Universally Composable Auditable Surveillance

Valerie Fetzer\textsuperscript{1,4}, Michael Kloß\textsuperscript{2}\textsuperscript{⋆}, Jörn Müller-Quade\textsuperscript{1,4}, Markus Raiber\textsuperscript{1,4}, and Andy Rupp\textsuperscript{3,4}

\textsuperscript{1} Karlsruhe Institute of Technology, Germany
\texttt{\{valerie.fetzer, joern.mueller-quade, markus.raiber\}@kit.edu}
\textsuperscript{2} Aalto University, Finland
\texttt{michael.klooss@aalto.fi}
\textsuperscript{3} University of Luxembourg, Luxembourg
\texttt{andy.rupp@uni.lu}
\textsuperscript{4} KASTEL Security Research Labs, Germany

Abstract. User privacy is becoming increasingly important in our digital society. Yet, many applications face legal requirements or regulations that prohibit unconditional anonymity guarantees, e.g., in electronic payments where surveillance is mandated to investigate suspected crimes. As a result, many systems have no effective privacy protections at all, or have backdoors, e.g., stored at the operator side of the system, that can be used by authorities to disclose a user’s private information (e.g., lawful interception). The problem with such backdoors is that they also enable silent mass surveillance within the system. To prevent such misuse, various approaches have been suggested which limit possible abuse or ensure it can be detected. Many works consider auditability of surveillance actions but do not enforce that traces are left when backdoors are retrieved. A notable exception which offers retrospective and silent surveillance is the recent work on misuse-resistant surveillance by Green et al. (EUROCRYPT’21). However, their approach relies on extractable witness encryption, which is a very strong primitive with no known efficient and secure implementations.

In this work, we develop a building block for auditable surveillance. In our protocol, backdoors or escrow secrets of users are protected in multiple ways: (1) Backdoors are short-term and user-specific; (2) they are shared between trustworthy parties to avoid a single point of failure; and (3) backdoor access is given conditionally. Moreover (4) there are audit trails and public statistics for every (granted) backdoor request; and (5) surveillance remains silent, i.e., users do not know they are surveilled. Concretely, we present an abstract UC-functionality which can be used to augment applications with auditable surveillance capabilities. Our realization makes use of threshold encryption to protect user secrets, and is concretely built in a blockchain context with committee-based YOSO MPC. As a consequence, the committee can verify that the conditions for backdoor access are given, e.g., that law enforcement is in possession of a valid surveillance warrant (via a zero-knowledge proof). Moreover, access leaves an audit trail on the ledger, which allows an auditor to retrospectively examine surveillance decisions.

\textsuperscript{⋆} Research was conducted at Karlsruhe Institute of Technology.
As a toy example, we present an Auditably Sender-Traceable Encryption scheme, a PKE scheme where the sender can be deanonymized by law enforcement. We observe and solve problems posed by retrospective surveillance via a special non-interactive non-committing encryption scheme which allows zero-knowledge proofs over message, sender identity and (escrow) secrets.

**Keywords:** Anonymity · Auditability · Provable Security · Universal Composability · UC · YOSO · Protocols.

**Table of Contents**

1 Introduction ....................................................... 3
2 System Overview .................................................. 11
3 A Formal Model for Auditable Surveillance Systems .... 15
4 Realizing the Model ............................................... 19
5 Application ......................................................... 25
6 Limitations and Future Work ................................. 31
References ............................................................. 32
A Building Blocks .................................................. 37
B Full System Model ............................................... 46
C Auditably Sender-Traceable Encryption (ASTE) ....... 71
D Discussion of Design Decisions ............................... 89
E Security Proof: $\Pi_{AS}$ UC-realizes $F_{AS}$ .............. 91
F Security Proof: $\Pi_{AD}$ UC/YOSO-realizes $F_{AD}$ ........ 113
G Security Proof: $\Pi_{ASTE}$ UC-realizes $F_{ASTE}$ ....... 126
1 Introduction

In user-centric application scenarios such as communication services, electronic payments, internet search engines, etc., there is a strong tension between the need for user privacy and legal requirements or business interests that entail the monitoring of a user’s (meta-)data. This tension is also reflected by the recent European Council Resolution on “Security through encryption and security despite encryption” [21]. On the one hand, there is a strong demand for anonymity and confidentiality supported by the European General Data Protection Regulation. On the other hand, scenario-specific laws and regulations such as the European Council Resolution on the Lawful Interception of Telecommunications [49] or the EU Directive on Anti-Money Laundering and Countering the Financing of Terrorism (AML/CFT) [50], to name just a few, make it necessary to revoke a user’s anonymity or disclose its encrypted transaction data or messages under certain well-defined circumstances, e.g., when a warrant has been issued for a suspect.

The security research community has recognized and addressed the necessity to balance confidentiality/anonymity with accountability. Most proposed solutions follow a variant of the key escrow paradigm [2, 3, 10, 22, 41, 47, 51, 55, 60]. In key escrow systems one or more, typically fixed, trusted authorities (TAs) aka escrow agents, are equipped with (shares of) a trapdoor key which can be used to recover encrypted messages, revoke transaction anonymity, etc. However, this holds the risk that, by corrupting the publicly known TAs, trapdoors can be silently misused, e.g., for mass surveillance or spying on lawful individuals of public interest (e.g., politicians, business leaders, celebrities). Due to these issues, policymakers, security researchers, and practitioners are concerned about deploying key escrow without further measures to prevent or detect misuse, e.g., [1, 35]. Moreover, the lack of transparency concerning the lawful usage of trapdoors leaves citizens with the subconscious feeling of being under permanent surveillance.

To make surveillance actions more transparent and accountable, recent work [32, 24, 52] has discovered the usefulness of distributed ledgers. Here, judges, law enforcement, and companies publish commitments to information about surveillance measures on the ledger and can provide zero-knowledge proofs that they behave according to the laws. However, these proposals do not enforce that, in order to access user trapdoors, evidence must be put onto the ledger first. That means, a secure and accountable escrow and disclosure of end-to-end encryption keys is not considered. Hence, trapdoors kept by a company are still at risk of being covertly misused without leaving any trace.

Very recently, Green et al. propose a misuse-resistant surveillance scheme [34] which compels law enforcement to leaving a warrant on the ledger in order to disclose the encrypted communication of a suspect. To this end, the authors build on extractable witness encryption (EWE) [30], instead of a key escrow scheme, where the publication of a warrant serves as the key (aka witness) to decrypt the communication. Unfortunately, it seems implausible that extractable
witness encryption (for general NP languages) does actually exist [26]. Moreover, they prove that any protocol secure in their model would already imply extractable witness encryption (for a highly non-trivial language). The goal of our system is to combine ledger-based auditability of surveillance actions with corruption-resilient key escrow mechanisms. Similar to [34], leaving a warrant on the ledger to access trapdoors is enforced, but without relying on extractable witness encryption.

Goyal et al. [33] model storing secrets on a blockchain (via secret-sharing them among members of an evolving committee) and retrieving them under some condition. If we apply that idea to our scenario, we could secret-share user-specific trapdoors at committees who only release their shares if they find a warrant on the ledger requesting their specific shares. Benhamouda et al. [8] also model storing of a secret on a blockchain, but additionally make committee members anonymous until they finished their work to prevent targeted corruption. Both methods [33, 8] can result in a lot of overhead for the committee who potentially has to manage millions of secrets. To reduce the committee’s workload during the handover phase, we take this idea and combine it with a key-escrow approach. Instead of secret-sharing millions of trapdoors on the blockchain, we only secret-share one item: A secret key for a threshold encryption scheme. The trapdoors are not secret-shared on the blockchain, but encrypted under a public key and stored off-chain to reduce blockchain workload. The secret key for that ciphertext is secret-shared on the blockchain and instead of directly retrieving a secret from the blockchain, law enforcement can request the decryption of the ciphertext. To increase security, only a part of the trapdoor is decryptable by the blockchain committee; the other part is stored offline at the system operator. Thus, the system remains secure even if the blockchain committee’s majority is corrupted. See Appendix D for a discussion of important design decisions.

While some other systems [44, 27] only enable prospective surveillance, where law enforcement must prepare surveillance of each user individually before this user conducts any transactions, we achieve retrospective surveillance, where law enforcement can also access transactions that were conducted in the past. We additionally model this functionality as a building block in the UC framework and prevent users from learning whether their trapdoor was retrieved or not (in contrast to [8, 33]). Unlike [34], we achieve retrospective surveillance without implying EWE. Moreover, the efficiency estimate of our application shows that our system is realizable and scales favorably: The judge is only involved in granting warrants, law enforcement is only involved when surveilling users and the work on the blockchain only depends on the number of surveilled users, not the number of registered users (cf. Section 4.3). We thus present the first UC-secure proof of concept. Unfortunately, it is not practically efficient due to large ciphertexts (cf. Section 5.3).

Additionally, we add the possibility to audit the surveillance decisions of law enforcement. A special party, the auditor, has the power to retrospectively examine law enforcement’s surveillance decisions and inform the public about possible

---

5 Even (non-extractable) witness encryption currently has no efficient constructions.
abuse. Since this party is very powerful, care must be taken as to who takes on this role; for example, a neutral investigative committee might be suitable.

1.1 Contribution

We formalize an auditable surveillance system as a building block which can be used to enhance many different (existing) applications with the capability to revoke a user’s anonymity or to reveal a user’s transaction data under the condition that a law enforcement agency has a court-signed warrant that legally empowers them to do so. Our auditable surveillance system is the first to combine key escrow with accountability without requiring unlikely assumptions (e.g., EWE) while scaling well. We accomplish this through the following three contributions.

First and foremost, we provide an abstract ideal functionality for auditable surveillance, \( F_{AS} \). The functionality \( F_{AS} \) serves as a building block which allows to enhance protocols with auditable surveillance with relative ease. In particular, this provides a basic target functionality to realize and serves as a separation between the low-level implementation of the auditability mechanism, and the high-level decision of adding auditability to protocols, e.g., choosing what data to make available to law enforcement in anonymous electronic payment systems with auditable surveillance. We formalize our auditable surveillance system in the Universal Composability (UC) framework [17, 16], which ensures that the system’s security and privacy guarantees still hold if the system is run in combination with many different other protocols.

Our second contribution is the protocol \( \Pi_{AS} \) which UC-realizes \( F_{AS} \). It uses several cryptographic building blocks like commitments, signatures and zero-knowledge proofs and is based on an ideal functionality for auditable decryption, \( F_{AD} \), for managing the secret-shared secret key for the threshold encryption scheme and answering decryption queries. Our modeling of this building block achieves auditability of requests for secrets and privacy regarding which secrets are released. The auditable decryption functionality \( F_{AD} \) is also modeled in the UC framework to enable a flexible use of this building block and we provide a protocol \( \Pi_{AD} \) that UC-realizes it. We cast our protocol \( \Pi_{AD} \) in the YOSO (You-Only-Speak-Once) model [28] (see Section 4.3 for a brief introduction). In this model, protocols are executed by roles, where each role is only allowed to send messages once. Which party is executing a specific role is hidden until it sends its messages. This prevents targeted corruption of parties that comprise the current committee, and thus allows for leveraging global honest-majority (for the set of all nodes operating the blockchain) assumptions for smaller committee sizes, and achieving security against mobile adaptive adversaries.\(^6\)

Our third contribution is a (toy) ideal functionality, \( F_{ASTE} \), for Auditably Sender-Traceable Encryption (ASTE). This functionality demonstrates the usefulness of \( F_{AS} \) as a building block. Effectively, \( F_{ASTE} \) allows a registered user to anonymously encrypt a confidential message to another registered user, while

\(^6\) Mobile adversaries can adaptively corrupt and uncorrupt parties as long as they do not exceed a certain threshold of simultaneously corrupted parties.
ensuring that law enforcement is able to deanonymize the resulting ciphertext. In our toy example, law enforcement does not learn the message of a ciphertext, fully separating “content” from “identity”. But it is easy to modify the example, so that law enforcement learns any function of message and identity. At first glance, this may be a trivial application of encryption and zero-knowledge. However, to attain retrospective surveillance which is somewhat practical, we need to combine non-interactive non-committing encryption (NINCE) with zero-knowledge proofs, which is highly non-trivial (cf. Section 1.2).

To summarize our contribution:

1. We identify and define an ideal UC-functionality, called $F_{AS}$, which acts as a building block for auditable surveillance systems or more generally auditable access systems.
2. We realize this functionality in a setting where the deployment of such systems is of interest (namely, the “blockchain space”). Therein we also specify a UC-functionality $F_{AD}$ for auditable decryption, which may be of independent interest. We stress however, that the realization of $F_{AS}$ is not in any way restricted to this setting.
3. We demonstrate the applicability of $F_{AS}$ by building $F_{ASTE}$ on top of it, and provide techniques to overcome the challenges posed by retrospective surveillance namely, techniques for ZK-compatible NINCE in the PROM.

1.2 Overview of Technical Challenges in Building Applications

To be used in an application, the auditable surveillance functionality $F_{AS}$ must provide a suitable interface. In most applications, users need to be able to prove that they escrowed a secret, e.g., a secret for the current period in our messenger application ASTE. A first idea to realize that, is for $F_{AS}$ to provide a digital signature on user identity, escrowed secret and current period to the user. However, formulating a usable (let alone zero-knowledge-compatible) UC signature scheme turns out to be a daunting task, since signatures in UC are riddled with subtleties [4, 18, 45, 13], and indeed many modeling artifacts occur.

To circumvent such problems, $F_{AS}$ directly provides the possibility for users to prove statements about their identity $uid$, their escrowed secret $secret$, some validity period $vper$ and other information. Such a proof can show a statement of interest w.r.t. $vper$, for a witness which includes $(uid, secret)$, and the proof ensures that the secret $secret$ for $uid$ is stored for period $vper$. Overall, this approach is more general and easier to use.

Our Auditably Sender-Traceable Encryption (ASTE) functionality encapsulates the main challenges encountered with auditable systems:

Privacy: Different information must be hidden from different parties: For $F_{ASTE}$, message recipients must learn the message, but not the sender's identity. Law enforcement must not learn the message, but must learn the sender’s identity.

Soundness (of Surveillance): Despite anonymity of ciphertexts in $F_{ASTE}$, it must be infeasible to produce ciphertexts in the name of another user.
Retrospective Surveillance: The surveillance requests of law enforcement essentially behave like adaptive partial corruptions.

The second point (soundness) can make good use of the non-interactive proving capabilities of $\mathcal{F}_{\text{AS}}$, which also greatly helps in achieving the first (privacy). However, the last point (retrospective surveillance) is surprisingly difficult to achieve, even with $\mathcal{F}_{\text{AS}}$. Indeed, it is a case of the well-known “simulator commitment problem” [39]: In the ideal execution, the simulator must generate a ciphertext for an unknown message (due to confidentiality) and an unknown user (due to anonymity). However, when law enforcement (lawfully) deanonymizes the user’s identity $uid$ in that ciphertext, it must be correct. In other words, the identity $uid$ is unknown to the simulator when it generates a ciphertext $c$ (which “commits” to $uid$). Yet, when law enforcement obtains the escrow secret, all (affected) ciphertexts must correctly decrypt to the user’s identity $uid$. If non-interactive decryption for law enforcement is assumed, then the simulator must retroactively choose the identity $uid$ of $c$. This asks for a (form of) non-interactive non-committing encryption (NINCE), which is known to be impossible in the standard model [48]. Thus, we rely on a NINCE-like construction in the programmable random oracle model (PROM). However, this entails a well-known problem, namely, that it is not possible to prove statements about (random) oracles with zero-knowledge proofs for NP — NP statements cannot have oracles. To overcome this, we use a non-trivial construction of NINCE which incorporates cut-and-choose techniques to obtain black-box proofs for statements over the NINCE-encrypted values (and even the encryption secret keys). In light of an apparent necessity for zero-knowledge-compatible NINCE, we consider this construction as part of our toolkit for basing applications on $\mathcal{F}_{\text{AS}}$.

1.3 Related Work

Key escrow. Since the 1990s, many papers, e.g., [41, 60, 51], have been dealing with different variants of key escrow mechanisms in various domains, where key material is deposited with one or more trusted parties who can then decrypt targeted communications or access devices. In particular, key escrow has also been applied in the scope of e-cash [10] to balance anonymity and accountability. Also, more recent work follows this paradigm. For instance, in [2] the authors propose protocols for secure-channel establishment of mobile communications that offer a session-specific opening mechanism. A session key is escrowed with $n$ authorities which all need to agree for recovering the session key. The system comes with some addition of security guarantees, e.g., non-frameability. The work [55] considers lawful device access while protecting from mass surveillance. The authors propose the use of self-escrow passcodes which are written to the device itself and can only be retrieved by means of physical access, e.g., via dedicated pins.

Accountable access. Some works have tried to extend key escrow with basic accountability features. We compare ourselves with the most relevant of them
<table>
<thead>
<tr>
<th>Paper</th>
<th>Lawful-only&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Retro-spective&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Silent&lt;sup&gt;3&lt;/sup&gt;</th>
<th>No EWE needed&lt;sup&gt;4&lt;/sup&gt;</th>
<th>Public Statistics&lt;sup&gt;5&lt;/sup&gt;</th>
<th>UC-secure&lt;sup&gt;6&lt;/sup&gt;</th>
<th>Flexible Framework&lt;sup&gt;7&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[44], [27]</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>[22]</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>[11]</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>[47]</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>partially&lt;sup&gt;8&lt;/sup&gt;</td>
</tr>
<tr>
<td>[38]</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>[44]</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Our System</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

1. Whether a warrant is needed for surveillance actions.
2. Whether surveillance is retrospective or only prospective.
3. Whether surveillance is silent, e.g., users are not aware that they are surveilled.
4. Whether the system does not assume the existence of EWE (extractable witness encryption).
5. Whether the system supports public statistics.
6. Whether the system is UC-secure.
7. Whether the work contains a framework that can be used for many different applications.
8. The system support a limited set of applications.

Table 1: Comparison of our system with the most relevant related work on accountable surveillance systems

In Table 1. A more detailed comparison with [34] can be found in Section 5.4, after our application ASTE is presented.

In [3] an anonymous yet accountable access control system is proposed. Regarding accountability, the user needs to escrow its identity with a TTP which is revealed by the TTP if some previously agreed condition bound to the ID using verifiable encryption with labels (where the condition is encoded as label) is satisfied. Liu et al. [47] propose an accountable escrow system focusing on encrypted email communication. In their system users escrow their decryption capability (instead of their private key), essentially by means of a 3-party Diffie-Hellman key exchange, to trusted custodians. Custodians perform decryptions upon request by means of their own private keys and are trusted to log each decryption request to hold the government accountable. Still, the private keys of custodians can be stolen or they can be corrupted in particular as they are well-known to government organizations. The works [44] and [27] deal with auditable tracing techniques in the context of e-cash and cryptocurrencies, respectively. The underlying idea is to provide a user either with a randomized version of the authority’s public key, where the corresponding secret key is the revocation trapdoor, or a completely random key, which is useless for tracing. The user cannot tell which key it received until later when the authority is enforced to reveal this. The big disadvantage of their approach is that the authority has to decide in advance (e.g., at the beginning of each month) which users should be traced. This could result in practical issues, e.g., when money laundering is suspected, but the transactions of suspects cannot be traced since tracing was not turned on for them. In [22] a “mutual accountability layer” is added to systems to make operators accountable for opening key-escrowed user transactions. However, this accountability feature only results in the current key escrow committee learning that some transaction was opened. The lawfulness of opening a transaction is
not verified and information about opening requests are not persistently and publicly stored.

Some proposals try to avoid key-escrow and its misuse potential from the outset. Very recently, a misuse-resistant surveillance scheme with retrospective exceptional access to end-to-end encrypted user communication has been proposed in [34]. Instead of building on key escrow, users are forced to (additionally) encrypt their messages with extractable witness encryption [31]. Loosely speaking, witness encryption allows to define a policy under which a ciphertext can be decrypted. In their scheme, this is the case when a warrant for the corresponding user signed by a judge has been published on a public ledger. The major disadvantage of their approach is that it is implausible that extractable witness encryption schemes actually exist [26].

In [38], the authors also abstain from using key escrow. Instead, given a warrant, a user has to (verifiable) reveal the transactions of interest itself to the judge. Unfortunately, this prevents silent investigations and leaves a suspect with the option to deny cooperation.

Several works deal with the collection of auditable logs of surveillance actions. In [6] the authors propose a distributed auditing system for CALEA-compliant wiretaps. The idea is to add Encryptor devices to the wiretaps which send encrypted audit records to a log. With the help of the log, audit statistics can be computed from ciphertexts using homomorphic encryption. Kroll et al. in two not formally published manuscripts [42, 43] propose different systems for accountable access control to user data. Here, law enforcement needs to interact with a set of decryption authorities to decrypt user records, for which a single encryption key is used. Accesses are logged by an auditor party with whom the other parties need to interact continuously in order to confirm the different protocol steps. No end-to-end encryption of user data or the revocation of anonymous records is considered. Also, formal security model and proofs are missing. The work [24] extending [32] uses ledgers to collect accountable information about surveillance action and a hierarchical form of MPC to compute aggregate statistics. However, a secure and accountable escrow and disclosure of user trapdoors is not considered. In particular, if such trapdoors are kept by a company, they can still be misused by the company, stolen by an intelligence agency, etc. without leaving any trace on the ledger. The work [52] addresses some of the issues of [24] like, e.g., that government agencies and companies are trusted to regularly post (correct) information on the ledger. This is done by introducing an independent party called Enforcer who serves as the conduit for interactions between them and ensure compliance. However, for this it is assumed that government agencies and companies do never directly communicate with each other. Moreover, they do not make use of ledgers to control access to user trapdoors to disclose end-to-end encrypted communication or revoke anonymous transactions.

Finally, there are number of rather unconventional proposals to impede mass surveillance. In [7], the authors introduce the concept of translucent cryptography which, based on oblivious transfer and without relying on key escrow, allows law enforcement to access encrypted messages with a certain probability \( p \). In
[58], a system is proposed where law enforcement needs to provide a hash-based proof of work in order to recover an encrypted message, intentionally resulting in a high monetary cost (e.g., $1K-$1M per message). The work [57] considers exceptional access schemes for unlocking devices such as smartphones. To unlock a device, law enforcement needs to get approval by a set of custodians and subsequently locate and get physical access to a randomly selected set of delegate devices to obtain an unlock token. This requires both human and monetary resources of the law enforcement agency. In [56] Scafuro introduces the concept of break-glass encryption for cloud storage where the confidentiality of encrypted cloud data can be revoked exactly once for emergency reasons. This “break” is detectable by the data owner. The author’s construction relies on trusted hardware and is a feasibility result rather than a practical solution. Persiano, Phan and Yung [53] recently introduce the concept of anamorphic encryption. The idea behind this is that even an attacker who can dictate the messages sent and demand the surrender of a decryption key (e.g., a government), cannot prevent a second, hidden message from being sent along that only the dedicated recipient can decipher. While this means that one cannot prevent the exchange of hidden messages despite of surveillance, law enforcement agencies still consider surveillance as a useful measure as not all criminals have the knowledge or skill set to exploit this fact or find it convenient.

