Output". If the party for which output is generated is semi-honest, the adversary is additionally provided with (ssid, msg). Additionally, whenever a semi-honest party P provides input $(\texttt{cmd}, \mathsf{ssid}, msg)$, it sends $(\texttt{cmd}, \mathsf{ssid}, msg, P)$ to the adversary. When an honest operator O provides input, it sends $(\texttt{cmd}, \mathsf{ssid}, O)$ to the adversary. When an honest user U provides input, if $\texttt{cmd} = \texttt{ureg}$ it sends $(\texttt{ureg}, \mathsf{ssid}, U)$ to the adversary, for $\texttt{cmd} \neq \texttt{ureg}$ it instead sends $(\texttt{cmd}, \mathsf{ssid}, \texttt{user})$ to the adversary, i.e., it keeps the identity of the user secret.

Init To model required setup steps in the real protocol, the functionality only starts working after each operator called this task.

UReg Before participating in the system, a user has to select a home operator HO from the list of operators and register with this operator. This ensures that information about user accounts, which may also contain (static) personal data (e.g., billing information), is only held by one operator. All other operators need to acknowledge the new user. The functionality then adds the user to the list of registered users with that home operator.

CreateEntry This task initiates creation of a new logbook entry. First, user and operator agree (out-of-protocol) on the content *entry* of the new entry and which period p it belongs to. Then, the functionality ensures that the user has previously registered an account, while keeping the user's identity secret. The entry is then added to a list of pending entries that have to be inserted into the correct logbook using the next task.

InsertEntry In this task, all operators work together to move an entry from the pending list into the correct logbook. Only after an entry has been inserted by this task can it be included in analytics computations.

ProveEntry As mentioned above, in some applications it is useful for users to be able to prove that their logbook contains certain (pending) entries, e.g., spot checks can be implemented by users showing that they created a check-in entry for the current train. This task does just that: A user can prove to an operator that they produced a certain logbook entry *entry* in period p. To do this, the functionality sends the identity of the

Functionality $\mathcal{F}_{\text{POBA}}$

State:
- Set of all operators $\mathcal{O}$, logbook size *size*
- Counter id^{O}_{last} for each $O \in \mathcal{O}$ to enumerate pending log entries
- List $\mathsf{L}^{O}_{\mathsf{users}}$ for each $O \in \mathcal{O}$ containing a list of users registered with that operator
- List $\mathsf{L}^{O}_{\mathsf{Pending}}$ for each $O \in \mathcal{O}$ to store pending log entries
- Logbooks Log^{p}_{U} for each user U and period p
- Variable *initialized*, initially set to 0

Init:
- Input each O: (`init`)
- Behavior:
  1. After receiving input from all $O \in \mathcal{O}$, set $initialized := 1$
- Output each O: (`init`, `ok`)

Ignore messages other than `init` until $initialized = 1$.

UReg:
- Input U: (`ureg`, ssid, pid_{HO})
- Input HO: (`ureg`, ssid, pid_{U})
- Input all other O': (`ureg`, ssid)
- Behavior:
  1. If U is already in $\bigcup_{O \in \mathcal{O}} \mathsf{L}^{O}_{\mathsf{users}}$ or U does not have pid pid_{U} or HO does not have pid pid_{HO}, ignore this call
  2. Upon receiving input from HO, send ((`ureg`, ssid, pid_{U}), HO) to the adversary
  3. Add (pid_{U}) to $\mathsf{L}^{\mathsf{HO}}_{\mathsf{users}}$
- Output U: (ssid, `ok`)
- Output HO: (ssid, `ok`)

CreateEntry:
- Input U: (`createentry`, ssid, $period_U$, $entry_U$, pid_{O})
- Input O: (`createentry`, ssid, $period_O$, $entry_O$)
- Behavior:
  1. If pid_{U} is not in $\bigcup_{O \in \mathcal{O}} \mathsf{L}^{O}_{\mathsf{users}}$ or $entry_U \neq entry_O$ or $period_U \neq period_O$ or O does not have pid pid_{O}, ignore this call
  2. Set $id^{O}_{\mathsf{last}} := id^{O}_{\mathsf{last}} + 1$, $id := id^{O}_{\mathsf{last}}$
  3. Add (id, pid_{U}, $period$, $entry_O$) to $\mathsf{L}^{O}_{\mathsf{Pending}}$
- Output U: (ssid, `ok`)
- Output O: (ssid, `ok`, id)

InsertEntry:
- Input O: (`insertentry`, ssid, $period$, id)
- Input all other O': (`insertentry`, ssid)
- Behavior:
  1. If no entry (id, pid_{U}, $period$, $entry$) exists in $\mathsf{L}^{O}_{\mathsf{Pending}}$, ignore this call
  2. If at least one O' is corrupted and O is honest, leak ((`insertentry`, ssid, $period$, $entry$), O) to the adversary
  3. If $|\mathsf{Log}^{period}_{U}| = size$, output (ssid, `error`) to all O
  4. Otherwise:
     (a) Remove that entry from $\mathsf{L}^{O}_{\mathsf{Pending}}$
     (b) Append ($entry$) to $\mathsf{Log}^{period}_{U}$
- Output all O: (ssid, `ok`)

ProveEntry:
- Input U: (`proveentry`, ssid, $entry$, $period$, pid_{O})
- Behavior:
  1. If no entry (id, pid_{U}, $period$, $entry$) in $\mathsf{L}_{\mathsf{Pending}}{}^{O'}_{O' \in \mathcal{O}}$ or ($entry$, O') in $\mathsf{Log}^{period}_{U}$ exists, ignore this input
  2. If both U and O' such that the entry was in $\mathsf{L}_{\mathsf{Pending}}{}^{O'}$ resp. the entry ($entry$, O') existed in $\mathsf{Log}^{period}_{U}$ are honest, send (`proveentry`, ssid, leak, O') to the adversary
- Output O: (ssid, `proveentry`, pid_{U}, $entry$, $period$)

Analytics:
- Input all O: (`analytics`, ssid, $func$, $\{period_i\}$, $\{U_i\}$, aux_O)
- Behavior:
  1. If not all parties provided the same $func$, $\{period_i\}$ and $\{U_i\}$ as input, ignore this call
  2. Set $\mathsf{Logs} := \left\{\mathsf{Log}^{p}_{U}\right\}_{U \in \{U_i\}, p \in \{period_i\}}$
  3. Evaluate $\{\mathtt{result}_O\}_{\mathcal{O}} \leftarrow func(\mathsf{Logs}, \{aux_O\}_{\mathcal{O}})$, setting $sequence := (p_1, p_2, \ldots, p_n)$ with an entry p_j for each time $func$ reads an entry from $\mathsf{Log}^{p_j}_{U}$
  4. If at least one operator is corrupted, leak (`analytics`, ssid, $sequence$) to the adversary
- Output each O: (ssid, $\mathtt{result}_O$)

Figure 2: Ideal functionality $\mathcal{F}_{\text{POBA}}$

user together with the period and entry to the operator if the entry is stored in either the user's logbook or the list of pending entries.

Analytics In this task, all operators work together to evaluate an analysis function that they have previously agreed on out-of-band. The analysis function can use the logbooks of several users from several periods as a data basis. The functionality first ensures that all operators agree on what function to evaluate on which users and periods. Each operator receives an output from the analysis, but it is possible that each operator receives a different output (depending on the specific analysis function). It depends on the application what exactly the analysis function calculates. For example, it can be used to create customer invoices and conduct market research, compute statistics to improve a service, etc.

3 A Protocol to Realize the Model

In this section we present our protocol Π_{POBA} that UC-realizes the ideal functionality $\mathcal{F}_{\text{POBA}}$.

At the heart of our system is the oblivious logbook storage, which is built upon oblivious RAM (ORAM), or more precisely an MPC protocol with ORAM. For each user, memory in the ORAM is reserved and logbook entries are written there. To allow for efficient user-facing protocols, logbook entries are first stored locally at the respective operator with an encrypted user identifier and later written into the actual logbook in batches. For better efficiency, we also split logbooks into periods, where each period starts with a new empty logbook. This is realized by simply having different instances of ORAM for each period, where a new instance is initialized as empty the first time an entry is generated for it. Note that analytics can still be based on multiple periods if desired.

3.1 Building Blocks

Before delving into the actual protocol, we briefly introduce the building blocks we use to achieve the desired security guarantees on a high level. Formal definitions are given in Appendix B.

Threshold Public Key Encryption (PKE) We use a threshold PKE to encrypt user identities. A threshold PKE with n parties and threshold t has a key generation algorithm that outputs an encryption key ek along with n decryption key shares dk_i. It has a standard encryption algorithm, but for decryption it has a partial decryption algorithm that, given a ciphertext and a decryption key share, outputs a decryption share, and a combine algorithm that combines $t+1$ of those decryption shares to recover the plaintext. We use this to encrypt user identities when creating new logbook entries to prevent a colluding minority of operators from decrypting those identities. IND-CPA-security guarantees that, given the encryption key and at most t decryption key shares, it is hard to decide whether a given ciphertext is an encryption of m_0 or m_1 for any same-length messages m_0, m_1.

See Definition 2 for a formal definition. Additionally, we require a distributed key generation protocol, formalized as $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$, see Fig. 8a.

Digital Signatures A digital signature scheme consists of a key generation algorithm that outputs a verification key vk and a signing key sk, and algorithms for signing messages with the signing key and verifying signatures with the verification key. EUF-CMA-security guarantees that it is hard to generate valid signatures given only the verification key. We use a (regular) signature scheme to authenticate log entries.

In aggregate signature schemes it is possible to combine different signatures into one aggregate signature, which reduces signature size and speeds up verification. We use this to efficiently verify system keys.

In a threshold signature scheme multiple parties have to work together to create a signature on a message. We use this to authenticate user credentials. It prevents a single leaked operator signing key from creating fake credentials for honest users, which would allow an adversary to forge entries for that user's logbook. The threshold version again has a key generation algorithm that outputs a verification key vk and signing key shares sk_i. Signing messages consists of generating a partial signature with a signing key share and then combining $t+1$ such partial signatures into a full signature. Here we use the TS-UF-1-security definition, which guarantees that given at most t signing key shares, the verification key, and access to an oracle that generates partial signatures, it is hard to generate a valid signature on a message that has not been queried to the oracle.

Formal definitions are given in Definition 3 for digital signatures, in Definition 4 for EUF-CMA-security, in Definition 5 for aggregate signatures, in Definition 6 for threshold signatures, and in Definition 7 for TS-UF-1-security. Again, we additionally require a distributed key generation protocol, formalized as $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$, see Fig. 8b.

Non-Interactive Zero-Knowledge Proofs To allow user anonymity, we use non-interactive zero-knowledge proofs of knowledge in the common reference string (*crs*) model. Such a proof allows a party to convince another party that a statement is in some NP relation, and even that they know a witness for this fact, without revealing information about the witness. More precisely, given a relation R, a statement *stmt*, and a witness *wit*, such that $(stmt, wit) \in R$, there is a prove algorithm for generating a proof and a verification algorithm that, given the relation R, the statement *stmt*, and the proof, verifies that it is valid. Additionally, given some trapdoor information that could be generated during setup, there exist algorithms to simulate proofs without actually knowing a witness, and to extract the witness from proofs. The zero-knowledge property guarantees that simulated proofs are indistinguishable from real proofs and *simulation-extractability* guarantees that extraction with the trapdoor is possible even if the adversary has seen simulated proofs before.

For formal definitions, see Definitions 8 and 9.

Secure Computation of RAM Programs We use MPC to manage the logbooks and compute analytics. To make this feasible with a large number of users, we need a secure computation protocol that allows the evaluation of RAM programs: Usually, secure computation is defined in terms of circuits, which has the advantage of being side-channel free. However, in order to access an input-dependent entry of an array, input wires for all entries of that array must go into a large multiplexer circuit that selects the correct entry. In our case, the array corresponds to the collection of logbooks for all users, which can range from tens of thousands to several million, so this is clearly not practical. When using a RAM program, on the other hand, accessing such an entry simply corresponds to a single memory access at the desired location, the complexity of which is independent of the size of the array. However, doing so naively reveals *which* entry of the array was accessed. So we require our MPC protocol to access the RAM in an oblivious way. To achieve this, oblivious RAM [Gol87] was introduced. It can be viewed as a compiler that transforms logical memory accesses into a sequence of physical memory accesses that look random. Combined with secure multi-party computation, this allows us to access a single logbook without revealing which one it was (and thus which user it belongs to), with a complexity that grows only logarithmically with the number of users. For modeling this, we follow the approach of [BPRS23] and use an ideal functionality $\mathcal{F}_{\mathrm{MPC}(f)}$ to securely evaluate a RAM program f. To this end, we require a secret-sharing scheme S which consists of algorithms to generate n shares from a secret such that up to t of those shares reveal no information about the secret, but any $t+1$ shares can be used to reconstruct the secret. See Definition 1 for a formal definition. $\mathcal{F}_{\mathrm{MPC}(f)}$ then takes as input secret-shares of the memory M and from each party P_i some input $\mathtt{input}_i$, reconstructs M from these shares, executes $f((\mathtt{input}_1, \ldots, \mathtt{input}_n), M)$ and outputs the result together with the number of memory accesses performed by f, as well as new shares of the updated memory. In addition, we use an extended functionality $\mathcal{F}^{\mathrm{ext}}_{\mathrm{MPC}(f')}$ that takes a collection of memories $M_1, \ldots, M_n$ as input and an extended RAM program f' that can perform memory accesses on any of these memories by specifying the index of the memory. Given a protocol realizing $\mathcal{F}_{\mathrm{MPC}(f)}$, it is straightforward to obtain a protocol realizing $\mathcal{F}^{\mathrm{ext}}_{\mathrm{MPC}(f)}$ by simply using multiple independent instances of the mechanism to access the memory.

For the formal definitions of these functionalities, see Figs. 3a and 3b.

Functionality $\mathcal{F}_{\mathbf{MPC}(f)}$

Parameters: A next-step function NS, number of participating parties n

- Upon receiving $(\text{Run}, \langle M \rangle, input_i)$ from all parties P_i, $i < n$:
  1. $M = \mathsf{S.Recon}(\langle M \rangle)$
  2. Initialize $\mathsf{st_{NS}} := (\mathsf{start}, (input_1, \ldots, input_n))$, $data := 0$ and $round := 0$
  3. While $\mathsf{st_{NS}} \neq (\mathsf{stop}, (output_1, \ldots, output_n))$, do:
     (a) Run $(\mathsf{st_{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st_{NS}}, data)$
     (b) If $I = (\mathsf{read}, addr, \bot)$, set $data := M[addr].val$
     (c) Else if $I = (\mathsf{write}, addr, val)$, set $data := M[addr].val$ and update $M[addr].val = I.val$
     (d) Update $round := round + 1$
  4. $\langle M^{\text{new}} \rangle \leftarrow \mathsf{S.Share}(M)$
  5. Output $(\langle M \rangle_i, output_i, round)$ to each party P_i, $i < n$.

(a) Functionality $\mathcal{F}_{\text{MPC}(f)}$ for securely evaluating RAM programs.

Functionality $\mathcal{F}^{\mathbf{ext}}_{\mathbf{MPC}(f)}$

Parameters: A next-step function NS, number of participating parties n

- Upon receiving $(\text{Run}, \{\langle M_j \rangle\}_{j<k}, input_i)$ from all parties P_i, $i < n$:
  1. For each $j < k$: $M_j = \mathsf{S.Recon}(\langle M_j \rangle)$
  2. Initialize $\mathsf{st_{NS}} := (\mathsf{start}, (input_1, \ldots, input_n))$, $data := 0$, $round := 0$ and $sequence = ()$
  3. While $\mathsf{st_{NS}} \neq (\mathsf{stop}, (output_1, \ldots, output_n))$, do:
     (a) Run $(\mathsf{st_{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st_{NS}}, data)$
     (b) If $I = (\mathsf{read}, index, addr, \bot)$:
        i. Ensure $index < k$, otherwise abort
        ii. Set $data := M_{index}[addr].val$
     (c) Else if $I = (\mathsf{write}, index, addr, val)$
        i. Ensure $index < k$, otherwise abort
        ii. Set $data := M_{index}[addr].val$
        iii. Update $M_{index}[addr].val = I.val$
     (d) Update $round := round + 1$ and $sequence := (sequence, index)$
  4. For each $j < k$: $\langle M_j^{\text{new}} \rangle \leftarrow \mathsf{S.Share}(M_j)$
  5. Output $(\{\langle M_j \rangle_i\}_{j<k}, output_i, sequence)$ to each party P_i, $i < n$.

(b) Functionality $\mathcal{F}^{\text{ext}}_{\text{MPC}(f)}$ for securely evaluating extended RAM programs.

Figure 3: Functionalities for secure computation.

(Semi-)Authenticated Secure Communication Our protocol relies on proper authentication during account creation, but also on the ability of users to anonymously communicate with operators during entry creation. We formalize this by using $\mathcal{F}_{\text{SMT}}$ for normal secure authenticated communication and $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ for secure communication where the initial receiver of the message is authenticated whereas the initial sender remains anonymous. $\mathcal{F}_{\text{SMT}}$ simply forwards a message msg together with the sender pid pid_S to the receiver R with pid pid_R, thus authenticating both the sender and receiver and keeping the content confidential. $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ on the other hand only forwards msg to the receiver R with pid pid_R but keeps the identity of the sender S secret, thus only authenticating the receiver of the message. To enable communication in both directions, $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ additionally allows a receiver to respond to the message. For the formal definitions of these functionalities, see Fig. 11. Note that we assume all communication between operators to use secure channels and omit the usage of $\mathcal{F}_{\text{SMT}}$ there and will simply write $O \to O' : (msg)$ as a shorthand for transmitting msg from O to O' using $\mathcal{F}_{\text{SMT}}$.

In practice, this can be achieved through anonymous wireless communication such as NFC or Bluetooth LE or the usage of anonymization networks such as TOR when communication over the internet is required.

3.2 Protocol Description

We now give a high-level description of our protocol, along with a formal one in Figs. 5 to 6. The NP-relations used in the NIZK proofs are explained in Fig. 4.

Logbooks are identified by their logical memory address, and a list of free addresses $\mathsf{L_{Free}}$ is synchronized between the operators. When a new user registers, a mapping is created between that user (represented by their public key) and a logbook (represented by its address). The number of logbooks (i.e., both assigned and free addresses) determines the size of the shared memory and the cost of accessing it. Thus, if the ORAM used does not allow the memory size to be increased dynamically, it must be increased for a new period when the number of free addresses becomes too small.

Details of the relation $\mathcal{R}_{\mathtt{UR}}$

$stmt \coloneqq (\mathsf{pk}_U)$
$wit \coloneqq (\mathsf{sk}_U)$

$\mathcal{R}_{\mathtt{UR}} = \{(stmt, wit) \mid$
- $\mathsf{KeyVerify}(\mathsf{pk}_U, \mathsf{sk}_U) = 1$

$\}$

(a) Relation $\mathcal{R}_{\mathtt{UR}}$ used during user registration.

Details of the relation $\mathcal{R}_{\mathtt{CE}}$

$stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
$wit \coloneqq (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r)$

$\mathcal{R}_{\mathtt{CE}} = \{(stmt, wit) \mid$
- $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
- $\mathsf{TSIG.Vf}(crs_{\mathsf{TSIG}}, \mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) = 1$
- $\mathsf{KeyVerify}(\mathsf{pk}_U, \mathsf{sk}_U) = 1$

$\}$

(b) Relation $\mathcal{R}_{\mathtt{CE}}$ used when creating new entries.

Details of the relation $\mathcal{R}_{\mathtt{PE}}$

$stmt \coloneqq (\mathsf{pk}_U, addr, ct_U, \mathsf{ek})$
$wit \coloneqq (r, \mathsf{sk}_U)$

$\mathcal{R}_{\mathtt{PE}} = \{(stmt, wit) \mid$
- $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
- $\mathsf{KeyVerify}(\mathsf{pk}_U, \mathsf{sk}_U) = 1$

$\}$

(c) Relation $\mathcal{R}_{\mathtt{PE}}$ used when proving an entry was created.

Figure 4: Zero-knowledge relations used in Π_{POBA}.

Init To set up the system, each operator generates a signature keypair $(\mathsf{sk}_O, \mathsf{vk}_O)$. They also perform a distributed key generation for the threshold encryption scheme and the threshold signature scheme, captured by $\mathcal{F}_{\mathrm{Keygen}}^{\mathsf{TPKE}}$ and $\mathcal{F}_{\mathrm{Keygen}}^{\mathsf{TSIG}}$ (cf. Fig. 8), to get a common encryption key ek with decryption key shares dk_i as well as a common verification key vk together with signing key shares sk_i. All operators sign the resulting $(\mathsf{ek}, \mathsf{vk})$ with their individual signing key, exchange these signatures, and store the aggregated signature.

User Registration To register, a user chooses one of the system operators as their home operator and creates an account with them. The user generates a signature key pair $(\mathsf{sk}_U, \mathsf{pk}_U)$ and proves knowledge of the signing key sk_U in zero-knowledge to the home operator. The home operator then picks a free logbook address $addr$ from $\mathsf{L}_{\mathsf{Free}}$ and the logbook index $\mathsf{L}_{\mathsf{Keys}}$ is updated with the new mapping of the user's verification key pk_U to the logbook address $addr$. Then, the home operator communicates with all other operators to ensure that everyone's logbook index is updated accordingly, and to obtain partial signatures on the mapping $(\mathsf{pk}_U, addr)$. Finally, the user receives the (combined) signature σ from the operators.