Storing secrets on a blockchain. There are prior works which model the capabilities of storing secrets and retrieving them under certain conditions thereby replacing the need for extractable witness encryption by relying on a blockchain.

In the recently published “eWEB” system [33] a dynamic proactive secret sharing (PSS) scheme with an efficient handover phase is constructed and used in a black-box way to store and retrieve secrets on a blockchain. Their system is secure against an adversary that statically corrupts less than half of the blockchain nodes. However, the members of the current committee are publicly known and the identifier of a retrieved secret is revealed to the general public. Hence, in our case, a user would learn that law enforcement is exposing its transactions, which should be prevented.

The system by Benhamouda et al. [8] uses a similar approach. They introduce an evolving-committee PSS scheme instead, where the selection of the next committee is a part of the secret-sharing scheme itself and the members of the current committee remain anonymous (to anyone, including the members of the previous committee) until they finished their work and hand over the role to the next committee. This enables the system to handle a mobile adversary corrupting $\approx 29\%$ of blockchain nodes. However, they do not clearly state how their PSS scheme can be used to retrieve secrets from the blockchain, in particular it is unclear who learns which secrets are retrieved, whether the current committee learns this witness and whether the secret is revealed to the public or not.

Our usage of a blockchain has similarities to [33, 8], in particular we also use the anonymous committees from [8], but we additionally model the system as a building block in the UC framework, which [33, 8] do not. Also, we prevent users from learning whether their secret was decrypted or not (in most cases).
Erwig, Faust and Riahi [23] propose a standalone protocol for threshold decryption in the YOSO model. Our building block for auditable decryption is similar to their protocol, but augmented to satisfy our auditing needs and formalized in the UC-model instead of using game-based security notions.

Brorsson et al. [11] recently published PAPR, an anonymous credential scheme with a retrospective auditable surveillance feature. PAPR and our work both model retrospective auditable surveillance with a detailed UC framework, utilizing techniques such as bulletin boards/blockchains, YOSO, secret-sharing, ZK, and TPKE. Regarding the committee structure, our system differs from PAPR as the sets of users and committee member candidates are distinct, while PAPR’s sets are identical. However, it is worth mentioning that PAPR proposes this separation as future work. Notably, our system supports silent anonymity revocation, whereas anonymity revocation in PAPR is inherently non-silent. Additionally, we implement different validity periods, allowing law enforcement to revoke a user’s anonymity only for specific periods, offering a more nuanced approach compared to PAPR, where revocation affects all credential showings.

2 System Overview

In this section, we provide an overview of our system. We start by introducing the parties, followed by a high-level description of their interactions. Lastly, we give a brief discussion of its (intuitively) captured security properties.

2.1 Parties

Our system consists of the following parties:

**System Operator:** The system operator SO operates the system (e.g., anonymous payment system, confidential instant messenger service) that is enhanced with auditable surveillance.

**Law Enforcement:** Law enforcement LE can access a user’s escrowed secrets if it is in possession of a valid warrant.

**Judge:** The judge J grants or rejects warrants.

**Auditor:** The auditor AU can access information about all used warrants.

**Users:** The set \( \{U_i\} \) of users who want to make use of the application provided.

**Nodes:** The set \( \{N_i\} \) of nodes that is available to execute assigned roles on the blockchain (the shareholder committee).

There is one each of system operator, law enforcement, judge and auditor, but there may be arbitrary many users and blockchain nodes. We consider static corruption of SO, LE and the users \( U_i \). For the nodes \( N_i \) we achieve mobile adaptive corruption since each role a node can play is cast in the YOSO model [28] and only sends messages once. J and AU\(^7\) are assumed to always be honest. Apart from the explicitly modeled parties, some portions of our system are available for everyone, for example reading information from the blockchain.

\(^7\) While a corrupt AU would learn all warrants, it is not possible for a corrupt AU to convince a third party of false claims about those warrants.


Fig. 1: High Level System Overview. The numbers represent the intended order of task execution.

### 2.2 High-Level System Overview

An overview of the tasks the different parties can execute can be found in Fig. 1. We now describe the intended usage of our system. In the following we focus on a specific application and instantiation to make the description clearer. We apply our functionality $F_{AS}$ to an anonymous message transfer service and instantiate it with the protocols $\Pi_{AS}$ and $\Pi_{AD}$ that utilize a blockchain.\(^8\) Note that of course different applications and instantiations are possible as well.

To initialize the system, each party generates their respective cryptographic keys. The public keys of SO, J and AU are posted on the blockchain and available to everyone. The blockchain selects its first committees and creates a threshold encryption keypair, whose public key is available to everyone and whose secret key is secret-shared among the committee members.

Users need to register themselves with SO and create an account to use the system. Each period (e.g., monthly) the user and SO create together a fresh escrow secret that could be used to expose the user’s identity in messages sent during that period. The secret is then secret-shared into two partial secrets: One part is directly stored at SO\(^9\) and the other part (only known to the user) is encrypted under the committee’s threshold public key. The ciphertext itself reveals nothing about the user’s identity, but it is linked to the user’s identity and the period the secret is valid for with a zero-knowledge (ZK) proof. After depositing a secret for the current period, the user can use the anonymous messenger service during that period. To use the anonymous messenger application, the sender would prove in

---

\(^8\) The features of a blockchain include publicly viewable and non-editable information as well as easy committee formation. Alternatively, an append-only bulletin board can be used. In this case, a committee has to be formed by some other means.

\(^9\) Since we use a Blum coin toss to jointly generate the secret, SO knows a share of the secret anyway.
zero-knowledge that it has stored a secret for the current period and appended its identity (but encrypted under its escrow secret for the current period) to the current message. This proof can be verified by anyone, in particular the receiver and SO, where the latter can block non-complying messages.

If the law enforcement agency LE has a legitimate interest in disclosing one or multiple user secrets, it can request a warrant from the judge J. Each warrant consists of a set of sub-warrants and each sub-warrant states which user should be surveilled in which period. Additional information is included in each sub-warrant, for example the name of the court, the reason for surveillance, etc. If J signs the warrant, LE can get information about each user from SO, i.e., the partial secret stored by SO and the ciphertexts containing the other part of the secret. With a signed warrant and the additional information from SO, LE is able to retrieve the partial secrets for the users and periods in that warrant from the blockchain. In particular, LE convinces the current committee in zero-knowledge that it indeed has a warrant signed by J for that set of ciphertexts. Upon verifying the proof, the committee then decrypts the ciphertexts and reveals the other partial secrets to LE, who can then reconstruct the users’ full secrets. Given the (anonymous) message data of the messenger system operator, LE can now match the previously anonymous messages to users in the warrant (but only for messages sent during the period specified in the warrant).

To prevent misuse of the system or even mass surveillance, several countermeasures are supported by the system. Firstly, a system-wide policy function prevents any warrants violating that policy. For example, it may disallow surveillance of an individual user for more than 12 months (by a single warrant). Secondly, LE needs to be in possession of a valid warrant signed by J to request a secret and the blockchain committee first verifies the validity of this warrant before decrypting a secret. There exists the risk that the signing key of the judge may get stolen and an attacker could use the stolen key to sign a warrant itself and thus retrieve some secrets. We cannot prevent this entirely (other than suggest that judges secure their secret keys properly), but have mitigation measures. To retrieve a decrypted secret from the blockchain, the request needs to publish some information about the respective warrant on the blockchain, thus leaving visible proof that such a request took place. What information about each warrant should be publicly available is declared by a system-wide transparency function. Choosing a suitable transparency function for the system can be a challenging task since one needs to balance the public’s desire for information, the privacy of surveilled users, and LE’s need for silent surveillance. The transparency function should, for example, never publish the names of the surveilled individuals. But it may, for example, publish the number of different users in the warrant, the periods in which the retrieved secrets are valid and the name of the court that signed the warrant. Assuming the latter information is published, each court can monitor the blockchain and upon seeing more warrants signed by its key than they actually signed, they can
detect key theft and henceforth take appropriate measures.\textsuperscript{10} Thirdly, assuming an appropriate transparency function, anyone can get useful information about all enforced warrants through publicly available statistics. For example, if the transparency functions reveals the number of different users in the warrant, the number of currently surveilled users becomes public. If this number suddenly skyrockets, it is an indication of misuse of power. Fourthly, the auditor AU has the capability (as it can access the “full” warrant information on the blockchain) to conduct a detailed investigation of enforced warrants. It may check for any peculiarities and prove statements about the stored warrants to an arbitrary entity without revealing any further information. For instance, such a statement might be that none of the warrants issued within a certain period of time involve a certain user.

2.3 Security Properties and Trust Assumptions

We now summarize the desired security and privacy properties of our building block for auditable surveillance in an informal and intuitive manner:

- (Non-colluding\textsuperscript{11}) users are not aware whether LE issued a request to recover one of their secrets.
- To use the Prove task (cf. Fig. 1) for an application on top, e.g., to prove some statement involving the user’s identity, the user needs to have escrowed a secret for the respective period.
- The privacy guarantees of the application on top are only breached if a warrant was granted. And even in that case, only the users covered by the warrant have (some of) their data exposed.
- LE can only request secrets for warrants that comply with the system policy.
- After LE requested a decryption by the blockchain, the publicly available information about the warrant (for statistical purposes) and the information about the warrant that is only available to AU are permanently stored on the blockchain and can not be modified or removed afterwards.
- Anyone is able to retrieve publicly available statistics on all enforced warrants.
- AU has the capability to provide the general public with provably correct statistics about the enforced warrants.\textsuperscript{12}
- Even if SO and LE collude, they can not expose any escrowed user secrets.
- Likewise, even if a majority of the blockchain nodes collude, they can not reconstruct a user’s secret.

\textsuperscript{10} Our system actually only supports a single non-revocable judge key to keep the model from being overly complex, but the extension to several different and revocable judge keys is straightforward.

\textsuperscript{11} This security property holds for a user colluding with other users and blockchain nodes but not one colluding with SO or LE.

\textsuperscript{12} Since AU has access to the full warrants, its statistics can be more detailed than those the general public can compute. AU could even prove to third parties (e.g., a parliament) facts about specific warrants without revealing the full warrant.
We assume the judge J and auditor AU to be honest. Since AU has the power to read all warrants in the clear, it is a very powerful entity that could potentially misuse that power. In reality, one could decentralize that trust by having multiple auditor parties that utilize threshold cryptography or multi-party computation. Since we trust the jurisdiction, we also model J as an honest party and assume that it honestly follows the protocol. Warrants that were granted but legally not justified can be detected upon request to AU. The existence of judge-signed warrants also ensures that LE can also only request user information from SO for which it has a signed warrant.

Although SO is corruptible in our system, we implicitly have to trust it in some aspects: We assume that SO cooperates with other parties and responds to messages. A SO that ignores messages from other parties could bring the system to a halt or deny individual users’ participation in the system. Since SO has a monetary motivation to keep the system running for a long time, we believe these assumptions to be reasonable. Note that while a corrupt SO cannot send false data (e.g., pretend that a ciphertext belongs to another honest user), it can omit some data (e.g., data from colluding corrupt users) when sending it to LE. This is a general problem that can be discouraged through laws.

As an exemplary application that is enhanced by auditable surveillance, we consider Auditably Sender-Traceable Encryption (ASTE). This application achieves (among others) the following core security guarantees:

- Any ciphertext which decrypts successfully (to \( m \neq \bot \)) for an honest user can be deanonymized by LE.
- Finding ciphertexts which falsely deanonymize to an honest user is infeasible.
- Plaintext and identities remain secret to parties which are not allowed access to them. That is: Without a warrant, LE learns nothing from a ciphertext. With a valid warrant, LE can deanonymize the user, but the plaintext remains hidden.

We give LE only the ability to deanonymize instead of reading the message (and deanonymizing), since corrupted users may use secure encryption to encrypt their messages anyway. However, by straightforward modifications, LE may learn any function of the message and the user’s identity.

3 A Formal Model for Auditable Surveillance Systems

In this section we introduce the functionality \( F_{AS} \), our formal model for our auditable surveillance system, which is independent of the application. We model our system in the Universal Composability (UC) framework [17, 16]. For a very brief introduction to UC, see Appendix A.8, and for some writing conventions see Appendix B.1.

We first introduce some variables that are essential for the functionality in Table 2. We describe how our auditable surveillance system is modeled in the UC framework by presenting the ideal functionality \( F_{AS} \) in Figs. 2 and 3.
pid  The party identifier for the UC framework (e.g., a physical identifier)
uid  The (self-chosen) identity of a user
secret The escrowed secret
vper  The validity period for a single secret (e.g., the current month)
W    A warrant from LE to retrieve (multiple) user secrets with $W = (W_1, \ldots, W_v)$ and $W_i = (uid_i, vper_i, meta_i)$
v    number of subwarrants $W_i$ inside a warrant $W$
meta  Meta information about a warrant. Some of it is public information and some is only meant for the auditor
Wpub The information about the warrant that should be known to the public

Table 2: Variables used in $F_{AS}$

that the functionality implicitly checks the pid of calling parties: If the functionality description states “Input P: (…)” for $P \in \{SO, AU, J, LE\}$, then the functionality first checks if the calling party has the correct role and ignores the message if the role does not match. We now also sketch the individual tasks.

System Init and Party Init. These tasks are just for initializing the system and other parties.

User Registration. To participate in the auditable surveillance system (and to use the application on top) the user needs to create an account with SO. It is ensured that each user can only create one account. The user can choose an unique identity (uid) under which it will be known henceforth.

Store Secret. Each period (e.g., monthly) the user needs to deposit a new secret. Since $F_{AS}$ is a trustworthy incorruptible entity, we do not need to encrypt it and directly store the secret inside $F_{AS}$. $F_{AS}$ first checks that the user is registered in the system and has not yet stored a secret for the current period. Then, $F_{AS}$ draws a fresh secret (so the user can not influence what his secret will be) and stores it in its internal storage. After this task is finished, the user can now use the application, which is not modeled directly in $F_{AS}$. For an example on how to use $F_{AS}$ for Auditably Sender-Traceable Encryption, see Section 5.

Request Warrant. In this task LE can request a warrant from J. The functionality itself first checks if the proposed warrant $W$ complies with the system’s policy function $f_p$. Then it gives J the opportunity to approve ($b = 1$) or deny ($b = 0$) that warrant. $F_{AS}$ also ensures that each warrant is only processed once (this ensures that the statistics calculated later will be correct).

Get Secrets. With a granted warrant LE can now retrieve the secrets for all $(uid, vper)$ pairs inside the warrant. $F_{AS}$ first verifies that the warrant was approved by J and then outputs all secrets corresponding to that warrant to LE.

Get Statistics. Every party of the system can query this task to enable the general public to access statistics about the warrants. For every warrant granted by J the transparency function $f_t$ is computed to get the publicly available information $W_{pub}$ about that warrant. The publicly available information about each warrant are then returned to the party asking for statistics.
## System Parameters:
- $f_t$ — Transparency function. Outputs public information of a warrant. Interface is $W^{pub} \leftarrow f_t(W)$.
- $f_p$ — Policy Function. Checks whether a given warrant is allowed by system policy. Interface is $[0, 1] \leftarrow f_p(W)$.
- $S$ — Space of secrets
- $\mathcal{R}$ — an NP relation for statements about stored secrets: Contains $\langle \text{stmt}_R, \text{wit}_R \rangle \in \mathcal{R}$ pairs where $\text{stmt}_R$ is a statement and $\text{wit}_R$ is a witness for that statement
- System pids: $\text{pid}_S, \text{pid}_J, \text{pid}_A$

## Functionality State:
- $L_i$: List of initialized parties (initially empty). Contains $(\text{pid}_i)$ entries.
- $\text{L}_U$: List of registered users (initially empty). Contains $(\text{uid}_i, \text{vper}_i, \text{secret}_i)$ entries.
- $L_{W}$: List of warrants that were requested by LE from J (initially empty). Entries are of the form $(W, b_i)$, where $b_i$ is a bit that states whether the warrant was granted or denied by the judge.
- $L_{\text{secret}}$: List of proofs (initially empty). Entries are of the form $(\text{stmt}_R, \text{wit}_R, \pi, b_i)$, where the bit $b_i$ states whether the relation is fulfilled or not.

## System Init:
- Input SO & J & AU: $(\text{Init, SO} | J | \text{AU})$
  - If this is the first time, this task is invoked, do whatever is stated in “Behavior”. If this is not the first time this task is invoked, ignore the messages.
  - Ignore all other messages until this task has been invoked.
- Behavior:
  1. Create empty lists $L_i, L_U, L_{\text{secret}}$, and $L_{W}$.
  2. Send $(\text{Init})$ to the adversary
  3. Add $\text{pid}_SO$ and $\text{pid}_J$ to $L_i$.
- Output SO & J & AU: $(\text{InitFinished})$

## Party Init:
- Input some Party P: $(\text{PInit})$
- Behavior:
  1. If $\text{pid} \not\in L_i$, store $\text{pid}$ in $L_i$
  2. Send $(\text{PInit})$ to the adversary
- Output to P: $(\text{PInitFinished})$

## User Registration:
- Input U: $(\text{Register, uid})$
- Input SO: $(\text{Register, uid'})$
- Behavior:
  1. As soon as U gave input, send $(\text{Register, pid, uid})$ to the adversary.
- Wait for (Ok) from the adversary and input from SO before continuing.
  2. If $\text{uid} \neq \text{uid}'$, abort. (wrong inputs)
  3. If $(\text{pid} \cdot) \in L_U$, abort. (User already registered.)
  4. If $(\cdot, \text{uid}) \in L_U$, abort. (Identity already taken.)
  5. Store $(\text{pid}, \text{uid})$ of the user in list of registered Users $L_U$
  6. Store $\text{pid}$ in $L_i$
- Output U & SO: $(\text{Registered})$

## Store Secret:
- Input U: $(\text{StoreSecret, uid, vper})$
- Input SO: $(\text{StoreSecret, vper'})$
- Behavior:
  1. If SO is corrupted (and U is honest): As soon as U gave input, send $(\text{StoreSecret, vper})$ to the adversary and then wait for input from SO
  2. If vper $\neq$ vper', abort
  3. Check if user is registered: If $(\text{pid}, \text{uid}) \not\in L_U$, abort.
  4. Check if user already registered a secret for the current validity period: If there exists an entry $(\text{uid}, \text{vper}, \cdot)$ in the list of stored secrets $L_{\text{secret}}$, abort
  5. Generate secret: secret $\leftarrow S$
  6. Store $(\text{uid}, \text{vper}, \text{secret})$ in the list of stored secrets $L_{\text{secret}}$
- Output U: $(\text{SecretStored, secret})$
- Output SO: $(\text{SecretStored, uid})$

## Request Warrant:
- Input LE: $(\text{RequestWarrant, W})$
- Behavior:
  1. Check if policy function allows that warrant: If $0 \leftarrow f_p(W)$, abort (Warrant not allowed by policy function).
  2. Output J: $(\text{RequestWarrant, W})$
  3. Input J: $(b)$
  4. Behavior:
    1. If there already exists an entry $(W, \cdot) \in L_{\text{W}}$, abort (Warrant already processed).
    2. If SO is honest: If $b = 1$, send $(\text{RequestWarrant}, f_i(W), |W|, v)$ to the adversary and wait for message (Ok) from the adversary.
    3. If SO is corrupted: Send $(\text{RequestWarrant}, W, b)$ to the adversary and wait for message (Ok) from the adversary.
  4. Append $(W, b)$ to $L_{\text{W}}$
- Output LE: $(\text{RequestWarrant, b})$

---

**Fig. 2:** The ideal functionality $F_{AS}$
Audit. When AU gets tasked with calculating detailed statistics or with investigating the case of a specific user, it can call this task. AU is then provided with all warrants that were approved by J. Since we assume AU to be a trustworthy entity (or a group of entities that perform this task in a multi-party computation) we can provide AU with the warrants in the clear. The actual execution of AU’s task takes place outside our system, we only provide AU with the necessary information.

Prove and Verify. The zero-knowledge proof interface is used to build applications on top of $F_{AS}$. The Prove task allows a prover to generate a proof $\pi$ for a statement $stmt_R$ in some NP-relation $R$, where the witness $wit_R$ includes the
user’s identity and escrow secret for the chosen period. The Verify task allows to check the validity of a proof. As an example, a user may prove that it correctly encrypted its identity under its escrow secret key. Since we do not limit our system to a single application, this generic proof/verify interface enables the flexible use of different applications.

Remark 1. In the description of $F_{AS}$ (c.f. Figs. 2 and 3) there are several messages of the form “Send (value) to the adversary”. This is due to modeling the system in the UC framework: In UC, privacy guarantees are modeled by explicitly sending all information that an adversary could learn in the real world to the adversary. Additionally, in UC there exists only a single adversarial party that can corrupt several other parties. In the real world, this corresponds to the scenario that all dishonest parties collude and share all the information they gathered with each other. Consequently, a UC-adversary learns a lot of information. However, one should note that in reality dishonest parties learn significantly less information if they do not cooperate.

4 Realizing the Model

After giving an idealized formalization of the system we want to achieve, we now elaborate on how to actually build such a system.

4.1 A Protocol $\Pi_{AS}$ for Realizing $F_{AS}$

The functionality $F_{AS}$ represents an ideal version of the system we want to achieve. Since in practice we do not want to rely on trusted third parties to perform our calculations for us, we build a protocol $\Pi_{AS}$ in the real world that achieves the same security guarantees as $F_{AS}$. We later prove that our constructed protocol $\Pi_{AS}$ UC-realizes $F_{AS}$ in the $\{F_{AD}, F_{CRS}, F_{BB}, G_{CLOCK}\}$-hybrid model. As a setup assumption, we use the well-known functionality $F_{CRS}$ which enables access for all parties to a common reference string (CRS), set up by a trusted party with a given distribution. We also use an external bulletin board functionality $F_{BB}$, where any party can register a single $(uid, v)$ pair associated with its identity, where the $uid$s need to be unique. Any party can retrieve registered values $v$, which in our case are public encryption keys.