Signing both the user's public key and the address with the threshold signature scheme ensures that no one can impersonate honest users when creating entries: we require the user to prove knowledge of a signature of the entry under that public key to create a new entry, as well as knowledge of the threshold signature on $(\mathsf{pk}_U, addr)$. Even if a minority of operators are corrupted, an adversary cannot create such a signature for $(\mathsf{pk}'_U, addr)$ with a different public key but the address of an honest user, and the threshold signature on $(\mathsf{pk}_U, addr)$ cannot be used to impersonate the honest user, since knowledge of sk_U is also required to create a new entry.

CreateEntry As mentioned before, adding entries to a logbook is divided into two steps: First, the CreateEntry task is run between an (anonymous) user and an operator. In this step, a pending logbook entry is created and authenticated by the user. Specifically, the user and operator first agree on the content *entry* of the entry and the current period *period* (out-of-protocol). Then, the user (threshold) encrypts their logbook address $addr$ as ct_U and proves the following in zero-knowledge:

- the user knows a valid (threshold) signature σ for their public key pk_U and logbook address $addr$
- the (threshold) encryption ct_U contains this logbook address $addr$
- the user knows the secret key sk_U corresponding to pk_U

Additionally, a hash of *period*, ct_U and the content of the entry is part of the statement, which thanks to the simulation-extractability ties the proof to the entry, preventing reuse

Protocol $\Pi_{\text{POBA}}(1^\lambda, n, t)$, Part 1

Building Blocks:
- Collision-resistant hash function H
- IND-CPA-secure threshold PKE scheme TPKE
- TS-UF-01-secure threshold signature scheme TSIG
- EUF-CMA-secure signature scheme SIG
- Simulation-extractable non-interactive zero-knowledge proof system ZK

Party State:
- Each operator O stores:
  - ZK CRS crs_{ZK}, TSIG parameters crs_{TSIG}
  - System encryption key ek, verification key vk and aggregated signature $\hat{\sigma}$
  - Corresponding secret key shares dk_i and sk_i
  - An operator keypair $(\mathsf{sk}_O, \mathsf{vk}_O)$
  - List $\mathsf{L}_{\mathsf{Pending}}$ of pending entries
  - Value id_{last} for indexing pending entries
  - Shamir secret-share $\langle M_j \rangle$ of the oblivious memory M_j for each period j, all initially set to 0
- The operators store synchronized copies of:
  - List $\mathsf{L}_{\mathsf{Free}}$ of free logbook addresses
  - Logbook index $\mathsf{L}_{\mathsf{Keys}}$ that maps user public keys to logbook addresses
- Each user U stores:
  - ZK CRS crs_{ZK}, TSIG parameters crs_{TSIG}
  - System encryption key ek and verification key vk
  - Credentials $creds := (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  - List $\mathsf{L}_{\mathsf{Receipt}}$ of receipts for logbook entries

Init:
- Input each O: $(\texttt{init})$
- Behavior (each O):
  1. Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$ (cf. Fig. 11a)
  2. Generate and store $(\mathsf{sk}_O, \mathsf{vk}_O) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  3. Send $(\mathsf{Keygen}, 1^\lambda)$ to $\mathcal{F}^{\mathsf{TPKE}}_{\text{Keygen}}$ `// Distributed key generation (cf. Fig. 8a)`
  4. Wait for output $(\mathsf{ek}, \mathsf{dk}_i)$ from $\mathcal{F}^{\mathsf{TPKE}}_{\text{Keygen}}$
  5. Send $(\mathsf{Keygen}, crs_{\mathsf{TSIG}})$ to $\mathcal{F}^{\mathsf{TSIG}}_{\text{Keygen}}$ `// Distributed key generation (cf. Fig. 8b)`
  6. Wait for output $(\mathsf{vk}, \mathsf{sk}_i)$ from $\mathcal{F}^{\mathsf{TSIG}}_{\text{Keygen}}$
  7. Store $\mathsf{ek}, \mathsf{vk}, \mathsf{dk}_i, \mathsf{sk}_i$
- Output O: $(\texttt{init}, \texttt{ok})$

UReg:
- Input U: $(\texttt{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO})$
- Input HO: $(\texttt{ureg}, \mathsf{ssid}, \mathsf{pid}_U)$
- Input all other O': $(\texttt{ureg}, \mathsf{ssid})$
- Behavior:
  - HO: Send $(\mathsf{ssid}\|1, \mathsf{pid}_U, (\mathsf{ek}, \mathsf{vk}, \hat{\sigma}))$ to $\mathcal{F}_{\text{SMT}}$
  - U: Receive $(\mathsf{ssid}\|1, \mathsf{pid}_{HO}, (\mathsf{ek}, \mathsf{vk}, \hat{\sigma}))$ from $\mathcal{F}_{\text{SMT}}$
  - U: Store ek and vk
  - U: Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$
  - U: $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  - U: $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, (\mathsf{pk}_U), (\mathsf{sk}_U))$
  - U: Send $(\mathsf{ssid}\|2, \mathsf{pid}_{HO}, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ to $\mathcal{F}_{\text{SMT}}$
  - HO: Wait for output $(\mathsf{ssid}\|2, \mathsf{pid}_U, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$
  - HO: If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, (\mathsf{pk}_U), \pi_{\mathsf{sk}_U}) \neq 1$, abort
  - HO: Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  - HO: Add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  - $HO \rightarrow$ all O': $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$
  - all O': $stmt := (\mathsf{pk}_U)$
  - all O': If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\mathsf{Free}}$, abort
  - all O': Remove $addr$ from $\mathsf{L}_{\mathsf{Free}}$
  - all O': Add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  - all O': $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  - all O' $\rightarrow HO$: $(\mathsf{ssid}, \sigma_i)$
  - HO: $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  - HO: Set T to be the first $t+1$ σ_i
  - HO: $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\mathsf{TSIG}}, T)$
  - HO: Send $(\mathsf{ssid}\|3, \mathsf{pid}_U, (addr, \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  - U: Receive $(\mathsf{ssid}\|3, \mathsf{pid}_{HO}, (addr, \sigma))$ from $\mathcal{F}_{\text{SMT}}$
  - U: Abort if $\mathsf{TSIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$
  - U: Store $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
- Output U: $(\mathsf{ssid}, \mathsf{ok})$
- Output HO: $(\mathsf{ssid}, \mathsf{ok})$

Figure 5: Protocol Π_{POBA} part 1, initialization and user registration

of a proof generated for a different entry. As previously mentioned, this ensures that the user has previously registered an account and prevents an adversary from impersonating an honest user. The operator checks this proof, stores the pending entry in a list of pending entries, and provides the user with a receipt *receipt* for the entry, which contains a signature σ_{entry} on the above hash.

InsertEntry The second step is performed later jointly by the operators, where the pending entries are actually written to the correct logbooks. This is done by each operator first locally performing a partial decryption of the user's encrypted logbook address and then running a secure multiparty protocol $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$ (cf. Figs. 3a and 7) that takes the pending entry along with the decryption shares as input, reconstructs the user's logbook address, and then writes the entry to the logbook with that address.

Doing partial decryption in the clear and only having to do the (additive) reconstruction of the plaintext from the partial decryptions in MPC gives a significant performance advantage over doing the whole decryption in MPC. Decoupling the (lightweight) interaction with the user from the execution of this secure multi-party protocol not only allows faster

Protocol $\Pi_{\text{POBA}}(1^\lambda, n, t)$, Part 2

CreateEntry:
- Input U: $(\texttt{createentry}, \mathsf{ssid}, period_U, entry_U, \mathsf{pid}_O)$
- Input O: $(\texttt{createentry}, \mathsf{ssid}, period_O, entry_O)$
- Behavior:
  - U: Retrieve $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$, ek, vk and vk_{HO}, abort if those values are not yet stored
  - U: $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
  - U: $h \coloneqq H(period_U, ct_U, entry_U)$
  - U: $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  - U: $wit \coloneqq (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r)$
  - U: $\pi \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, wit)$
  - U: Send $(\mathsf{send}, \mathsf{ssid}\|1, \mathsf{pid}_O, (entry_U, ct_U, \pi))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - O: Wait for output $(\mathsf{send}, \mathsf{ssid}\|1, (entry_U, ct_U, \pi))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - O: If $entry_U \neq entry_O$, abort
  - O: $h \coloneqq H(period_O, ct_U, entry_O)$
  - O: $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  - O: If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, \pi) \neq 1$, abort
  - O: $id_{\mathsf{last}} \coloneqq id_{\mathsf{last}} + 1$
  - O: Add $(id_{\mathsf{last}}, period_O, ct_U, entry_O, \pi)$ to $\mathsf{L}_{\mathsf{Pending}}$
  - O: $\sigma_{entry} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O, h)$
  - O: Send $(\mathsf{respond}, \mathsf{ssid}\|1, (\sigma_{entry}, \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - U: Wait for output $(\mathsf{respond}, \mathsf{ssid}\|1, (\sigma_{entry}, \mathsf{vk}_O))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - U: Check $\mathsf{SIG.Vf}(\mathsf{vk}_O, h, \sigma_{entry})$
  - U: Store $receipt \coloneqq (entry_U, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ in $\mathsf{L}_{\mathsf{Receipt}}$
- Output U: $(\mathsf{ssid}, \mathsf{ok})$
- Output O: $(\mathsf{ssid}, \mathsf{ok}, id_{\mathsf{last}})$

InsertEntry:
- Input O: $(\texttt{insertentry}, \mathsf{ssid}, period, id)$
- Input all other O': $(\texttt{insertentry}, \mathsf{ssid})$
- Behavior:
  - O: Retrieve $(id, period, ct_U, entry, \pi)$ from $\mathsf{L}_{\mathsf{Pending}}$ and abort if no entry with $(id, period)$ exists
  - $O \rightarrow$ all O': $(\mathsf{ssid}, period, ct_U, entry, \pi)$
  - all: $h \coloneqq H(period, ct_U, entry)$
  - all: $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  - all: If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, \pi) \neq 1$, abort[a]
  - all: $ct_i \coloneqq \mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct_U)$
  - all: Send $(\text{Run}, (ct_U, ct_i, entry), \mathsf{Share}M_{period})$ to $\mathcal{F}_{\text{MPC(insert)}}$ and wait for output $(written)$
  - all: If $written = \mathsf{false}$ output $(\mathsf{ssid}, \texttt{error})$
  - all: Otherwise output $(\mathsf{ssid}, \mathsf{ok})$
- Output all O: $(\mathsf{ssid}, \mathsf{ok}/\texttt{error})$

ProveEntry:
- Input U: $(\texttt{proveentry}, \mathsf{ssid}, entry, period, \mathsf{pid}_O)$
- Behavior:
  - U: Retrieve $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$, ek and vk
  - U: Retrieve $receipt \coloneqq (entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ from $\mathsf{L}_{\mathsf{Receipt}}$. If no such receipt exists, abort
  - U: $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U, \mathsf{ek})$, $wit \coloneqq (r, \mathsf{sk}_U)$
  - U: $\pi \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, wit)$
  - U: Send $(\mathsf{ssid}\|1, \mathsf{pid}_O, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ to $\mathcal{F}_{\text{SMT}}$
  - O: Wait for output $(\mathsf{ssid}\|1, \mathsf{pid}_U, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ from $\mathcal{F}_{\text{SMT}}$
  - O: Check if vk_O is a valid operator key, else abort
  - O: $h \coloneqq H(period, ct_U, entry)$, $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U, \mathsf{ek})$
  - O: If $\mathsf{SIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  - O: If $\mathsf{SIG.Vf}(\mathsf{vk}_O, h, \sigma_{entry}) \neq 1$, abort
  - O: If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, \pi) \neq 1$, abort
  - O: Find $(\mathsf{pk}_U, addr, U)$ in $\mathsf{L}_{\mathsf{Keys}}$
- Output O: $(\mathsf{ssid}, \texttt{proveentry}, \mathsf{pid}_U, entry, period)$

Analytics:
- Input each O: $(\texttt{analytics}, \mathsf{ssid}, func, \{period_i\}, \{U_i\}, aux_O)$
- Behavior:
  - all: For each $U \in \{U_i\}$ lookup $addr_i$ in $\mathsf{L}_{\mathsf{Keys}}$
  - all: Send $(\text{Run}, \mathsf{ssid}\|1, \{\langle M_j \rangle\}_{j \in \{period_i\}}, (\{addr_i\}, aux_O))$ to $\mathcal{F}_{\text{MPC}(func)}^{\text{ext}}$ and wait for output $(\{\langle M_j \rangle_i\}_{j<k}, output_i, round, sequence)$
  - all: Set $\texttt{result}_O \coloneqq output_i$
- Output each O: $(\mathsf{ssid}, \texttt{result}_O)$

[a] This is not strictly required for semi-honest security

Figure 6: Protocol Π_{POBA} part 2, logbook interaction and management

user interactions, but also to handle peaks higher than the throughput of the MPC, as long as the average remains below the throughput.

ProveEntry This task is used when a user wants to prove to an operator that they have made a certain logbook entry. Note that this operator intentionally learns the user's identity, the content of the entry, the period for which the entry was made, and the operator with which the entry was made. The basis for this task is the receipt *receipt* that the user received from CreateEntry as confirmation of their entry. The operator is given a period *period*, a message *msg*, a user public key pk_U, a logbook address *addr*, a (threshold) signature σ, an encrypted logbook address ct_U, an entry signature σ_{entry}, an operator verification key vk_O, and a zk proof π, and checks

- that there exists a logbook entry for the message *msg* in period *period* (for *some* user): for this, the operator checks the validity of the signature σ_{entry} (from the CreateEntry task) on the logbook entry *msg*, the period *period*, and the encrypted logbook address ct_U under the operator verification key vk_O
- whether the user who made the entry is a registered user: for this, the (threshold) signature σ (from the UReg task) on $(\mathsf{pk}_U, addr)$ is checked
- that the user who made the entry is the same user who is currently executing this

RAM-program $\mathsf{insert}(\{ct^O, ct_i^O, msg^O\}, M)$

1. Set $ct := ct^O, msg := msg^O$ for the first operator O
2. Set $addr := \mathsf{TPKE.Combine}(ct, (ct_i^{O_1}, \ldots, ct_i^{O_t}))$
3. Set $\mathsf{Log} := M[addr]$
4. Set $written := \mathsf{false}$
5. For $j = 0, \ldots, maxentries$:
   - If $written = \mathsf{false} \wedge \mathsf{Log}[j] = \bot$:
     (a) $\mathsf{Log}[j] := msg$
     (b) $written := \mathsf{true}$
6. Write the updated logbook back: $M[addr] := \mathsf{Log}$
7. Output $written$

Figure 7: The RAM program insert to write log entries into the correct logbook.

task: to do this, the user proves to the operator in zero-knowledge that the encrypted logbook address ct_U contains $addr$ and that they know the secret key associated with the public key pk_U of the user who made the entry

If all checks succeed, the operator is convinced that this user indeed made the entry msg during period $period$ with the operator whose public key is vk_O.

Analytics To perform analytics on the user logbooks, all operators have to agree on some admissible function $func$ to evaluate. This function is then evaluated using the secure multi-party protocol (cf. Fig. 3b). If data from multiple periods should be used, then it is gathered from each ORAM independently and combined.

3.3 Security

First, we informally argue why our protocol satisfies the desired security properties: To create logbook entries, users need a threshold signature on their public key and their assigned logbook address. The security of TSIG implies that entries can only be created for registered users. Additionally, a proof of knowledge of the user's secret key is provided. This prevents anyone from creating logbook entries in the name of honest users, and since the zero-knowledge proof is non-malleable (implied by simulation-extractability) and the (hash of the) entry is part of the statement, it also prevents changing entries before they are inserted into the logbook. Furthermore, a logbook entry also requires the signature of the operator involved in order to be used in ProveEntry, and it will only end up in the logbook if the involved operator agrees and forwards it to the other operators. Thus, entries are only created if both user and operator agree. The use of threshold encryption for the logbook address and NIZK proofs during creation ensures that the user's identity remains hidden. When inserting entries, the secure computation functionality ensures that no information about the logbook involved is revealed. To later prove having created an entry, a user simply presents the signature they received if and only if they successfully completed a CreateEntry task for that message, along with the signature on their public key and logbook address. Computation of analytics functions is done by $\mathcal{F}_{\mathrm{MPC}(func)}^{\mathrm{ext}}$, which does not reveal anything except the desired output and the number of memory accesses, which we do not consider a privacy violation.[8]

Security is captured by the following theorem, which we formally prove in Appendix E.

Theorem 1 (informal)**.** Π_{POBA} *UC-realizes* $\mathcal{F}_{\mathrm{POBA}}$ *wrt. adversaries* $\mathcal{A}$ *that may statically corrupt any number of users maliciously and up to* $t \leq \frac{n-1}{2}$ *operators semi-honestly.*

[8] While in theory a function could be designed in a way that the number of memory accesses encodes some private information, such behavior can be interpreted as part of the output and honest operators should not agree to compute them.

4 Applications

POBA has various applications in areas where users should be able to perform transactions in an anonymous and unlinkable way, but operators still need to be able to analyze the collected transaction data. These applications include, but are not limited to, check-in/check-out based payment systems for public transportation, charging/discharging electric vehicle batteries in smart grids, coalition loyalty programs, and behavior-based car insurance. We now take a closer look at the first application to illustrate how POBA can be used in practice. In Appendix C we additionally consider the charging/discharging electric vehicle batteries in smart grids application to showcase POBA's versatility.

As an exemplary application for POBA, we propose a check-in/check-out based payment system for public transportation. In the context of mobility-as-a-service (MaaS), it will become beneficial to offer user-friendly systems that incorporate the services of several mobility providers. Therefore, we sketch here a payment system for multimodal public transportation where a customer's trip data is collected and analyzed in a privacy-preserving way. Note that users are anonymous during trips, but spot checks are identifying.

Registration When registering, users choose a home mobility provider to create an account with.

Check-in/Check-out A check-in/check-out-based system allows users to board transportation vehicles without first purchasing a ticket. Instead, the fare is calculated at a later time. More precisely, when using trains, for example, users submit a 'Check-in' message to their logbook upon entering the train, and a 'Check-out' message upon exiting. Fare calculation is deferred until the end of the billing period, e.g., the end of the month. This enables the use of flexible tariffs, such as combining multiple individual trips into a cheaper daily flat rate. The check-in and check-out process is handled by the CreateEntry task with the mobility provider with whom the trip takes place. Communication may actually be with a subsidiary of the mobility provider (e.g., a device in a train). A logbook entry may look like this: $msg \coloneqq$ (`check_in`/`check_out`, *train_type*, *train_number*, *terminal_station*, *current_station*, *current_time*). In practice, all communication required for check-in/check-out can be achieved locally through short-distance wireless communication such as NFC or Bluetooth LE and requires neither the provider device nor the user device to have internet connectivity.

Spot Checks In public transportation systems without physical barriers to control access, ticket inspectors are commonly used to realize spot checks for passengers. This can be done in our system with the ProveEntry and CreateEntry tasks.

The customer first utilizes the ProveEntry task to prove that a message of type $msg \coloneqq$ (`check_in`, *train_type*, *train_number*, *terminal_station*, *entry_station*, *entry_time*) is in the database that matches the current train and a timestamp *entry_time* that is reasonably close to the current time. The ticket inspector[9] is now convinced that the user has a valid account at one of the operators and has previously checked-in to this train. Note that the ticket inspector learns the contents of the message *msg*, but everything except the entry station is already known by context.

To ensure that the user has not already checked out, CreateEntry is used: The ticket inspector and the user create a logbook entry for the message $msg \coloneqq$ (`inspection`, *train_type*, *train_number*, *terminal_station*, *current_station*, *current_time*). The actual check that a user has not checked out yet is then postponed to the invoice calculation. In the case that the user had already checked out before the spot check, a fine can be added to the user's invoice during invoice generation.

[9] We assume that the ticket inspector acts as a subsidiary of the train's operator

Invoice Generation At the end of a billing period, for each user an invoice is generated, utilizing the Analytics task. There, all logbook entries for a user for the current billing period are loaded. Then they are checked for validity: Are there always matching `check_in` and `check_out` entries? Are `inspection` entries in between appropriate `check_in` and `check_out` entries? If all is valid, the invoice can be generated. If not, a fine is added to the invoice. Then, the invoice (consisting of the aggregate sum for the month and any extra fines with information about the respective spot-check) is output to the respective home provider. Note that each user only gets an invoice from their home provider, regardless of the mobility providers they took trips with.

Clearing between Providers After issuing invoices to all users, operators initiate their clearing phase. The trip fees must be redistributed to the mobility providers with whom the trips were actually taken, as users pay their invoices to their home operators. To achieve this, the system also calculates during each invoice generation which trips were made with which mobility provider. The revenue for each mobility provider is then written to an array, secret-shared, and each mobility provider receives a share as output. When the next user's invoice is generated, all operators input their current share as auxiliary input aux, the MPC reconstructs the revenue array, updates it, and then secret-shares and outputs it again. Once all invoices for the current period have been created, the operators can initiate the actual clearing. For this, the shares are input into the MPC again, which then calculates the distribution of revenues among providers.

Route Load Analysis The Analytics task can also be used by mobility providers to analyze their route load to answer questions like "On which routes do more trains need to be deployed, where can trains be reduced, etc.?" By accessing data from all mobility providers, these analytics can take into account information beyond the boundaries of the provider's own network. This means that connections between different transportation networks can be better planned, resulting in a better overall experience for users.

5 Evaluation and Benchmarks

To evaluate the practical feasibility of POBA, we implemented our protocol and measured the execution times of the different interactions. We ran experiments for 3, 5 and 9 operators, each running on a server with an Intel® Xeon® E-2388G CPU @ 3.20 GHz and 64 GB RAM and connected via a 10 Gbit/s switch. We implemented the full protocol, but since user networking is difficult to control, we ignore the communication time between users and operators in our results.

5.1 Implementation details

Our implementation is written in C++. To achieve efficient zero-knowledge proofs, we use the Groth–Sahai [GS08; EG14] proof system, which enables efficient proofs for pairing-product equations. To ensure all relations used in our protocol can be expressed as such pairing-product equations, we instantiate our building blocks in a compatible way over the pairing-friendly Barreto-Naehrig curve BN-382 [BN06], which provides roughly 128 bit of security [APR21], using the RELIC toolkit v.0.7.0 [AG20] as our arithmetic backend. To make the Groth–Sahai proof system simulation-extractable, we use the standard transformation as described in e.g. [HJ12], with the one-time signature described there along with BLS signatures for the simulation trapdoor. The other building blocks are instantiated as follows: For TPKE, we chose standard IND-CPA secure threshold ElGamal encryption over $\mathbb{G}_1$. For SIG, we use BLS signatures [BLS01] over $\mathbb{G}_1$. For TSIG, we use the structure-preserving threshold signature scheme of [MMS+24]. When signing and

encrypting the logbook addresses of users, we use the Koblitz encoding scheme [Kob87] to embed them in the x-coordinate of a point in $\mathbb{G}_1$. To implement $\mathcal{F}_{\text{MPC}(\cdot)}$ and $\mathcal{F}^{\text{ext}}_{\text{MPC}(\cdot)}$, we use the MP-SPDZ framework [Kel20] in the semi-honest honest majority setting with Shamir sharing. For better efficiency during entry insertion, we split the computation into two parts, which are run with different parameters: First, the partial decryptions of ct_U are combined in an instance MPC_1 over the field $\mathbb{F}_p$ over which BN-382 is defined. This allows us to obtain Shamir shares of the x-coordinate which encodes the logbook address. We then use the share conversion method described in [DT08] to convert them to Shamir shares over a field $\mathbb{F}_q$ with a 128 bit prime q chosen by MP-SPDZ. The rest of the computation is then done in an instance MPC_2 over $\mathbb{F}_q$, which uses PathORAM [KS14] for oblivious memory accesses. This splitting is done so that most of the computation, including the expensive ORAM accesses, does not have to be done over a 382 bit prime field but instead can be done in a smaller and MPC-friendly field.

Additionally, we also implemented the InsertEntry protocol for the special case of $n = 3, t = 1$ using GigaDORAM [FOSZ23] for memory accesses. Concretely, we still make use of the MP-SPDZ instance MPC_1 to combine the partial decryptions, but then use the emp-toolkit [WMK16] as MPC_2, which is the MPC-backend of GigaDORAM. Since GigaDORAM uses replicated XOR secret sharing, we now need to perform bit-decomposition in MP-SPDZ to convert shares between these MPC-instances. Since this now represents the majority of the time spent in MPC_1 and involves many rounds of communication, we perform this step on batches of 20 entries together. To represent our logbook in GigaDORAM, which only supports memory cells with 64 bit, we assign $100 \times \frac{\text{entry-size}}{64}$ consecutive addresses to each logbook, encrypting the starting address of this block in the ciphertext from the user. For our concrete example, logbook entries consist of 6 32 bit values, so we assigned 300 addresses to each logbook. To avoid having to perform 300 oblivious reads and 300 writes per insert, we store a pointer to the first free entry at the start of the logbook. Insertion then works by reading that pointer, calculating the addresses of the 3 relevant memory cells from it (i.e., using the pointer, pointer+1 and pointer+2) and updating the stored pointer to the value of the next free entry (i.e., old pointer +3). Then the entry gets written to the three relevant memory cells, resulting in a total of 1 oblivious read and 4 oblivious writes.

5.2 Results

We use the check-in/check-out based payment system for public transportation from Section 4 as an application to benchmark. Thus, our logbooks contain 100 slots of 6×32 bit each to store check-in/check-out messages. We benchmark ORAMs of size 2^{15} ($\approx$32 000) and 2^{20} ($\approx$1 million) entries, which results in the same number of maximum registered users in the protocol. As analysis function, we use the invoice generation.

To obtain representative results, we average our measurements over 50 runs. First, as expected, the running times of protocols involving users, i.e., UReg, CreateEntry and ProveEntry, are very fast and independent of the number of operators or other users involved. Without taking communication time between user and home operator into consideration, registering a new account takes 0.45 s. This allows for account creation to finish in under one second when performed over the internet. Creating a new entry takes 0.28 s, which again allows the interaction including communication over NFC or BLE to finish in one to two seconds, a time common for other NFC-based transactions. Proving an entry was created (e.g., during spot-checks) takes 0.22 s and consists of a single message from user to inspector which could be transmitted through scanning of a QR-code or again NFC. Thus, we achieved our goal of having very efficient client-facing protocols.

On the operator end, both tasks can also be parallelized to improve throughput: user registration only requires synchronizing free logbook addresses or assigning pools of free addresses to each instance, and CreateEntry and ProveEntry require no synchronization.

Table 1: MP-SPDZ with PathORAM: Average time (in ms) and transmitted data (in kB/MB) to complete the InsertEntry and Analytics tasks, where $\sum$ refers to the total time of InsertEntry, and MPC_1 and MPC_2 to the (amortized) time spent within the respective MPC protocol for InsertEntry. The setting differentiates between the number of operators n and the number of users #.

Setting	InsertEntry					Analytics
	MPC_1		MPC_2		$\sum$	
n=3, #=2^{15}	0.2 ms	1 kB	679 ms	8.5 MB	916 ms	652 ms
n=3, #=2^{20}	0.2 ms	1 kB	867 ms	10.2 MB	1143 ms	765 ms
n=5, #=2^{15}	0.4 ms	4 kB	1000 ms	21.6 MB	1269 ms	1003 ms
n=5, #=2^{20}	0.4 ms	4 kB	1281 ms	25.4 MB	1531 ms	1196 ms
n=9, #=2^{15}	1.2 ms	13 kB	2146 ms	30.1 MB	2468 ms	2782 ms
n=9, #=2^{20}	1.2 ms	13 kB	2851 ms	70.6 MB	3119 ms	3286 ms

For InsertEntry and Analytics, the runtime depends on both the number of operators and the number of registered users (i.e., the size of the ORAM). Results for $n = 3, 5, 9$ operators and number of registered users $\# = 2^{15}, 2^{20}$ when using MP-SPDZ are shown in Table 1. For all results, we used the maximum number of corrupted operators, namely $t = \frac{n-1}{2}$. The majority of the time spent in the second MPC protocol is reading and writing the logbook from memory: For $n = 9$ operators and 2^{15} users, out of the 2.146 s for MPC_2, 1.016 s are spent reading the logbook from memory, and 0.999 s are spent writing the updated logbook back. Updating the logbook with the new entry, which involves iterating over all 100 slots to find the first empty one, only takes the remaining 0.132 s. For InsertEntry, it is possible to parallelize the verification of the zero-knowledge proof and the partial decryption, and the reconstruction of the shared logbook address from the partial decryption, i.e., execution of MPC_1. MPC_2 must be executed sequentially because PathORAM does not support parallel memory access. Thus, the bottleneck for throughput is the time required for MPC_2.

Overall, ORAM in MP-SPDZ currently does not scale well with an increasing number of parties, with the throughput for 9 operators already severely limiting the number of users. Further increasing the number of operators to e.g., $n = 20$ reduces throughput to roughly 5000 entries per day.

For the special case of $n = 3$, optimized protocols for distributed ORAM significantly increase throughput. When using GigaDORAM [FOSZ23] and the emp-toolkit (on which the implementation of GigaDORAM is based) for MPC_2, the time to access ORAM decreases drastically, with the whole of MPC_2 only taking 39 ms per entry with 2^{15} logbooks and 42 ms with 2^{20} logbooks, see Table 2. While performance seems to allow for much greater numbers of logbooks and logbook sizes, the current implementation of

Table 2: InsertEntry with GigaDORAM and $n = 3$ operators. MPC_1 is the (amortized) time per entry spent in MP-SPDZ to combine the partial decryptions to obtain the logbook address and convert the resulting Shamir-sharing into xor-sharing required for GigaDORAM. MPC_2 is the time per entry spent in emp-toolkit to write the entry into ORAM. $\sum$ is the total (amortized) time for the whole InsertEntry protocol, additionally consisting of verification of the zero-knowledge proof and partial decryption of the ciphertext.

Logbooks	MPC_1	MPC_2	$\sum$
2^{15}	80 ms	39 ms	126 ms
2^{20}	80 ms	42 ms	128 ms

GigaDORAM limits the size of the memory to $< 2^{30}$, and with each of our logbooks taking $300 \approx 2^9$ our implementation is limited to $\approx 2^{20}$ logbooks. The running time of MPC_1 on the other hand increases to 1.6 s for 20 entries, or on average 80 ms per entry. Performing the required steps sequentially, our implementation takes ≈0.13 s per entry, allowing ≈650 000 entries per day. However, only MPC_2 needs to be sequentialised (since GigaDORAM also does not support parallel accesses), whereas verification of the zero-knowledge proof, partial decryption and MPC_1 can be parallelized across multiple servers per operator. This way, the 0.042 s for MPC_2 limit the throughput, allowing over two million entries per day with a million registered users.

A public transport operator serving a rural area with a population of around half a million told us that they have around 370 000 registered users, peaking at around 10 million trips in a rainy November, with an average of around 700 000 entries (two for each trip) per day. Thus, a consortium of three operators could already use our system in this setting. On the extreme end however, Transport for London reports 26 million daily trips for the greater London area, which is currently far beyond the limit of our protocol.

Acknowledgments

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. The authors would like to thank Marcel Keller for his support when implementing our protocol in MP-SPDZ and Marc Leinweber for his help during benchmarking.

References

[AG20] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic, 2020.

[AGJ+22] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: privacy preserving aggregate statistics via boolean shares. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 516–539. Springer, 2022. DOI: 10.1007/978-3-031-14791-3_23. URL: https://doi.org/10.1007/978-3-031-14791-3%5C_23.

[ALF+13] Man Ho Au, Joseph K Liu, Junbin Fang, Zoe L Jiang, Willy Susilo, and Jianying Zhou. A new payment system for enhancing location privacy of electric vehicles. *IEEE transactions on vehicular technology*, 63(1):3–18, 2013.

[ALT+15] Ghada Arfaoui, Jean-François Lalande, Jacques Traoré, Nicolas Desmoulins, Pascal Berthomé, and Saïd Gharout. A practical set-membership proof for privacy-preserving NFC mobile ticketing. *Proceedings on Privacy Enhancing Technologies*, 2015(2):25–45, April 2015. DOI: 10.1515/popets-2015-0019.

[APR21] Diego F. Aranha, Elena Pagnin, and Francisco Rodríguez-Henríquez. LOVE a pairing. In Patrick Longa and Carla Ràfols, editors, *Progress in Cryptology - LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America*, volume 12912 of *Lecture Notes in Computer Science*, pages 320–340, Bogotá, Colombia. Springer, Heidelberg, October 2021. DOI: 10.1007/978-3-030-88238-9_16.

[BBC+21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy*, pages 762–776. IEEE Computer Society Press, May 2021. DOI: 10.1109/SP40001.2021.00048.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464. Springer, Heidelberg, December 2018. DOI: 10.1007/978-3-030-03329-3_15.

[BGL+22] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. ACORN: input validation for secure aggregation. Cryptology ePrint Archive, Report 2022/1461, 2022. https://eprint.iacr.org/2022/1461.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, Heidelberg, December 2001. DOI: 10.1007/3-540-45682-1_30.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005: 12th Annual International Workshop on Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, Heidelberg, August 2006. DOI: 10.1007/11693383_22.

[BPRS23] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: souper fast three-party computation for RAM programs. Cryptology ePrint Archive, Report 2023/310, 2023. https://eprint.iacr.org/2023/310.

[Can00] Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[Can20] Ran Canetti. Universally Composable Security. *Journal of the ACM*, 67(5):28:1–28:94, September 2020. DOI: 10.1145/3402457. URL: https://doi.org/10.1145/3402457.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: private, robust, and scalable computation of aggregate statistics. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 259–282. USENIX Association, 2017. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs.

[CHB+24] Ying Chen, Debiao He, Zijian Bao, Min Luo, and Kim-Kwang Raymond Choo. A post-quantum privacy-preserving payment protocol in vehicle to grid networks. *IEEE Transactions on Intelligent Vehicles*, 2024.

[DPRS23] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable distributed aggregation functions. *Proc. Priv. Enhancing Technol.*, 2023(4):578–592, 2023. DOI: 10.56553/POPETS-2023-0126. URL: https://doi.org/10.56553/popets-2023-0126.

[DT08] Ivan Damgard and Rune Thorbek. Efficient conversion of secret-shared values between different fields. Cryptology ePrint Archive, Report 2008/221, 2008. https://eprint.iacr.org/2008/221.

[EG14] Alex Escala and Jens Groth. Fine-tuning Groth-Sahai proofs. In Hugo Krawczyk, editor, *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*, volume 8383 of *Lecture Notes in Computer Science*, pages 630–649. Springer, Heidelberg, March 2014. DOI: 10.1007/978-3-642-54631-0_36.

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014: 21st Conference on Computer and Communications Security*, pages 1054–1067. ACM Press, November 2014. DOI: 10.1145/2660267.2660348.

[FKM+22] Valerie Fetzer, Marcel Keller, Sven Maier, Markus Raiber, Andy Rupp, and Rebecca Schwerdt. PUBA: privacy-preserving user-data bookkeeping and analytics. *Proceedings on Privacy Enhancing Technologies*, 2022(2):447–516, April 2022. DOI: 10.2478/popets-2022-0054.

[FOSZ23] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. Gigadoram: breaking the billion address barrier. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 3871–3888. USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/falk.

[FPE16] Giulia C. Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the unknown: privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies*, 2016(3):41–61, July 2016.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004. ISBN: 0-521-83084-2. DOI: 10.1017/CBO9780511721656. URL: http://www.wisdom.weizmann.ac.il/%5C%7Eoded/foc-vol2.html.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194. ACM Press, May 1987. DOI: 10.1145/28395.28416.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 444–459. Springer, Heidelberg, December 2006. DOI: 10.1007/11935230_29.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 415–432. Springer, Heidelberg, April 2008. DOI: 10.1007/978-3-540-78967-3_24.

[Gud14] Ivan Gudymenko. A privacy-preserving e-ticketing system for public transportation supporting fine-granular billing and local validation. In *Proceedings of the 7th International Conference on Security of Information and Networks*, pages 101–108, 2014.

[HCDF06] Thomas S. Heydt-Benjamin, Hee-Jin Chae, Benessa Defend, and Kevin Fu. Privacy for public transportation. In George Danezis and Philippe Golle, editors, *PET 2006: 6th International Workshop on Privacy Enhancing Technologies*, volume 4258 of *Lecture Notes in Computer Science*, pages 1–19. Springer, Heidelberg, June 2006. DOI: 10.1007/11957454_1.

[HCS+19] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. Privacy-preserving electronic ticket scheme with attribute-based credentials. *IEEE Transactions on Dependable and Secure Computing*, 18(4):1836–1849, 2019.

[HJ12] Dennis Hofheinz and Tibor Jager. Tightly secure signatures and public-key encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 590–607. Springer, Heidelberg, August 2012. DOI: 10.1007/978-3-642-32009-5_35.

[JJQ07] Oliver Jorns, Oliver Jung, and Gerald Quirchmayr. A privacy enhancing service architecture for ticket-based mobile applications. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 139–146. IEEE, 2007.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1575–1590. ACM Press, November 2020. DOI: 10.1145/3372297.3417872.

[KLG13] Florian Kerschbaum, Hoon Wei Lim, and Ivan Gudymenko. Privacy-preserving billing for e-ticketing systems in public transportation. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 143–154, New York, NY, USA. Association for Computing Machinery, November 2013. ISBN: 978-1-4503-2485-4. DOI: 10.1145/2517840.2517848. (Visited on 02/20/2024).

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer, Heidelberg, December 2014. DOI: 10.1007/978-3-662-45608-8_27.

[LP09] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):59–98, April 2009. DOI: 10.29012/jpc.v1i1.566.

[MDD16] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. In *ISOC Network and Distributed System Security Symposium – NDSS 2016*. The Internet Society, February 2016.

[MDND15] Milica Milutinovic, Koen Decroix, Vincent Naessens, and Bart De Decker. Privacy-preserving public transport ticketing system. In *Data and Applications Security and Privacy XXIX: 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings 29*, pages 135–150. Springer, 2015.

[MMS+24] Aikaterini Mitrokotsa, Sayantan Mukherjee, Mahdi Sedaghat, Daniel Slamanig, and Jenit Tomy. Threshold structure-preserving signatures: strong and adaptive security under standard assumptions. In Qiang Tang and Vanessa Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 14601 of *Lecture Notes in Computer Science*, pages 163–195. Springer, Heidelberg, April 2024. DOI: 10.1007/978-3-031-57718-5_6.

[MST23] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. PLASMA: private, lightweight aggregated statistics against malicious adversaries with full security. Cryptology ePrint Archive, Report 2023/080, 2023. https://eprint.iacr.org/2023/080.

[RD11] Alfredo Rial and George Danezis. Privacy-preserving smart metering. In Yan Chen and Jaideep Vaidya, editors, *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, WPES 2011, Chicago, IL, USA, October 17, 2011*, pages 49–60. ACM, 2011. DOI: 10.1145/2046556.2046564. URL: https://doi.org/10.1145/2046556.2046564.

[RDK18] Alfredo Rial, George Danezis, and Markulf Kohlweiss. Privacy-preserving smart metering revisited. *Int. J. Inf. Sec.*, 17(1):1–31, 2018. DOI: 10.1007/S10207-016-0355-8. URL: https://doi.org/10.1007/s10207-016-0355-8.

[RHBP13] Andy Rupp, Gesine Hinterwälder, Foteini Baldimtsi, and Christof Paar. P4R: privacy-preserving pre-payments with refunds for transportation systems. In Ahmad-Reza Sadeghi, editor, *FC 2013: 17th International Conference on Financial Cryptography and Data Security*, volume 7859 of *Lecture Notes in Computer Science*, pages 205–212. Springer, Heidelberg, April 2013. DOI: 10.1007/978-3-642-39884-1_17.

[SG02] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *Journal of Cryptology*, 15(2):75–96, March 2002. DOI: 10.1007/s00145-001-0020-9.

[SNF+19] Rebecca Schwerdt, Matthias Nagel, Valerie Fetzer, Tobias Gräf, and Andy Rupp. P6V2G: a privacy-preserving V2G scheme for two-way payments and reputation. *Energy Inform.*, 2(S1), 2019. DOI: 10.1186/S42162-019-0075-1. URL: https://doi.org/10.1186/s42162-019-0075-1.

[SvS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 299–310. ACM Press, November 2013. DOI: 10.1145/2508859.2516660.

[VCPM10] Arnau Vives-Guasch, Jordi Castellà-Roca, M Magdalena Payeras-Capella, and Macià Mut-Puigserver. An electronic and secure automatic fare collection system with revocable anonymity for users. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*, pages 387–392, 2010.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WZL+21] Zhiguo Wan, Tong Zhang, Weizhuang Liu, Mingqiang Wang, and Liehuang Zhu. Decentralized privacy-preserving fair exchange scheme for v2g based on blockchain. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2442–2456, 2021.

[YBY+23] Xiaohan Yue, Xue Bi, Haibo Yang, Shi Bai, and Yuan He. Pap: a privacy-preserving authentication scheme with anonymous payment for v2g networks. Cryptology ePrint Archive, Paper 2023/1189, 2023. URL: https://eprint.iacr.org/2023/1189. https://eprint.iacr.org/2023/1189.

[ZCWY14] Tianyu Zhao, Chang Chen, Lingbo Wei, and Mengke Yu. An anonymous payment system to protect the privacy of electric vehicles. In *2014 Sixth International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, October 2014. DOI: 10.1109/WCSP.2014.6992208. (Visited on 04/13/2024).

A Application-Specific Related Work

We now review the related work on solutions tailored to different applications. It is worth noting that the vast majority of them focus on billing and do not consider other analyses.

Privacy-Preserving Ticketing for Public Transportation Several proposals exist for privacy-preserving ticketing systems in public transportation [MDND15; HCDF06; ALT+15; Gud14; RHBP13; KLG13; VCPM10; JJQ07; HCS+19]. However, most of these systems focus solely on payment functions and do not address the privacy-preserving analysis of resulting data. Kerschbaum et al. [KLG13] appear to be an exception, as they are developing a privacy-preserving ticketing system for public transportation that stores the (encrypted) data in the backend. The travel data is encrypted using an additively homomorphic encryption scheme, where the decryption key is held by a trusted third party, who needs to be involved in entry operations, invoice calculation, and any analysis function (modeled as a circuit) that requires negation. Unlike POBA, their system only contains a single provider and they have no formal model or even UC proof. Furthermore, of these systems only [HCS+19] has a formal security proof, however only in a standalone simulation-based model, leaving security under arbitrary composition open.

Privacy-Preserving Electric Vehicle Charging/Discharging To enable privacy-preserving billing in smart-grid scenarios where electric vehicles are charged/discharged based on the grid state, several works [ALF+13; ZCWY14; SNF+19; WZL+21; YBY+23; CHB+24] propose solutions for billing. Of these, [SNF+19; WZL+21] have proofs in the UC framework. However, they all only compute billing information and do not allow any further analytics.

B Building Blocks

We use a variety of cryptographic building blocks in our protocol, which we formally define here.

B.1 Secret-Sharing

A secret-sharing scheme consists of a sharing algorithm that, given some secret, outputs n shares such that up to t shares reveal no information about the secret, and a reconstruction algorithm that given at least $t+1$ shares reconstructs the secret.