We modularize our realization by outsourcing the auditable decryption of ciphertexts to another hybrid functionality $F_{AD}$, which we describe in Section 4.2. This hybrid functionality idealizes the primitive of a blockchain with an evolving set of committees where the committee members are anonymous until they finished their work. The functionality is parameterized by a (threshold) PKE scheme and provides everyone with access to a public key under which secrets can be encrypted. Then, given suitable auditing and authorization information, decryption of a ciphertext can be requested. In our case this information will be a judge-signed warrant for that ciphertext. We outsource the auditable decryption for several reasons: 1. A primitive for auditable threshold decryption is
an interesting building block in itself and to the best of our knowledge no UC formalization of that primitive exists yet. 2. It simplifies the security analysis of our system, since we can first assume that the decryption is handled by a trustworthy party. In a second step we then replace the auditable decryption functionality \( F_{AD} \) with a protocol that realizes it.

Since blockchains with evolving committees generally require some concept of time (e.g., to ensure that the committee changes daily), the functionality \( F_{AD} \) utilizes a global clock functionality \( G_{CLOCK} \) (from [5]) to model time.

We now elaborate on some of the core techniques we use in \( \Pi_{AS} \) and refer for the complete description of \( \Pi_{AS} \) to Appendix B.2 and for the used hybrid functionalities (except \( F_{AD} \)) to Appendix B.1. The hybrid functionality \( F_{AD} \) is briefly discussed in Section 4.2 and in full detail in Appendix B.3.

System Init. SO and J each create a signing keypair. Then SO, J, and AU initialize the auditable decryption functionality \( F_{AD} \) together. The functionality \( F_{AD} \) is then used to provide all parties with access to the needed public keys.

User Registration. Users create a signing keypair \((vk_U, sk_U)\) during registration. To ensure at most one account per user, we use an idealized bulletin board \( F_{BB} \), where any party can register a single \((uid, vk_U)\) pair associated with its identity, where the \( uids \) need to be unique. Any party can retrieve registered values.

Store Secret. To store a fresh secret with validity period \( vper \), user and SO jointly create a fresh secret \( secret \) with a Blum coin toss, where each party draws a partial secret: SO draws \( sec_1 \) and the user draws \( sec_2 \). The full secret is then \( secret := sec_1 \oplus sec_2 \), but only learned by the user. The operator directly stores the partial secret \( sec_1 \) (along with the user’s identity \( uid \) and the current period \( vper \)). The user encrypts the partial secret \( sec_2 \) under the public threshold encryption key \( pk \) of \( F_{AD} \), sends the resulting ciphertext \( ct \) to SO and proves in zero-knowledge that it calculated all values honestly. SO also stores the ciphertext \( ct \) and provides the user with a (blinded) signature on \((uid, vper, secret)\) (without learning \( secret \)). The user can then utilize this signature in the application on top to prove to another party that it indeed stored a secret for that validity period.

Request Warrant. First, J signs the warrant \( W \) proposed by LE to convince third parties that it indeed has approved the warrant. Since the auditable decryption functionality \( F_{AD} \) needs to know which ciphertexts should be decrypted, we need J to also sign all ciphertexts \( ct \) containing the partial secrets \( sec_2 \) associated with the warrant \( W \). Therefore, we additionally include SO in this protocol: LE sends the signed warrant to SO and asks for the corresponding ciphertexts \( ct \) along with the stored partial secrets \( sec_1 \). Afterwards, LE provides J with the ciphertexts \( ct \) to get a signature on \( \tilde{W} \), which is the warrant \( W \) including the

\[13\] This of course enables SO to guess which users are or will be tracked by LE. But in practice this could be amended either by SO just sending all its information to LE or by LE using private information retrieval (PIR) to get just the ciphertexts for the current warrant without SO learning which ciphertexts were retrieved.
ciphertexts. LE can now utilize the hybrid functionality $F_{AD}$ to request the decryption of all ciphertexts corresponding to the warrant $\tilde{W}$.

Get Secrets. After $F_{AD}$ processed the decryption request, LE can retrieve the partial secrets $sec_2$ for a warrant from $F_{AD}$. Of course, $F_{AD}$ verified the validity of all requests and partial secrets for invalid requests cannot be retrieved. Then, LE uses the already stored partial secrets $sec_1$ (obtained from SO) to reconstruct the full user secrets $secret := sec_1 \oplus sec_2$ for all users in the warrant.

Get Statistics and Audit. Since $F_{AD}$ knows all requested warrants, $F_{AD}$ can directly give the desired information for those tasks.

Prove and Verify. The Prove task is a local task in which the user uses a NIZKPoK to create the proof $\pi$ itself. Similarly, the Verify task is also a local task in which the validity of the statement is verified using the NIZKPoK.

Security. In Appendix E we show that our protocol $\Pi_{AS}$ UC-realizes $F_{AS}$. In particular, we show the following theorem.

**Theorem 1.** $\Pi_{AS}$ UC-realizes $F_{AS}$ in the \{F_{AD}, F_{CRS}, F_{BB}, G_{CLOCK}\}-hybrid model under the assumptions that COM is a (computationally) hiding, (statistically) binding and (dual-mode) extractable and equivocable commitment scheme, $\Sigma$ is a EUF-CMA secure signature scheme, NIZK is a straight-line simulation-extractable non-interactive zero-knowledge proof system, and TPKE is IND-CPA secure against all PPT-adversaries $A$ who statically corrupts either (1) a subset of the users, (2) LE and a subset of the users, (3) SO and a subset of the users, or (4) SO, LE and a subset of the users.

4.2 Decrypting Secrets with $F_{AD}$

We now want to briefly describe our ideal auditable decryption functionality $F_{AD}$. The formal description can be found in Appendix B.3. To enable protocols based on the YOSO approach, our functionality makes use of the global clock functionality $G_{CLOCK}$ and proceeds in rounds, where a round lasts a predefined amount of time units and decryptions only become available in following rounds. The functionality is also parameterized by a (threshold) PKE scheme.

Init, Get Tasks and Interaction with $G_{CLOCK}$. $F_{AD}$ starts by creating an encryption keypair and registering with the global clock $G_{CLOCK}$. In $\Pi_{AS}$ SO and J each create a signing keypair and pass their public verification keys to $F_{AD}$, where they are stored. There are also tasks to provide the parties in $\Pi_{AS}$ with the public keys of SO, J and the public encryption key of the functionality. This ensures that both systems use the same keys.

Before sending the request to J, LE checks the users’ signatures. Before answering the request, J checks the (same) signatures as well. Since we assume that J is always honest, it would be sufficient for only J to check the signatures. But we intentionally let LE check the signatures first to filter out invalid requests before forwarding them to J, to reduce J’s workload.

Note that these tasks provide the parties with information about all requested warrants, independently of whether the secrets were actually retrieved or not.
**Request Decryption and Retrieve Secret.** LE can send a signed warrant to \( \mathcal{F}_{\text{AD}} \) to request decryption of all ciphertexts \( ct \) listed in that warrant. \( \mathcal{F}_{\text{AD}} \) checks if that warrant is valid and then adds the listed ciphertexts \( ct \) to a list of pending decryption requests. To allow our implementation \( \Pi_{\text{AD}} \) to be YOSO, requests for decryption are only processed during committee handovers. To emulate that behavior in the ideal world, \( \mathcal{F}_{\text{AD}} \) separates the request of a secret and the actual retrieval of a secret into two different inquiries by LE, with the requirement that the committee switches between the two calls.

**Role Execute.** To emulate the passing of time in the real world and the fact that YOSO parties can only send a message once, \( \mathcal{F}_{\text{AD}} \) interacts with the clock \( \mathcal{G}_{\text{CLOCK}} \): After all honest nodes have activated \( \mathcal{F}_{\text{AD}} \), it advances the current time. After the required amount of “time” passes, \( \mathcal{F}_{\text{AD}} \) emulates a committee handover as follows: \( \mathcal{F}_{\text{AD}} \) handles all pending decryption requests and adds the decrypted (partial) secrets to a list of processed decryption requests. After this “committee handover”, the list of processed decryption requests contains all (partial) secrets that are ready for retrieval by LE.

**Get Statistics and Audit.** \( \mathcal{F}_{\text{AD}} \) keeps track of all warrants for which LE requested secrets. If AU initiates an investigation, \( \mathcal{F}_{\text{AD}} \) provides it with all valid warrants. Likewise, \( \mathcal{F}_{\text{AD}} \) can also provide a party asking for statistics with the outputs of the transparency function for all warrants. Therefore, \( \mathcal{F}_{\text{AD}} \) provides the same statistics and audit information as \( \mathcal{F}_{\text{AS}} \).

### 4.3 A Protocol \( \Pi_{\text{AD}} \) for Realizing \( \mathcal{F}_{\text{AD}} \)

We cast our protocol \( \Pi_{\text{AD}} \) in the YOSO model, which we now introduce briefly.

**The YOSO Model.** In the YOSO (You-Only-Speak-Once) model introduced by [28], protocols are run between roles, where each role is only allowed to send one message and has no lasting state. These roles can then be assigned to actual machines executing them through some form of role assignment mechanism. With a way to anonymously receive messages (e.g., by reading ciphertexts stored on a public blockchain) and a role assignment mechanism that privately assigns roles, this prevents targeted attack against roles: The identity of a machine executing a role can only be learned when it sends its message, but at that point it finished execution and is no longer in possession of any secret state.

This allows both resilience against denial-of-service attacks as well as against strong adversaries trying to corrupt roles that are part of a protocol execution of interest. Assuming a large enough pool of machines willing to execute roles, an attacker able to corrupt any machine of its choosing, but limited in the number of machines it can corrupt at once, cannot break the security of a protocol even when run between only a number of roles smaller than the corruption limit.

To achieve this in our protocol, we make use of a blockchain with role assignment functionality \( \mathcal{F}_{\text{BCRA}} \), which provides a public append-only ledger together with a mechanism that anonymously selects parties for the next committee by posting public encryption and verification keys for the individual roles on the ledger and privately sending the corresponding decryption and signing keys to
the assigned party. For more details, see Fig. 4. Note that since it is unclear how to realize the role assignment functionality provided in [28], our functionality differs from theirs in the following ways:

1. We integrated the global clock $G_{CLOCK}$.
2. $F_{BCRA}$ only allows assigning roles of the next committee, not roles at an arbitrary time in the future.
3. The adversary can control a portion of the public keys (and $F_{BCRA}$ does not get to know the corresponding secret keys).

We deem it plausible that the committee selection protocol from [8] with suitable corruption thresholds in combination with a suitable blockchain can be used to implement this, although more efficient approaches such as described in [15] are also possible. We want to stress here that we are in the “near future” setting of [15], compared to the “far future” setting that would imply witness encryption. Given that the mechanisms for assigning roles and the criteria by which parties should be chosen are subject to ongoing research, we believe using $F_{BCRA}$ to abstract from the details is a suitable approach that allows incorporation of future research.

Now we give a brief description of $Π_{AD}$ and refer to Appendix B.4 for further details. Our instantiation $Π_{AD}$ UC-realizes $F_{AD}$ in the $\{F_{BCRA}, F_{CRS}, G_{CLOCK}\}$-hybrid model.

*Init, Get Tasks and Interaction with $G_{CLOCK}$. During initialization, AU creates an encryption keypair and SO, J and AU post their public keys to the ledger $F_{BCRA}$. The common reference string is obtained by querying $F_{CRS}$, the other Get Tasks are handled by reading from the ledger $F_{BCRA}$. All honest nodes $N$ register with the clock $G_{CLOCK}$ upon first activation. Additionally, the distributed key generation protocol from [23] is run by the first roles assigned through $F_{BCRA}$, resulting in a public encryption key $pk$ and a threshold-sharing of the corresponding decryption key among the first committee.

*Request Decryption and Retrieve Secret. To request decryption of ciphertexts and subsequently receive user secrets, LE needs to be in possession of a judge-signed warrant listing the relevant ciphertexts $ct$. To ensure privacy of the warrant $W$, instead of simply posting the warrant to $F_{BCRA}$, LE instead posts an encryption $W^{\text{enc}}$ of the warrant $W$ under AU’s public key, the output $W^{\text{pub}}$ of the transparency function and a NIZK-proof that it knows a valid signature under J’s public key on $W$ and both $W^{\text{enc}}$ and $W^{\text{pub}}$ were computed correctly. Additionally, instead of posting the ciphertexts $ct$ directly, LE re-randomizes them and also proves in zero-knowledge that the ciphertexts $\hat{ct}$ are indeed re-randomizations of the ciphertexts $ct$ listed in the warrant. This ensures that the users under surveillance can not be identified from the ciphertexts. The request additionally contains a public encryption key of LE under which the responses from the committee members will be encrypted.

16 Alternatively, a suitable variant of the committee selection protocol from [8] or the “encryption to the current winner” scheme from [15] are good candidates as well.
The functionality is parameterized by an encryption scheme $\text{PKE}$, a signature scheme $\Sigma$, a threshold $\epsilon$, a maximum delay $\delta_{\text{post}}$, and a set $\text{MACHINE}$ which is the set of parties allowed to use it. It has also access to a global clock $\text{GCLOCK}$. Upon receiving any input, $\mathcal{F}_{\text{BCRA}}$ first queries $\text{GCLOCK}$ and sets $t_{\text{now}}$ to the value returned.

**Init:** Let $\text{Posted}$ be the empty set, let $\text{ORDERED}$ be the empty sequence.

**Post:** On input $(\text{Post}, m)$ from $\text{pid} \in \text{MACHINE}$, add $(\text{pid}, t_{\text{now}}, m)$ to the set $\text{Posted}$ and output $(\text{Post}, \text{pid}, m)$ to the adversary.

**Order:** On input $(\text{Order}, \text{pid}, m)$ from the adversary where some $(\text{pid}, t, m) \in \text{Posted}$ and no $(\text{pid}, t', m) \in \text{ORDERED}$, append $(\text{pid}, t_{\text{now}}, m)$ to $\text{ORDERED}$. Then output $(\text{ORDERED})$ to $\text{pid}$.

**NextCommittee:** On input $(\text{NextCommittee}, R, n, S)$ from the adversary, with $|S| < \epsilon n$ containing entries of the form $(M_i \in \text{MACHINE}, (\text{ek}_i, \text{vk}_i))$, do the following:

- For $i \in \{1, \ldots, |S|\}$:
  1. Mark $M_i$ as corrupted
  2. Set $m = (\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i)$, add $(\text{roleassign}, t_{\text{now}}, m)^*$ to $\text{Posted}$ and leak $m$ to the adversary
- For $i \in \{|S| + 1, \ldots, n\}$:
  1. Sample $\text{ek}_i, \text{dk}_i \leftarrow \text{PKE}.\text{Gen}(1^\lambda)$
  2. Sample $\text{vk}_i, \text{sk}_i \leftarrow \Sigma.\text{Gen}(1^\lambda)$
  3. Sample a uniformly random $M_{\text{new}} \in \text{MACHINE}$
  4. Set $m = (\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i)$, add $(\text{roleassign}, t_{\text{now}}, m)$ to $\text{Posted}$ and leak $m$ to the adversary

When $(\text{roleassign}, t', m)$ is later added to $\text{ORDERED}$ output $(\text{generate}, R, \text{ek}_i, \text{dk}_i, \text{vk}_i, \text{sk}_i)$ to $M_i$ for $i \in \{|S| + 1, \ldots, n\}$

**Forward Security:** When $M_{\text{old}}$ becomes corrupted, output $(\text{generate}, R, \text{ek}_i, \text{dk}_i, \text{vk}_i, \text{sk}_i)$ to the adversary if $(\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i) \notin \text{ORDERED}$.

---

*a roleassign is a special pid used to represent role-assignment messages*

---

**Functionality $\mathcal{F}_{\text{BCRA}}$**

The functionality is parameterized by an encryption scheme $\text{PKE}$, a signature scheme $\Sigma$, a threshold $\epsilon$, a maximum delay $\delta_{\text{post}}$, and a set $\text{MACHINE}$ which is the set of parties allowed to use it. It has also access to a global clock $\text{GCLOCK}$. Upon receiving any input, $\mathcal{F}_{\text{BCRA}}$ first queries $\text{GCLOCK}$ and sets $t_{\text{now}}$ to the value returned.

**Init:** Let $\text{Posted}$ be the empty set, let $\text{ORDERED}$ be the empty sequence.

**Post:** On input $(\text{Post}, m)$ from $\text{pid} \in \text{MACHINE}$, add $(\text{pid}, t_{\text{now}}, m)$ to the set $\text{Posted}$ and output $(\text{Post}, \text{pid}, m)$ to the adversary.

**Order:** On input $(\text{Order}, \text{pid}, m)$ from the adversary where some $(\text{pid}, t, m) \in \text{Posted}$ and no $(\text{pid}, t', m) \in \text{ORDERED}$, append $(\text{pid}, t_{\text{now}}, m)$ to $\text{ORDERED}$. Then output $(\text{ORDERED})$ to $\text{pid}$.

**NextCommittee:** On input $(\text{NextCommittee}, R, n, S)$ from the adversary, with $|S| < \epsilon n$ containing entries of the form $(M_i \in \text{MACHINE}, (\text{ek}_i, \text{vk}_i))$, do the following:

- For $i \in \{1, \ldots, |S|\}$:
  1. Mark $M_i$ as corrupted
  2. Set $m = (\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i)$, add $(\text{roleassign}, t_{\text{now}}, m)^*$ to $\text{Posted}$ and leak $m$ to the adversary
- For $i \in \{|S| + 1, \ldots, n\}$:
  1. Sample $\text{ek}_i, \text{dk}_i \leftarrow \text{PKE}.\text{Gen}(1^\lambda)$
  2. Sample $\text{vk}_i, \text{sk}_i \leftarrow \Sigma.\text{Gen}(1^\lambda)$
  3. Sample a uniformly random $M_{\text{new}} \in \text{MACHINE}$
  4. Set $m = (\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i)$, add $(\text{roleassign}, t_{\text{now}}, m)$ to $\text{Posted}$ and leak $m$ to the adversary

When $(\text{roleassign}, t', m)$ is later added to $\text{ORDERED}$ output $(\text{generate}, R, \text{ek}_i, \text{dk}_i, \text{vk}_i, \text{sk}_i)$ to $M_i$ for $i \in \{|S| + 1, \ldots, n\}$

**Forward Security:** When $M_{\text{old}}$ becomes corrupted, output $(\text{generate}, R, \text{ek}_i, \text{dk}_i, \text{vk}_i, \text{sk}_i)$ to the adversary if $(\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i) \notin \text{ORDERED}$.

---

*a roleassign is a special pid used to represent role-assignment messages*

---

**Functionality $\mathcal{F}_{\text{BCRA}}$**

The functionality is parameterized by an encryption scheme $\text{PKE}$, a signature scheme $\Sigma$, a threshold $\epsilon$, a maximum delay $\delta_{\text{post}}$, and a set $\text{MACHINE}$ which is the set of parties allowed to use it. It has also access to a global clock $\text{GCLOCK}$. Upon receiving any input, $\mathcal{F}_{\text{BCRA}}$ first queries $\text{GCLOCK}$ and sets $t_{\text{now}}$ to the value returned.

**Init:** Let $\text{Posted}$ be the empty set, let $\text{ORDERED}$ be the empty sequence.

**Post:** On input $(\text{Post}, m)$ from $\text{pid} \in \text{MACHINE}$, add $(\text{pid}, t_{\text{now}}, m)$ to the set $\text{Posted}$ and output $(\text{Post}, \text{pid}, m)$ to the adversary.

**Order:** On input $(\text{Order}, \text{pid}, m)$ from the adversary where some $(\text{pid}, t, m) \in \text{Posted}$ and no $(\text{pid}, t', m) \in \text{ORDERED}$, append $(\text{pid}, t_{\text{now}}, m)$ to $\text{ORDERED}$. Then output $(\text{ORDERED})$ to $\text{pid}$.

**NextCommittee:** On input $(\text{NextCommittee}, R, n, S)$ from the adversary, with $|S| < \epsilon n$ containing entries of the form $(M_i \in \text{MACHINE}, (\text{ek}_i, \text{vk}_i))$, do the following:

- For $i \in \{1, \ldots, |S|\}$:
  1. Mark $M_i$ as corrupted
  2. Set $m = (\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i)$, add $(\text{roleassign}, t_{\text{now}}, m)^*$ to $\text{Posted}$ and leak $m$ to the adversary
- For $i \in \{|S| + 1, \ldots, n\}$:
  1. Sample $\text{ek}_i, \text{dk}_i \leftarrow \text{PKE}.\text{Gen}(1^\lambda)$
  2. Sample $\text{vk}_i, \text{sk}_i \leftarrow \Sigma.\text{Gen}(1^\lambda)$
  3. Sample a uniformly random $M_{\text{new}} \in \text{MACHINE}$
  4. Set $m = (\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i)$, add $(\text{roleassign}, t_{\text{now}}, m)$ to $\text{Posted}$ and leak $m$ to the adversary

When $(\text{roleassign}, t', m)$ is later added to $\text{ORDERED}$ output $(\text{generate}, R, \text{ek}_i, \text{dk}_i, \text{vk}_i, \text{sk}_i)$ to $M_i$ for $i \in \{|S| + 1, \ldots, n\}$

**Forward Security:** When $M_{\text{old}}$ becomes corrupted, output $(\text{generate}, R, \text{ek}_i, \text{dk}_i, \text{vk}_i, \text{sk}_i)$ to the adversary if $(\text{NextCommittee}, R, \text{ek}_i, \text{vk}_i) \notin \text{ORDERED}$.

---

*a roleassign is a special pid used to represent role-assignment messages*
obtain their share $sk_i$ of the threshold decryption key $sk$. To enable the committee in the next round to fulfill their duties as well, they reshare $sk_i$ again and encrypt each reshare to a committee member from the next round. This is again done in the same way as in [23].

For each valid decryption request, a (partial) threshold-decryption of the ciphertext $ct$ is performed using $sk_i$. The answer (including the partial decryption $ct^*$ and a proof of correct decryption) is encrypted under the public key of LE contained in the request.

All messages to be sent are signed using the role’s signing key. Before sending any message, all state except for the prepared messages is deleted. Finally, all messages are posted to $F_{BCRA}$.

Get Statistics and Audit. Obtaining statistics is achieved by reading the content of $F_{BCRA}$ and gathering all $W_{pub}$ accompanied by valid NIZK-proofs. Similarly, the audit is performed by AU reading the content of $F_{BCRA}$ and decrypting all $W_{enc}$ accompanied by valid NIZK-proofs.

Security. In Appendix F we show that our protocol $\Pi_{AD}$ UC-realizes $F_{AD}$. In particular, we show the following theorem.

**Theorem 2.** If NIZK is a straight-line simulation-extractable non-interactive zero-knowledge proof system, $\Sigma$ is an EUF-CMA secure signature scheme, the PKE scheme used by LE and AU is an IND-CPA secure public key encryption scheme, the PKE scheme that is a parameter of $F_{BCRA}$ is a RIND-SO secure public key encryption scheme, and TPKE is an IND-CPA secure, randomizable and binding $(t,n)$-threshold PKE, then $\Pi_{AD}$ UC-realizes $F_{AD}$ in the $\{F_{BCRA}, F_{CRS}, G_{CLOCK}\}$-hybrid model with respect to adversaries $A$ that may statically corrupt SO and/or LE as well as mobile adaptively corrupt at most a fraction $\frac{t}{n} - \epsilon$ of nodes $N$.

Efficiency. Given the need for a suitable incentive for nodes to participate, it is important to limit the amount of work roles have to do. In our protocol, the work of roles only depends on the number of decryption requests (which correspond to the number of users under surveillance in the combined system), but not on the number of ciphertexts created (which corresponds to the number of registered users in the combined system). A current bottleneck is the role assignment process and the communication required to transmit the shared secret key to the next committee. This is an active area of research and improvements along the lines of [20, 14, 29] are promising.

5 Application

In this section, we present our (toy) application $F_{ASTE}$ for Auditably Sender-Traceable Encryption. It is intentionally kept very simplistic, and chosen to exemplify the problems one encounters with creating a system with auditable

---

17 These include messages to the next committee and decryption answers to LE
surveillance based on top of $F_{AS}$. The construction and techniques developed for $F_{ASTE}$, in particular those used for cut-and-choose-based combination of NINCE with zero-knowledge, are applicable for many functionalities of interest.

5.1 The Functionality $F_{ASTE}$

We briefly describe the capabilities, i.e., the tasks, offered by $F_{ASTE}$. The full description is in Appendix C.2. Basically, $F_{ASTE}$ offers a public key encryption scheme to its users, with the tweak that only registered users can generate ciphertexts $c$ (which honest users accept), and that such ciphertexts can be deanonymized by an authority (namely LE), holding users accountable. To simplify the definition, given a ciphertext $c$, the intended receiver learns only the message, while law enforcement LE learns only the identity of the encryptor (and nothing about the message). This can be easily modified, cf. Remark 2. To prevent perpetual user surveillance, “time” is partitioned into validity periods. Ciphertexts are bound to a period, and honest users only accepts those bound to the current period. These periods are also the granularity of surveillance of LE. That is, a warrant specifies which users are under surveillance during which (past) validity periods.

Setup and Auditability. The tasks System Init, Request Warrant, Get Statistics, and Audit, are inherited from and identical to $F_{AS}$ with minimal changes to System Init to formally handle update periods (i.e., $G_{CLOCK}$).

Register, Update, and Next Period. Similar to $F_{AS}$, a user must first register to participate in the system, except that the user cannot choose its $uid$ anymore (as it will become the public encryption key in the protocol). The Next Period task allows the system operator to advance the current validity period, in particular, validity periods are not bound to (physical) time. Honest users will only accept ciphertexts of the current validity period. Hence, when the period changes, users must execute the Update task (which, intuitively, deposits a new escrow secret for the current period).

Encrypt Message and Decrypt Ciphertext. These tasks do exactly what one expects: They encrypt a message to a (registered) user, and decrypt received ciphertexts. Decryption ensures that the ciphertext is encrypted w.r.t. the current validity period. Ciphertexts under past periods are not accepted, otherwise, a warrant specifying surveillance of a certain user in a certain period would incorrectly omit (or include) ciphertexts which do not belong to said period.

Prepare Access. Law enforcement prepares the information necessary for access to ciphertexts $c$ covered by warrant $W$. This usually means that LE acquires the respective escrow secrets associated with $W$, i.e., it is almost identical to the Get Secrets task of $F_{AS}$.

Execute Access. In this task, LE can check whether a ciphertext $c$ was generated by the user with identity $uid$ during period $v_{per}$, where LE is supposed to have previously prepared access for a warrant which affects ($uid$, $v_{per}$).
5.2 The Protocol $\Pi_{\text{ASTE}}$

To realize $\Pi_{\text{ASTE}}$, we build a protocol $\Pi_{\text{ASTE}}$ (whose full description is in Appendix C.3). Firstly, we work in the $\{\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{AS}}, \mathcal{G}_{\text{CLOCK}}\}$-hybrid model. Thus, we can rely on $\mathcal{F}_{\text{AS}}$ to take care of the basic requirements for adding auditability, namely, key escrow with auditable access via warrants. Our second key building block is a significantly tweaked public key encryption scheme, $\text{PKE}_{\text{AS}}$.

In the protocol, the user identity $uid$ will be set to a $\text{PKE}_{\text{AS}}$ public key.

The scheme $\text{PKE}_{\text{AS}}$ provides the promised deanonymization capabilities. This is achieved by including a proof of consistency in the ciphertext (which requires the sender’s $uid$ and escrow secret $usk$) which demonstrates:

- One ciphertext component encrypts a message $m$ to the receiver (with receiver public key $pk_R = uid_R$).
- Another ciphertext component encrypts the sender $uid$ (i.e., sender’s public key) under the sender’s escrow secret key $usk$ (to enable deanonymization).

This is proven via the non-interactive proof capabilities provided by $\mathcal{F}_{\text{AS}}$, which additionally ensure that the tuple $(uid, usk, v_{\text{per}})$ is in $L_{\text{Secrets}}$, i.e., it is a stored user secret. At a first glance, these two properties, combined with any secure public key encryption scheme for the receiver, seem to be enough for our purposes. However, as mentioned in the technical overview (Section 1.2), the simulator-commitment-problem obstructs a security proof for this direct approach.

The scheme $\text{PKE}_{\text{AS}}$. To circumvent the impossibility of non-interactive non-committing encryption (NINCE) [48], we rely on the (programmable) random oracle model (PROM). It is trivial to construct public-key and secret-key NINCE schemes in the PROM, and we let $\text{PKE}_{\text{NCE}}$ and $\text{SKE}_{\text{NCE}}$ be such schemes. Intuitively, we use $\text{PKE}_{\text{NCE}}$ to encrypt the message for the receiver, and $\text{SKE}_{\text{NCE}}$ to encrypt the user’s identity $uid$ under its escrow key $usk$ for LE. We tie the ciphertexts together with a zero-knowledge proof. Unfortunately, the random oracle $\text{RO}$ has no “code”, so it is impossible to prove (correct) encryption of a message for $\text{PKE}_{\text{NCE}}$ and $\text{SKE}_{\text{NCE}}$ with the usual (circuit-based) zero-knowledge proofs. Thus, we use a somewhat elaborate cut-and-choose technique, in order to connect traditional zero-knowledge proofs for NP with $\text{PKE}_{\text{NCE}}$ and $\text{SKE}_{\text{NCE}}$.

Remark 2. For simplicity and concreteness, we only prove knowledge of the tuple $t = (m, uid, usk, v_{\text{per}})$ in $\text{PKE}_{\text{AS}}$, and prove that ciphertext components encrypt $m$ under the receiver’s key $pk_R$ and $uid$ under the escrow key $usk$ (for law enforcement access). However, it is straightforward to prove any efficient relation over $t$. For example, choosing a receiver message function $f$ and law enforcement message function $g$, one can encrypt $f(t)$ under $pk_R$ (by modifying the shares

---

18 We note that avoiding a cut-and-choose approach is challenging for provably secure constructions. There are impossibilities for black-box zero-knowledge proofs, e.g., [54, 46], which we have to avoid. We do so by using a cut-and-choose approach and additionally constructing the NINCE together with its zero-knowledge proof. (The positive results in [46] also use this idea.)
of $m$ to encrypt shares of $f(t)$ and adapting the proof statements), and similarly encrypt $g(t)$ under $usk$ (by analogous modifications). In this sense, the code in Fig. 5 has $f(t) = m$ and $g(t) = uid$ hard-coded for concreteness.

Setup, encryption and ciphertext verification algorithms of $\text{PKE}_{\text{AS}}$ are given in Fig. 5; the two latter algorithms depend on a public commitment key $ck$. Now, we explain the idea behind the proofs ($\pi_{\text{cut}}, \pi_{\text{con}}$) in $\text{PKE}_{\text{AS.Enc}}$ and $\text{PKE}_{\text{AS.Vfy}}$ (see Fig. 5). The values $m$, $uid$, $usk$ are first secret-shared, then used in encryptions $ct_{i,b}^{RE}$, $ct_{i,b}^{RE}$. Moreover, the shares of $m_{i,b}$, $uid_{i,b}$, $usk_{i,b}$ along with the encryption randomness are also committed to in $com_{i,b}$. The $com_{i,b}$ will be important for consistency proofs which we explain now. The extractability of $\pi_{\text{con}}$ (realized by $F_{\text{AS}}$) allows to recover the committed values in the security proof. Moreover, $\pi_{\text{con}}$ ensures that the shares committed in $com_{i,b}$ are consistent with unique shared values $m'$, $uid'$, $usk'$, e.g., $m' = m'_{i,0} + m'_{i,1}$ holds for all $i$. The proof $\pi_{\text{cut}}$ is the cut-and-choose part which ensures that enough $\text{PKE}_{\text{RE}}$ resp. $\text{SKE}_{\text{RE}}$ ciphertexts encrypt $m'$ under $pk_{R}$ resp. $uid'$ under $usk'$. For this, $\pi_{\text{cut}}$ forces the encryptor to open a randomly chosen share $\gamma_i \in \{0,1\}$ of $ct_{i,b}^{RE}$, $ct_{i,b}^{RE}$ and $com_{i,b}$ for each $i = 1, \ldots, \ell(\lambda)$, where the challenge $\gamma_i = \text{RO}(\text{stmt}, \pi_{\text{con}})$ is derived following the Fiat–Shamir paradigm. Note that if both for $\gamma_i = 0$ and $\gamma_i = 1$ can be opened, then $\pi_{\text{con}}$ ensures that the value reconstructed from the shares is $m'$ (resp. $uid'$, $usk'$), unless the binding property of $\text{COM}$ is broken. Assuming unconditionally binding $\text{COM}$, the latter cannot happen. Then, by a standard argument, if less than the majority of ciphertexts share reconstruct $m'$ (resp. $uid'$, $usk'$), the probability to succeed in $\pi_{\text{cut}}$ is about $2^{-\ell(\lambda)/2} = 2^{-\lambda}$. Consequently, if $\pi_{\text{con}}$ and $\pi_{\text{cut}}$ are accepting, then with overwhelming probability, the extracted values $m'_{i,b}$, which satisfy $m' = m'_{i,0} + m'_{i,1}$, agree with the decrypted values $m_{i,b}$ which yield $m_i = m_{i,0} + m_{i,1}$ in the majority of indices $i$. Thus, decrypting each $ct_{i,b}^{RE}$, computing $m_i$, and then picking the majority value $m_i$ (or $\perp$ if none exists) agrees with overwhelming probability with the extracted value $m'$ of $\pi_{\text{con}}$. Unsurprisingly, $\text{PKE}_{\text{AS.DecR}}$ will do just that.

Observe that $\text{PKE}_{\text{AS.Vfy}}$ enables public verifiability of well-formedness of ciphertexts. This verification allows the system operator to check ciphertext validity and remove any invalid ciphertexts, and it is also the first step of decryption procedures $\text{PKE}_{\text{AS.DecR}}$ and $\text{PKE}_{\text{AS.DecL}}$, which work as follows:

- $\text{PKE}_{\text{AS.DecR}}$ decrypts all $ct_{i,b}^{RE}$ using $sk_{R}$, reconstructs messages $m_i = m_{i,0} + m_{i,1}$, and outputs $m$ if this is the absolute majority of $m_i$, and $\perp$ otherwise.
- $\text{PKE}_{\text{AS.DecL}}$ uses the escrow secret key $usk$ and exploits the relation $usk_{i,1-\gamma_i} = usk - usk_{i,\gamma_i}$ to derive the second secret share $usk_{i,1-\gamma_i}$. With this, it trial-decrypts the symmetric ciphertext(s) $ct_{i,b}^{RE}$ to recover $uid_i = uid_{i,0} + uid_{i,1}$, and outputs $uid$ if this is the absolute majority of $uid_i$, and $\perp$ otherwise.

The majority decisions in decryption ensure that the extracted values $m'$ and $uid'$ from $\pi_{\text{con}}$ coincide with the actual results of decryption. In particular, if $\text{PKE}_{\text{AS.Vfy}}$ accepts $c$, then $c$ will decrypt for the receiver and LE.

Law-enforcement access. To provide access to LE, the Prepare Access task uses the Get Secrets task of the underlying $F_{\text{AS}}$-hybrid functionality. Thus, LE ob-
for $i = 1, \ldots, \ell(\lambda)$

// Prepare cut-and-choose encryption for LE

$m_{i,0} + m_{i,1} = m$  // Additive shares

$\text{usk}_{i,0} + \text{usk}_{i,1} = \text{usk}$

$\text{uid}_{i,0} + \text{uid}_{i,1} = \text{uid}$

for $b \in \{0, 1\}$

$\text{ct}_{i,b}^{RE} = \text{PKE}_{\text{AS}}.\text{Enc}((\text{pk}_{R}, m_{i,b}, r_{i,b}^{RE}))$

$\text{ct}_{i,b}^{LE} = \text{SKE}_{\text{NCE}}.\text{Enc}((\text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^{LE}))$

$\text{w}_{i,b} = (m_{i,b}, \text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^{RE}, r_{i,b}^{LE})$

$\text{com}_{i,b} = \text{COM}.\text{Com}(\text{ct}_{i,b})$

$d_{i,b} = (\text{w}_{i,b}, r_{i,b}^{com})$

// Consistency proof for commitments and ctxs.

$\text{wit} = (\text{ct}_{i,b}^{RE}, \text{ct}_{i,b}^{LE})$

$\text{stmt} = (\text{vper}, (\text{琉}_{i,b}, \text{ct}_{i,b}^{RE}, \text{ct}_{i,b}^{LE}))$

$\pi_{\text{con}} = \text{NIZK}_{\text{AS}}.\text{Prove}((\text{usk}, \text{uid}, \text{vper}), (\text{stmt}, \text{wit}, \text{R}_{\text{PKEAS}}))$ for relation $\text{R}_{\text{PKEAS}} = \{(\text{stmt}, \text{wit}) | (\text{uid}, \text{sk}_{\text{S}}) = \text{PKE}_{\text{AS}}.\text{Gen}(\text{key}_{\text{AS}}^\lambda, \text{sk}_{\text{S}})\}$

$\text{com}_{i,b} = \text{COM}.\text{Com}(\text{ct}_{i,b})$

$\forall i: m = m_{i,0} + m_{i,1}$

$\forall i: \text{usk} = \text{usk}_{i,0} + \text{usk}_{i,1}$

$\forall i: \text{uid} = \text{uid}_{i,0} + \text{uid}_{i,1}$

// Cut-and-choose: Query challenge $\gamma$

$\gamma = \text{RO}(\text{stmt}, \pi_{\text{con}}) \in \{0, 1\}^\ell$

// Verify $\text{PKE}_{\text{AS}}.\text{Gen}(\text{key}_{\text{AS}}^\lambda)$

$\text{pk}_{R} = (\text{com}_{i,b}, \text{ct}_{i,b}^{RE}, \text{ct}_{i,b}^{LE})$

parse $c = (\text{pk}, (\text{琉}_{i,b}, \text{ct}_{i,b}^{RE}, \text{ct}_{i,b}^{LE}))$

parse $\text{w}_{i,b} = (m_{i,b}, \text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^{RE}, r_{i,b}^{LE})$

if $\text{com}_{i,b} = \text{COM}.\text{Com}((\text{ct}_{i,b}^{LE}))$

or $\text{ct}_{i,b}^{RE} = \text{PKE}_{\text{AS}}.\text{Enc}((\text{pk}_{R}, m_{i,b}, r_{i,b}^{RE}))$

or $\text{ct}_{i,b}^{LE} = \text{SKE}_{\text{NCE}}.\text{Enc}((\text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^{LE}))$

then return 0

return 1

Fig. 5: Encryption and ciphertext verification subroutines of $\text{PKE}_{\text{AS}}$. 

29
tains escrow secrets $usk$ for all tuples $(uid, vper)$ covered by the warrant $W$. In the Execute Access task, the obtained escrow secrets $usk$ for $(uid, vper)$ is used with $\text{PKE}_{\text{AS}}\cdot\text{Dec}_{\text{LE}}$ to trial-decrypt a ciphertext $c$.

5.3 On Efficiency and the Necessity of (NI)NCE

We briefly sketch the apparent necessity of non-interactive (NI) non-committing encryption (NCE) to efficiently realize $\text{FASTE}$. The core problem appears in the security proof, where the simulator must generate ciphertexts for users whose identities are unknown to it. Once a warrant $W$ pertains a user with $uid$, all simulated ciphertexts $c_1, \ldots, c_n$ affected by $W$ must correspond to that user. Thus, the simulator can (and must) not be committed to an identity $uid$ for a ciphertext, unless these are affected by warrants. If the Execute Access task is non-interactive, this is very analogous to NINCE, which is impossible without (strong) setup assumptions [48]. As suitable setups are “black-box”, e.g., random oracles, they are incompatible with circuit-based zero-knowledge. Although expensive, cut-and-choose techniques are (to our best knowledge) the only known approach. Unfortunately, the cut-and-choose proof leads to relatively large ciphertexts. For $\lambda = 128$ bits of security, an optimistic estimate yields at least $1.5\text{MiB}$ for the ciphertexts, without accounting for $\pi_{\text{cut}}$ and $\pi_{\text{con}}$, see Appendix C.6 for details. These proofs allow trade-offs between size and computational efficiency (e.g., by using SNARKs).

5.4 Comparison with [34]

Green, Kaptchuk, and Laer [34] construct (non-anonymous) messaging with auditable surveillance. They face similar challenges for their protocols, and implicitly rely on NINCE in the PROM as well. However, their setting is considerably simpler: The recipient learns all information contained in the ciphertext, so that it can simply recompute the encryption to ensure a ciphertext is well-formed. This approach avoids zero-knowledge proofs, but is quite limited. It can, for example, not be used as an anonymous messenger, since the receiver needs the sender’s identity to recompute the ciphertext and verify the correctness of the message. Thus it cannot be used to realize ASTE. As an additional feature, [34] suggests to have the system operator remove invalid ciphertexts. This is possible for $\Pi_{\text{ASTE}}$, since ciphertext validity is publicly verifiable. However, it is not possible for [34], unless the system operator can read every message, defeating the purpose of the protocol.

---

19 Using interactive decryption circumvents the impossibility without strong setups, but is undesirable in practice.
20 We stress that, although $\text{PKE}_{\text{AS}}$ as defined in Fig. 5 encrypts only the user identity and not the message to law enforcement, this can easily be changed to also give the message to law enforcement. As noted in Remark 2, our approach allows a quite flexible choice of leakage, not just user identity and/or message.
While both [34] and we model system security in the UC framework, our ideal functionalities differ in the following aspects: In our system it can be decided separately for the sender identity and the content of the message whether the recipient, law enforcement (with a warrant), or both should learn it (cf. Remark 2). In [34] both parties learn both information.\textsuperscript{21} Also, [34] is limited to the messaging application scenario, while we also specify an auditable surveillance functionality $F_{AS}$ that can be used for many applications, e.g., for $F_{ASTE}$.

Compared to us, [34] offers greater flexibility w.r.t. warrants. While our warrants are fixed on user (identity, validity period) pairs, in [34] a warrant may specify a predicate on metadata of a ciphertext and law enforcement can enforce access when the predicate is true. We explicitly avoid this flexibility, as it negatively affects efficiency and requires strong(er) cryptographic primitives — our approach is practically feasible, but [34] is far from it. Recall that [34] uses extractable witness encryption (EWE) [30] whose existence is implausible [26] for general NP-relations. While the relation in [34] is specific, it is quite complex, so even if such EWE existed, its practical efficiency is implausible.

### 5.5 Other Applications

Offering a zero-knowledge proof interface allows broad use of $F_{AS}$ in applications, and our techniques to combine PROM-based encryption and zero-knowledge allow to overcome the problem of NINCE which naturally appears in most applications. For example, one may augment an anonymous e-cash or electronic payment system with auditable surveillance by adding a ciphertext for law-enforcement, which encrypts the identities of the parties of the transfer and proves that the encrypted contents (which law enforcement can learn with the escrow secret) are indeed correct and related to the transaction which was carried out.

### 6 Limitations and Future Work

In its current form, our building block and its realization are subject to certain limitations. Realizations of the $F_{BCRA}$ hybrid functionality are still novel and experimental [8, 14, 20], so the actual guarantees of such a realization may differ from our assumptions. For $PKE_{AS}$, ciphertext size is unacceptable in practice.

A realistic protocol must also cover the existence of many judges, law enforcement agencies, and auditors. Once the judge (and auditor) are not modeled as trusted parties anymore, key-revocation mechanisms become absolutely necessary, as otherwise a single key compromise allows a (state-level) adversary permanent unauthorized surveillance. Although our system ensures that such unauthorized surveillance will be noticed, it does not prevent it. See also Appendix D for a more fine-grained discussion of system limitations.

\textsuperscript{21} While the ideal functionality in [34] technically only supplies some \textit{metadata} to law enforcement (during the message sending process) and not the sender’s identity, it becomes apparent later in the paper that the authors assume the sender’s identity to be included in the metadata.
Moreover, to harden security, one should distribute trust over multiple parties, especially for the auditor, for example by using threshold cryptography or secure multi-party computation. Lastly, it is an interesting question how to achieve more flexibility w.r.t. warrants, e.g., surveillance based on metadata similar to [34], but without resorting to implausible primitives such as EWE.

Acknowledgements
This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. This work has been supported by Helsinki Institute for Information Technology HIIT.

References


A Building Blocks

We now review some of the used building blocks.

A.1 SKE, PKE and NINCE

In this section, we recall secret-key and public-key encryption, and define non-
interactive non-committing encryption and the security notions we require.

Definition 1 (Syntax of PKE and SKE). A public-key encryption (PKE) scheme PKE consists of a tuple \( (\text{PKE.} \text{Gen}, \text{PKE.} \text{Enc}, \text{PKE.} \text{Dec}) \) of PPT algorithms such that

- \( \text{PKE.} \text{Gen}(1^\lambda) \) outputs a pair of public key \( \text{pk} \) and secret key \( \text{sk} \).
- \( \text{PKE.} \text{Enc}(\text{pk},m) \) takes a public key \( \text{pk} \), a message \( m \) in the message space \( M \), and outputs a ciphertext \( c \). The message space may depend on \( \text{pk} \), and we sometimes write \( M_{\text{pk}} \) to emphasize this.
- \( \text{PKE.} \text{Dec}(\text{sk},c) \) takes a secret key \( \text{sk} \), a ciphertext \( c \), and outputs a message \( m \in M \) or \( \perp \).