Definition 1 (Syntax of a Secret-Sharing Scheme)**.** A secret-sharing scheme S with number of parties n and corruption threshold t is a tuple $\mathsf{S} = (\mathsf{S.Share}, \mathsf{S.Recon})$ of PPT algorithms, where

- $\mathsf{S.Share}(m)$ outputs shares $\langle m \rangle_1, \ldots, \langle m \rangle_n$ such that up to t shares are information-theoretically independent of m
- $\mathsf{S.Recon}(\langle m \rangle_1, \ldots, \langle m \rangle_n)$ outputs m if each $\langle m \rangle_i$ is either $\perp$ or the output of $\mathsf{S.Share}(m)$, and at least $t+1$ shares are not $\perp$

We will refer to $\langle m \rangle$ as a S-sharing of message m.

B.2 Threshold PKE

We follow the notation of [SG02] for threshold encryption:

Definition 2 (Syntax of Threshold PKE)**.** A (t, n)-*threshold PKE (TPKE)* scheme TPKE consists of a tuple $(\mathsf{Keygen}, \mathsf{Encrypt}, \mathsf{PartDec}, \mathsf{PartVerify}, \mathsf{Combine})$ of PPT algorithms such that

- $\mathsf{TPKE.Keygen}(1^\lambda, n, t)$ outputs a public encryption key ek and sets $\{\mathsf{vk}_i\}_{i\in[n]}$ of verification keys and $\{\mathsf{dk}_i\}_{i\in[n]}$ of decryption key shares.
- $\mathsf{TPKE.Encrypt}(\mathsf{ek}, m)$ takes an encryption key ek, a message m in the message space $\mathcal{M}$, and outputs a ciphertext ct.
- $\mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct)$ takes a decryption key share dk_i, a ciphertext ct, and outputs a decryption share ct_i.
- $\mathsf{TPKE.PartVerify}(ct, \mathsf{vk}_i, ct_i)$ takes a ciphertext ct, a verification key vk_i and a decryption share ct_i, and outputs either 1 or 0. If the output is 1, ct_i is called a *valid* decryption share.
- $\mathsf{TPKE.Combine}(ct, T)$ takes a ciphertext ct and a set of valid decryption shares T such that $|T| = t + 1$, and outputs a message $m \in \mathcal{M}$.

We require correctness, i.e., the following two conditions to hold for all $(\mathsf{ek}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{dk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Keygen}(1^\lambda)$:

1. $\forall m \in \mathcal{M}, ct \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, m), i \in [n]$ it holds that $\mathsf{TPKE.PartVerify}(ct, \mathsf{vk}_i, \mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct)) = 1$.
2. $\forall m \in \mathcal{M}, ct \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, m)$ and any set $T \coloneqq (ct_1, \ldots, ct_{t+1})$ of valid decryption shares $ct_i \leftarrow \mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct)$ for $t + 1$ distinct decryption key shares dk_i it holds that $\mathsf{TPKE.Combine}(ct, T) = m$.

IND-CPA security for a TPKE scheme is defined as usual, except that the adversary can choose up to t key shares to receive.

We make use of a UC-functionality for distributed key generation $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ defined in Fig. 8.

Functionality $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$

Parameter: Number of parties n, threshold t
- Upon receiving $(\mathsf{Keygen}, 1^\lambda)$ from all parties:
  1. Run $(\mathsf{ek}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{dk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Keygen}(1^\lambda, n, t)$
  2. Send output $(\mathsf{ek}, \mathsf{vk}_i, \mathsf{dk}_i)$ to party P_i

(a) Functionality $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ for TPKE

Functionality $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$

Parameters: Number of parties n, threshold t
- Upon receiving $(\mathsf{Keygen}, crs_{\mathsf{TSIG}})$ from all parties:
  1. Run $(\mathsf{vk}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{TSIG.Keygen}(crs_{\mathsf{TSIG}}, n, t)$
  2. Send output $(\mathsf{vk}, \mathsf{vk}_i, \mathsf{sk}_i)$ to party P_i

(b) Functionality $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$ for TSIG

Figure 8: Functionalities for distributed key generation

B.3 Digital Signatures and PKI

We use a correct and EUF-CMA-secure aggregate signature scheme SIG.

Definition 3 (Syntax of Digital Signatures)**.** A *digital signature scheme* SIG over message space $\mathcal{M}$ consists of a tuple $(\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Vf})$ of PPT algorithms such that:

- $\mathsf{SIG.Keygen}(1^\lambda)$ takes 1^λ as input and outputs a keypair $(\mathsf{vk}, \mathsf{sk})$.
- $\mathsf{SIG.Sign}(\mathsf{sk}, m)$ takes the signing key sk and a message $m \in \mathcal{M}$ as input and outputs a signature σ.
- $\mathsf{SIG.Vf}(\mathsf{vk}, m, \sigma)$ is deterministic and takes as input a verification key vk, a message $m \in \mathcal{M}$ and a signature σ. It returns either 0 or 1.

  (Intuitively, returning 1 means that σ is valid signature for message m under verification key vk. Returning 0 means it is invalid.)

SIG is *correct* if for all λ, $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$, $m \in \mathcal{M}$, and $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}, m)$ it holds that

$$1 = \mathsf{SIG.Vf}(\mathsf{vk}, m, \sigma).$$

We additionally require a deterministic PPT algorithm $\mathsf{KeyVerify}(\mathsf{vk}, \mathsf{sk})$ so that for $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{SIG.Keygen}(\lambda)$ it holds that $\mathsf{KeyVerify}(\mathsf{vk}, \mathsf{sk}) = 1$ and if $\mathsf{KeyVerify}(\mathsf{vk}, \mathsf{sk}) = 1$ then $\mathsf{SIG.Vf}(\mathsf{vk}, m, \mathsf{SIG.Sign}(\mathsf{sk}, m)) = 1$ for all $m \in \mathcal{M}$.

Definition 4 (EUF-CMA-Security [Gol04])**.** A signature scheme $\mathsf{SIG} = (\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Vf})$ is *EUF-CMA-secure* if for every PPT adversary $\mathcal{A}$ with access to a signing oracle $\mathcal{O}^{\mathsf{SIG}}$ it holds that the following probability is negligible in the security parameter λ:

$$\Pr[(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{SIG.Keygen}(1^\lambda), (\sigma^*, m^*) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{SIG}}(\mathsf{sk}, \cdot)}(\mathsf{vk}, 1^\lambda) : \mathsf{SIG.Vf}(\sigma^*, \mathsf{vk}, m^*)] \leq \mathsf{negl}(\lambda)(1^\lambda)$$

where σ^* was never returned from $\mathcal{O}^{\mathsf{SIG}}(\mathsf{sk}, \cdot)$.

An *aggregate signature scheme* is a digital signature scheme that additionally allows multiple signatures to be aggregated into a single signature. This single aggregate signature can then be used to verify all the original signatures collectively.

We assume that all signers sign the same message[10] and therefore directly use a syntax to highlight that.

Definition 5 (Aggregate Signature Scheme for Same Message)**.** We say that an aggregate signature scheme SIG over message space $\mathcal{M}$ consists of the algorithms $(\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Vf})$ as defined in Definition 3 as well as the following two additional algorithms:

- $\mathsf{SIG.Agg}((\mathsf{vk}_1, \sigma_1), \ldots, (\mathsf{vk}_\ell, \sigma_\ell))$ takes multiple verification key and signature tuples $(\mathsf{vk}_i, \sigma_i)$ and combines them to a single aggregate signature $\hat{\sigma}$.
- $\mathsf{SIG.AVf}((\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell), m, \hat{\sigma})$ is deterministic and takes as input multiple verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell)$, a message $m \in \mathcal{M}$ and a aggregated signature $\hat{\sigma}$. It returns either 0 or 1.

  (Intuitively, returning 1 means that $\hat{\sigma}$ is valid aggregated signature for message m under verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell)$. Returning 0 means it is invalid.)

Definition 6 (Syntax of Threshold Signatures, [MMS+24])**.** A (n, t)-*threshold signature scheme* TSIG over message space $\mathcal{M}$ consists of a tuple $(\mathsf{Setup}, \mathsf{Keygen}, \mathsf{PartSign}, \mathsf{PartVerify}, \mathsf{Combine}, \mathsf{Vf})$ of PPT algorithms such that:

[10] See [BDN18] how to adapt BLS signatures to this setting.

$\mathsf{Exp}^{\mathsf{TS-UF-1}}_{\mathcal{A}}(1^\lambda)$

1 : $crs \leftarrow \mathsf{Setup}(1^\lambda)$
2 : $(n, t, \mathrm{CS}, \mathrm{st}_0) \leftarrow \mathcal{A}(crs)$
3 : $\mathrm{HS} \coloneqq [1, n] \setminus \mathrm{CS}$
4 : $(\{\mathsf{sk}_i, \mathsf{vk}_i\}, \mathsf{vk}) \leftarrow \mathsf{Keygen}(crs, n, t)$
5 : $(m^*, \sigma^*, \mathrm{st}_1) \leftarrow \mathcal{A}^{\mathsf{O}^{\mathsf{PartSign}}}(\mathrm{st}_0, \mathsf{vk}, \{\mathsf{sk}_j\}_{j\in\mathrm{CS}}, \{\mathsf{vk}_i\}_{i\in[1,n]})$
6 : **return** $(\mathsf{Vf}(crs, \mathsf{vk}, m^*, \sigma^*) \wedge |\mathrm{CS}| \leq t \wedge |S_1(m^*)| \leq t - |\mathrm{CS}|)$

Oracle $\mathsf{O}^{\mathsf{PartSign}}(i, m)$

1 : **assert** $m \in \mathcal{M} \wedge i \in \mathrm{HS}$
2 : $\sigma_i \leftarrow \mathsf{PartSign}(crs, \mathsf{sk}_i, m)$
3 : **if** $\sigma_i \neq \perp$
4 : $\quad S_1(m) = S_1(m) \cup \{i\}$
5 : **return** σ_i

Figure 9: Security game defining TS-UF-1 security for threshold signatures.

Functionality $\mathcal{F}_{\mathrm{PKI}}$

- Upon receiving $(\mathrm{register}, v)$ from some party P_i:
  1. Check that no pair (P_i, v) is already recorded, otherwise ignore this input
  2. Record (P_i, v)
- Upon receiving $(\mathrm{retrieve}, P_i)$ from some party P_j or the adversary $\mathcal{S}$:
  1. Check if some pair (P_i, v) is recorded
  2. If yes:
     – Return (P_i, v) to P_j or $\mathcal{S}$
  3. Otherwise:
     – Return $(P_i, \perp)$ to P_j or $\mathcal{S}$

Figure 10: Functionality $\mathcal{F}_{\mathrm{PKI}}$ to implement a public key infrastructure.

- $\mathsf{TSIG.Setup}(1^\lambda)$ takes 1^λ as input and outputs a common reference string crs.
- $\mathsf{TSIG.Keygen}(crs, n, t)$ takes the common reference string along with integers n and t s.t. $1 \leq t \leq n$ as input and outputs signing/verification keys $(\mathsf{sk}_i, \mathsf{vk}_i)$ along with a global verification key vk.
- $\mathsf{TSIG.PartSign}(crs, \mathsf{sk}_i, m)$ takes the common reference string, the i-th signing key and a message vector $m \in \mathcal{M}$ as inputs and outputs a partial signature σ_i.
- $\mathsf{TSIG.PartVerify}(crs, \mathsf{vk}_i, m, \sigma_i)$ takes the crs, the i-th verification key, a message $m \in \mathcal{M}$ and a partial signature σ_i as input and outputs 1 if the partial signature is valid and 0 otherwise.
- $\mathsf{TSIG.Combine}(crs, T)$ takes the crs and a set of valid partial signatures T with $|T| = t + 1$ and outputs a signature σ.
- $\mathsf{TSIG.Vf}(crs, \mathsf{vk}, m, \sigma)$ takes the crs, a verification key, a message $m \in \mathcal{M}$ and a signature and outputs 1 if the signature is valid and 0 otherwise.

TSIG is *correct* if for all λ, $crs \leftarrow \mathsf{TSIG.Setup}(1^\lambda)$, $(\{\mathsf{sk}_i, \mathsf{vk}_i\}, \mathsf{vk}) \leftarrow \mathsf{TSIG.Keygen}(crs, n, t)$, $m \in \mathcal{M}$, $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs, \mathsf{sk}_i, m)$ for $i \in [1, n]$, $\forall T \subset \{\sigma_i\} : |T| = t + 1$ and $\sigma \leftarrow \mathsf{TSIG.Combine}(crs, T)$ it holds that $\mathsf{TSIG.Vf}(crs, \mathsf{vk}, m, \sigma) = 1$

Similar to EUF-CMA-security for normal signatures, we require TSIG to be threshold unforgeable.

Definition 7 (Threshold Unforgeability [MMS+24])**.** Let $\mathsf{TSIG} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{PartSign}, \mathsf{PartVerify}, \mathsf{Combine}, \mathsf{Vf})$ be an (n, t)-threshold signature scheme over message space $\mathcal{M}$. TSIG is called *TS-UF-1 secure* if for any PPT adversary $\mathcal{A}$ the advantage in the security game $\mathsf{Exp}^{TS-UF-1}_{\mathcal{A}}(1^\lambda)$ defined in Fig. 9 is negligible.

We make use of a simple bulletin-board based public key infrastructure, depicted in Fig. 10, that allows each party to register a single public key.

B.4 Non-interactive Zero-Knowledge Proofs

We define non-interactive zero-knowledge proofs of knowledge (NIZKPoK) in the common reference string (CRS) model. Our target security will be straight-line simulation-extractability.

Definition 8. A *non-interactive proof system* ZK for NP-relation R is a tuple ($\mathsf{ZK.Setup}$, $\mathsf{Prove}, \mathsf{Vf}$) of PPT algorithms, where

- $\mathsf{ZK.Setup}(1^\lambda)$ outputs (crs, td), where crs is the CRS and td is a trapdoor.
- $\mathsf{Prove}(crs, stmt, wit)$ generates a proof π given $(stmt, wit) \in R$.
- $\mathsf{Vf}(crs, stmt, \pi)$ verifies a proof π for statement $stmt$ and outputs 0 or 1.

It is a *NIZKPoK* if additionally there are algorithms $\mathsf{ZK.Sim}$ and $\mathsf{ZK.Ext}$ with

- $\mathsf{ZK.Sim}(td, stmt)$ outputs a proof π.
- $\mathsf{ZK.Ext}(td, stmt, \pi)$ outputs a witness wit with $(stmt, wit) \in R$ or $\perp$.

Note that we sometimes explicitly add the relation R as input to the algorithms to make it clear for which relation the NIZPoK is.

Definition 9. Let ZK be a NIZKPoK for NP-relation R. We define the *straight-line simulation-extractability* game as follows: Let $\mathcal{A}$ be an adversary.

1. Setup $(crs, td) \leftarrow \mathsf{ZK.Setup}(1^\lambda)$.
2. Run $\mathcal{A}(1^\lambda, crs)$, where $\mathcal{A}$ is given oracle access to oracle $(stmt, \pi) \mapsto \mathsf{ZK.Ext}(td, stmt, \pi)$ and either
   - $(stmt, wit) \mapsto \mathsf{Prove}(crs, stmt, wit)$, or
   - $(stmt, wit) \mapsto \mathsf{ZK.Sim}(td, stmt)$,

   with the choice made uniformly at random. Let $\mathsf{L}_{\mathrm{Ext}}$ be the list of queries with outputs $(stmt, \pi, wit)$ to $\mathsf{ZK.Ext}$, and $\mathsf{L}_{\mathrm{Prove}}$ be the list of queries with outputs $(stmt, wit, \pi)$ to Prove or $\mathsf{ZK.Sim}$.
3. Eventually, $\mathcal{A}$ outputs a guess b, where $b = 0$ (resp. $b = 1$) indicates that it interacted with Prove (reps. $\mathsf{ZK.Sim}$).
4. $\mathcal{A}$ wins the game if:
   - $\mathcal{A}$ never queried $(stmt, \pi)$ if π was previously returned from a query to Prove (or $\mathsf{ZK.Sim}$) with statement $stmt$. (I.e., $\mathcal{A}$ may not trivially distinguish simulated proofs from real proofs.)
   - For any $(stmt, wit, \pi) \in Q_{Ext}$, we have $wit = \perp$, but $\mathsf{Vf}(crs, stmt, \pi) = 1$. (I.e., extraction failed for an accepting proof.)
   - $\mathcal{A}$ guessed b correctly.

We say that ZK is *straight-line simulation-extractable*, if for any PPT adversary, the probability that it wins the straight-line simulation-extractability game is negligible.

Functionality $\mathcal{F}_{\text{CRS}}$

Parameters: A distribution $\mathcal{D}$
- On input (`value`) from party P:
  - If $d = \bot$, set $d \leftarrow \mathcal{D}$
  - Generate public delayed output (d) to party P

(a) Functionality $\mathcal{F}_{\text{CRS}}$ for common reference strings

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

- On input (ssid, pid_R, msg) from party S:
  - Leak (ssid, pid_S, pid_R, $|msg|$) to the adversary
  - Generate private delayed output ($\mathsf{ssid}, \mathsf{pid}_S, msg$) to party R with pid pid_R

(b) Functionality $\mathcal{F}_{\text{SMT}}$ for authenticated communication

Functionality $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

State: List L of open channels
- On input (`send`, ssid, pid_R, msg) from party S:
  - Store (ssid, pid_S, pid_R) in L
  - Leak (ssid, `send`, pid_R, $|msg|$) to the adversary
  - Generate private delayed output (`send`, ssid, msg) to party R with pid pid_R
- On input (`respond`, ssid, msg) from party R with pid pid_R:
  - Find entry (ssid, pid_S, pid_R) in L with matching ssid and pid_R, otherwise ignore this input
  - Leak (ssid, `respond`, pid_S, $|msg|$) to the adversary
  - Generate private delayed output (`respond`, ssid, msg) to party S with pid pid_S

(c) Functionality $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ for one-sided authenticated communication

Figure 11: Functionalities for authenticated communication and common reference strings

B.5 Additional Ideal Functionalities

We use the standard CRS functionality $\mathcal{F}_{\text{CRS}}$, see Fig. 11a for details.

For communication between user and operator, we want the user to remain anonymous while authenticating the operator, except for user registration, for which we require mutual authentication. For the latter case (as well as for all communication between operators), we make use of the standard UC functionality for a secure channel. For the former case, we use a modified version of a secure channel that only authenticates one of the parties involved: Any party S can initiate a secure connection to some other party R by sending (`send`, ssid, pid_R, msg) to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, which guarantees that the party with id pid_R will be the one receiving this message. Party R only receives the message, but no information about the sender. However, it can respond to the message by sending (`respond`, ssid, msg') to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, using the same subsession id ssid under which it received the message. $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ will then privately output msg' to party S together with the information that it came from party R. See Fig. 11 for the formal definitions of $\mathcal{F}_{\text{SMT}}$ and $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$.

C Another Application: Charging and Discharging Electric Vehicle Batteries in V2G Scenario

In the future smart grid, vehicle-to-grid (V2G) systems can use the batteries of plugged-in electric vehicles (EVs) as distributed energy storage. In this scenario, users are EV owners who want to charge their EVs at charging stations owned by different electricity providers. Operators are different electricity providers that own the EV charging stations. POBA can be used here as a privacy-preserving payment system for charging/discharging operations that handles the billing process between users and electricity providers and between the

electricity providers themselves.

Registration When registering for the system, users select an electricity provider to create an account with. That electricity provider becomes the user's home operator.

Charging/Discharging EVs When an EV is charged (or discharged), a logbook entry is created with the electricity provider the charging station belongs to. Several pieces of information about this (dis-)charging process are written to the logbook, so the entry might look like this: $msg := (\texttt{charge/discharge}, station_id, date, start_time, end_time, kWh, price)$.

Invoice Generation & Clearing between Providers As in the check-in/check-out-based payment system for multimodal public transportation described earlier, the Analytics task can be used to generate invoices for individual users and to handle the clearing between the electricity providers.

Possibility of Complaint If a user discovers any discrepancies in their invoice, they can use the ProveEntry task to file a complaint with their home operator and clarify the issue. For example, if a discharge transaction, which would benefit the user monetarily, is missing from the invoice, the user can prove to their home operator that this transaction actually took place and should be included in the invoice. This discrepancy may occur if a logbook entry was created, but for some reason never inserted into the actual logbook.

Power Consumption Analysis The Analytics task can also be used to analyze past power consumption across the grid and to forecast future power consumption in specific regions or time periods. Such forecasts are important for ensuring the continued stability of the power grid.

D Implementation Details of the Zero-Knowledge Proofs

To achieve efficient straight-line simulation-extractability for our statements, we augment the Groth–Sahai (GS) non-interactive proof system for pairing product equations [GS08; EG14] with the transformation described in [Gro06]. The GS proof system is a dual-mode system in the common reference string model, which can be setup either in zero-knowledge mode with a simulation trapdoor or in soundness mode with an extraction trapdoor. We achieve straight-line extraction by using the extraction trapdoor, achieve zero-knowledge simulation by standard transformation of generating an OR-proof of either the intended statement or knowledge of a new simulation trapdoor (a signature under a verification key that will also be part of the crs), and finally achieve simulation-extractability (and thus also non-malleability) by the transformation described in [Gro06], i.e., adding a one-time-signature scheme, generating a fresh keypair when generating a new proof, adding the verification key to the statement and then signing the complete statement and proof.

The concrete pairing product equations (PPE) proved in the Groth–Sahai proof system in our implementation are provided in Figs. 12, 13 and 15. The concrete PPE of the zero-knowledge branch are shown in Fig. 14. We use the notation of [EG14], where variables of the statement have type `G1_pub` or `G2_pub` for elements of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, and variables of the statement can be of type either `Gi_com` or `Gi_enc`. Additionally, there are variables `g1` and `g2` of type `G1_base` and `G2_base` representing the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$.