For PKE schemes in the ROM, \( \text{PKE.} \text{Enc} \) and \( \text{PKE.} \text{Dec} \) (but not \( \text{PKE.} \text{Gen} \)) additionally get access to (one or more) random oracle(s). A secret-key encryption (SKE) scheme SKE is defined analogous to PKE, except there is no public key, i.e. \( \text{SKE.} \text{Gen} \) outputs only \( \text{sk} \), and \( \text{SKE.} \text{Enc} \) takes \( \text{sk} \) as input.

Remark 3. We explicitly forbid key generation algorithms (\( \text{PKE.} \text{Gen} \) and \( \text{SKE.} \text{Gen} \)) access to the random oracle. This will later allow use to use zero-knowledge proofs for NP over key generation algorithms.

Definition 2 (Security notions for encryption). We make use of the usual security notions (in the ROM) for encryption:

- indistinguishability under chosen plaintext attacks (IND-CPA) for PKE and SKE,
- indistinguishability from randomness under chosen plaintext attacks (IND-$\dollar$-CPA) for PKE and SKE,
- indistinguishability under chosen ciphertext attacks (IND-CCA) for PKE and SKE,
- ciphertext integrity (INT-CTXT) for SKE

Definition 3 (Key-committing). Let SKE be a SKE scheme. We say SKE is (secret-)key-committing, if it is hard to find a tuple \( (\text{sk}_1, \text{sk}_2, c) \) such that \( \text{sk}_1 \neq \text{sk}_2 \) but \( \text{SKE.} \text{Dec}(\text{sk}_1, c) \neq \perp \) and \( \text{SKE.} \text{Dec}(\text{sk}_2, c) \neq \perp \).

Let PKE be a perfectly correct PKE scheme. We say PKE is (public-)key-committing, if it is hard to find a tuple \( (\text{pk}_1, \text{sk}_1), (\text{pk}_2, \text{sk}_2), c \) such that \( (\text{pk}_1, \text{sk}_1) \in \text{Supp}(\text{PKE.} \text{Gen}(1^\lambda)) \) and \( \text{pk}_1 \neq \text{pk}_2 \) but \( \text{PKE.} \text{Dec}(\text{sk}_1, c) \neq \perp \) and \( \text{PKE.} \text{Dec}(\text{sk}_2, c) \neq \perp \).
We define a strong form of non-interactive non-committing encryption in the PROM. Compared to more standard game-based definitions of NINCE (e.g. [59, 9]), strong NINCE ensures that simulation and explanation of ciphertexts is possible even for honestly generated public and secret keys. This makes strong NINCE very easy to use.

**Definition 4 (Strong NINCE).** Let \( \text{PKE} = (\text{PKE.Gen}, \text{PKE.Enc}, \text{PKE.Dec}) \) be a PKE scheme in the programmable random oracle model (PROM). Define the strong non-interactive non-committing encryption (NINCE) experiment for \( \text{PKE} \) with simulator \( (S, \text{Expln}) \) and adversary \( A \) as follows, where all algorithms (except \( \text{PKE.Gen} \)) have access to all ROs.

The simulator is split into two separate algorithms, \( S \) simulates ciphertexts, and \( \text{Expln} \) explains ciphertexts by programming the random oracle.

\[
\begin{align*}
\text{Exp}_{\text{PKE},S,A}^{\text{s-nince-real}}(1^\lambda) \\
(pk, sk) &\leftarrow \text{PKE.Gen}(1^\lambda) \\
&// A gets access to encryption oracle. \\
\text{state}_A &\leftarrow A^{\text{PKE.Enc}(pk,\cdot)}(1^\lambda, pk) \\
b &\leftarrow A(\text{state}_A, sk) \\
\text{return } b
\end{align*}
\]

\[
\begin{align*}
\text{Exp}_{\text{PKE},S,A}^{\text{s-nince-ideal}}(1^\lambda) \\
(pk, sk) &\leftarrow \text{PKE.Gen}(1^\lambda) \\
&// A gets access to simulation oracle. \\
\text{state}_A &\leftarrow A^{S(1^\lambda, pk, \cdot)}(pk) \\
&// Let m_1, \ldots, m_n be the queries of A. \\
&// Let state_s be the state of S \\
\text{Expln}(state_s, sk, m_1, \ldots, m_n) &// Program RO \\
b &\leftarrow A(\text{state}_A, sk) \\
\text{return } b
\end{align*}
\]

The advantage \( \text{Adv}_{\text{PKE},S,\text{Expln},A}^{\text{s-nince}}(\lambda) \) is defined as

\[
\left| \Pr \left[ \text{Exp}_{\text{PKE},S,A}^{\text{s-nince-real}}(\lambda) = 1 \right] - \Pr \left[ \text{Exp}_{\text{PKE},S,A}^{\text{s-nince-ideal}}(\lambda) = 1 \right] \right|.
\]

We say \( \text{PKE} \) is a strongly secure NINCE scheme (in the ROM), if there exist PPT algorithms \( S, \text{Expln} \) such that every PPT adversary has negligible advantage.

We define strong NINCE for SKE schemes analogously, except that \( pk \) is not given to \( A \) and \( S \).

**A.2 Instantiations of NINCE**

We construct a public-key NINCE scheme \( \text{PKE}_{\text{NCE}} \) with message space \( \{0, 1\}^{\text{poly}(\lambda)} \) from any secure PKE scheme with superpolynomial and uniformly sampleable
Additional transformations are necessary if $PKE$ is an IND-CPA-secure and perfectly correct PKE scheme. Then the NINCE scheme $PKE_{NCE} = (PKE_{NCE}.Gen, PKE_{NCE}.Enc, PKE_{NCE}.Dec, PKE_{NCE}.Sim, PKE_{NCE}.Expln)$ defined in Appendix A.2 is IND-CCA-secure, strong NINCE-secure, and perfectly correct.

The proof is standard.

---

The construction of $PKE_{NCE}$ above is essentially a variant of the Fujisaki–Okamoto transformation [25, 37] to achieve CCA secure key-encapsulation, plus stretching the key through $RO_M$ and using it as a one-time pad and using a tag $t$ to ensure integrity of the message. Moreover, the tag $t$ also serves as a key-commitment. By programming the one-time pad, any message can be "explained" later, ensuring the strong NINCE-security of the scheme.

**Lemma 1.** Let $PKE$ be an IND-CPA-secure and perfectly correct PKE scheme. Then the NINCE scheme $PKE_{NCE} = (PKE_{NCE}.Gen, PKE_{NCE}.Enc, PKE_{NCE}.Dec, PKE_{NCE}.Sim, PKE_{NCE}.Expln)$ defined in Appendix A.2 is IND-CCA-secure, strong NINCE-secure, and perfectly correct.

---

22 Additional transformations are necessary if $R, M \neq \{0,1\}^\lambda$ or $\text{poly}(\lambda) \neq \lambda$. We omit these standard details.
Proof (Proof sketch). Perfect correctness is evident. Strong NINCE can be argued as follows: For a ciphertext with ephemeral key $esk$ in $c_1$, either the adversary never queries $RO_M$ and $RO_I$ with $esk$, in which case programming in $PKE_{NCE}$ works, or it was queried before. Since $sk$ is only available to $PKE_{NCE}$, but not $PKE_{NCE}$. Sim, it is straightforward to obtain an IND-CPA adversary from a NINCE-adversary by testing whether a queried $esk'$ for $RO_M$ and $RO_I$ decrypt the IND-CPA challenge message correctly. Thus, the probability that $esk$ from (any) simulated ciphertext was previously queried by the adversary is negligible, since $PKE$ is IND-CPA-secure.

IND-CCA-security follows from standard arguments, as $PKE_{NCE}$ is only a variant of the Fujisaki–Okamoto transformation [25, 37].

Key-committment of $PKE_{NCE}$ follows immediately from the tag $t$ in the ciphertext, because it is a hash over $pk$. Thus, a ciphertext which decrypts to $m \neq \bot$ for two keys induces a tag-collision. However, random oracles are collision resistant, thus this happens with negligible probability.

For a secret-key NINCE construction, we basically replace the public key by a secret key and the ephemeral key by a nonce, which yields a standard PRF-based authenticated encryption scheme. The message space is again $\{0, 1\}^{poly(\lambda)}$.

\[
\text{SKE}_{NCE} \cdot \text{Gen}(1^\lambda) \\
\text{sk} \leftarrow \{0, 1\}^\lambda
\]

\[
\text{SKE}_{NCE} \cdot \text{Enc}(\text{sk}, m) \\
N \leftarrow \{0, 1\}^{2^\lambda} \text{ / Nonce} \\
c' = RO_M(\text{sk}, N) \oplus m \\
t = RO_I(\text{sk}, N, c', m) \\
\text{return } (N, c', t)
\]

\[
\text{SKE}_{NCE} \cdot \text{Dec}(\text{sk}, c) \\
\text{parse } (N, c', t) = c \\
m = RO_M(\text{sk}, N) \oplus c' \\
t' = RO_I(\text{sk}, N, c', m) \\
\text{if } t = t' \text{ return } m \\
\text{else return } \bot
\]

\[
\text{SKE}_{NCE} \cdot \text{Sim}() \\
N \leftarrow \{0, 1\}^{2^\lambda} \\
c' \leftarrow \{0, 1\}^{poly(\lambda)} \\
t \leftarrow \{0, 1\}^\lambda \\
\text{return } (N, c', t)
\]

\[
\text{SKE}_{NCE} \cdot \text{Expln}(\text{sk}, c, m) \\
\text{parse } (N, c', t) = c \\
\text{program } RO_M( esk ) := c' \oplus m \\
\text{program } RO_I(\text{sk}, N, c', m) := t
\]

Lemma 2. The NINCE scheme $SKE_{NCE} = (SKE_{NCE} \cdot \text{Gen}, SKE_{NCE} \cdot \text{Enc}, SKE_{NCE} \cdot \text{Dec}, SKE_{NCE} \cdot \text{Sim}, SKE_{NCE} \cdot \text{Expln})$ defined above is IND-CCA-secure, strong NINCE-secure, and perfectly correct. Moreover, $SKE_{NCE}$ is key-committing.

Proof (Proof sketch). By standard arguments, $SKE_{NCE}$ is actually a authenticated encryption scheme, so in particular IND-CCA secure. Similar to $PKE_{NCE}$,
it’s not hard to see that simulation works. Indeed, since the security of the scheme is information-theoretic (as it relies solely on random oracles), this is even simpler to prove.

Key-commitment of SKENKE follows immediately from the tag \( t \) in the ciphertext, because it is a hash over \( sk \). Thus, a ciphertext which decrypts to \( m \neq \bot \) for two keys induces a tag-collision. However, random oracles are collision resistant, thus this happens with negligible probability.

A.3 Threshold PKE

**Definition 5 (Syntax of Threshold PKE).** A \((t,n)\)-threshold PKE (TPKE) scheme TPKE consists of a tuple \((\text{TPKE.Gen}, \text{TPKE.Enc}, \text{TPKE.Dec}, \text{TPKE.TDec}, \text{TPKE.ShareVer}, \text{TPKE.Combine}, \text{TPKE.Sk2Pk})\) of PPT algorithms such that

- \( \text{TPKE.Gen}(1^\lambda) \) outputs a public key \( pk \), secret key \( sk \) and sets \( \{vk_i\}_{i \in [n]} \) of verification keys and \( \{sk_i\}_{i \in [n]} \) of secret key shares.
- \( \text{TPKE.Enc}(pk, m) \) takes a public key \( pk \), a message \( m \) in the message space \( \mathcal{M} \), and outputs a ciphertext \( ct \).
- \( \text{TPKE.Dec}(sk, ct) \) takes a secret key \( sk \), a ciphertext \( ct \) and outputs a message \( m \in \mathcal{M} \).
- \( \text{TPKE.TDec}(sk_i, ct) \) takes a secret key share \( sk_i \), a ciphertext \( ct \), and outputs a decryption share \( ct_i \).
- \( \text{TPKE.ShareVer}(ct, vk_i, ct_i) \) 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.
- \( \text{TPKE.Combine}(ct, T) \) takes a ciphertext \( ct \) and a set of valid decryption shares \( T \) such that \( |T| = t + 1 \), and outputs a message \( m \in \mathcal{M} \).
- \( \text{TPKE.Sk2Pk}(sk) \) takes a secret key \( sk \) (or a secret key share \( sk_i \)) as input and outputs the corresponding public key \( pk \) (or the corresponding verification key \( vk_i \)).

We require correctness, i.e. the following two conditions to hold for all \( (pk, \{vk_i\}_{i \in [n]}, \{sk_i\}_{i \in [n]}) \leftarrow \text{TPKE.Gen}(1^\lambda) \):

1. \( \forall m \in \mathcal{M}, ct \leftarrow \text{TPKE.Enc}(pk, m), i \in [n] \) it holds that \( \text{TPKE.ShareVer}(ct, vk_i, \text{TPKE.TDec}(sk_i, ct)) = 1 \).
2. \( \forall m \in \mathcal{M}, ct \leftarrow \text{TPKE.Enc}(pk, m) \) and any set \( T := (ct_1, \ldots, ct_{t+1}) \) of valid decryption shares \( ct_i \leftarrow \text{TPKE.TDec}(sk_i, ct) \) for \( t + 1 \) distinct secret key shares \( sk_i \): \( \text{TPKE.Combine}(ct, T) = m = \text{TPKE.Dec}(sk, ct) \).

In particular, in the ideal world we make use of \( \text{TPKE.Dec} \) as a shorthand for \( \text{TPKE.TDec} \) plus \( \text{TPKE.Combine} \).

A TPKE scheme is called simulatable if additionally an efficient simulation algorithm \( \text{TPKE.SimTDec}(pk, ct, m, \{ct_k\}_{k \in \mathbb{B} : |\mathbb{B}| \leq t}) \) exists that takes a public key, ciphertext, target message and up to \( t \) decryption shares of corrupted parties as input and outputs decryption shares \( \{ct_i\}_{i \in [n] \setminus \mathbb{B}} \) for the honest parties that cause
A TPKE scheme is called **re-randomizable** if it also has a PPT algorithm TPKE.Rand such that \( \forall (pk, \{vk_i\}_{i \in [n]}, \{sk_i\}_{i \in [n]}) \leftarrow TPKE.Gen(1^\lambda) : \)

- TPKE.Rand(ct) takes a ciphertext ct, and outputs a new ciphertext ct'.
- \( \forall m \in M : TPKE.Rand(TPKE.Enc(pk, m)) \approx TPKE.Enc(pk, m) \)

A TPKE scheme is called **binding** if it holds that \( \forall PPT_A : \Pr[(sk', sk), ct) \leftarrow A(1^\lambda) : TPKE.Sk2Pk(sk) = TPKE.Sk2Pk(sk') \wedge TPKE.Dec(sk, ct) \neq TPKE.Dec(sk', ct)] \neq \text{negl}(\lambda) \)

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

### A.4 Selective Opening Security

We recall receiver selective opening security from Hazay et al. [36].

**Definition 6 (RIND-SO Security).** A PKE scheme PKE is called RIND-SO secure if for all PPT adversaries \( A \) and all efficiently resamplable distributions \( D \) we have that

\[
\text{Adv}_{PKE}^{\text{RIND-SO}}(A, \lambda) := \left| \Pr_{\text{Exp}_{PKE}^{\text{RIND-SO}}}(A, \lambda) - \frac{1}{2} \right| \leq \text{negl}(\lambda)
\]

**Exp_{PKE}^{\text{RIND-SO}}(A, \lambda)**

1: \( b \leftarrow \{0, 1\} \)
2: \( (pk, sk) : (pk_i, sk_i) \leftarrow (PKE.Gen(1^\lambda))_{i \in [n]} \)
3: \( m : = (m_i)_{i \in [n]} \leftarrow D \)
4: \( c : = (c_i)_{i \in [n]} \leftarrow (PKE.Enc(pk, m_i))_{i \in [n]} \)
5: \( \text{state} \leftarrow A^{C(i)}(c) \)
6: \( m' \leftarrow \text{Resample}_D(m_I) \)
7: \( m^* = m \text{ if } b = 0, \text{ else } m^* = m' \)
8: \( b' \leftarrow A(m^*, \text{state}) \)
9: \( \text{return } b = b' \)

**Corruption Oracle C(i)**

1: \( \text{if } i \notin [n] \text{return } \bot \)
2: \( \mathcal{I} : = \mathcal{I} \cup \{i\} \)
3: \( \text{return } sk_i \)

Fig. 6: Experiment for receiver selective opening security.
A.5 Digital Signature Scheme

We use a correct and EUF-CMA-secure signature scheme $\Sigma$.

Definition 7 (Syntax of Digital Signatures). A digital signature scheme $\Sigma = (\Sigma.\text{Gen}, \Sigma.\text{Sign}, \Sigma.\text{Vfy})$ consists of the following algorithms:

- $\Sigma.\text{Gen}(1^\lambda)$ is a PPT algorithm, which takes $1^\lambda$ as input and outputs a keypair $(vk, sk)$.
- $\Sigma.\text{Sign}(sk, m)$ is a PPT algorithm, which takes the signing key $sk$ and a message $m$ as input and outputs a signature $\sigma$.
- $\Sigma.\text{Vfy}(vk, m, \sigma)$ is a deterministic polynomial-time algorithm, which takes as input a verification key $vk$, a message $m$ and a signature $\sigma$. It returns either $0$ or $1$.
  (Intuitively, returning $1$ means that $\sigma$ is valid signature for message $m$ under verification key $vk$. Returning $0$ means it is invalid.)

$\Sigma$ is correct if for all $\lambda$, $(vk, sk) \leftarrow \Sigma.\text{Gen}(1^\lambda)$, $m \in M$, and $\sigma \leftarrow \Sigma.\text{Sign}(sk, m)$ it holds that

$$1 = \Sigma.\text{Vfy}(vk, m, \sigma).$$

A.6 Commitment Scheme

We use a correct, (computationally) hiding, (statistically) binding, extractable and equivocable commitment scheme $\text{COM}$.

Definition 8 (Syntax Commitment Scheme). A (non-interactive) commitment scheme $\text{COM} := (\text{COM}.\text{Setup}, \text{COM}.\text{Com}, \text{COM}.\text{Open})$ consists of the following three algorithms:

- $\text{COM}.\text{Setup}$ is a PPT algorithm, which takes $1^\lambda$ as input and outputs public parameters $\text{crs}_{\text{com}}$.
- $\text{COM}.\text{Com}$ is a PPT algorithm, which takes as input parameters $\text{crs}_{\text{com}}$ and a message $m \in M$ and outputs a commitment $\text{com}_m$ to $m$ and some decommitment value $\text{decom}_m$.
- $\text{COM}.\text{Open}$ is a deterministic polynomial-time algorithm, which takes as input parameters $\text{crs}_{\text{com}}$, a commitment $\text{com}_m$, a decommitment value $\text{decom}_m$ and a message $m$. It returns either $0$ or $1$.

$\text{COM}$ is correct if for all $\lambda$, $\text{crs}_{\text{com}} \leftarrow \text{COM}.\text{Setup}(1^\lambda)$, $m \in M$, and $(\text{com}_m, \text{decom}_m) \leftarrow \text{COM.\text{Com}}(\text{crs}_{\text{com}}, m)$ it holds that

$$1 = \text{COM.\text{Open}}(\text{crs}_{\text{com}}, \text{com}_m, \text{decom}_m, m).$$

An extractable commitment scheme has the following additional algorithms:

- $(\text{crs}_{\text{ext}}^{\text{com}}, \text{td}_{\text{com}}^{\text{ext}}) \leftarrow \text{COM.\text{Setup}}^{\text{Ext}}(1^\lambda)$
- $m \leftarrow \text{COM.\text{Extract}}(\text{td}_{\text{com}}^{\text{ext}}, \text{com}_m)$

An equivocable commitment scheme has the following additional algorithms:
Definition 9 (Equivocal Commitment scheme). A commitment scheme \( \text{COM} \) is equivocal defined by

\[
\Pr \left[ 1 \leftarrow \mathcal{A}(\text{crs}_\text{com}) \mid \text{crs}_\text{com} \leftarrow \text{Setup}^{\text{Equiv}}(1^\lambda) \right] - \Pr \left[ 1 \leftarrow \mathcal{A}(\text{crs}_\text{com}, \text{td}_\text{com}) \mid (\text{crs}_\text{com}, \text{td}_\text{com}) \leftarrow \text{Setup}^{\text{Equiv}}(1^\lambda) \right] \lambda \text{ is negligible in } \lambda \text{ and we have that the advantage defined by}
\]

\[
\Pr \left[ 1 \leftarrow \mathcal{A}(\text{crs}_\text{com}, \text{td}_\text{com}) \mid (\text{crs}_\text{com}, \text{td}_\text{com}) \leftarrow \text{Setup}^{\text{Equiv}}(1^\lambda), m \leftarrow \mathcal{M}, (\text{com}_\text{eq}, \text{decom}_\text{eq}) \leftarrow \text{COM}^{\text{Equiv}}(\text{crs}_\text{com}, m) \right] - \Pr \left[ 1 \leftarrow \mathcal{A}(\text{crs}_\text{com}, \text{td}_\text{com}) \mid (\text{crs}_\text{com}, \text{td}_\text{com}) \leftarrow \text{Setup}^{\text{Equiv}}(1^\lambda), m \leftarrow \mathcal{M}, \text{com}_\text{eq} \leftarrow \text{COM}^{\text{Equiv}}(\text{crs}_\text{com}, m), \text{decom}_\text{eq} \leftarrow \text{COM}^{\text{Equiv}}(\text{crs}_\text{com}, m, \text{eq}) \right] \lambda \text{ is negligible in } \lambda \text{ and we have that the advantage defined by}
\]

\[
\Pr \left[ 1 \leftarrow \mathcal{A}(\text{crs}_\text{com}) \mid \text{crs}_\text{com} \leftarrow \text{Setup}^{\text{Ext}}(1^\lambda) \right] - \Pr \left[ 1 \leftarrow \mathcal{A}(\text{crs}_\text{com}) \mid (\text{crs}_\text{com}, \text{td}_\text{com}) \leftarrow \text{Setup}^{\text{Ext}}(1^\lambda) \right] \lambda \text{ is negligible in } \lambda \text{ and we have that the advantage defined by}
\]

\[
\Pr \left[ \text{COM}^{\text{Ext}}(\text{td}_\text{com}, \text{com}_\text{eq}) \neq m \mid (\text{td}_\text{com}, \text{com}_\text{eq}) \leftarrow \text{Setup}^{\text{Ext}}(1^\lambda), c \leftarrow \mathcal{A}(\text{crs}_\text{com}), \exists m \in \mathcal{M}, r \text{ such that } c \leftarrow \text{COM}^{\text{Ext}}(\text{td}_\text{com}, \text{com}_\text{eq}, m, r) \right] \lambda \text{ is negligible in } \lambda \text{ and we have that the advantage defined by}
\]

\[
\text{A.7 Zero-Knowledge Proof System}
\]

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 11. A non-interactive proof system \( \text{NIZK} \) for NP-relation \( \mathcal{R} \) is a tuple \( (\text{NIZK}.\text{Setup}, \text{NIZK}.\text{Prove}, \text{NIZK}.\text{Verify}) \) of PPT algorithms, where

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

It is a \text{NIZKPoK} if additionally there are algorithms \text{NIZK}.\text{Sim} \text{ and } \text{NIZK}.\text{Ext} \text{ with }

\begin{itemize}
  \item \text{NIZK}.\text{Sim}(\text{td}, \text{stmt}) \text{ outputs a proof } \pi.
  \item \text{NIZK}.\text{Ext}(\text{td}, \text{stmt}, \pi) \text{ outputs a witness } \text{wit} \text{ with } (\text{stmt}, \text{wit}) \in \mathcal{R} \text{ or } \bot.
\end{itemize}

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

**Definition 12.** Let \text{NIZK} be a \text{NIZKPoK} for \text{NP-relation} \mathcal{R}. We define the straight-line simulation-extractability game as follows: Let \textbf{A} be an adversary.