PPE for $\mathcal{R}_{\mathtt{UR}}$

```
    // Statement
1 : G1_pub pk
    // Witness
2 : G2_enc sk
    // Equations
3 : e(pk,g2) + e(g1,sk) = 0
```

Figure 12: The concrete equations to implement $\mathcal{R}_{\mathtt{UR}}$.

PPE for $\mathcal{R}_{\mathtt{PE}}$

```
    // Statement
1 : G1_pub pk    // pk_U
2 : G1_pub addr  // addr
3 : G1_pub ek    // ek
4 : G1_pub ct1   // ciphertext of addr
5 : G1_pub ct2
    // Witness
6 : G2_enc sk    // sk_U
7 : G2_com rinv  // r^-1
    // Equations
8 : e(ct1,g2) + e(g1,rinv) = 0    // encryption of addr
9 : e(ct2,g2) + e(ek,rinv) - e(addr,g2) = 0
10 : e(pk,g2) + e(g1,sk) = 0    // knowledge of sk_U
```

Figure 13: The concrete equations to implement $\mathcal{R}_{\mathtt{PE}}$.

PPE for ZK simulation branch

```
    // Statement
1 : G1_pub zkPK    // simulation pk from crs
2 : G2_pub zkHash  // hash of statement
    // Witness
3 : G2_enc zkSig   // BLS signature on zkHash
    // Equations
4 : e(zkPK,g2) + e(g1,zkHash) - e(g1,zkSig) = 0
```

Figure 14: The concrete equations of the simulation branch.

PPE for $\mathcal{R}_{\mathsf{CE}}$

// Statement
1 : G1_pub ct1 // ciphertext of *addr*
2 : G1_pub ct2
3 : G1_pub ek // ek
4 : G2_pub ppA11 // crs of TSIG
5 : G2_pub ppA21
6 : G2_pub ppUA11
7 : G2_pub ppUA21
8 : G2_pub ppVA11
9 : G2_pub ppVA21
10 : G1_pub ppB11
11 : G1_pub ppB21
12 : G1_pub ppBtU11
13 : G1_pub ppBtU12
14 : G1_pub ppBtV11
15 : G1_pub ppBtV12
16 : G2_pub vk11
17 : G2_pub vk21
18 : G2_pub vk31
19 : G2_pub h // hash of *entry*
// Witness
20 : G1_com pk
21 : G1_com addr
22 : G2_enc sk
23 : G2_com rinv // r^{-1}
24 : G1_com fsig11 // TSIG signature
25 : G1_com fsig12
26 : G1_com ssig11
27 : G1_com ssig12
28 : G1_com tsig11
29 : G1_com tsig12
30 : G2_enc fosig
// Equations
31 : e(ct1,g2) + e(g1,rinv) = 0 // encryption of *addr*
32 : e(ct2,g2) + e(ek,rinv) - e(addr,g2) = 0
33 : e(pk,g2) + e(g1,sk) = 0 // knowledge of sk_U
34 : e(g1,vk11) + e(pk,vk21) + e(addr,vk31) = 0 // valid TSIG signature
35 : e(fsig11,ppA11) + e(fsig12,ppA21) = 0
36 : e(ssig11,ppUA11) + e(ssig12,ppUA21) = 0
37 : e(tsig11, ppVA11) + e(tsig12,ppVA21) = 0
38 : e(tsig11, ppVA11) + e(tsig12,ppVA21) + e(ssig11,ppUA11) + e(ssig12,ppUA21) + e(g1,vk11)
+ e(addr,vk21) + e(pk,vk31) - e(fsig11,ppA11) - e(fsig12,ppA21) = 0
39 : e(ssig11, fosig) - e(tsig11,g2) = 0
40 : e(ssig12, fosig) - e(tsig12,g2) = 0

Figure 15: The concrete equations to implement $\mathcal{R}_{\mathsf{CE}}$.

E Security Proof

We now provide the formal proof of Theorem 1, i.e., of the UC-security of our protocol Π_{POBA}. First, we re-state the theorem formally:

Theorem 2 (formal)**.** *If* ZK *is a straight-line simulation-extractable non-interactive zero-knowledge proof system,* SIG *is an EUF-CMA-secure signature scheme,* TSIG *is a TS-UF-1-secure threshold signature scheme, and* TPKE *is an IND-CPA-secure threshold PKE, then* Π_{POBA} *UC-realizes* $\mathcal{F}_{\text{POBA}}$ *in the* $\{\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}, \mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{SMT}}^{\text{os-auth}}, \mathcal{F}_{MPC(\cdot)}, \mathcal{F}_{MPC(\cdot)}^{ext}\}$*-hybrid model wrt. adversaries* $\mathcal{A}$ *that may statically corrupt any number of users maliciously and up to* $t \leq \frac{n-1}{2}$ *operators passively.*

To prove Theorem 2, we first provide a simulator $\mathcal{S}$ and then show indistinguishability to the real protocol through a series of games.

Simulator $\mathcal{S}$

State of the Simulator

- Setup
  - Common reference string crs_{ZK}
  - Signature parameters crs_{TSIG}
  - Trapdoor td_{ZK} for extracting and simulating zero-knowledge proofs
  - Consortium encryption key ek and decryption key shares $\{\mathsf{vk}_i, \mathsf{dk}_i\}_{i\in[n]}$
  - Consortium verification key vk and signing key shares $\{\mathsf{vk}_i, sk_i\}_{i\in[n]}$
- Global state
  - A list $\mathsf{L}_{\mathsf{Free}}^{\mathcal{S}}$ of free logbook addresses
  - The logbook index $\mathsf{L}_{\mathsf{Keys}}^{\mathcal{S}}$ that maps user public keys to logbook addresses
  - The memory M_j for each period j
- For each honest operator:
  - Its signature keypair $(\mathsf{sk}_O, \mathsf{vk}_O)$
  - A list $\mathsf{L}_{\mathsf{Pending}}$ of pending entries
  - A value id_{last} for indexing pending entries
- For each semi-honest operator:
  - Its signature keypair $(\mathsf{sk}_O, \mathsf{vk}_O)$
  - A share of the consortium signing key $(\mathsf{vk}_i, \mathsf{sk}_i)$
  - A share of the consortium decryption key $(\mathsf{vk}_i, \mathsf{dk}_i)$
  - A list $\mathsf{L}_{\mathsf{Pending}}$ of pending entries
  - A value id_{last} for indexing pending entries
  - A sharing of the oblivious memory $\mathsf{Share}M_j$ for each period j
  - A list $\mathsf{L}_{\mathsf{Free}}$ of free logbook addresses
  - The logbook index $\mathsf{L}_{\mathsf{Keys}}$ that maps user public keys to logbook addresses
- For each honest user:
  - Verification key of home operator vk_{HO}^{U}
  - Credentials $creds_U := (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  - List $\mathsf{L}_{\mathsf{Receipt}}^{\mathsf{U}}$ of receipts for logbook entries

Simulation of $\mathcal{F}_{\text{CRS}}$:
A corrupted party can issue calls to $\mathcal{F}_{\text{CRS}}$ whenever it wants. Since simulation of $\mathcal{F}_{\text{CRS}}$ is straightforward, we specify here once how all calls from corrupted U and semi-honest O to $\mathcal{F}_{\text{CRS}}$ are handled and omit calls to $\mathcal{F}_{\text{CRS}}$ in the following description of the simulator.

- Upon receiving $(\texttt{value})$ from a party P to $\mathcal{F}_{\text{CRS}}$:
  1. If this is the first time such a message is received:
     (a) Generate $(crs_{\text{ZK}}, td_{\text{ZK}}) \leftarrow \mathsf{ZK.Setup}(1^\lambda)$ and store them
     (b) Generate $crs_{\mathsf{TSIG}} \leftarrow \mathsf{TSIG.Setup}(1^\lambda)$ and store it
  2. Report message $(crs_{\text{ZK}}, crs_{\mathsf{TSIG}})$ from $\mathcal{F}_{\text{CRS}}$ to P

Simulation of $\mathcal{F}_{\text{SMT}}$ and $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
The simulator simply follows the description of these functionalities.

Simulation of $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$:

- Upon receiving $(\mathsf{Keygen}, 1^\lambda)$ from a semi-honest operator O:
  1. If $\mathsf{ek} = \bot$
     (a) Run $(\mathsf{ek}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{dk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Keygen}(1^\lambda, n, t)$
     (b) Store ek and $\{\mathsf{vk}_i, \mathsf{dk}_i\}_{i\in[n]}$
  2. Retrieve ek and $(\mathsf{vk}_i, \mathsf{dk}_i) \in \{\mathsf{vk}_i, \mathsf{dk}_i\}_{i\in[n]}$ for O
  3. After receiving input from all operators, output $(\mathsf{ek}, \mathsf{vk}_i, \mathsf{dk}_i)$ to O

Simulation of $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$:

- Upon receiving $(\mathsf{Keygen}, crs_{\mathsf{TSIG}}{}')$ from a semi-honest operator O:
  1. If $\mathsf{vk} = \bot$
     (a) Run $(\mathsf{vk}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{TSIG.Keygen}(crs_{\mathsf{TSIG}}, n, t)$
     (b) Store vk and $\{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]}$
  2. Retrieve vk and $(\mathsf{vk}_i, \mathsf{sk}_i) \in \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]}$ for O
  3. After receiving input from all operators, output $(\mathsf{vk}, \mathsf{vk}_i, \mathsf{dk}_i)$

Init (honest O):

- Upon receiving $(\texttt{init}, O)$ from $\mathcal{F}_{\text{POBA}}$:
  1. Record input from O to $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$
  2. Wait for the simulations of $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$ to deliver output to O
  3. If $M_0 = \bot$, initialize the memory for period 0
  4. Generate and store $(\mathsf{sk}_O, \mathsf{vk}_O) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output

Init (semi-honest O):

- The simulator executes the protocol honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to other parties. If the adversary did not make the protocol abort, allow $\mathcal{F}_{\text{POBA}}$ to deliver output.

UReg (honest U, honest HO):

- Upon receiving $((\mathsf{ureg}, \mathsf{ssid}, \mathsf{pid}_U), HO)$ from $\mathcal{F}_{\text{POBA}}$:
  1. Report output $(\mathsf{ssid}\|1,\ \mathsf{pid}_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ from $\mathcal{F}_{\text{SMT}}$ to U
- When the adversary allows delivery of that output:
  1. Wait until receiving $((\mathsf{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO}), U)$ from $\mathcal{F}_{\text{POBA}}$ (if it has not yet been received)
  2. Store vk_O as vk_{HO}^U
  3. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  4. Generate $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt := \mathsf{pk}_U, td_{\text{ZK}})$
  5. Report output $(\mathsf{ssid}\|2,\ \mathsf{pid}_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to O
- When the adversary allows delivery of that output:
  1. Get next free logbook address $addr \leftarrow \mathsf{L_{Free}.pop}()$
  2. Add $(\mathsf{pk}_U, \bot, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}^{\mathcal{S}}$
  3. Report a message $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ from HO to each other operator O'
- For each semi-honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\mathsf{ureg}, \mathsf{ssid}), O')$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. Execute the protocol honestly for O', sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- For each honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\mathsf{ureg}, \mathsf{ssid}), O')$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. Set $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$ where i is the index of O'
  3. Report a message of length $|(\mathsf{ssid}, \sigma_i)|$ from O' to HO
- After the adversary allowed delivery of all messages from O' to HO:
  1. Set T to be the first $t+1$ partial signatures σ_i
  2. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\mathsf{TSIG}}, T)$
  3. Store $(\mathsf{pk}_U, \bot)$ in $\mathsf{L}_{\mathsf{Accounts}}^{\mathsf{HO}}$ and $creds_U := (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  4. Report output $(\mathsf{ssid}\|3,\ \mathsf{pid}_{HO},\ (addr,\ \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U
  5. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to HO
- After the adversary allowed delivery of the output of $\mathcal{F}_{\text{SMT}}$ to U:
  1. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to U

UReg (honest U, semi-honest HO):

- The simulator executes the protocol honestly for HO, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to U.
- When the adversary allows delivery of the output $(\mathsf{ssid}\|1,\ \mathsf{pid}_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ from $\mathcal{F}_{\text{SMT}}$ to U:
  1. Wait until receiving $((\mathsf{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO}), U)$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  3. Store vk_O as vk_{HO}^U
  4. $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}$
  5. $stmt := (\mathsf{pk}_U), td_{\text{ZK}})$
  6. Report output $(\mathsf{ssid}\|2,\ \mathsf{pid}_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO
- For each O', when the adversary allows delivery of the message $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ from HO:
  1. Wait until receiving $((\mathsf{ureg}, \mathsf{ssid}), O')$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. For the first O':
     (a) Remove $addr$ from $\mathsf{L_{Free}}$
     (b) Add $(\mathsf{pk}_U, \bot, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}^{\mathcal{S}}$
  3. Execute the protocol honestly for O'
  4. If O' is semi-honest, send the party's state to the adversary after each activation and wait for `continue` from the adversary before sending out any messages.
- When the adversary allows delivery of the output $(\mathsf{ssid}\|3,\ \mathsf{pid}_{HO},\ (addr,\ \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U:
  1. If $\mathsf{SIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  2. Store $creds_U := (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to U

UReg (corrupted U):
- The simulator executes the protocol honestly, sending each semi-honest party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Additionally, when receiving $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U,})$ from U, if the proof verifies successfully:
  1. Extract $wit := (\mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathtt{UR}}, stmt, \pi_{\mathsf{sk}_U}, td_{\mathrm{ZK}})$
  2. Store $(\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ U)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Send $(\mathtt{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO})$ to $\mathcal{F}_{\mathrm{POBA}}$ on behalf of U
- When simulation of HO sends the message $(\mathsf{ssid}\|3, \mathsf{pid}_U, (addr, \sigma))$ to $\mathcal{F}_{\mathrm{SMT}}$, allow $\mathcal{F}_{\mathrm{POBA}}$ to deliver output to HO

CreateEntry (honest U, honest O):
- Simply report messages of the appropriate length.

CreateEntry (honest U, semi-honest O):
- The simulator runs the protocol honestly for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Upon receiving $((\mathtt{createentry},\ \mathsf{ssid}),\ \mathtt{user})$ and $((\mathtt{createentry},\ \mathsf{ssid},\ period,\ entry),\ O)$ from $\mathcal{F}_{\mathrm{POBA}}$:
  1. Set $addr := 0$
  2. Generate $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
  3. Set $h := H(period, ct_U, entry)$
  4. Set $stmt := (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  5. Generate $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathtt{CE}}, stmt, td_{\mathrm{ZK}})$
  6. Report output $(\mathsf{send},\ \mathsf{ssid}\|1,\ (ct_U,\ \pi))$ from $\mathcal{F}^{\mathrm{os-auth}}_{\mathrm{SMT}}$ to O
- When the adversary allows delivery of output $(\mathtt{respond},\ \mathsf{ssid}\|1,\ (\sigma_{entry},\ \mathsf{vk}_O))$ from $\mathcal{F}^{\mathrm{os-auth}}_{\mathrm{SMT}}$:
  1. Store $(entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ in $\mathsf{L}^{\mathsf{U}}_{\mathsf{Receipt}}$
  2. Allow $\mathcal{F}_{\mathrm{POBA}}$ to deliver output

CreateEntry (corrupted U, semi-honest O):
- The simulator executes the protocol honestly for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Additionally, the simulator does the following:
  - When the adversary allows delivery of output $(\mathsf{send},\ \mathsf{ssid}\|1,\ (entry_U,\ ct_U,\ \pi))$ from $\mathcal{F}^{\mathrm{os-auth}}_{\mathrm{SMT}}$, if the honest protocol did not abort as a result of this message:
    1. Get $wit := (\mathsf{pk}_U, \mathsf{sk}_U, addr, r, \sigma) \leftarrow \mathsf{ZK.Ext}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathtt{CE}}, stmt, \pi, td_{\mathrm{ZK}})$
    2. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no such entry exists
    3. If U^* is honest, abort
    4. Send $(\mathtt{createentry}, \mathsf{ssid}, period, entry_U)$ to $\mathcal{F}_{\mathrm{POBA}}$ on behalf of user U^*

CreateEntry (corrupted U, honest O):
- After receiving $((\mathtt{createentry}, \mathsf{ssid}), O)$ from $\mathcal{F}_{\mathrm{POBA}}$ and the adversary allowing delivery of the output $(\mathsf{send}, \mathsf{ssid}\|1, (entry_U, ct_U, \pi))$ from $\mathcal{F}_{\mathrm{SMT}}$ to O:
  1. $h := H(period, ct_U, entry_U)$, $stmt := (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  2. If $\mathsf{Vf}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathtt{CE}}, stmt, \pi) \neq 1$, ignore this message
  3. Otherwise, get $wit := (\mathsf{pk}_U, \mathsf{sk}_U, addr, r, \sigma) \leftarrow \mathsf{ZK.Ext}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathtt{CE}}, stmt, \pi, td_{\mathrm{ZK}})$, abort if this fails
  4. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no such entry exists
  5. If U^* is honest, abort
  6. Send $(\mathtt{createentry}, \mathsf{ssid}, period, entry_U, \mathsf{pid}_O)$ to $\mathcal{F}_{\mathrm{POBA}}$ on behalf of user U^*

InsertEntry:
- Upon receiving $((\mathsf{insert}, \mathsf{ssid}, period, id), O)$ from $\mathcal{F}_{\mathrm{POBA}}$ (i.e., O is semi-honest):
  1. Honestly execute the protocol for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Upon receiving $((\mathsf{insert}, \mathsf{ssid}), O)$ and $((\mathsf{insert}, \mathsf{ssid}, period, id), O)$ from $\mathcal{F}_{\mathrm{POBA}}$ (i.e., O is honest):
  1. Find $(id, period, ct_U, entry, \pi) \in \mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$ and ignore this message if no entry with $(id, period)$ exists
  2. Send $(\mathsf{ssid}, ct_U, entry, \pi)$ to all O'
  3. Honestly run the checks from the real protocol
  4. Treat O as having provided input to $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$
- Upon receiving $((\mathsf{insert}, \mathsf{ssid}), O')$ from $\mathcal{F}_{\mathrm{POBA}}$ for honest O':
  1. Wait until the adversary allowed delivery of the message $(ct_U, entry, \pi)$ from O
  2. Honestly run the checks from the real protocol
  3. Treat O' as having provided input to $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$
- Upon receiving $((\mathsf{insert}, \mathsf{ssid}), O')$ from $\mathcal{F}_{\mathrm{POBA}}$ for semi-honest O':
  1. Honestly execute the protocol for O', sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Simulate $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$ as follows:
  - Upon receiving $(\mathrm{Run}, \mathsf{Share}M, \mathsf{Share}x)$ from all semi-honest operators and having treated all honest operators as having provided input to $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$:
    1. Chose a random $addr^* \in \mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
    2. Ignore $\mathsf{Share}M$ and $\mathsf{Share}x$, use M and partial decryptions of $\mathsf{TPKE.Encrypt}(\mathsf{ek}, addr^*)$ instead
    3. Follow the description of insert honestly
    4. When delivering output to semi-honest operators, instead of $\mathsf{Share}M$ use $\mathsf{Share}0$, where 0 is a memory the same size as M but consisting only of 0-entries

- When the adversary allows delivery of output from $\mathcal{F}_{\text{MPC(insert)}}$ to an operator O, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

ProveEntry (honest U, honest O):
- Simply report a message of the appropriate length from U to O and allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O after the adversary allowed that message to be delivered

ProveEntry (honest U, semi-honest O):
- The simulator executes the protocol for O honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- When $\mathcal{F}_{\text{POBA}}$ wants to deliver output (ssid, `proveentry`, pid_U, *entry*, *period*) to O:
  1. Retrieve ($entry$, ct_U, r, σ_{entry}, vk_O) from $\mathsf{L}^{\mathsf{U}}_{\mathsf{Receipt}}$ and (pk_U, $\perp$, $addr$, U) from $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  2. Set $stmt := (\mathsf{pk}_U, addr, ct_U)$ and $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, td_{\text{ZK}})$
  3. Report output ($\mathsf{ssid}\|1$, pid_U, ($period$, $entry$, pk_U, $addr$, σ, ct_U, σ_{entry}, vk_O, π)) to O from $\mathcal{F}_{\text{SMT}}$
- When the adversary allows O to generate output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

ProveEntry (corrupted U)
- When the adversary allows delivery of output ($\mathsf{ssid}\|1$, pid_U, ($period$, $entry$, pk_U, $addr$, σ, ct_U, σ_{entry}, vk_O, π)) from $\mathcal{F}_{\text{SMT}}$:
  1. Follow the steps of the honest protocol, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages if O is semi-honest
  2. Before asking the adversary for permission to deliver output:
     (a) Set $stmt := (\mathsf{pk}_U, addr, ct_U)$
     (b) Get $wit := (r, \mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, \pi, td_{\text{ZK}})$, abort if this fails
     (c) Find the entry $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no matching entry exists
     (d) If U^* is honest, abort
     (e) Send message (`proveentry`, ssid, $entry$, $period$, O) to $\mathcal{F}_{\text{POBA}}$ on behalf of U^*
  3. When the adversary allows O to deliver output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

Analytics:
- If all operators are honest, simply report messages for revealing output shares of the appropriate lengths.
- Otherwise (at least one operator is semi-honest):
- Upon receiving ((`analytics`, ssid, $func$, $\{period_i\}$, $\{U_i\}$, $aux_{O'}$), O') from $\mathcal{F}_{\text{POBA}}$ (i.e., O' is semi-honest):
  1. Simulate O' honestly, sending the state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
  2. When the adversary allows generation of output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O'
- Upon receiving ((`analytics`, ssid), O) from $\mathcal{F}_{\text{POBA}}$ (i.e., O is honest):
  1. Do nothing
- Wait until receiving (`analytics`, ssid, $sequence$) and outputs (ssid, $\mathtt{result}_{O'}$) for each semi-honest operator O', then simulate $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ as follows:
  1. For each honest O^*, set $\mathtt{result}_{O^*} := 0$
  2. Set $\mathtt{result} := \bigcup \mathtt{result}_O$ (this is 0 for honest operators, and the output of $\mathcal{F}_{\text{POBA}}$ for semi-honest operators)
  3. Output shares $\mathsf{Share}(\mathsf{stop}, \mathtt{result})$, $\{\mathsf{Share}M_i\}$, $round$, $sequence$ to all parties
- When the simulation of $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ has delivered output to honest O:
  1. For each semi-honest O', reveal $\mathsf{Share}\mathtt{result}_{O'}$ to O'
  2. For each honest O^*, report a message of the correct length from O to O^*
- When the adversary allowed delivery of all share reveals towards an honest operator O:
  1. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

We proceed in a series of games, starting with the real experiment, which we gradually transform into the ideal experiment. In these games, $\mathcal{F}^{i+1}$ behaves the same as $\mathcal{F}^i$ except for the stated changes, and $\mathcal{S}^{i+1}$ behaves the same as $\mathcal{S}^i$ except for the stated changes.