1. \textbf{Setup} \((\text{crs}, \text{td}) \leftarrow \text{NIZK}.\text{Setup}(1^{\lambda})\).
2. \textbf{Run} \(\textbf{A}(1^{\lambda}, \text{crs})\), where \textbf{A} is given oracle access to oracle \((\text{stmt}, \pi) \mapsto \text{NIZK}.\text{Ext}(\text{td}, \text{stmt}, \pi)\) and either
   \begin{itemize}
     \item \((\text{stmt}, \text{wit}) \mapsto \text{NIZK}.\text{Prove}(\text{crs}, \text{stmt}, \text{wit})\), or
     \item \((\text{stmt}, \text{wit}) \mapsto \text{NIZK}.\text{Sim}(\text{td}, \text{stmt})\),
   \end{itemize}
   with the choice made uniformly at random. Let \(L_{\text{Ext}}\) be the list of queries with outputs \((\text{stmt}, \pi, \text{wit})\) to \text{NIZK}.\text{Ext}, and \(L_{\text{Prove}}\) be the list of queries with outputs \((\text{stmt}, \text{wit}, \pi)\) to \text{NIZK}.\text{Prove} or \text{NIZK}.\text{Sim}.
3. Eventually, \textbf{A} outputs a guess \(b\), where \(b = 0\) (resp. \(b = 1\)) indicates that it interacted with \text{NIZK}.\text{Prove} (resp. \text{NIZK}.\text{Sim}).
4. \textbf{A} wins the game if:
   \begin{itemize}
     \item \textbf{A} never queried \((\text{stmt}, \pi)\) if \(\pi\) was previously returned from a query to \text{NIZK}.\text{Prove} (or \text{NIZK}.\text{Sim}) with statement \text{stmt}. (I.e., \textbf{A} may not trivially distinguish simulated proofs from real proofs.)
     \item For any \((\text{stmt}, \text{wit}, \pi) \in Q_{\text{Ext}}\), we have \(\text{wit} = \bot\), but \text{NIZK}.\text{Verify}(\text{crs}, \text{stmt}, \pi) = 1. \text{(I.e., extraction failed for an accepting proof.)}
     \item \textbf{A} guessed \(b\) correctly.
   \end{itemize}

We say that \text{NIZK} is straight-line simulation-extractable, if for any PPT adversary, the probability that it wins the straight-line simulation-extractability game is negligible.

Note that weaker notions of simulation-extractability exist, which are not straight-line. But there are simple and efficient transformations to ensure straight-line extractability (e.g., by adding an encryption of the witness under a public key of a PKE which is put into the \text{crs} (and proving consistency as part of the relation)).

**A.8 Introduction to the UC Framework**

The Universal Composability (UC) framework \cite{17, 16} is a model to analyze cryptographic protocols which offers strong security guarantees. Security is proven by showing that two different worlds are indistinguishable from one another.
In the *ideal world* an incorruptible entity – the ideal functionality $F$ – exists. Every party just sends their private input to the functionality, the functionality honestly computes the output based on the inputs from the parties and sends the output back to all the parties.

In the *real world*, no such functionality exists. Instead, the (mutually distrustful) parties themselves execute a protocol $II$ that computes the same function.

Informally said, the protocol $II$ is as secure as the ideal functionality $F$ if no PPT-machine (called the environment $Z$) can distinguish between the two worlds. One then says that $II$ UC-realizes $F$.

### B Full System Model

In the following, we provide our full system model, apart from the auditable surveillance functionality $F_{AS}$ which was already presented in Section 3. We start with some preliminary remarks in Appendix B.1. Then we present the protocol $II_{AS}$ that realizes $F_{AS}$ in Appendix B.2. We continue by presenting the auditable decryption functionality $F_{AD}$ in Appendix B.3 and the protocol $II_{AD}$ realizing it in Appendix B.4.

#### B.1 Preliminary UC Remarks

We use several functionalities as hybrid functionalities in our system. In particular, we use a global clock functionality (see Fig. 7) to model time, a random oracle functionality (see Fig. 8), a CRS functionality (see Fig. 9) and a bulletin board functionality (see Fig. 10).

We also use the following conventions when describing the functionalities $F_{AS}$, $F_{AD}$ and $F_{ASTE}$:

- As usual, we assume authenticated channels between all parties.
- We omit (sub)session identifiers, written sid (resp. ssid), which are used to identify separate protocol instances (resp. separate instances of tasks in a protocol instance). Adding sids and ssids is straightforward.
- When we write “Input P: (…)” for $P \in \{SO, AU, J, LE\}$, the functionality first checks that the party calling the functionality has indeed that role.\(^{23}\) If that is not the case, the functionality ignores the message.
- The first task that has to be called is the Init task. If any parties send any other messages to each functionality before the Init task has been completed, those messages are ignored.
- We use *delayed output* to some party P as a short-hand notation for the following protocol: Send (OUTPUT, TASKNAME, P) to the adversary and wait for message (ALLOW, TASKNAME, P) from the adversary. Then send (value) to P.

\(^{23}\) The functionality can do that because it knows the party identifiers (pids) of all parties in the system.
The functionality manages the set \( \mathcal{P} \) of registered identities, i.e., parties \( P = (\text{pid}, \text{sid}) \). It also manages the set \( F \) of functionalities (together with their session identifier). Initially, \( \mathcal{P} := \emptyset \) and \( F := \emptyset \).

For each session \( \text{sid} \) the clock maintains a variable \( \tau_{\text{sid}} \). For each identity \( P := (\text{pid}, \text{sid}) \in \mathcal{P} \) it manages a variable \( d_P \). For each pair \( (F, \text{sid}) \in F \) it manages a variable \( d_{F, \text{sid}} \) (all integer variables are initially 0).

**Synchronization:**
- Upon receiving \((\text{Clock-Update}, \text{sid}_C)\) from some party \( P \in \mathcal{P} \) set \( d_P := 1 \); execute Round-Update and forward \((\text{Clock-Update}, \text{sid}_C, P)\) to \( A \).
- Upon receiving \((\text{Clock-Update}, \text{sid}_C)\) from some functionality \( F \) in a session \( \text{sid} \) such that \((F, \text{sid}) \in F\) set \( d_{F, \text{sid}} := 1 \), execute Round-Update and return \((\text{Clock-Update}, \text{sid}_C, F)\) to this instance of \( F \).
- Upon receiving \((\text{Clock-Read}, \text{sid}_C)\) from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return \((\text{Clock-Read}, \text{sid}_C, \tau_{\text{sid}})\) to the requestor (where \( \text{sid} \) is the session id of the calling instance).

**Party Management:**
- Upon receiving \((\text{Register}, \text{sid}_C)\) from some party \( P_i \) (or from \( A \) on behalf of a corrupted \( P_i \)), set \( \mathcal{P} := \mathcal{P} \cup \{P_i\} \). Return \((\text{Register}, \text{sid}_C, P_i)\) to the caller.
- Upon receiving \((\text{De-Register}, \text{sid}_C)\) from some party \( P_i \in \mathcal{P} \), the functionality updates \( \mathcal{P} := \mathcal{P} \setminus \{P_i\} \) and returns \((\text{De-Register}, \text{sid}_C, P_i)\) to \( P_i \).
- Upon receiving \((\text{Is-Registered}, \text{sid}_C)\) from some party \( P_i \), return \((\text{Register}, \text{sid}_C, b)\) to the caller, where the bit \( b \) is 1 if and only if \( P_i \in \mathcal{P} \).
- Upon receiving \((\text{Get-Registered}, \text{sid}_C)\) from \( A \), the functionality returns the response \((\text{Get-Registered}, \text{sid}_C, \mathcal{P})\) to \( A \).
- Upon receiving \((\text{Register}, \text{sid}_C)\) from a functionality \( F \) (with session-id \( \text{sid} \)), update \( F := F \cup \{(F, \text{sid})\} \).
- Upon receiving \((\text{De-Register}, \text{sid}_C)\) from a functionality \( F \) (with session-id \( \text{sid} \)), update \( F := F \setminus \{(F, \text{sid})\} \).
- Upon receiving \((\text{Get-Registered-F}, \text{sid}_C)\) from \( A \), return \((\text{Get-Registered-F}, \text{sid}_C, F)\) to \( A \).

**Procedure Round-Update:** For each session \( \text{sid} \) do: If \( d_{F, \text{sid}} = 1 \) for all \( F \in F \) and \( d_P = 1 \) for all honest parties \( P = (\cdot, \text{sid}) \in \mathcal{P} \), then set \( \tau_{\text{sid}} := \tau_{\text{sid}} + 1 \) and reset \( d_{F, \text{sid}} := 0 \) and \( d_P := 0 \) for all parties \( P = (\cdot, \text{sid}) \in \mathcal{P} \).

Fig. 7: The global clock functionality from [5]
### Functionality $\mathcal{F}_{RO}$

The functionality models a random oracle $\mathcal{RO}: \mathcal{X} \rightarrow \mathcal{Y}$. Initially $\mathcal{L}_{\mathcal{RO}} = \emptyset$.

**Hash:** Upon receiving $(\text{Hash}, m)$ from party $P$: If $m \notin \mathcal{X}$ abort. If $\exists (m, h') \in \mathcal{L}_{\mathcal{RO}}$, let $h = h'$; else pick uniformly at random $h \leftarrow \mathcal{Y}$; store $(m, h)$ in $\mathcal{L}_{\mathcal{RO}}$. Send $(\text{HashConfirm}, m, h)$ to $P$.

Fig. 8: Random oracle functionality from [12]

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

$\mathcal{F}_{CRS}$ proceeds as follows, when parameterized by a distribution $\mathcal{D}$.

1. When activated for the first time on input $(\text{VALUE}, sid)$, choose $d \leftarrow \mathcal{D}$ and send $d$ back to the activating party. In each other activation return the valued $d$ to the activating party.

Fig. 9: The Common Reference String functionality from [19]

### Functionality $\mathcal{F}_{BB}$

The functionality models a bulletin-board. Any party can register a single $(uid, v)$ pair associated with its identity, where the $uid$s need to be unique. Any party can retrieve registered values. Initially $\mathcal{L}_{\mathcal{BB}} = \emptyset$.

**Register:** Upon receiving $(\text{REGISTER}, uid, v)$ from party $P$, send $(\text{REGISTER}, uid, v)$ to the adversary. Upon receiving $(\text{OK})$ from the adversary, if there is no record $(P, \cdot, \cdot) \in \mathcal{L}_{\mathcal{BB}}$ and no record $(\cdot, uid, \cdot) \in \mathcal{L}_{\mathcal{BB}}$ record the pair $(P, uid, v)$. Otherwise, do nothing.

**Retrieve by Identity:** Upon receiving $(\text{RETRIEVE}, P_i)$ from party $P_j$ (including the adversary), generate public delayed output $(\text{RETRIEVE}, P_i, uid, v)$ to $P_j$ where $(uid, v)$ is such that $(P_i, uid, v) \in \mathcal{L}_{\mathcal{BB}}$ or $(\bot, \bot)$ if no entry is recorded in $\mathcal{L}_{\mathcal{BB}}$.

**Retrieve by UID:** Upon receiving $(\text{RETRIEVE}, uid)$ from party $P_j$ (including the adversary), generate public delayed output $(\text{RETRIEVE}, P_i, uid, v)$ to $P_j$ where $(P_i, v)$ is such that $(P_i, uid, v) \in \mathcal{L}_{\mathcal{BB}}$ or $(\bot, \bot)$ if no entry is recorded in $\mathcal{L}_{\mathcal{BB}}$.

Fig. 10: Bulletin Board functionality, adapted from [40]
– When we write “Output P: (value)”, this is a shorthand for “generate delayed output (value) to party P”.
– If, in contrast, we write “immediately output (value) to P”, the message is directly sent to party P without notifying the adversary.
– We use notion of “urgent request”/“immediate response” from [13], as well as responsive environments, to model that the adversary can not make any other calls to functionalities or hybrid functionalities before sending a response to the current request. By UC-realization, we mean realization w.r.t. to responsive environments.
– The notation “Send (value) to the adversary” is also a shorthand for “Send (value) to the adversary and wait for message (Ok) from the adversary before continuing”.

B.2 The Protocol $\Pi_{AS}$

**Details of the relation $R^{AS}_{ss}$**

\[
\begin{align*}
\text{wit} & := (\text{secret}, \text{sec}_2, \text{decom}_{\text{sec}_2}, \text{decom}_{\text{sec}_2}, r) \\
\text{stmt} & := (\text{sec}_1, \text{com}_{\text{sec}}, \text{com}_{\text{sec}_2}, \text{pk}, ct, \text{crs}_{\text{com}}) \\
R^{AS}_{ss} & = \{(\text{stmt}, \text{wit}) \mid \\
& - \text{secret} = \text{sec}_1 \oplus \text{sec}_2 \\
& - \text{COM.Open}(\text{crs}_{\text{com}}, \text{com}_{\text{sec}}, \text{decom}_{\text{sec}}, \text{secret}) = 1 \\
& - \text{COM.Open}(\text{crs}_{\text{com}}, \text{com}_{\text{sec}_2}, \text{decom}_{\text{sec}_2}, \text{sec}_2) = 1 \\
& - ct = \text{TPKE.Enc}(\text{pk}, \text{sec}_2; r) \}
\end{align*}
\]

Fig. 11: ZK-Relation $R^{AS}_{ss}$

In $\Pi_{AS}$, we use the following NP relations for zero-knowledge proofs:

– Relation $R^{AS}_{ss}$ (see Fig. 11) is used in Store Secret by a user to prove to the system operator that they honestly encrypted the partial secret $\text{sec}_2$
– Relation $R^{AS}_{zk}$ (see Fig. 12) is the NP relation used in the Prove and Verify tasks to prove some statement about a user’s identity, secret and the current period

We now give the complete description of the protocol $\Pi_{AS}$ that UC-realizes the auditable surveillance functionality $\mathcal{F}_{AS}$.
### Details of the relation $\mathcal{R}_{AS}^{zk}$

\[
\begin{align*}
\text{wit} & := (\text{uid}, \text{secret}, \text{wit}', \sigma_{ss}, \text{com}_{uid}, \text{decom}_{uid}, \text{com}_{sec}, \\
& \quad \text{decom}_{sec}, \text{com}_{vper}, \text{decom}_{vper}) \\
\text{stmt} & := (\text{vper}, \text{stmt}', \text{vk}_{SO}, \text{crs}_{com}) \\
\mathcal{R}_{AS}^{zk} & = \{(\text{stmt}, \text{wit}) | \\
& \quad \Sigma.\text{Vfy}(\text{vk}_{SO}, (\text{com}_{uid}, \text{com}_{sec}, \text{com}_{vper}), \sigma_{ss}) = 1 \\
& \quad \text{COM.Open}(\text{crs}_{com}, \text{com}_{uid}, \text{decom}_{uid}, \text{uid}) = 1 \\
& \quad \text{COM.Open}(\text{crs}_{com}, \text{com}_{sec}, \text{decom}_{sec}, \text{secret}) = 1 \\
& \quad \text{COM.Open}(\text{crs}_{com}, \text{com}_{vper}, \text{decom}_{vper}, \text{vper}) = 1 \\
& \quad \text{wit}_R = (\text{uid}, \text{secret}, \text{wit}') \\
& \quad \text{stmt}_R = (\text{vper}, \text{stmt}') \\
& \quad \mathcal{R}(\text{stmt}_R, \text{wit}_R) = 1 \}\end{align*}
\]

Fig. 12: ZK-Relation $\mathcal{R}_{AS}^{zk}$

### $H_{AS}$

**System Parameters:**
- $f_t$ — Transparency function. Gets a preliminary warrant $W$ as input and outputs what should be publicly known about that warrant. Interface is $W^\text{pub} \leftarrow f_t(W)$.
- $f_p$ — Policy Function. Checks whether a given warrant is allowed by system policy. Interface is $\{0, 1\} \leftarrow f_p(W)$.
- $n$ — Number of committee members
- $S$ — Space from which the secrets are drawn
- Signature scheme $\Sigma = (\Sigma.\text{Gen}, \Sigma.\text{Sign}, \Sigma.\text{Vfy})$, where all algorithms are PPT and $\Sigma.\text{Vfy}$ is deterministic.
- Commitment scheme $\text{COM} = (\text{COM.\text{Setup}}, \text{COM.\text{Com}}, \text{COM.\text{Open}})$, where all algorithms are PPT and $\text{COM.\text{Open}}$ is deterministic.
- Threshold Public Key Encryption scheme $\text{TPKE} = (\text{TPKE.\text{Gen}}, \text{TPKE.\text{Enc}},$ $\text{TPKE.\text{Dec}}, \text{TPKE.\text{TDec}}, \text{TPKE.\text{ShareVer}}, \text{TPKE.\text{Combine}}, \text{TPKE.\text{Rand}}, \text{TPKE.\text{Sk2Pk}})$, where all are PPT and $\text{TPKE.\text{Dec}}, \text{TPKE.\text{TDec}}, \text{TPKE.\text{ShareVer}}, \text{TPKE.\text{Combine}}$ and $\text{TPKE.\text{Sk2Pk}}$ are deterministic.
- Public Key Encryption scheme $\text{PKE} = (\text{PKE.\text{Gen}}, \text{PKE.\text{Enc}}, \text{PKE.\text{Dec}})$, where all are PPT and $\text{PKE.\text{Dec}}$ is deterministic.
- Non-interactive Zero-Knowledge scheme $\text{NIZK} = (\text{NIZK.\text{Setup}}, \text{NIZK.\text{Prove}}, \text{NIZK.\text{Verify}})$, where all are PPT and $\text{NIZK.\text{Verify}}$ is deterministic.
- $\mathcal{R}$ — an NP relation. (The code treats $\mathcal{R}$ as a binary function.)
- Relations $\mathcal{R}_{AS}^s$ (see Fig. 11), and $\mathcal{R}_{AS}^{zk}$ (see Fig. 12)

50
– System pids: $pid_{SO}$, $pid_J$, $pid_{AU}$

**States of the Parties:**

– Each user stores:
  • CRS: $crs = (crs_{zk}, crs_{com})$
  • SO public key $vk_{SO}$ (to verify signatures from SO)
  • Public threshold encryption key $pk$
  • Own signing keypair: $(vk_U, sk_U)$
  • Registration data: $(uid, com_{uid}, decom_{uid}, \sigma_{reg})$
  • List of own secrets. Entries are of the form $(secret, com_{sec}, decom_{sec}, vper, com_{vper}, decom_{vper}, \sigma_{ss})$

– LE stores:
  • List of warrants. Entries are of the form $(W, b, \bar{W}, \sigma_{\bar{W}})$
  • List of ciphertexts requested for decryption. Entries are of the form $(uid_i, vper_i, ct_i, sec_{i,1})$

– J stores:
  • Signature keypair $(vk_J, sk_J)$ (to sign warrants)
  • CRS $crs_{com}$
  • List of already processed preliminary warrants. Entries are of the form $(W, b)$
  • List of already processed enhanced warrants. Entries are of the form $(\bar{W}, \sigma_{\bar{W}})$

– SO stores:
  • CRS: $crs = (crs_{zk}, crs_{com})$
  • Public verification key of the judge: $vk_J$
  • Public threshold encryption key $pk$
  • Signature keypair $(vk_{SO}, sk_{SO})$. Messages are either of the form $com_{uid}$ or of the form $(com_{uid}, com_{sec}, com_{vper})$
  • List of registered users. Entries are of the form $(uid, com_{uid}, decom_{uid}, \sigma_{reg}, vk_U)$
  • List of stored partial secrets. Entries are of the form $(uid, vper, ct, \sigma_U, sec_1, com_{sec}, decom_{sec_1}, \sigma_{sec_1})$

**System Setup ($F_{CRS}$):**

– $crs_{zk} \leftarrow \text{NIZK.Setup}(1^\lambda)$
– $crs_{com} \leftarrow \text{COM.Setup}(1^\lambda)$,
– Return $(crs_{zk}, crs_{com})$

**System Init:**

– Input SO: (INIT, SO)
– Input J: (INIT, J)
– Input AU: (INIT, AU)
– Note:
  • If this is the first time, this task is invoked, do whatever is stated in “Behavior”.
• If this is not the first time this task is invoked, ignore the messages.
• Ignore all other messages until this task has been invoked.
• If the parties invoking the task do not have te pids (pid_{SO}, pid_{J}, pid_{AU}), ignore the messages.

Behavior:
1. SO: Create signing keypair \((vk_{SO}, sk_{SO}) \leftarrow \Sigma.Gen(1^\lambda)\) and store it
2. J: Create signing keypair \((vk_{J}, sk_{J}) \leftarrow \Sigma.Gen(1^\lambda)\) and store it
3. Initialize \(\mathcal{F}_{AD}\):
   \[SO \rightarrow \mathcal{F}_{AD}: (\text{Init, SO, } vk_{SO})\]
   \[\mathcal{F}_{AD} \rightarrow SO: (\text{InitFinished})\]
   \[J \rightarrow \mathcal{F}_{AD}: (\text{Init, J, } vk_{J})\]
   \[\mathcal{F}_{AD} \rightarrow J: (\text{InitFinished})\]
   \[AU \rightarrow \mathcal{F}_{AD}: (\text{Init, AU})\]
   \[\mathcal{F}_{AD} \rightarrow AU: (\text{InitFinished})\]
4. SO: Get CRS and keys
   \[SO \rightarrow \mathcal{F}_{CRS}: (\text{value})\]
   \[\mathcal{F}_{CRS} \rightarrow SO: (\text{crs}_{AS}^{zk}, crs_{com})\]
   \[SO \rightarrow \mathcal{F}_{AD}: (\text{GetPK})\]
   \[\mathcal{F}_{AD} \rightarrow SO: (\text{GotPK, pk})\]
   \[SO \rightarrow \mathcal{F}_{AD}: (\text{GetSKeys})\]
   \[\mathcal{F}_{AD} \rightarrow SO: (\text{GotSKeys}, vk_{SO}, vk_{J})\]
5. SO: Store \(crs = (\text{crs}_{AS}^{zk}, crs_{com})\), \(pk\) and \(vk_{J}\)
6. J: Get CRS
   \[J \rightarrow \mathcal{F}_{CRS}: (\text{value})\]
   \[\mathcal{F}_{CRS} \rightarrow J: (\text{crs}_{AS}^{zk}, crs_{com})\]
7. J: Store \(crs = (\text{crs}_{com})\)
   – Output SO: (InitFinished)
   – Output J: (InitFinished)
   – Output AU: (InitFinished)

Party Init:
– Input some Party P: (PInit)
– Behavior P:
  1. Get CRS and keys
     \[P \rightarrow \mathcal{F}_{CRS}: (\text{value})\]
     \[\mathcal{F}_{CRS} \rightarrow P: (\text{crs}_{AS}^S, crs_{com})\]
     \[P \rightarrow \mathcal{F}_{AD}: (\text{GetSKeys})\]
     \[\mathcal{F}_{AD} \rightarrow P: (\text{GotSKeys, } vk_{SO}, vk_{J})\]
     \[P \rightarrow \mathcal{F}_{AD}: (\text{GetPK})\]
     \[\mathcal{F}_{AD} \rightarrow P: (\text{GotPK, pk})\]
  2. Store \(crs = (\text{crs}_{AS}^S, crs_{com})\) and \((pk, vk_{J}, vk_{SO})\)
     – Output to P: (PInitFinished)

User Registration:
– Input U: (Register, uid)
– Input SO: \((\text{Register}, \text{uid}')\)

– Behavior:
  1. U: If no keypair is stored yet, generate keypair \((vk_U, sk_U) \leftarrow \Sigma.\text{Gen}(1^\lambda)\)
     and store it
  2. U: Send \((uid)\) to SO
  3. SO: If \(uid \neq uid'\), abort (wrong inputs)
  4. SO: If there is already an entry \((uid, \cdot, \cdot, \cdot)\) in the list of registered users, abort
     (User already successfully registered.)
  5. SO: Send \((\text{UID_OK})\) to U
  6. U \rightarrow F_{BB}: \((\text{Register}, uid, vk_U)\)
  7. U: Check if registration was successful:
     \(U \rightarrow F_{BB}: \((\text{Retrieve}, uid)\)\)
     \(F_{BB} \rightarrow U: \((\text{Retrieve}, pid_U^*, uid, vk_U^*)\)\)
  8. U: If \(pid_U^* \neq pid_U\) or \(vk_U^* \neq vk_U\), abort (UID already taken by someone else)
  9. U: Send \((\text{OK})\) to SO
  10. SO \rightarrow F_{BB}: \((\text{Retrieve}, uid)\)
     \(F_{BB} \rightarrow U: \((\text{Retrieve}, pid_U^*, uid, vk_U^*)\)\)
  11. SO: If \(pid_U^* \neq pid_U\), abort (UID already taken by another user)
  12. SO: Set \(vk_U := vk_U^*\).
  13. SO: Commit-and-Sign uid:
      \((com_{uid}, decom_{uid}) = \text{COM}.\text{Com}(crs_{com}, uid)\) and \(\sigma_{reg} = \Sigma.\text{Sign}(sk_{SO}, com_{uid})\)
  14. SO: Send \((\sigma_{reg}, com_{uid}, decom_{uid})\) to U
  15. U: Get public key of SO:
      \(U \rightarrow F_{AD}: \((\text{GETSKEYS})\)\)
      \(F_{AD} \rightarrow U: \((\text{GETSKEYS}, vk_{SO}, vk_J)\)\)
  16. U: Store \(vk_{SO}\)
  17. U: Get CRS:
      \(U \rightarrow F_{CRS}: \((\text{VALUE})\)\)
      \(F_{CRS} \rightarrow U: \((crs_{AS}^A, crs_{com})\)\)
  18. U: Store \(crs = (crs_{AS}^A, crs_{com})\)
  19. U: Get public key of TPKE
      \(U \rightarrow F_{AD}: \((\text{GETPK})\)\)
      \(F_{AD} \rightarrow U: \((\text{GETPK}, pk)\)\)
  20. U: Store \(pk\)
  21. U: Verify commitment and signature: If \(\text{COM}.\text{Open}(crs_{com}, com_{uid}, decom_{uid}, uid) = 0\) or \(\Sigma.\text{Vfy}(vk_{SO}, \sigma_{reg}, com_{uid}) = 0\), abort.
  22. U: Store \((uid, com_{uid}, decom_{uid}, \sigma_{reg})\)
  23. SO: Store \((uid, com_{uid}, decom_{uid}, \sigma_{reg}, vk_U)\)
– Output U: \((\text{REGISTERED})\)
– Output SO: \((\text{REGISTERED})\)

\textbf{Store Secret:}
– Input U: \((\text{STORESECRET}, uid, vper)\)
Input SO: \( \text{STORESECRET}, v\text{per}' \)

Behavior:
1. U: Draw part of secret: \( \text{sec}_2 \leftarrow \mathcal{S} \)
2. U: Commit on \( \text{sec}_2: (\text{com}_\text{sec}_2, \text{decom}_\text{sec}_2) \leftarrow \text{COM} (\text{crs}_\text{com}, \text{sec}_2) \)
3. U: Commit on \( v\text{per}: (\text{com}_v\text{per}, \text{decom}_v\text{per}) \leftarrow \text{COM} (\text{crs}_\text{com}, v\text{per}) \)
4. U: Send \( (\text{uid}, \sigma_{\text{reg}}, \text{com}_\text{uid}, \text{decom}_\text{uid}, \text{com}_\text{sec}_2, \text{com}_v\text{per}, \text{decom}_v\text{per}) \) to SO
5. SO: Verify \( \text{com}_v\text{per} \): If \( \text{COM} (\text{crs}_\text{com}, \text{com}_v\text{per}, \text{decom}_v\text{per}, v\text{per}') = 0 \), abort.
6. SO: If \( (\text{uid}, \text{com}_\text{uid}, \text{decom}_\text{uid}, \sigma_{\text{reg}}, \cdot) \) is not in the list of registered Users, abort.
7. SO: If there already exists an entry \( (\text{uid}, v\text{per}, \cdot, \cdot) \) in the list of stored secrets, abort (user already registered a secret for the current validity period).
8. SO: Draw part of secret: \( \text{sec}_1 \leftarrow \mathcal{S} \)
9. SO: Send \( (\text{sec}_1') \) to U
10. U: Compute \( \text{secret} := \text{sec}_1 \oplus \text{sec}_2 \) (\( \oplus \) can be XOR or +, but it must hold that \( \text{secret} \in \mathcal{S} \)).
11. U: Commit on \( \text{secret}: (\text{com}_\text{sec}, \text{decom}_\text{sec}) \leftarrow \text{COM} (\text{crs}_\text{com}, \text{secret}) \)
12. U: Commit and sign \( \text{sec}_1: (\text{com}_\text{sec}_1, \text{decom}_\text{sec}_1) \leftarrow \text{COM} (\text{crs}_\text{com}, \text{sec}_1) \) and \( \sigma_{\text{sec}_1} \leftarrow \Sigma \text{Sign}(\text{sk}_\text{U}, \text{com}_\text{sec}_1) \)
13. U: Encrypt \( \text{secret} \): \( \text{ct} \leftarrow \text{TPKE} (\text{Enc}(\text{pk}, \text{sec}_1); r) \) for fresh randomness \( r \)
14. U: Sign ciphertext: \( \sigma_{\text{U}} \leftarrow \Sigma \text{Sign}(\text{sk}_\text{U}, (\text{ct}, \text{uid}, \text{vper})) \)
15. U: Assemble ZK-Witness \( \text{wit}_\text{ss} := (\text{sec}_1, \text{com}_\text{sec}, \text{com}_\text{sec}_2, \text{ct}, \text{crs}_\text{com}) \)
16. U: Assemble ZK-Statement: \( \text{stmt}_\text{ss} := (\text{sec}_1, \text{com}_\text{sec}, \text{com}_\text{sec}_2, \text{pt}, \text{ct}, \text{crs}_\text{com}) \)
17. U: Compute Proof: \( \pi_{\text{ss}} \leftarrow \text{NIZK} \text{Prove}(\text{crs}_\text{ss}^{\text{AS}}, \text{stmt}_\text{ss}, \text{wit}_\text{ss}, \text{crs}_\text{ss}^{\text{AS}}) \)
18. U: Send \( (\text{ct}, \text{com}_\text{sec}, \pi_{\text{ss}}, \sigma_{\text{U}}, \sigma_{\text{sec}_1}, \text{com}_\text{sec}_1, \text{decom}_\text{sec}_1) \) to SO
19. SO: Assemble ZK-Statement: \( \text{stmt}_\text{U} := (\text{sec}_1, \text{com}_\text{sec}, \text{com}_\text{sec}_2, \text{ct}, \text{crs}_\text{com}) \)
20. SO: Verify Proof: If \( \text{NIZK} \text{Verify}(\text{crs}_\text{ss}^{\text{AS}}, \text{stmt}_\text{ss}, \pi_{\text{ss}}, \text{crs}_\text{ss}^{\text{AS}}) = 0 \), abort.
21. SO: For \( \text{uid} \), retrieve entry \( (\text{uid}, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \) from list of registered users.
22. SO: Verify Commitment: If \( \text{COM} (\text{crs}_\text{com}, \text{com}_\text{sec}, \text{decom}_\text{sec}_1, \text{sec}_1) = 0 \), abort.
23. SO: Verify Signatures: If \( \Sigma \text{Vfy}(\text{vk}_\text{U}, (\text{ct}, \text{uid}, \text{vper}), \sigma_{\text{U}}) = 0 \) or \( \Sigma \text{Vfy}(\text{vk}_\text{U}, \text{com}_\text{sec}, \sigma_{\text{sec}_1}) = 0 \), abort.
24. SO: \( \sigma_{\text{ss}} = \Sigma \text{Sign}(\text{sk}_\text{SO}, (\text{com}_\text{uid}, \text{com}_\text{sec}, \text{com}_\text{vper})) \)
25. SO: Send \( (\sigma_{\text{ss}}) \) to U
26. U: Verify signature: If \( \Sigma \text{Vfy}(\text{vk}_\text{SO}, (\text{com}_\text{uid}, \text{com}_\text{sec}, \text{com}_\text{vper}), \sigma_{\text{ss}}) = 0 \), abort.
27. U: Store \( (\text{secret}, \text{com}_\text{sec}, \text{com}_\text{sec}_1, \text{com}_\text{vper}, \text{decom}_\text{vper}, \sigma_{\text{ss}}) \)
28. SO: Store \( (\text{uid}, \text{vper}, \text{ct}, \sigma_{\text{U}}, \text{sec}_1, \text{com}_\text{sec}_1, \text{decom}_\text{sec}_1, \sigma_{\text{sec}_1}) \)

Output U: \( (\text{SECRETSTORED}, \text{secret}) \)
Output SO: \( (\text{SECRETSTORED}, \text{uid}) \)
Request Warrant:

- Input LE: (RequestWarrant, W)
- Behavior:
  1. LE: Send message (W) to J
  2. J: Check if policy function allows that warrant: If 0 ← fp(W), abort the protocol (Warrant not allowed by policy function).
- Output J: (RequestWarrant, W)
- Input J: (b)
- Behavior:
  1. J: If b = 0, set σW = ⊥. Else, set σW = Σ.Sign(skJ, W)
  2. J: Store (W, b) in list of already requested preliminary warrants
  3. J: Send (b, σW) to LE
  4. LE: Store (W, b, ⊥, ⊥) in list of warrants
  5. LE: If b = 0, skip the next steps and directly go to output
  6. LE → FAD: (GetSKeys)
  7. FAD → LE: (GotSKeys, vkSO, vkJ)
  8. LE: Verify Signature: If Σ.Vfy(vkJ, W, σW) = 0, abort.
  9. LE: Send (W, σW) to SO
  10. SO: Verify Signature: If Σ.Vfy(vkJ, W, σW) = 0, abort.
  11. SO: Parse preliminary warrant: (W1, ..., Wv) ← W and (uidi, vperi, metai) ← Wi
  12. SO: For each (uidi, vperi, ·) entry in W, retrieve the matching (uidi, vperi, ci, σU,i, sec1,i, comsec1,i, decomsec1,i, σsec1,i) entry from internal storage. If none exists, abort (User has not Stored a Secret)
  13. SO: Build enhanced warrant ˜W by adding ci to each ˜Wi
  14. SO: Send (˜W, {σU,i, sec1,i, comsec1,i, decomsec1,i, σsec1,i}i∈{1,...,v}) to LE
  15. LE: Parse enhanced warrant: (˜W1, ..., ˜Wv) ← ˜W and (uidi, vperi, metai, ci) ← ˜Wi
  16. LE: If ˜W ≠ W, abort. (SO sent wrong ˜W)
  17. LE: For i from 1 to v:
     • LE → FBB: (RETRIEVE, uidi)
     • FBB → LE: (RETRIEVE, uidi, vkU,i)
     • Check signatures: If Σ.Vfy(vkU,i, (ci, uidi, vperi), σU,i) = 0 or Σ.Vfy(vkU,i, comsec1,i, σsec1,i) = 0, abort.
     • Check commitment: If COM.Open(crscom, comsec1,i, decomsec1,i, sec1,i) = 0, abort.
  18. LE: Send (˜W, {σU,i, sec1,i, comsec1,i, decomsec1,i, σsec1,i}i∈{1,...,v}) to J
  19. J: Build preliminary warrant W'' out of ˜W by deleting ci from each ˜Wi
  20. J: If there is no entry (W'', 1) in the list of already processed preliminary warrants, abort.
21. J: Check if \((W, \cdot)\) is already in list of stored enhanced warrants. If yes, abort (Warrant already processed).

22. J: Parse enhanced warrant: \((\widetilde{W}_1, \ldots, \widetilde{W}_v) \leftarrow \widetilde{W}\) and \((uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i\)

23. J: For \(i\) from 1 to \(v\):
   - J \(\rightarrow F_{BB}\): (Retrieve, \(uid_i\))
     \(F_{BB} \rightarrow J:\) (Retrieve, \(uid_i, vk_{U,i}\))
   - Check signatures: If \(\Sigma.Vfy(vk_{U,i}, (ct_i, uid_i, vper_i), \sigma_{U,i}) = 0\) or \(\Sigma.Vfy(vk_{U,i}, com_{sec,i}, \sigma_{sec,i}) = 0\), abort.
   - Check commitment: If \(\text{COM.} \text{Open}(\text{crs}_{com}, \text{com}_{sec,i}, \text{decom}_{sec,i}, \text{sec}_{1,i}) = 0\), abort.

24. J: Sign enhanced warrant: \(\sigma_{\widetilde{W}} = \Sigma.\text{Sign}(sk_J, \widetilde{W})\)

25. J: Store \((\widetilde{W}, \sigma_{\widetilde{W}})\) in list of stored enhanced warrants

26. J: Send \((\sigma_{\widetilde{W}})\) to LE

27. LE: Update the entry \((W, b, \perp, \perp)\) in the list of warrants to \((W, b, \widetilde{W}, \sigma_{\widetilde{W}})\)

28. LE \(\rightarrow F_{AD}\): (Request, \(\widetilde{W}, \sigma_{\widetilde{W}}\))
   \(F_{AD} \rightarrow LE:\) (Request)

29. LE: Store each \((uid_i, vper_i, ct_i, sec_{1,i})\) pair for \(i \in \{1, \ldots, v\}\)
   - Output LE: (REQUESTWARRANT, \(b\))

Get Secrets:
- Input LE: (GETSECRETS, \(W\))
- Behavior LE:
  1. Retrieve the entry \((W, 1, \widetilde{W}, \sigma_{\widetilde{W}})\) from the list of warrants. If none exists, abort (warrant not requested or not granted)
  2. Retrieve all \((uid_i, vper_i, ct_i)\) tuples from \(\widetilde{W}\)
  3. For all \((uid_i, vper_i, ct_i)\), retrieve the corresponding entry \((uid_i, vper_i, ct_i, \sigma_{U,i}, sec_{1,i}, com_{sec,i}, \text{decom}_{sec,i}, \text{sec}_{2,i})\) from the list of stored partial secrets
  4. LE \(\rightarrow F_{AD}\): (RETRIEVE)
    \(F_{AD} \rightarrow LE:\) (RETRIEVE, \(L_{\text{ReadyRequests}}\))
  5. For each \((uid_i, vper_i, ct_i, sec_{1,i})\):
     a. Search for an entry \((ct_j, sec_{2,j})\) in \(L_{\text{Requests}}\) where \(ct_i = ct_j\).
     b. If one exists, set \(secret_i := sec_{1,i} \oplus sec_{2,j}\).
     c. If none exists, set \(secret_i := \perp\).
   - Output LE: (GOTSECRETS, \((secret_1, \ldots, secret_v)\))

Get Statistics:
- Input some Party P: (GETSTATISTICS)
- Behavior P:
  1. P \(\rightarrow F_{AD}\): (GETSTATISTICS)
    \(F_{AD} \rightarrow P:\) (GOTSTATISTICS, \(L_{\text{Stats}}\))
B.3 The Functionality $\mathcal{F}_{\text{AD}}$

We now give the complete description of the secret storing functionality $\mathcal{F}_{\text{AD}}$.

<table>
<thead>
<tr>
<th>$\mathcal{F}_{\text{AD}}$</th>
</tr>
</thead>
</table>

System Parameters:
- $t_{\text{com}}$ — Number of clock ticks between committee handovers
- $f_t$ — Transparency function. Gets a warrant $W$ as input and outputs what should be publicly known about that warrant. Interface is $W^{\text{pub}} \leftarrow f_t(W)$.
- Signature scheme $\Sigma = (\Sigma.\text{Gen}, \Sigma.\text{Sign}, \Sigma.\text{Vfy})$, where all are PPT and $\Sigma.\text{Vfy}$ is deterministic.
- Threshold Public Key Encryption scheme $\text{TPKE} = (\text{TPKE.}\text{Gen}, \text{TPKE.}\text{Enc}, \text{TPKE.}\text{Dec}, \text{TPKE.}\text{TDec}, \text{TPKE.}\text{ShareVer}, \text{TPKE.}\text{Combine}, \text{TPKE.}\text{Rand}, \text{TPKE.}\text{Sk2Pk})$, where all are PPT and $\text{TPKE.}\text{Dec}$, $\text{TPKE.}\text{TDec}$, $\text{TPKE.}\text{ShareVer}$, $\text{TPKE.}\text{Combine}$ and $\text{TPKE.}\text{Sk2Pk}$ are deterministic.
- Public Key Encryption scheme $\text{PKE} = (\text{PKE.}\text{Gen}, \text{PKE.}\text{Enc}, \text{PKE.}\text{Dec})$, where all are PPT and $\text{PKE.}\text{Dec}$ is deterministic.
- Non-interactive Zero-Knowledge scheme $\text{NIZK} = (\text{NIZK.}\text{Setup}, \text{NIZK.}\text{Prove}, \text{NIZK.}\text{Verify})$, where all are PPT and $\text{NIZK.}\text{Verify}$ is deterministic.
- System pids: $\text{pid}_{SO}, \text{pid}_J, \text{pid}_{AU}$
- The functionality has access to a global clock $G_{\text{CLOCK}}$.

**Functionality State:**
- $t_{\text{last}}$: Time of last committee handover.
- $\{\text{pk, sk}\}$: Keypair for TPKE
- $\text{crs}$: Common reference string for NIZK
- $L_{\text{Warrants}}$: List of Warrants on the blockchain. Entries are of the form $\tilde{W}$.
- $L_{\text{Requests}}$: List of Requests for decryption not yet ready for retrieval.
- $L_{\text{Ready}}$: Lists of Requests for decryption ready to be retrieved.
- $\text{vk}_{SO}$: Public signing key of SO. Not needed in this functionality, it only exists so parties in $\Pi_{\text{AS}}$ can get it.
- $\text{vk}_{J}$: Public signing key of J. Used to verify warrant signatures and so parties in $\Pi_{\text{AS}}$ can get it.
- $\{d_{\text{pid}}, \text{pid} \in \{N_i\}\}$: Flag whether the node $\text{pid}$ has called $\text{EXECUTEROLE}$ since the last clock tick.