Game 0. Hybrid 0 is the real experiment. That is,

$$H_0 := \mathsf{Exp}^{\Pi_{\text{POBA}}}_{\mathcal{F}_{\text{CRS}}, \mathcal{F}^{\mathsf{TPKE}}_{\text{Keygen}}, \mathcal{F}^{\mathsf{TSIG}}_{\text{Keygen}}, \mathcal{F}_{\text{MPC}(\cdot)}, \mathcal{F}_{\text{MPC}}, \mathcal{S}^0, \mathsf{Enc}}$$

with $\mathcal{S}^0$ being the dummy adversary and all parties executing the real protocol.

Game 1. In hybrid 1, the simulator $\mathcal{S}^1$ assumes control of $\mathcal{F}_{\text{CRS}}$, $\mathcal{F}^{\mathsf{TPKE}}_{\text{Keygen}}$, $\mathcal{F}^{\mathsf{TSIG}}_{\text{Keygen}}$, and $\mathcal{F}_{\text{MPC}(\cdot)}$ but still executes them honestly. We also introduce $\mathcal{F}^1$ (see Fig. 16) and all honest parties are replaced by dummy parties that forward their inputs to $\mathcal{F}^1$ and when receiving output from $\mathcal{F}^1$ forward it to the environment. $\mathcal{F}^1$, upon receiving any message from a dummy party, forwards it to the simulator $\mathcal{S}^1$ and asks it for output. $\mathcal{S}^1$ simply executes the real protocol for all honest parties using the inputs received from $\mathcal{F}^1$ and instructs $\mathcal{F}^1$ to deliver the resulting outputs.

Proof Sketch: Game 0 ≈ Game 1.
$\mathcal{S}^1$ executes the same code as the real parties on the same inputs. Thus, Game 0 and

$\mathcal{F}^1$

- Upon receiving some input (msg) from party P:
  1. Send $(\texttt{input}, P, \text{msg})$ to the simulator
- Upon receiving $(\texttt{output}, P, \text{msg})$ from the simulator:
  1. Send (msg) to party P

Figure 16: $\mathcal{F}^1$

Handling of (`init`) in $\mathcal{F}^2$

- Upon receiving input (`init`) from operator O:
  1. Send $(\texttt{input}, O, (\texttt{init}))$ to the simulator.
- Upon receiving $(\texttt{output}, O, (\texttt{init}, \texttt{ok}))$ from the simulator,
  1. Send $(\texttt{init}, \texttt{ok})$ to O.

Handling of (`init`) in $\mathcal{F}^3$

- Upon receiving input (`init`) from operator O:
  1. Send $((\texttt{init}), O)$ to the simulator
  2. After receiving input from all $O \in \mathcal{O}$, set $\mathit{initialized} := 1$
  3. Deliver private delayed output $(\texttt{init}, \texttt{ok})$ to O
- Upon receiving $(\texttt{output}, O, (\texttt{init}, \texttt{ok}))$ from the simulator:
  1. Ignore this message.

Figure 17: Changes to $\mathcal{F}_{\text{POBA}}$

Game 1 are identical from the environment's view.

Game 2. In this game, $\mathcal{S}^2$ generates the common reference string for the zero-knowledge proof system with an extraction and simulation trapdoor td_{ZK} as $(crs_{\text{ZK}}, td_{\text{ZK}}) \leftarrow \mathsf{ZK.Setup}(1^\lambda)$.

Proof Sketch: Game 1 ≈ Game 2.
Indistinguishability follows from the simulation-extractability of ZK.

Game 3. In this game, $\mathcal{F}^3$ handles (`init`) inputs the same way as $\mathcal{F}_{\text{POBA}}$ and $\mathcal{S}^3$ handles them the same way as $\mathcal{S}$. The changes to $\mathcal{F}^3$ are detailed in Fig. 17 and to $\mathcal{S}^3$ in Fig. 18.

Proof Sketch: Game 2 ≈ Game 3.
Simulation for semi-honest operators is identical between these games (only the format of the message from $\mathcal{F}^3$ changed). The only visible effect of honest operators is $\mathcal{F}^{\mathsf{TPKE}}_{\text{Keygen}}$ and $\mathcal{F}^{\mathsf{TSIG}}_{\text{Keygen}}$ delivering output to semi-honest operators only after all honest operators have called it.

Game 4. In this game, $\mathcal{F}^4$ handles messages $(\texttt{ureg}, \ldots)$ the way $\mathcal{F}_{\text{POBA}}$ does if both the user and the home operator involved are honest. The changes to $\mathcal{S}^4$ are detailed in Fig. 19.

Proof Sketch: Game 3 ≈ Game 4.
The only change visible to the environment between these two games is how π_{sk_U} is generated, the lengths of messages between honest parties and the other parts of the messages to semi-honest parties are identical in both games. Thus, the probability of the environment to distinguish these games is bounded by the probability to break zero-knowledge of ZK, and hence these games are indistinguishable.

Game 5. In this game, $\mathcal{F}^5$ handles messages $(\texttt{ureg}, \ldots)$ the way $\mathcal{F}_{\text{POBA}}$ does if the user involved is honest but the home operator is semi-honest. The changes to $\mathcal{S}^5$ are detailed in Fig. 20.

Proof Sketch: Game 4 ≈ Game 5.
Again, the only change visible to the environment is the generation of π_{sk_U}, all other messages are the same in both games. Thus, the probability of the environment to distinguish these games is again bounded by the probability to break zero-knowledge of ZK, and hence these games are indistinguishable.

Game 6. In this game, $\mathcal{F}^6$ handles messages $(\texttt{ureg}, \ldots)$ the way $\mathcal{F}_{\text{POBA}}$ does if the user involved is corrupted. The changes to $\mathcal{S}^6$ are detailed in Fig. 21.

Handling of (init) in $\mathcal{S}^2$

Upon receiving $(\texttt{input}, O, (\texttt{init}))$ from $\mathcal{F}^2$:
1. Run the protocol honestly for O
2. When the protocol would generate output $(\texttt{init}, \texttt{ok})$ for O, send $(\texttt{output}, O, (\texttt{init}, \texttt{ok}))$ to $\mathcal{F}^2$

Handling of (init) in $\mathcal{S}^3$

Upon receiving $((\texttt{init}), O)$ from $\mathcal{F}^3$:
- If O is semi-honest:
  1. Run the protocol honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to other parties. If the adversary did not make the protocol abort, allow $\mathcal{F}^3$ to deliver output.
- If O is honest:
  1. Record input from O to $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$
  2. Wait for the simulations of $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\mathsf{TSIG}}$ to deliver output to O
  3. If $M_0 = \bot$, initialize the memory for period 0
  4. Generate and store $(\mathsf{sk}_O, \mathsf{vk}_O) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. Allow $\mathcal{F}^3$ to deliver output

Figure 18: Changes to $\mathcal{S}$

Proof Sketch: Game 5 ≈ Game 6.
All messages to the user and semi-honest operators are generated honestly, thus these games only differ if $\mathcal{S}^6$ aborts because extraction of the witness failed. The probability of this is bound by the probability of breaking the simulation-extractability of ZK and hence these games are indistinguishable.

Game 7. In this game, when simulating $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$, $\mathcal{S}^7$ ignores the input $\mathsf{Share}M_{period}$ and instead uses M_{period} from it's global state, see Fig. 22.

Proof Sketch: Game 6 ≈ Game 7.
These games are identical from the environment's view since all M_i start empty and semi-honest operators always input the correct share, which corresponds to either the empty initial memory or the memory last output by $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$.

Game 8. In this game, when simulating $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$, $\mathcal{S}^8$ outputs shares of an all-0 memory instead of $\mathsf{Share}M$, see Fig. 23.

Proof Sketch: Game 7 ≈ Game 8.
These games are identical from the environment's view since at most t operators are corrupted, and t out of the n shares reveal no information about M.

Game 9. In this game, $\mathcal{F}^9$ on input $(\texttt{createentry}, \mathsf{ssid}, period, entry)$ from U ignores the message if $\mathsf{pid}_U \notin \cup_{O \in \mathcal{O}} \mathsf{L}_{\mathsf{users}}^{O}$. Otherwise, it still sends the input to the simulator and delivers output as instructed by the simulator. Furthermore, $\mathcal{F}^9$ fills $\mathsf{L}_{\mathsf{Pending}_O}$ the same way as $\mathcal{F}_{\text{POBA}}$. The concrete behavior of $\mathcal{F}^9$ is detailed in Fig. 24.

Proof Sketch: Game 8 ≈ Game 9.
Storing internal data in $\mathcal{F}^9$ has no effect on the view of the environment. As such, the only observable change is $\mathcal{F}^9$ ignoring an input $(\texttt{createentry}, \ldots)$. $\mathcal{F}^9$ only receives those inputs for honest users, and it only ignores the message if it did not receive a message $(\texttt{ureg}, \ldots)$ before. Thus, whenever $\mathcal{F}^9$ ignores the message, the simulation of U by $\mathcal{S}^8$ would also abort since no values $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$ would have been stored for that user. Thus, from the environment's view, these games are identical.

Game 10. In this game, when receiving a message $(\mathsf{ssid}, entry_U, ct_U, \pi)$ from a corrupted user, the simulator extracts the proper inputs and calls $\mathcal{F}^{10}$ in the name of the correct user, see Fig. 25.

Proof Sketch: Game 9 ≈ Game 10.
The only change to the environment's view happens when the simulator aborts. This can happen because 1. extraction of the witness failed, 2. no matching entry in $\mathsf{L}_{\mathsf{Keys}}^{\mathcal{S}}$ exists, or 3. U^* is honest. The probability for (1) is bounded by the probability to break

UReg in $\mathcal{S}^3$, honest U, honest HO

- Upon receiving $(\texttt{input}, HO, (\texttt{ureg}, \mathsf{pid}_U, \mathsf{ssid}))$ from $\mathcal{F}^3$, perform as HO:
  1. Send message $(\mathsf{ssid}\|1,\ \mathsf{pid}_U,\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}$
- Upon receiving $(\texttt{input},\ U,\ (\texttt{ureg},\ \mathsf{ssid},\ \mathsf{pid}_{HO}))$ from $\mathcal{F}^3$, perform as U:
  1. Wait until $(\mathsf{ssid}\|1,\ \mathsf{pid}_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ was received from $\mathcal{F}_{\text{SMT}}$
  2. Store ek, vk and vk_O as vk_{HO}
  3. Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$
  4. $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. $stmt := (\mathsf{pk}_U)$, $wit := (\mathsf{sk}_U)$
  6. $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt, wit)$
  7. Send message $(\mathsf{ssid}\|2,\ \mathsf{pid}_{HO},\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ to $\mathcal{F}_{\text{SMT}}$
- After delivering output $(\mathsf{ssid}\|2,\ \mathsf{pid}_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO, perform as HO:
  1. $stmt := (\mathsf{pk}_U)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ abort
  3. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  4. Add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. Send message $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ to each O'
- Upon receiving $(\texttt{input}, O', (\texttt{ureg}, \mathsf{ssid}))$ from $\mathcal{F}^3$, perform as O' (for each O'):
  1. Wait until $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ was sent from HO to O'
  2. $stmt := (\mathsf{pk}_U)$
  3. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\mathsf{Free}}$ abort
  4. Remove $addr$ from $\mathsf{L}_{\mathsf{Free}}$, add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  6. Send message $(\mathsf{ssid}, \sigma_i)$ to HO
- After all messages $(\mathsf{ssid}, \sigma_i)$ from honest and semi-honest O' have been received by HO, perform as HO:
  1. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  2. Set T to be the first valid $t{+}1$ partial signatures σ_i
  3. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\mathsf{TSIG}}, T)$
  4. Store (pk_U) in $\mathsf{L}^{\mathsf{HO}}_{\mathsf{Accounts}}$
  5. Send message $(\mathsf{ssid}\|3,\ \mathsf{pid}_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  6. Send $(\texttt{output}, HO, (\texttt{ok}))$ to $\mathcal{F}^3$
- After delivering output $(\mathsf{ssid}\|3,\ \mathsf{pid}_{HO},\ (addr, \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U, perform as U:
  1. If $\mathsf{TSIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  2. Store $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Send $(\texttt{output}, U, (\texttt{ok}))$ to $\mathcal{F}^3$

UReg in $\mathcal{S}^4$, honest U, honest HO

- Upon receiving $((\texttt{ureg}, \mathsf{ssid}, \mathsf{pid}_U), HO)$ from $\mathcal{F}^4$:
  1. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(\mathsf{ssid}\|1,\ \mathsf{pid}_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ to U
- When the adversary allows delivery of that output:
  1. Wait until receiving $((\texttt{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO}), U)$ from $\mathcal{F}^4$ (if it has not yet been received)
  2. Store vk_O as vk^U_{HO}
  3. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  4. Generate $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}},\ \mathcal{R}_{\texttt{UR}},\ stmt := (\mathsf{pk}_U),\ td_{\text{ZK}})$
  5. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(\mathsf{ssid}\|2,\ \mathsf{pid}_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ to O
- When the adversary allows delivery of that output:
  1. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  2. Add $(\mathsf{pk}_U, \perp, addr, U)$ to $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Report a message $(\mathsf{ssid},\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ from HO to each other operator O'
- For each semi-honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\texttt{ureg}, \mathsf{ssid}), O')$ from $\mathcal{F}^4$ (if it has not been received yet)
  2. Execute the protocol honestly for O', sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- For each honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\texttt{ureg}, \mathsf{ssid}), O')$ from $\mathcal{F}^4$ (if it has not been received yet)
  2. Set $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$, where i is the index of O'
  3. Report message $(\mathsf{ssid},\ \sigma_i)$ from O' to HO
- After the adversary allowed delivery of all messages from O' to HO:
  1. Set T to be the first $t+1$ σ_i
  2. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\mathsf{TSIG}}, T)$
  3. Store $(\mathsf{pk}_U, \perp)$ in $\mathsf{L}^{\mathsf{HO}}_{\mathsf{Accounts}}$ and $creds_U := (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  4. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(\mathsf{ssid}\|3,\ \mathsf{pid}_{HO},\ (addr,\ \sigma))$ to U
  5. Allow $\mathcal{F}^4$ to deliver output to HO
- After the adversary allowed delivery of the output from $\mathcal{F}_{\text{SMT}}$ to U:
  1. Allow $\mathcal{F}^4$ to deliver output to U

Figure 19: Handling registration for honest user and home operator

UReg in $\mathcal{S}^4$, honest U,semi-honest HO

- Upon receiving $(\texttt{input}, HO, (\texttt{ureg}, \mathsf{pid}_U, \mathsf{ssid}))$ from $\mathcal{F}^4$, perform as HO:
  1. Send message $(\mathsf{ssid}\|1, \mathsf{pid}_U, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}$
- Upon receiving $(\texttt{input}, U, (\texttt{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO}))$ from $\mathcal{F}^4$, perform as U:
  1. Wait until $(\mathsf{ssid}\|1, \mathsf{pid}_{HO}, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ was received from $\mathcal{F}_{\text{SMT}}$
  2. Store ek, vk and vk_O as vk_{HO}
  3. Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$
  4. $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. $stmt := (\mathsf{pk}_U)$, $wit := (\mathsf{sk}_U)$
  6. $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt, wit)$
  7. Send message $(\mathsf{ssid}\|2, \mathsf{pid}_{HO}, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ to $\mathcal{F}_{\text{SMT}}$
- After delivering output $(\mathsf{ssid}\|2, \mathsf{pid}_U, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO, perform as HO:
  1. $stmt := (\mathsf{pk}_U)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ abort
  3. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  4. Add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. Send message $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ to each O'
- Upon receiving $(\texttt{input}, O', (\texttt{ureg}, \mathsf{ssid}))$ from $\mathcal{F}^4$, perform as O' (for each O'):
  1. Wait until $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ was sent from HO to O'
  2. $stmt := (\mathsf{pk}_U)$
  3. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\mathsf{Free}}$ abort
  4. Remove $addr$ from $\mathsf{L}_{\mathsf{Free}}$, add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  6. Send message $(\mathsf{ssid}, \sigma_i)$ to HO
- After all messages $(\mathsf{ssid}, \sigma_i)$ from honest and semi-honest O' have been received by HO, perform as HO:
  1. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  2. Set T to be the first valid $t+1$ partial signatures σ_i
  3. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\mathsf{TSIG}}, T)$
  4. Store (pk_U) in $\mathsf{L}^{\mathsf{HO}}_{\mathsf{Accounts}}$
  5. Send message $(\mathsf{ssid}\|3, \mathsf{pid}_U, (addr, \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  6. Send $(\texttt{output}, HO, (\texttt{ok}))$ to $\mathcal{F}^4$
- After delivering output $(\mathsf{ssid}\|3, \mathsf{pid}_{HO}, (addr, \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U, perform as U:
  1. If $\mathsf{TSIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  2. Store $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Send $(\texttt{output}, U, (\texttt{ok}))$ to $\mathcal{F}^4$

UReg in $\mathcal{S}^5$, honest U, semi-honest HO

- The simulator executes the protocol honestly for HO, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to U.
- When the adversary allows delivery of output $(\mathsf{ssid}\|1, \mathsf{pid}_{HO}, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ to U:
  1. Wait until receiving $((\texttt{ureg}, \mathsf{ssid}, \mathsf{pid}_{HO}), U)$ from $\mathcal{F}^5$ (if it has not yet been received)
  2. Store vk_O as vk^U_{HO}
  3. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  4. Generate $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{UR}}, stmt := (\mathsf{pk}_U), td_{\text{ZK}})$
  5. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(\mathsf{ssid}\|2, \mathsf{pid}_U, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ to O
- For each O', when the adversary allows delivery of the message $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ from HO:
  1. Wait until receiving $((\texttt{ureg}, \mathsf{ssid}), O')$ from $\mathcal{F}^5$ (if it has not been received yet)
  2. For the first O':
     (a) Remove $addr$ from $\mathsf{L}_{\mathsf{Free}}$
     (b) Add $(\mathsf{pk}_U, \perp, addr, U)$ to $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Execute the protocol honestly for O'
  4. If O' is semi-honest, send the party's state to the adversary after each activation and wait for `continue` from the adversary before sending out any messages.
- After the adversary allowed delivery of the output from $\mathcal{F}_{\text{SMT}}$ to U:
  1. If $\mathsf{SIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$ abort
  2. Store $creds_U := (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Allow $\mathcal{F}^5$ to deliver output to U

Figure 20: Handling registration for honest user and semi-honest home operator

UReg in $\mathcal{S}^5$, corrupted U

- Upon receiving $(\texttt{input}, HO, (\texttt{ureg}, \mathsf{pid}_U, \mathsf{ssid}))$ from $\mathcal{F}^5$, perform as HO:
  1. Send message $(\mathsf{ssid}\|1,\ \mathsf{pid}_U,\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}$
- When the adversary allows delivery of output $(\mathsf{ssid}\|2,\ \mathsf{pid}_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO, perform as HO:
  1. $stmt \coloneqq (\mathsf{pk}_U)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ abort
  3. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  4. Add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. Send message $(\mathsf{ssid},\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ to each O'
- Upon receiving $(\texttt{input}, O', (\texttt{ureg}, \mathsf{ssid}))$ from $\mathcal{F}^5$, perform as O' (for each O'):
  1. Wait until $(\mathsf{ssid},\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ was delivered from HO to O'
  2. $stmt \coloneqq (\mathsf{pk}_U)$
  3. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\mathsf{Free}}$ abort
  4. Remove $addr$ from $\mathsf{L}_{\mathsf{Free}}$, add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  6. Send message $(\mathsf{ssid}, \sigma_i)$ to HO
- After all messages $(\mathsf{ssid}, \sigma_i)$ from honest and semi-honest O' have been received by HO, perform as HO:
  1. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\mathsf{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  2. Set T to be the first valid $t{+}1$ partial signatures σ_i
  3. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\mathsf{TSIG}}, T)$
  4. Store (pk_U) in $\mathsf{L}^{\mathsf{HO}}_{\mathsf{Accounts}}$
  5. Send message $(\mathsf{ssid}\|3,\ \mathsf{pid}_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  6. Send $(\texttt{output}, HO, (\texttt{ok}))$ to $\mathcal{F}^5$

UReg in $\mathcal{S}^6$, corrupted U

- The simulator executes the protocol honestly, sending each semi-honest party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Additionally, when receiving $(\mathsf{ssid}, \mathsf{pk}_U, \pi_{\mathsf{sk}_U},)$ from U, if the proof verifies successfully:
  1. Extract $wit \coloneqq (\mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}, td_{\text{ZK}})$
  2. Store $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Send $(\mathsf{ureg}, \mathsf{ssid}, HO)$ to $\mathcal{F}^6$ on behalf of U
- When simulation of HO sends the message $(\mathsf{ssid}\|3,\ \mathsf{pid}_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$, allow $\mathcal{F}^6$ to deliver output to HO

Figure 21: Handling registration for corrupted user

$\mathcal{F}_{\text{MPC(insert)}}$ in Game 6

Parameters: A next-step function NS

- Upon receiving (Run, Share$M$, Share$x$) from all parties:
  1. Reconstruct memory M and input x
  2. Initialize $\mathsf{st}_{\mathsf{NS}} \coloneqq (\mathsf{start}, x)$, $data \coloneqq 0$ and $round \coloneqq 0$
  3. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, z)$, do:
     (a) Run $(\mathsf{st}_{\mathsf{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, data)$
     (b) If $I = (\mathsf{read}, addr, \bot)$, set $data \coloneqq M[addr].val$
     (c) Else if $I = (\mathsf{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     (d) Update $round \coloneqq round + 1$
  4. Output shares $\mathsf{Share}^{\mathsf{t}}_{\mathsf{NS}}$, Share$M$ and $round$ to all parties

$\mathcal{F}_{\text{MPC(insert)}}$ in Game 7

Parameters: A next-step function NS

- Upon receiving (Run, Share$M$, Share$x$) from all parties:
  1. Ignore Share$M$, use $M \coloneqq M_{period}$ from the simulator state instead
  2. Reconstruct input x
  3. Initialize $\mathsf{st}_{\mathsf{NS}} \coloneqq (\mathsf{start}, x)$, $data \coloneqq 0$ and $round \coloneqq 0$
  4. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, z)$, do:
     (a) Run $(\mathsf{st}_{\mathsf{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, data)$
     (b) If $I = (\mathsf{read}, addr, \bot)$, set $data \coloneqq M[addr].val$
     (c) Else if $I = (\mathsf{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     (d) Update $round \coloneqq round + 1$
  5. Output shares $\mathsf{Share}^{\mathsf{t}}_{\mathsf{NS}}$, Share$M$ and $round$ to all parties

Figure 22: Simulation of $\mathcal{F}_{\text{MPC(insert)}}$, Part 1

$\mathcal{F}_{\mathbf{MPC}(\mathsf{insert})}$ in Game 7

Parameters: A next-step function NS

- Upon receiving (Run, $\mathsf{Share}M$, $\mathsf{Share}x$) from all parties:
  1. Ignore $\mathsf{Share}M$, use $M := M_{period}$ from the simulator state instead
  2. Reconstruct input x
  3. Initialize $\mathsf{st}_{\mathsf{NS}} := (\mathsf{start}, x)$, $data := 0$ and $round := 0$
  4. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, z)$, do:
     (a) Run $(\mathsf{st}_{\mathsf{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, data)$
     (b) If $I = (\mathsf{read}, addr, \bot)$, set $data := M[addr].val$
     (c) Else if $I = (\mathsf{write}, addr, val)$, set $data := M[addr].val$ and update $M[addr].val = I.val$
     (d) Update $round := round + 1$
  5. Output shares $\mathsf{Sharest}_{\mathsf{NS}}$, $\mathsf{Share}M$ and $round$ to all parties

$\mathcal{F}_{\mathbf{MPC}(\mathsf{insert})}$ in Game 8

Parameters: A next-step function NS

- Upon receiving (Run, $\mathsf{Share}M$, $\mathsf{Share}x$) from all parties:
  1. Ignore $\mathsf{Share}M$, use $M := M_{period}$ from the simulator state instead
  2. Reconstruct input x
  3. Initialize $\mathsf{st}_{\mathsf{NS}} := (\mathsf{start}, x)$, $data := 0$ and $round := 0$
  4. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, z)$, do:
     (a) Run $(\mathsf{st}_{\mathsf{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, data)$
     (b) If $I = (\mathsf{read}, addr, \bot)$, set $data := M[addr].val$
     (c) Else if $I = (\mathsf{write}, addr, val)$, set $data := M[addr].val$ and update $M[addr].val = I.val$
     (d) Update $round := round + 1$
  5. Output shares $\mathsf{Sharest}_{\mathsf{NS}}$, $\mathsf{Share}0$ and $round$ to all parties

Figure 23: Simulation of $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$, Part 2

Handling of $(\texttt{createentry}, \ldots)$ in $\mathcal{F}^8$

- Upon receiving input ($\texttt{createentry}$, ssid, $period$, $entry$) from party P:
  1. Send ($\texttt{input}$, P, ($\texttt{createentry}$, ssid, $period$, $entry$)) to the simulator
- Upon receiving ($\texttt{output}$, P, msg) from the simulator:
  1. Send (msg) to party P

Handling of $(\texttt{createentry}, \ldots)$ in $\mathcal{F}^9$

- Upon receiving input ($\texttt{createentry}$, ssid, $period$, $entry$) from party P:
  1. If pid_U is not in $\bigcup_{O \in \mathcal{O}} \mathsf{L}^{O}_{\mathsf{users}}$, ignore this call
  2. Set $id^{O}_{\mathsf{last}} := id^{O}_{\mathsf{last}} + 1$, $id := id^{O}_{\mathsf{last}}$
  3. Add $(id, \mathsf{pid}_U, period, entry)$ to $\mathsf{L}_{\mathsf{Pending}\,O}$
  4. Send ($\texttt{input}$, P, ($\texttt{createentry}$, ssid, $period$, $entry$)) to the simulator
- Upon receiving ($\texttt{output}$, P, msg) from the simulator:
  1. Send (msg) to party P

Figure 24: First step in handling creation of log entries

Handling of `createentry` messages from corrupt users in $\mathcal{S}^9$

- When the adversary allows delivery of output (send, $\mathsf{ssid}\|1$, $(entry_U, ct_U, \pi)$):
  1. Wait until receiving ($\texttt{input}$, ($\texttt{createentry}$, ssid, $period$, $entry_O$), O) from $\mathcal{F}^9$
  2. If $entry_U \neq entry_O$, abort
  3. $h := H(period, ct_U, entry_O)$, $stmt := (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  4. If $\mathsf{Vf}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathrm{CE}}, stmt, \pi) \neq 1$, abort
  5. $id_{\mathsf{last}} := id_{\mathsf{last}} + 1$
  6. Add $(id_{\mathsf{last}}, period, ct_U, entry, \pi)$ to $\mathsf{L}_{\mathsf{Pending}}$
  7. $\sigma_{entry} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O, h)$
  8. Send ($\mathsf{respond}$, $\mathsf{ssid}\|1$, $(\sigma_{entry}, \mathsf{vk}_O)$) to $\mathcal{F}^{\mathrm{os-auth}}_{\mathrm{SMT}}$ and ($\texttt{output}$, O, (ssid, $\texttt{ok}$, id_{last})) to $\mathcal{F}^9$

Handling of `createentry` messages from corrupt users in $\mathcal{S}^{10}$

- When the adversary allows delivery of output (send, $\mathsf{ssid}\|1$, $(entry_U, ct_U, \pi)$):
  1. Wait until receiving ($\texttt{input}$, ($\texttt{createentry}$, ssid, $period$, $entry_O$), O) from $\mathcal{F}^{10}$
  2. If $entry_U \neq entry_O$, abort
  3. $h := H(period, ct_U, entry_O)$, $stmt := (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  4. If $\mathsf{Vf}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathrm{CE}}, stmt, \pi) \neq 1$, abort
  5. $id_{\mathsf{last}} := id_{\mathsf{last}} + 1$
  6. Add $(id_{\mathsf{last}}, period, ct_U, entry, \pi)$ to $\mathsf{L}_{\mathsf{Pending}}$
  7. $\sigma_{entry} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O, h)$
  8. Extract $wit := (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r) \leftarrow \mathsf{ZK.Ext}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathrm{CE}}, stmt, \pi, td_{\mathrm{ZK}})$, abort if this fails
  9. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no such entry exists
  10. If U^* is honest, abort
  11. Send ($\texttt{createentry}$, ssid, $period$, $entry_U$) to $\mathcal{F}_{\mathrm{POBA}}$ on behalf of user U^*
  12. Send ($\mathsf{respond}$, $\mathsf{ssid}\|1$, $(\sigma_{entry}, \mathsf{vk}_O)$) to $\mathcal{F}^{\mathrm{os-auth}}_{\mathrm{SMT}}$ and ($\texttt{output}$, O, (ssid, $\texttt{ok}$, id_{last})) to $\mathcal{F}^{10}$

Figure 25: Calling $\mathcal{F}_{\mathrm{POBA}}$ with $(\texttt{createentry}, \ldots)$ for corrupted users

Handling of $(\mathsf{insert}, \ldots)$ **in** $\mathcal{F}^{10}$

- Upon receiving input $(\mathsf{insert}, \mathsf{ssid}, \mathit{period}, \mathit{id})$ from party P:
  1. Send $(\mathtt{input}, P, (\mathsf{insert}, \mathsf{ssid}, \mathit{period}, \mathit{id}))$ to the simulator
- Upon receiving $(\mathtt{output}, P, \mathrm{msg})$ from the simulator:
  1. Send (msg) to party P

Handling of $(\mathsf{insert}, \ldots)$ **in** $\mathcal{F}^{11}$

- Upon receiving input $(\mathsf{insert}, \mathsf{ssid}, \mathit{period}, \mathit{id})$ from party P:
  1. If no entry $(\mathit{id}, \mathsf{pid}_U, \mathit{period}, \mathit{entry})$ exists in $\mathsf{L}_{\mathsf{Pending}_O}$, ignore this call
  2. Otherwise, remove that entry and append (entry) to $\mathsf{Log}_U^{\mathit{period}}$
  3. Send $(\mathtt{input}, P, (\mathsf{insert}, \mathsf{ssid}, \mathit{period}, \mathit{id}))$ to the simulator.
- Upon receiving $(\mathtt{output}, P, \mathrm{msg})$ from the simulator:
  1. Send (msg) to party P

Figure 26: First step in handling creation of log entries

simulation-extractability of ZK. For (2), note that a valid signature σ on $(\mathsf{pk}_U, \mathit{addr})$ has been extracted. That signature is generated during execution of the `ureg` task, and since Game 6 the simulator creates the appropriate entry in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$ when generating this signature. Thus, no matching entry existing means σ was not created by the simulator, and thus the probability of (2) is bound by the probability to break TS-UF-1-security of TSIG (for the reduction, the simulator uses vk and the obtained sk_i for corrupted parties in the TS-UF-1 experiment during simulation of $\mathcal{F}^{\mathsf{TSIG}}_{\mathrm{Keygen}}$ and replaces all instances of $\mathsf{TSIG.PartSign}$ of honest parties with calls to the signing oracle of the experiment). Lastly, for (3), extraction also provided the secret signing key of U^*. For a honest user, obtaining this is bounded by the probability to break EUF-CMA-security of SIG (for the reduction, the simulator uses vk of the EUF-CMA experiment. The user secret key is never actually used to generate a signature, and the simulator is already simulating all zero-knowledge proofs that would use the secret key. The extracted signing key can then be used to forge a valid signature). Thus, the distinguishing advantage is negligible.

Game 11. In this game, $\mathcal{F}^{11}$ on input $(\mathsf{insert}, \mathsf{ssid}, \mathit{period}, \mathit{id})$ from some O, ignores the call if no entry $(\mathit{id}, \mathsf{pid}_U, \mathit{period}, \mathit{entry})$ exists in $\mathsf{L}_{\mathsf{Pending}_O}$. Otherwise, it still sends the input to the simulator and delivers output as instructed by the simulator. Furthermore, $\mathcal{F}^{11}$ fills $\mathsf{Log}_U^{\mathit{period}}$ the same way as $\mathcal{F}_{\mathrm{POBA}}$. The concrete behavior of $\mathcal{F}^{11}$ is detailed in Fig. 26.

Proof Sketch: Game 10 ≈ Game 11.
Storing internal data in $\mathcal{F}^{11}$ has no effect on the view of the environment. As such, the only observable change is $\mathcal{F}^{11}$ ignoring a call instead of forwarding the input to the simulator. $\mathcal{F}^{11}$ ignores the call only if no matching entry $(\mathit{id}, \mathsf{pid}_U, \mathit{period}, \mathit{entry})$ exists in $\mathsf{L}_{\mathsf{Pending}_O}$. Since Game 10, such an entry is created whenever O successfully completes the `createentry` task. Thus, whenever $\mathcal{F}^{11}$ ignores the call, the simulation of the (semi-)honest O would also abort because no entry $(\mathit{id}, \mathit{period}, \mathit{ct}_U, \mathit{entry}, \pi)$ exists in $\mathsf{L}_{\mathsf{Pending}}$. Thus, from the environment's view, these games are identical.

Game 12. In this game, $\mathcal{F}^{12}$ ignores inputs $(\mathtt{proveentry}, \mathsf{ssid}, \mathit{entry}, \mathit{period}, O)$ from some user U if no entry $(\mathit{id}, \mathsf{pid}_U, \mathit{period}, \mathit{entry})$ in $\mathsf{L}_{\mathsf{Pending}_{O' \in \mathcal{O}}}$ or (entry) in $\mathsf{Log}_U^{\mathit{period}}$ exists. Otherwise, it still forwards these inputs to the simulator and forwards outputs from the simulator to O. For the concrete changes of this game, see Fig. 27.

Proof Sketch: Game 11 ≈ Game 12.
$\mathcal{F}^{12}$ ignores inputs only if no matching entry exists. These entries were created since Game 10 resp. Game 11 whenever the user actually created the respective log entry. Thus, whenever $\mathcal{F}^{12}$ ignores an input, simulation of the honest user would also abort and thus not create any messages because no entry in $\mathsf{L}_{\mathsf{Receipt}}$ exists. Therefore, these games are identical to the environment.

Game 13. In this game, $\mathcal{F}^{13}$ handles inputs $(\mathtt{proveentry}, \mathsf{ssid}, \mathit{entry}, \mathit{period}, O)$

Handling of (proveentry, ...) in $\mathcal{F}^{11}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, *O*) from user *U*:
  1. Send (input, *P*, (proveentry, ssid, *entry*, *period*, *O*)) to the simulator
- Upon receiving (output, *O*, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Send (ssid, proveentry, pid_U, *entry*, *period*) to *O*

Handling of (proveentry, ...) in $\mathcal{F}^{12}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. Otherwise, send (input, *P*, (proveentry, ssid, *entry*, *period*, *O*)) to the simulator
- Upon receiving (output, *O*, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Send (ssid, proveentry, pid_U, *entry*, *period*) to *O*

Figure 27: Handling of (proveentry) for honest *U* and *O* in $\mathcal{F}_{\text{POBA}}$

Handling of (proveentry, ...) in $\mathcal{F}^{12}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. Otherwise, send (input, *P*, (proveentry, ssid, *entry*, *period*, *O*)) to the simulator
- Upon receiving (output, *O*, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Send (ssid, proveentry, pid_U, *entry*, *period*) to *O*

Handling of (proveentry, ...) in $\mathcal{F}^{13}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. If *O* is honest:
     (a) Send ((proveentry, ssid), *U*) to the adversary
     (b) Output (ssid, proveentry, *U*, *entry*, *period*) to *O*
  3. Otherwise, send (input, *P*, (proveentry, ssid, *entry*, *period*, *O*)) to the simulator
- Upon receiving (output, *O*, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If *U* and *O* are honest, ignore this message
  2. Otherwise, send (ssid, proveentry, pid_U, *entry*, *period*) to *O*

Figure 28: Handling of (proveentry) for honest *U* and *O* in $\mathcal{F}_{\text{POBA}}$

from some user *U* the same way as $\mathcal{F}_{\text{POBA}}$ does if both *U* and *O* are honest, and $\mathcal{S}^{13}$ handles simulation of them the same way as $\mathcal{S}$ does. For the concrete changes of this game, see Figs. 28 and 29.

Proof Sketch: Game 12 ≈ Game 13.
This is indistinguishable for the environment as running the real protocol reveals nothing besides the length of the message from *U* to *O*.

Game 14. In this game, $\mathcal{F}^{14}$ handles inputs (proveentry, ssid, *entry*, *period*, *O*) for honest users same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{14}$ handles simulation of them the same way as $\mathcal{S}$ does for honest users. For corrupted users, $\mathcal{S}^{14}$ still handles them the same way as $\mathcal{S}^{13}$ and $\mathcal{F}^{14}$ still forwards output to *O*. For the concrete changes of this game, see Figs. 30 and 31.

Proof Sketch: Game 13 ≈ Game 14.
The only visible difference between these games for the environment is how π in the message to the semi-honest operator is generated. Thus, the probability of distinguishing these games is bounded by the probability to break zero-knowledge of ZK and therefore the games are indistinguishable.

Game 15. In this game, $\mathcal{F}^{15}$ handles inputs (proveentry, ssid, *entry*, *period*, *O*) same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{15}$ handles simulation of them the same way as $\mathcal{S}$ does for both honest and corrupted users. For the concrete changes of this game, see Figs. 32 and 33.

Proof Sketch: Game 14 ≈ Game 15.
Let us first consider when the simulator aborts. This is when 1. extraction of the witness

Handling of (proveentry, ...) **in** $\mathcal{S}^{12}$

- Upon receiving (input, U, (proveentry, ssid, *entry*, *period*, O)) from $\mathcal{F}^{12}$:
  1. Run the protocol honestly for U and O
  2. When the protocol would generate output (ssid, proveentry, pid_U, *entry*, *period*) for O, send (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) to $\mathcal{F}^{12}$

Handling of (proveentry, ...) **in** $\mathcal{S}^{13}$

- Upon receiving (input, U, (proveentry, ssid, *entry*, *period*, O)) from $\mathcal{F}^{13}$:
  1. Run the protocol honestly for U and O
  2. When the protocol would generate output (ssid, proveentry, pid_U, *entry*, *period*) for O, send (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) to $\mathcal{F}^{13}$
- Upon receiving ((proveentry, ssid), U) from $\mathcal{F}^{13}$:
  1. Simply report a message of the appropriate length from U to O

Figure 29: Handling of (proveentry) for honest U and O in $\mathcal{S}$

Handling of (proveentry, ...) **in** $\mathcal{F}^{13}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, O) from user U:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. If O is honest:
     (a) Send ((proveentry, ssid), U) to the adversary
     (b) Output (ssid, proveentry, pid_U, *entry*, *period*) to O
  3. Otherwise, send (input, P, (proveentry, ssid, *entry*, *period*, O)) to the simulator
- Upon receiving (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If U and O are honest, ignore this message
  2. Otherwise, send (ssid, proveentry, pid_U, *entry*, *period*) to O

Handling of (proveentry, ...) **in** $\mathcal{F}^{14}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, O) from some user U:
  1. Check if an entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or an entry (*entry*) exists in Log_U^{period}, otherwise ignore this input
  2. If O is corrupted, send ((proveentry, ssid, *entry*, *period*, O), U) to the adversary
     Otherwise, send ((proveentry, ssid), U) to the adversary.
  3. Output (ssid, proveentry, U, *entry*, *period*) to O
- Upon receiving (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If U is honest, ignore this message
  2. Otherwise, send (ssid, proveentry, pid_U, *entry*, *period*) to O

Figure 30: Handling of (proveentry) in $\mathcal{F}_{\text{POBA}}$

Handling of (proveentry, ...) **in** $\mathcal{S}^{13}$

- Upon receiving ((proveentry, ssid), U) from $\mathcal{F}^{13}$:
  1. Simply report a message of the appropriate length from U to O
- Upon receiving (input, U, (proveentry, ssid, *entry*, *period*, O)) from $\mathcal{F}^{13}$:
  1. Run the protocol honestly for U and O
  2. When the protocol would generate output (ssid, proveentry, pid_U, *entry*, *period*) for O, send (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) to $\mathcal{F}^{13}$

Handling of (proveentry, ...) **in** $\mathcal{S}^{14}$

- Upon receiving ((proveentry, ssid), U) from $\mathcal{F}^{14}$:
  1. Simply report a message of the appropriate length from U to O
- Upon receiving ((proveentry, ssid, *entry*, *period*, O), U) from $\mathcal{F}^{14}$:
  1. Retrieve $(entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ from $\mathsf{L}^{\mathsf{U}}_{\mathsf{Receipt}}$
  2. Retrieve $(\mathsf{pk}_U, \bot, addr, U)$ from $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Set $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U)$
  4. Set $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, td_{\text{ZK}})$
  5. Report output (ssid‖1, pid_U, (*period*, *entry*, pk_U, *addr*, σ, ct_U, σ_{entry}, vk_O, π)) to O from $\mathcal{F}_{\text{SMT}}$
- When the adversary allows delivery of that output:
  1. Execute the protocol for O honestly, sending the party's state to the adversary after each activation and waiting for continue from the adversary before sending out any messages.
  2. When the adversary allows O to generate output, allow $\mathcal{F}^{14}$ to deliver output to O

Figure 31: Handling of (proveentry) for honest U in $\mathcal{S}$

Handling of (proveentry, ...) **in** $\mathcal{F}^{14}$

- When the adversary allows delivery of output (ssid‖1, pid_U, (*period*, *entry*, pk_U, *addr*, σ, ct_U, σ_{entry}, vk_O, π)) from $\mathcal{F}_{\text{SMT}}$:
  1. Check if an entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or an entry (*entry*) exists in Log_U^{period}, otherwise ignore this input
  2. If O is corrupted, send ((proveentry, ssid, *entry*, *period*, O), U) to the adversary, otherwise send ((proveentry, ssid), U) to the adversary
  3. Output (ssid, proveentry, U, *entry*, *period*) to O
- Upon receiving (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If U is honest, ignore this message
  2. Otherwise, send (ssid, proveentry, pid_U, *entry*, *period*) to O

Handling of (proveentry, ...) **in** $\mathcal{F}^{15}$

- Upon receiving input (proveentry, ssid, *entry*, *period*, O) from some user U:
  1. Check if an entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\mathsf{Pending}\,O'\in\mathcal{O}}$ or an entry (*entry*) exists in Log_U^{period}, otherwise ignore this input
  2. If O is corrupted, send ((proveentry, ssid, *entry*, *period*, O), U) to the adversary, otherwise send ((proveentry, ssid), U) to the adversary
  3. Output (ssid, proveentry, U, *entry*, *period*) to O
- Upon receiving (output, O, (ssid, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Ignore this message

Figure 32: Handling of (proveentry) in $\mathcal{F}_{\text{POBA}}$

Handling of (proveentry, ...) **in** $\mathcal{S}^{14}$

- Upon receiving ((proveentry, ssid), U) from $\mathcal{F}^{15}$:
  1. Simply report a message of the appropriate length from U to O
- Upon receiving ((proveentry, ssid, *entry*, *period*, O), U) from $\mathcal{F}^{15}$:
  1. [...]
- Upon receiving (ssid, *period*, *entry*, pk_U, *addr*, σ, ct_U, σ_{entry}, vk_O, π) from a corrupted U:
  1. Follow the protocol of O honestly, sending the party's state to the adversary after each activation and waiting for continue from the adversary before sending out any messages if O is semi-honest
  2. When O generates output, send the appropriate (output, O, ...) message to $\mathcal{F}^{14}$

Handling of (proveentry, ...) **in** $\mathcal{S}^{15}$

- Upon receiving ((proveentry, ssid), U) from $\mathcal{F}^{15}$:
  1. Simply report a message of the appropriate length from U to O
- Upon receiving ((proveentry, ssid, *entry*, *period*, O), U) from $\mathcal{F}^{15}$:
  1. [...]
- Upon receiving (ssid, *period*, *entry*, pk_U, *addr*, σ, ct_U, σ_{entry}, vk_O, π) from a corrupted U:
  1. Follow the steps of the honest protocol, sending the party's state to the adversary after each activation and waiting for continue from the adversary before sending out any messages if O is semi-honest
  2. Before asking the adversary for permission to deliver output:
     (a) Set $stmt := (\mathsf{pk}_U, addr, ct_U)$
     (b) Get $wit := (r, \mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, \pi, td_{\text{ZK}})$, abort if this fails
     (c) Find the entry $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no matching entry exists
     (d) If U^* is honest, abort
     (e) Send message (proveentry, *entry*, *period*, O, ssid) to $\mathcal{F}^{15}$ on behalf of U^*
  3. When the adversary allows O to deliver output, allow $\mathcal{F}^{15}$ to deliver output to O

Figure 33: Handling of (proveentry) for honest U in $\mathcal{S}$

Handling of `analytics` in $\mathcal{F}^{15}$

- Upon receiving input (`analytics`, `ssid`, *func*, $\{period_i\}$, $\{U_i\}$, aux_O) from O:
  1. Send (`input`, O, (`analytics`, `ssid`, *func*, $\{period_i\}$, $\{U_i\}$)) to the simulator
- Upon receiving (`output`, O, (`ssid`, $\texttt{result}_O$)) from the simulator:
  1. Send (`ssid`, $\texttt{result}_O$) to O

Handling of `analytics` in $\mathcal{F}^{16}$

- Upon receiving input (`analytics`, `ssid`, *func*, $\{period_i\}$, $\{U_i\}$, aux_O) from O:
  1. If not all parties provided the same *func*, $\{period_i\}$ and $\{U_i\}$ as input, ignore this call
  2. Set $\mathsf{Logs} := \left\{\mathsf{Log}_U^p\right\}_{U\in\{U_i\},p\in\{period_i\}}$
     Evaluate $\{\texttt{result}_O\}_{O\in\mathcal{O}} \leftarrow func(\mathsf{Logs}, \{aux_O\}_{O\in\mathcal{O}})$, setting $sequence := (p_1, p_2, \ldots, p_n)$ with an entry $p_j := period_j$ for each time *func* reads an entry from $\mathsf{Log}_U^{period_j}$
  3. If at least one operator is semi-honest, send (`analytics`, `ssid`, *sequence*) to the adversary
  4. For each operator O, generate delayed private output (`ssid`, $\texttt{result}_O$)

Figure 34: Handling of analytics in $\mathcal{F}_{\text{POBA}}$

fails, 2. no matching entry in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$ exists, or 3. U^* is honest. The probability of (1) is bounded by the probability to break simulation-extractability of ZK. The message contains a valid signature σ on $(\mathsf{pk}_U, addr)$. Whenever such a signature is created, the appropriate entry in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$ is also generated, thus the probability for (2) is bounded by the probability to break TS-UF-1-security of TSIG. Regarding (3), extraction provided sk_U, obtaining of which for an honest user is as hard as breaking EUF-CMA-security of SIG. Therefore, the probability that $\mathcal{S}^{15}$ aborts is negligible. Note that when $\mathcal{S}^{15}$ sends (`proveentry`, ...) to $\mathcal{F}^{15}$, simulation of O outputs (`ssid`, `proveentry`, ...). We thus need to ensure that $\mathcal{F}^{15}$ also delivers that output to O. This is the case because if $\mathcal{S}^{15}$ did not abort, an appropriate call of (`createentry`, ...) has been made.

Game 16. In this game, $\mathcal{F}^{16}$ handles inputs (`analytics`, ...) the same way as $\mathcal{F}_{\text{POBA}}$ and $\mathcal{S}^{16}$ handles simulation of those inputs and $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ the same way as $\mathcal{S}$. For the concrete changes, see Figs. 34 and 35.

Proof Sketch: Game 15 ≈ Game 16.
If all operators are honest, the only thing visible to the environment are the lengths of the messages required to reveal shares of the outputs. Since shares have a fixed length, simulation of these is trivial and perfect. With at least one semi-honest operator, $\mathcal{F}^{16}$ leaks *sequence* required to simulate $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ as well as the correct output for each semi-honest operator. With these information, simulation of $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ is perfect. The only other messages visible to semi-honest operators and thus the environment are those required to reveal shares of their output, and since $\mathcal{S}^{16}$ was using the correct outputs when generating these shares, these are also identically distributed.

Game 17. In this game, $\mathcal{F}^{17}$ handles inputs ($\mathsf{insert}, \ldots$) the same way as $\mathcal{F}_{\text{POBA}}$ and $\mathcal{S}^{17}$ handles simulation of them the same way as $\mathcal{S}$. For the concrete changes, see Figs. 36 and 37.

Proof Sketch: Game 16 ≈ Game 17.
Simulation of $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$ is the same in both games as insert has no output and the number of memory accesses is constant, independent of $addr^*$. The simulator now does not correctly update M anymore, but since Game 16 this is not used in a way visible to the environment anymore, so this change is indistinguishable for the environment. Lastly, the messages sent to semi-honest parties are distributed the same in both games since ct_U,*entry* and π are the same in both games. Thus, these games are perfectly indistinguishable to the environment.

Game 18. In this game, when both user and operator are honest while calling `createentry`, $\mathcal{F}^{18}$ no longer sends their inputs to the simulator. $\mathcal{S}^{18}$ simulates this

Handling of `analytics` in $\mathcal{S}^{15}$

- Upon receiving (`input`, O, (`analytics`, `ssid`, ...)) from $\mathcal{F}^{15}$
  1. Run the protocol honestly from O
  2. When the protocol would generate output (`ssid`, $\texttt{result}_O$) for O send (`output`, O, (`ssid`, $\texttt{result}_O$)) to $\mathcal{F}^{15}$

Handling of `analytics` in $\mathcal{S}^{16}$

- If all operators are honest, simply report messages for revealing output shares of the appropriate lengths.
- Otherwise (at least one operator is semi-honest):
- Upon receiving ((`analytics`, `ssid`, *func*, $\{period_i\}$, $\{U_i\}$), O') from $\mathcal{F}_{\text{POBA}}$ (i.e., O' is semi-honest):
  1. Simulate O' honestly, sending the state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages. When the adversary allows generation of output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O'
- Upon receiving ((`analytics`, `ssid`), O) from $\mathcal{F}_{\text{POBA}}$ (i.e., O is honest):
  1. Do nothing
- Wait until receiving (`analytics`, `ssid`, *sequence*) and outputs ($\mathsf{ssid}, \texttt{result}_{O'}$) for each semi-honest operator O', then simulate $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ as follows:
  1. For each honest O^*, set $\texttt{result}_{O^*} := 0$
  2. Set $\texttt{result} := \bigcup \texttt{result}_O$ (this is 0 for honest operators, and the output of $\mathcal{F}_{\text{POBA}}$ for semi-honest operators)
  3. Output shares $\mathsf{Share}(\mathsf{stop}, \texttt{result})$, $\{\mathsf{Share}M_i\}$, *round*, *sequence* to all parties
- When the simulation of $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ has delivered output to honest O:
  1. For each semi-honest O', reveal $\mathsf{Share}\texttt{result}_{O'}$ to O'
  2. For each honest O^*, report a message of the correct length from O to O^*
- When the adversary allowed delivery of all share reveals towards an honest operator O:
  1. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

Figure 35: Handling of analytics in $\mathcal{S}$

Handling of $(\mathsf{insert}, \ldots)$ in $\mathcal{F}^{16}$

- Upon receiving input (`insert`, `ssid`, *period*, *id*) from O:
  1. If no entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\mathsf{Pending}\,O}$, ignore this call
  2. Otherwise, remove that entry and append (*entry*) to Log_U^{period}
  3. Send (`input`, P, (`insert`, `ssid`, *period*, *id*)) to the simulator
- Upon receiving (`output`, P, msg) from the simulator:
  1. Send (msg) to party P

Handling of $(\mathsf{insert}, \ldots)$ in $\mathcal{F}^{17}$

- Upon receiving input (`insert`, `ssid`, *period*, *id*) from O and (`insert`, `ssid`) from all other O':
  1. If at least one O' is corrupted, send ((`insert`, `ssid`, *period*, *id*), O) to the adversary
  2. If no entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\mathsf{Pending}\,O}$, ignore this call
  3. Otherwise, remove that entry and append (*entry*) to Log_U^{period}
  4. Generate delayed private output (`ssid`, `ok`) for each O

Figure 36: Handling insertion of log entries in $\mathcal{F}_{\text{POBA}}$

Handling of (insert, ...) **in** $\mathcal{S}^{16}$

- Upon receiving (`input`, O, (insert, ...)) from $\mathcal{F}^{16}$
  1. Run the protocol honestly for O
  2. When the protocol would generate output ($entry$) for O, send (`output`, O, $entry$) to $\mathcal{F}^{16}$

Handling of (insert, ...) **in** $\mathcal{S}^{17}$

- Upon receiving ((insert, ssid, $period$, id), O) from $\mathcal{F}^{17}$ (i.e., O is semi-honest):
  1. Honestly execute the protocol for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Upon receiving ((insert, ssid), O) and ((insert, ssid, $period$, $entry^*$), O) from $\mathcal{F}^{17}$ (i.e., O is honest):
  1. Find (id, $period$, ct_U, $entry$, π) $\in \mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$ and ignore this message if no entry with (id, $period$) exists
  2. Send (ssid, ct_U, $entry$, π) to all O'
  3. Honestly run the checks from the real protocol
  4. Treat O as having provided input to $\mathcal{F}_{\text{MPC(insert)}}$
- Upon receiving ((insert, ssid), O') from $\mathcal{F}^{17}$ for honest O':
  1. Wait until the adversary allowed delivery of the message (ct_U, $entry$, π) from O
  2. Honestly run the checks from the real protocol
  3. Treat O' as having provided input to $\mathcal{F}_{\text{MPC(insert)}}$

Simulate $\mathcal{F}_{\text{MPC(insert)}}$ as follows:

- Upon receiving (Run, Share$M$, Share$x$) from all semi-honest operators and having treated all honest operators as having provided input to $\mathcal{F}_{\text{MPC(insert)}}$:
  1. Chose a random $addr^* \in \mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  2. Ignore Share$M$ and Share$x$, use M and partial decryptions of TPKE.Encrypt(ek, $addr^*$) instead
  3. Follow the description of insert honestly
  4. When delivering output to semi-honest operators, instead of Share$M$ use Share0, where 0 is a memory the same size as M but consisting only of 0-entries
- When the adversary allows delivery of output from $\mathcal{F}_{\text{MPC(insert)}}$ to an operator O
  1. Allow $\mathcal{F}^{17}$ to deliver output to O

Figure 37: Handling insertion of log entries in $\mathcal{S}$

Handling of $(\mathsf{insert}, \ldots)$ **in** $\mathcal{S}^{17}$

- Upon receiving $((\mathsf{insert}, \mathsf{ssid}), O)$ and $((\mathsf{insert}, \mathsf{ssid}, period, entry^*), O)$ from $\mathcal{F}^{18}$ (i.e., O is honest):
  1. Find $(id, period, ct_U, entry, \pi) \in \mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$ and ignore this message if no entry with $(id, period)$ exists
  2. Send $(\mathsf{ssid}, ct_U, entry, \pi)$ to all O'
  3. Honestly run the checks from the real protocol
  4. Treat O as having provided input to $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$

Handling of $(\mathsf{insert}, \ldots)$ **in** $\mathcal{S}^{18}$

- Upon receiving $((\mathsf{insert}, \mathsf{ssid}), O)$ and $((\mathsf{insert}, \mathsf{ssid}, period, entry^*), O)$ from $\mathcal{F}^{18}$ (i.e., O is honest):
  1. Create $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, 0)$
  2. Set $h := H(period, ct_U, entry^*)$
  3. Set $stmt := (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  4. Set $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathrm{CE}}, stmt, td_{\mathrm{ZK}})$
  5. Send $(\mathsf{ssid}, ct_U, entry^*, \pi)$ to all O'
  6. Treat O as having provided input to $\mathcal{F}_{\mathrm{MPC}(\mathsf{insert})}$

Figure 38: Handling insertion of log entries in $\mathcal{S}$

Handling of `createentry` **in** $\mathcal{S}^{18}$

- Upon receiving $(\mathtt{input}, O, (\mathtt{createentry}, \mathsf{ssid}, period, entry_O))$ and $(\mathtt{input}, U, (\mathtt{createentry}, \mathsf{ssid}, period, entry_U))$:
  1. Run the protocol honestly
- When the adversary allows delivery of the output:
  1. Forward that output to $\mathcal{F}^{18}$

Handling of `createentry` **in** $\mathcal{S}^{19}$

- Honestly run the protocol for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Upon receiving $((\mathtt{createentry}, \mathsf{ssid}), \mathtt{user})$ and $((\mathtt{createentry}, \mathsf{ssid}, period, entry), O)$ from $\mathcal{F}_{\mathrm{POBA}}$:
  1. Set $addr := 0$
  2. Generate $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
  3. Set $h := H(period, ct_U, entry)$
  4. Set $stmt := (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  5. Set $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathrm{CE}}, stmt, td_{\mathrm{ZK}})$
  6. Report that $\mathcal{F}^{\mathrm{os-auth}}_{\mathrm{SMT}}$ wants to generate output $(\mathtt{send}, \mathsf{ssid}\|1, (ct_U, \pi))$ to O
- When the adversary allows delivery of output $(\mathtt{respond}, \mathsf{ssid}\|1, (\sigma_{entry}, \mathsf{vk}_O))$:
  1. Store $(entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ in $\mathsf{L}^{\mathsf{U}}_{\mathsf{Receipt}}$
  2. Allow $\mathcal{F}_{\mathrm{POBA}}$ to deliver output

Figure 39: Handling creation of log entries in $\mathcal{S}$

case by simply reporting messages of appropriate size between the honest parties. Since this results in $\mathcal{S}^{18}$ no longer creating entries $(id, period, ct_U, entry, \pi)$ in $\mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$, it now uses the leak of $(period, entry)$ provided by $\mathcal{F}^{18}$ to simulate InsertEntry for such entries. For the detailed changes to $\mathcal{S}^{18}$ regarding InsertEntry, see Fig. 38.

Proof Sketch: Game 17 ≈ Game 18.
First, let us observe the changes to InsertEntry: The probability of distinguishing the change of ct_U during insert is bounded by the probability to break IND-CPA-security of TPKE, and of π by the probability to break zero-knowledge of ZK. The message is still the same, as the one leaked by $\mathcal{F}^{18}$ is the same that was sent by $\mathcal{F}^{17}$ during CreateEntry.

Game 19. We now further change `createentry`: In this game, when the user is honest but the operator is only semi-honest, $\mathcal{F}^{19}$ no longer forwards the user's input or output from the simulator. Since the operator is semi-honest, $\mathcal{S}^{19}$ still gets their input to simulate. See Fig. 39 for the detailed changes.

Proof Sketch: Game 18 ≈ Game 19.
Again, these games are indistinguishable because the only changes are the generation of ct_U and π. Thus, the distinguishing advantage is bounded by the probability to break either IND-CPA of TPKE or zero-knowledge of ZK.

Game 20. Another change to `createentry`: In this game, when the user is corrupt and the operator is semi-honest, $\mathcal{F}^{20}$ handles this task the same way as $\mathcal{F}_{\mathrm{POBA}}$ does, and $\mathcal{S}^{20}$ the same way as $\mathcal{S}$ does. Since the operator is semi-honest, $\mathcal{S}^{20}$ still gets the operator's input to simulate. See Fig. 40 for the detailed changes.

Handling of `createentry` in $\mathcal{S}^{19}$

- Upon receiving (`input`, O, (`createentry`, `ssid`, $period$, $entry_O$)) and outputs from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
  1. Run the protocol honestly
- When the adversary allows delivery of output for the operator:
  1. Forward that output to $\mathcal{F}^{19}$

Handling of `createentry` in $\mathcal{S}^{20}$

- The simulator executes the protocol honestly for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Additionally, the simulator does the following:
  - When the adversary allows delivery of output (`send`, `ssid`$\|1$, ($entry_U$, ct_U, π)) from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, if the honest protocol did not abort as a result of this message:
    1. Get $wit := (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r) \leftarrow \mathsf{ZK.Ext}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, \pi, td_{\mathsf{ZK}})$
    2. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no such entry exists
    3. If U^* is honest, abort
    4. Send (`createentry`, `ssid`, $period$, $entry_U$) to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*

Figure 40: Handling creation of log entries in $\mathcal{S}$

Proof Sketch: Game 19 ≈ Game 20.
The simulation of the operator stays the same. Thus, these games differ either because the simulator aborts or because $\mathcal{F}^{20}$ delivers different output than the real protocol. The probability for the first abort is bound by the probability of breaking simulation-extractability of ZK. The probability of the second abort is bounded by the probability of breaking TS-UF-01-security of TSIG, because extraction provided a valid signature, and whenever that signature is generated, the simulator creates the respective entry in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$. The probability for the third abort is bounded by the EUF-CMA-security of SIG, since extraction provided the secret key of the user, which can be used to win the EUF-CMA game. Note that this is possible because by this point, the secret key of honest users is not used anywhere by the simulator anymore. If $\mathcal{S}^{20}$ does not abort, then $\mathcal{F}^{20}$ delivers the same output as the real protocol: It does not ignore the call because U^* is corrupted, has called `ureg` previously and the messages match, and id is calculated the same way by $\mathcal{F}_{\text{POBA}}$ and the protocol.

Game 21. We again change `createentry`: In this game, when the user is corrupt and the operator is honest, $\mathcal{F}^{21}$ handles this task the same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{21}$ the same way as $\mathcal{S}$ does. This means that in this game, $\mathcal{F}^{21}$ handles `createentry` as a whole the same as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{21}$ handles it the same as $\mathcal{S}$ does. See Fig. 41 for the detailed changes.

Proof Sketch: Game 20 ≈ Game 21.
These games are indistinguishable for the same reason as the two previous games. The simulator does not abort because of simulation-extractability of ZK, TS-UF-01-security of TSIG and EUF-CMA-security of SIG. When it does not abort, the output of the ideal functionality matches that of the real protocol.

Note that $\mathcal{S}^{21}$ equals the final simulator $\mathcal{S}$. Since we showed that Game i is indistinguishable from Game $i+1$ for $i \in \{0, \ldots, 20\}$, it follows that Game 0 (the real protocol) is indistinguishable from Game 21 (ideal execution) and thus Theorem 2 follows. □

Handling of `createentry` in $\mathcal{S}^{20}$

- Upon receiving (`input`, O, (`createentry`, `ssid`, $period$, $entry_O$)) and outputs from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
  1. Run the protocol honestly
- When the adversary allows delivery of output for the operator:
  1. Forward that output to $\mathcal{F}^{20}$

Handling of `createentry` in $\mathcal{S}^{21}$

- When the adversary allows delivery of output (`send`, `ssid`$\|1$, ($entry_U$, ct_U, π)) from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ and after receiving ((`createentry`, `ssid`), O) from $\mathcal{F}^{21}$:
  1. $h := H(period,\ ct_U,\ entry_U)$, $stmt := (ct_U,$ $\mathsf{ek},\ crs_{\mathsf{TSIG}},\ \mathsf{vk},\ h)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, \pi) \neq 1$, ignore this message
  3. Otherwise, get $wit := (\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ \sigma,$ $r) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}},\ \mathcal{R}_{\text{CE}},\ stmt,\ \pi,\ td_{\text{ZK}})$, abort if this fails
  4. Find $(\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ U^*)$ in $\mathsf{L}_{\mathsf{Keys}}^{\mathcal{S}}$, abort if no such entry exists
  5. If U^* is honest, abort
  6. Send (`createentry`, `ssid`, $period$, $entry_U$) to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*

Figure 41: Handling creation of log entries in $\mathcal{S}$