**Init**
Note: for each party only execute this task once. Ignore messages other than (Init) or (GetCRS) until SO, J and AU each have called this task.
- Input SO: (Init, SO, $\text{vk}_{SO}$)
- Behavior:
  1. Send (Register) to $G_{\text{CLOCK}}$
  2. Set $\text{cnm} := 0$
  3. Generate encryption key pair: $(\text{pk}, \text{sk}, \{\text{vk}_i\}, \{\text{sk}_i\}) \leftarrow \text{TPKE.}\text{Gen}(1^\lambda)$ and store $(\text{pk}, \text{sk})$
  4. Set $t_{\text{last}} := 0$
  5. Send (Init, SO, $\text{vk}_{SO}.\text{pk}$) to $A$
  6. Store $\text{vk}_{SO}$
- Output SO: (InitFinished)
– Input J: (INIT, J, vk_j)
– Behavior:
1. Send (INIT, J, vk_j) to A
2. Store vk_j
– Output J: (INITFinished)
– Input AU: (INIT, AU)
– Behavior:
1. Send (INIT, AU) to A
– Output AU: (INITFinished)

Get Public Key:
– Input some Party P: (GETPK)
– Behavior:
1. Send (GETPK) to the adversary
2. Retrieve (pk, sk) := (pk, sk)
– Output P: (GotPK, pk)

Get System Keys:
– Input some Party P: (GETSKEYS)
– Behavior:
1. Send (GETSKEYS) to the adversary
2. Retrieve vk_SO and vk_j
– Output P: (GotSKEYS, vk_SO, vk_j)

Request Decryption:
– Input LE: \( \left( \text{REQUEST}, \bar{W}, \sigma_{\bar{W}} \right) \)
– Behavior:
1. Check Warrant Signature: \( b \leftarrow \sum Vfy(vk_j, \bar{W}, \sigma_{\bar{W}}) \)
2. If \( b = 0 \), abort (Warrant not valid)
3. Parse warrant: \( \left( \bar{W}_1, \ldots, \bar{W}_v \right) \leftarrow \bar{W} \) and \( (uid_i, vper_i, meta_i, ct_i) \leftarrow W_i \)
4. For each \( i \), store \( ct_i \) in list of requests \( L_{\text{Requests}} \)
5. Send (REQUEST, pid, \( f_t(W), |\bar{W}|, v \)) to the adversary, where \( pid \) is the id of the calling party
6. When the adversary allows to deliver output, store \( \bar{W} \) in List of Warrants \( L_{\text{Warrants}} \)
– Output LE: (REQUEST)

Retrieve Secret:
– Input LE: (RETRIEVE)
– Behavior:
1. If LE is honest, send (RETRIEVE) to the adversary
2. If LE is corrupted, send \( (\text{RETRIEVE}, L_{\text{PendingRequests}}) \) to the adversary
Output LE: \((\text{Retrieve}, L_\text{Requests})\)

Get Statistics:
- Input some Party P: (GetStatistics)
- Behavior:
  1. Send \((\text{GetStatistics})\) to the adversary
  2. Initialize empty list \(L_\text{Stats}\)
  3. Retrieve list of Warrants \(L_\text{Warrants}\), entries are of the form \(\bar{W}\)
  4. For each \(\bar{W} \in L_\text{Warrants}\): Build \(W\) from \(\bar{W}\), calculate \(W^{\text{pub}} \leftarrow f_t(W)\)
     and append \(W^{\text{pub}}\) to \(L_\text{Stats}\)
- Output P (immediately): (GotStatistics, \(L_\text{Stats}\))

Audit:
- Input AU: (AuditRequest)
- Behavior:
  1. If the party invoking this task has not the pid \(pid_\text{AU}\), then abort.
  2. Send \((\text{AuditRequest})\) to the adversary
  3. Retrieve list of Warrants \(L_\text{Warrants}\)
- Output AU (immediately): (AuditAnswer, \(L_\text{Warrants}\))

RoleExecute:
- Input N: (ExecuteRole)
- Behavior:
  1. Send \((\text{ExecuteRole}, pid_\text{N})\) to the adversary
  2. Set \(d_\text{N} := 1\)
  3. If for all honest nodes \(N_i\), \(d_\text{N}_i = 1\), execute ClockTick
- Output N: (ExecuteRole)

Handling corruptions:
- Upon receiving a message \((\text{Corrupt}, pid)\) for \(pid \in \{N_i\}\)
  1. Mark \(N_i\) as corrupted
  2. If as a result of this operation for all remaining honest nodes \(N_i\), \(d_\text{N}_i = 1\), execute ClockTick

ClockTick:
1. Reset \(d_{\text{pid}} := 0\) for all nodes
2. Send \((\text{Clock-Update}, sid)\) to \(G_{\text{CLOCK}}\)
3. Send \((\text{Clock-Read}, sid)\) to \(G_{\text{CLOCK}}\) and receive \((\text{Clock-Read}, sid, t_{\text{Now}})\)
4. If \(t_{\text{last}} + t_{\text{com}} \leq t_{\text{Now}}\) execute Handover.

Handover:
1. Set \(L_{\text{Pending Requests}} := L_{\text{Requests}} \cup L_{\text{Pending Requests}}\) and clear \(L_{\text{Requests}}\)
2. For each entry \(ct_i \in L_{\text{Requests}}\):
   (a) \(secret_i \leftarrow \text{TPKE.Dec(sk, ct_i)}\)
   (b) Add \((ct_i, secret_i)\) to \(L_{\text{Pending Requests}}\)
3. Clear \(L_{\text{Requests}}\)
4. Set time of last handover: \(t_{\text{last}} := t_{\text{Now}}\)
B.4 The Protocol $Π_{AD}$

Details of the relation $R_{w}^{AD}$

\[
\text{wit} := (\overline{W}, r, \sigma_{\overline{W}}, \{ct_i, r_i\}_{i \in [v]}) \\
\text{stmt} := (ek_{AU}, vk_{J}, W_{enc}, W_{pub}, \{\hat{ct}_i\}) \\
R_{w}^{AD} = \{(\text{stmt}, \text{wit}) | \\
- W_{enc} = \text{PKE.Enc}(ek_{AU}, \overline{W}; r) \\
- W_{pub} = f_{i}(W) \\
- \Sigma.Vfy(vk_{J}, \overline{W}, \sigma_{\overline{W}}) \\
- W \text{ is } \overline{W} \text{ but without the ciphertexts} \\
- \text{For each } i \in [v]: \\
- \hat{ct}_i \leftarrow \text{TPKE.Rand}(ct_i, r_i) \}
\]

Fig. 13: ZK-Relation $R_{w}^{AD}$

In $Π_{AD}$, we use the following NP relations for zero-knowledge proofs:

- Relation $R_{w}^{AD}$ (see Fig. 13) is used in Request Secrets by law enforcement to prove that they possess a warrant signed by the judge and correctly evaluated the transparency function.
- Relations $R_{KG1}^{AD}$ and $R_{KG2}^{AD}$ (see Fig. 14) are used during initial key generation to ensure the resulting keypair is generated properly.
- Relation $R_{KS}^{AD}$ (see Fig. 15) is used during each committee handover to ensure correct resharing of the secret key.
- Relation $R_{Dec}^{AD}$ (see Fig. 16) is used when answering decryption requests to prove correct partial decryption.

We now give the complete description of the protocol $Π_{AD}$ that UC-realizes the auditable decryption functionality $F_{AD}$. 
Details of the relation $R_{ADKG1}$

\[
\text{wit} := (sk_i, \{r_j\}_{j \in [n]}, F)
\]
\[
\text{stmt} := (pk_i, \{ct_j, ek_{R_j}\}_{j \in [n]})
\]
\[
R_{ADKG1} = \{(\text{stmt, wit}) | \quad - \quad F \text{ is a degree } t \text{ polynomial}
\]
\[
\quad - \quad \text{For each } j \in [n]:
\quad \quad \bullet \quad ct_j \leftarrow \text{PKE.Enc}(ek_{R_j}, F(j); r_j)
\}

Details of the relation $R_{ADKG2}$

\[
\text{wit} := (sk_i, \{r_j\}_{j \in [n]}, dk_{R}, F)
\]
\[
\text{stmt} := (pk_i, \{ct^k\}_{k \in [t+1]}, \{ct_j, ek_{R_j}\}_{j \in [n]}, ek_{R})
\]
\[
R_{ADKG2} = \{(\text{stmt, wit}) | \quad - \quad \text{PKE.Sk2Pk}(dk_{R}) = ek_{R}
\]
\[
\quad - \quad sk_i = \sum_{k=1}^{t+1} \text{PKE.Dec}(dk_{R}, ct^k)
\]
\[
\quad - \quad \text{TPKE.Sk2Pk}(sk_i) = pk_i
\]
\[
\quad - \quad F \text{ is a degree } t \text{ polynomial with } F(0) = sk_i
\]
\[
\quad - \quad \text{For each } j \in [n]:
\quad \quad \bullet \quad ct_j \leftarrow \text{PKE.Enc}(ek_{R_j}, F(j); r_j)
\}

Fig. 14: ZK-Relations $R_{ADKG1}$ and $R_{ADKG2}$
Details of the relation $R_{AD}^{KS}$

\[
\text{wit} := (dk_R, sk_i, \{r_j\}_{j \in [n]}, F, G)
\]
\[
\text{stmt} := (ek_R, \{ct^k\}_{k \in [t+1]}, \{ct_j, ek_R\}_{j \in [n]}, pk_i)
\]
\[
R_{AD}^{KS} = \{ (\text{stmt}, \text{wit}) | \]
- PKE. Sk(Pk(dk_R)) = ek_R  
- TPKE. Sk(Pk(sk_i)) = pk_i  
- $G$ and $F$ are both degree $t$ polynomials  
- $G(i) = F(0) = sk_i$  
- For each $k \in [t+1]$:  
  - $sh_k \leftarrow \text{PKE. Dec}(dk_R, ct^k)$  
  - $G$ is obtained by interpolating $sh_1, \ldots, sh_{t+1}$  
- For each $j \in [n]$:  
  - $ct_j \leftarrow \text{PKE.Enc}(ek_R, G(j); r_j)$  
\}

Fig. 15: ZK-Relation $R_{AD}^{KS}$

Details of the relation $R_{Dec}^{AD}$

\[
\text{wit} := (sk_i)
\]
\[
\text{stmt} := (vk_i, ct, ct^*)
\]
\[
R_{Dec}^{AD} = \{ (\text{stmt}, \text{wit}) | \]
- TPKE. Sk(Pk(sk_i)) = vk_i  
- ct^* \leftarrow \text{TPKE.TDec}(sk_i, ct)
\}

Fig. 16: ZK-Relation $R_{Dec}^{AD}$
Building Blocks:
- NIZK proof system \( \text{NIZK} = (\text{NIZK}.\text{Setup}, \text{NIZK}.\text{Prove}, \text{NIZK}.\text{Verify}) \)

System Parameters:
- \( n \) — Number of committee members
- \( t_{\text{com}} \) — Number of clock ticks between committee handovers
- \( f_t \) — Transparency function. Gets a preliminary warrant \( W \) as input and outputs what should be publicly known about that warrant. Interface is \( W^{\text{pub}} \leftarrow f_t(W) \).
- Signature scheme \( \Sigma = (\Sigma.\text{Gen}, \Sigma.\text{Sign}, \Sigma.\text{Vfy}) \), where all algorithms are PPT and \( \Sigma.\text{Vfy} \) is deterministic.
- Threshold Public Key Encryption scheme \( \text{TPKE} = (\text{TPKE.\text{Gen}}, \text{TPKE.\text{Enc}}, \text{TPKE.\text{Dec}}, \text{TPKE.\text{TDec}}, \text{TPKE.\text{ShareVer}}, \text{TPKE.\text{Combine}}, \text{TPKE.\text{Rand}}, \text{TPKE.\text{Sk2Pk}}) \), where all algorithms are PPT and \( \text{TPKE.\text{Dec}}, \text{TPKE.\text{TDec}}, \text{TPKE.\text{ShareVer}}, \text{TPKE.\text{Combine}} \) and \( \text{TPKE.\text{Sk2Pk}} \) are deterministic.
- Public Key Encryption scheme \( \text{PKE} = (\text{PKE.\text{Gen}}, \text{PKE.\text{Enc}}, \text{PKE.\text{Dec}}) \), where all are PPT and \( \text{PKE.\text{Dec}} \) is deterministic.
- Non-interactive Zero-Knowledge scheme \( \text{NIZK} = (\text{NIZK}.\text{Setup}, \text{NIZK}.\text{Prove}, \text{NIZK}.\text{Verify}) \), where all are PPT and \( \text{NIZK.\text{Verify}} \) is deterministic.
- ZK-Relations \( R_{\text{AD}}^{\text{W}} \) (see Fig. 13), \( R_{\text{AD}}^{\text{KG}_1}, R_{\text{AD}}^{\text{KG}_2} \) (see Fig. 14), \( R_{\text{AD}}^{\text{KS}} \) (see Fig. 15) and \( R_{\text{AD}}^{\text{Dec}} \) (see Fig. 16).
- System pids: \( \text{pid}_S, \text{pid}_J, \text{pid}_A, \text{pid}_L \)

State of the Parties:
- LE stores:
  - \( (\text{ek}_L, \text{sk}_L) \)
- AU stores:
  - \( (\text{ek}_A, \text{sk}_A) \)

System Setup (\( \mathcal{F}_{\text{CRS}} \)):
- \( \text{crs}_{\text{zk}}^{\text{SS}} \leftarrow \text{NIZK.\text{Setup}}(1^\lambda) \)
- Return \( \text{crs}_{\text{zk}}^{\text{SS}} \)

Init:
- Note:
  - Each of SO, J, AU only executes the following the first time this input is received and ignores all further inputs of this form.
  - Each of SO, J, AU ignores all other inputs and messages until this has been executed.
- Input SO: (\( \text{Init}, \text{SO}, \text{vk}_S \))
- Behavior SO:
  1. Send (\( \text{POST}, (\text{OperatorKey}, \text{vk}_S) \)) to \( \mathcal{F}_{\text{BCRA}} \)
- Output SO: (\( \text{InitFinished} \))
Input J: \((\text{Init}, J, \text{vk}_J)\)

Behavior J:
1. Send \((\text{Post}, (\text{JudgeKey}, \text{vk}_J))\) to \(\mathcal{F}_{\text{BCRA}}\)
2. Output J: \((\text{InitFinished})\)

Input AU: \((\text{Init}, AU)\)

Behavior AU:
1. Generate \((\text{ek}_AU, \text{sk}_AU) \leftarrow \text{PKE.Gen}(1^\lambda)\)
2. Send \((\text{Post}, (\text{AuditorKey}, \text{ek}_AU))\) to \(\mathcal{F}_{\text{BCRA}}\)
3. Store \((\text{ek}_AU, \text{sk}_AU)\)
4. Output AU: \((\text{InitFinished})\)

Get Public Key:

Input some Party P: \((\text{GetPK})\)

Behavior:
1. Send \((\text{Read})\) to \(\mathcal{F}_{\text{BCRA}}\) and receive \((\text{Ordered})\)
2. For each role \(R_i\) in the first key generation committee do the following:
   a. Find \((\text{roleassign}, t, (\text{Generate}, R_i, \text{ek}_R, \text{vk}_R))\) in \(\text{Ordered}\) and retrieve \(\text{vk}_R\)
   b. Find \((R, msg, \sigma)\) with \(\sigma \equiv (\text{KeyGen1}, \{(ct_j)_{j \in [n]}, \pi_{KG}\})\) in \(\text{Ordered}\)
   c. Check \(\Sigma.\text{Vfy}(\text{vk}_R, msg, \sigma) = 1\), otherwise ignore this role
   d. Assemble ZK-Statement:
      \[\text{stmt}_{KG} := (pk_i, \{ct_j, \text{ek}_R\}_{j \in [n]})\]
   e. Check \(\text{NIZK.\text{Verify}}(crs_{\text{SS}}, \text{stmt}_{KG}, \pi_{KG}, \mathcal{R}_{AD_{KG1}}) = 1\), otherwise ignore this role
   f. Add \(\{ct_j\}_{j \in [n]}\) to \(L^t_{\text{Qual}}\)
3. Sort \(L^t_{\text{Qual}}\) lexicographically
4. For each role \(R_i\) in the second key generation committee do the following:
   a. Find \((\text{roleassign}, t, (\text{Generate}, R_i, \text{ek}_R, \text{vk}_R))\) in \(\text{Ordered}\) and retrieve \(\text{ek}_R, \text{vk}_R\)
   b. Find \((R, msg, \sigma)\) with \(\sigma \equiv (\text{KeyGen2}, \{(pk_i, \{ct_j\}_{j \in [n]}, \pi_{KG}\})\) in \(\text{Ordered}\)
   c. Check \(\Sigma.\text{Vfy}(\text{vk}_R, msg, \sigma) = 1\), otherwise ignore this role
   d. Retrieve the first \(t + 1\) ciphertexts \(ct_1^t, \ldots, ct_{t+1}^t\) for this role from \(L^t_{\text{Qual}}\) as \(\{ct_k\}_{k \in [t+1]}\)
   e. Assemble ZK-Statement:
      \[\text{stmt}_{KG} := (pk_i, \{ct_k\}_{k \in [t+1]}, \{ct_i\}_{i \in [n]}, \text{ek}_R)\]
   f. Check \(\text{NIZK.\text{Verify}}(crs_{\text{SS}}, \text{stmt}_{KG}, \pi_{KG}, \mathcal{R}_{AD_{KG2}}) = 1\), otherwise ignore this role
   g. Add \(pk_i\) to \(L_{\text{Qual}}\)
5. Sort \(L_{\text{Qual}}\) lexicographically

65
6. Reconstruct \( \text{pk} \) via Lagrange-interpolation from the first \( t + 1 \) entries in \( \text{L}_\text{Qual} \)

- Output P: \((\text{GotPK}, \text{pk})\)

Get System Keys:

- Input some Party P: \(\text{(GetSKeys)}\)
- Behavior P:
  1. Send \(\text{(READ)}\) to \(\mathcal{F}_{\text{BCRA}}\) and receive \(\text{(ORDERED)}\)
  2. Find \((\text{pid}_{SO}, t_1, (\text{OperatorKey}, \text{vk}_{SO}))\) in ORDERED, otherwise abort
  3. Find \((\text{pid}_j, t_2, (\text{JudgeKey}, \text{vk}_j))\) in ORDERED, otherwise abort
- Output P: \((\text{GotSKeys}, \text{vk}_{SO}, \text{vk}_j)\)

Request Decryption:

- Input LE: \(\text{(REQUEST, } \overline{W}, \sigma_{\overline{W}})\)
- Behavior LE:
  1. Initialize empty list \(\text{L}_{\text{CommitteePK}}\)
  2. If \(\text{sk}_{LE} = \bot\) run \((\text{ek}_{LE}, \text{sk}_{LE}) \leftarrow \text{PKE.Gen}(1^\lambda)\)
  3. Send \(\text{(READ)}\) to \(\mathcal{F}_{\text{BCRA}}\) and receive \(\text{(ORDERED)}\)
  4. Find \((\text{pid}_J, t_2, (\text{JudgeKey}, \text{vk}_J))\) in ORDERED, otherwise abort
  5. Find \((\text{pid}_{AU}, t_3, (\text{AuditorKey}, \text{ek}_{AU}))\) in ORDERED, otherwise abort
  6. Build preliminary warrant \(W\) out of \(\tilde{W}\) by deleting the ciphertext from each entry.
  7. Compute transparency function of warrant: \(W^{\text{pub}} \leftarrow f_t(W)\)
  8. Encrypt warrant to the auditor: \(W^{\text{enc}} \leftarrow \text{PKE.Enc}(\text{ek}_{AU}, \overline{W}; r)\) with randomness \(r \in \mathcal{R}\)
  9. Parse warrant: \((\tilde{W}_1, \ldots, \tilde{W}_v) \leftarrow \overline{W}\) and \((\text{uid}_i, \text{eper}, \text{meta}_i, \text{ct}_i) \leftarrow \tilde{W}_i\)
  10. For each \(i \in [v]\), re-randomize \(\hat{\text{ct}}_i \leftarrow \text{TPKE.Rand}(\text{ct}_i; r_i)\)
  11. Assemble ZK-Witness: \(\text{wtr}_{\overline{W}} := (\overline{W}, r, \sigma_{\overline{W}}, \{\text{ct}_i\}_{i \in [v]})\)
  12. Assemble ZK-Statement:
      \[\text{stmt}_{\overline{W}} := (\text{ek}_{AU}, \text{vk}_J, W^{\text{enc}}, W^{\text{pub}}, \{\hat{\text{ct}}_i\}_{i \in [v]})\]
  13. Compute Proof:
      \[\pi_{\overline{W}} \leftarrow \text{NIZK.Prove}(\text{crs}_{\text{SS}}, \text{stmt}_{\overline{W}}, \text{wtr}_{\overline{W}}, \text{RAP})\]
  14. Send \((\text{POST}, \text{(REQUEST, } W^{\text{pub}}, W^{\text{enc}}, \{\hat{\text{ct}}_i\}_{i \in [v]}, \pi_{\overline{W}}, \text{ek}_{LE}))\) to \(\mathcal{F}_{\text{BCRA}}\)
- Output LE: \(\text{(REQUEST)}\)

Retrieve Secret:

- Input LE: \(\text{(RETRIEVE)}\)
- Behavior of LE:
  1. Send \(\text{(READ)}\) to \(\mathcal{F}_{\text{BCRA}}\) and receive \(\text{(ORDERED)}\)
  2. For each entry \((\text{pid}, t, R, (\text{REQUEST, } C), \sigma)\) in ORDERED:
     (a) Find \(\text{roleassign}(t, R, \text{ek}_R, \text{vk}_R)\) in ORDERED
     (b) Verify \(\Sigma.\text{Vfy}((\text{vk}_R, \text{REQUEST}, C), \sigma)\), otherwise ignore this entry
     (c) Find \((\text{pid}, t, R, \text{msg}, \sigma')\) with \(\text{msg} := (\text{KeyShare}, (\text{vk}_i, \{\text{ct}_k\}, \pi_{KS}))\)
        in ORDERED

66
(d) Verify \( \Sigma.\text{Vfy}(vk_R, msg, \sigma') \), otherwise ignore this entry
(e) Decrypt \((ct, ct^*, \pi_{Dec}) \leftarrow \text{PKE Dec}(sk_LE, C)\)
(g) Assemble ZK-Statement:
\[ stmt_{Dec} := (vk, ct, ct^*) \]
(h) Verify \( \text{NIZK Verify}(crs_{SS}^{rk}, stmt_{Dec}, \pi_{Dec}, \mathcal{R}_{AO}) \), otherwise ignore this entry

(i) Add \( ct^* \) to \( L_{ct Qual} \)
3. Sort all lists \( L_{ct Qual}^{t+1} \) lexicographically
4. For each list \( L_{ct Qual}^{t+1} \):
   (a) retrieve the first \( t+1 \) values \( ct^* \) from \( L_{ct Qual}^{t+1} \), ignore this list if \( |L_{ct Qual}^{t+1}| < t+1 \)
   (b) Obtain decryption \( secret \leftarrow \text{TPKE Combine}(\{ct^*_k\}_{k \in [t+1]}, ct) \)
   (c) Add \((ct, secret)\) to \( L_{ct Qual} \)

Output LE: \( \{\text{RETRIEVE}, L_{ct Qual}^{t+1}\} \)

Get Statistics:
- Input some Party P: \( \text{(GetStatistics)} \)
  - Behavior P:
    1. Initialize empty list \( L_{Stats} \)
    2. Send \( \text{(READ)} \) to \( J_{BCRA} \) and receive \( \text{(ORDERED)} \)
    3. Find \((pid, t_2, (\text{JudgeKey}, vk_J))\) in \text{ORDERED}, otherwise abort
    4. Find \((pid_{AU}, t_3, (\text{AuditorKey}, ek_{AU}))\) in \text{ORDERED}, otherwise abort
    5. For each entry \((pid, t, (id, W_{pub}, W_{enc}, \pi_{\widetilde{W}}))\):
       (a) Assemble ZK-Statement:
           \[ stmt_{\widetilde{W}} := (ek_{AU}, vk_J, W_{enc}, W_{pub}) \]
       (b) Verify Proof:
           \[ b \leftarrow \text{NIZK Verify}(crs_{SS}^{rk}, stmt_{\widetilde{W}}, \pi_{\widetilde{W}}) \]
           If \( b = 1 \) and \( W_{pub} \notin L_{Stats}, \) store \( W_{pub} \) in \( L_{Stats} \)
       (c) Output P: \( \text{(GotStatistics, L_{Stats})} \)

Audit:
- Input AU: \( \text{(AuditRequest)} \)
  - Behavior AU:
    1. Initialize empty list \( L_{Warrants} \)
    2. Retrieve stored \((ek_{AU}, sk_{AU})\)
    3. Send \( \text{(READ)} \) to \( J_{BCRA} \) and receive \( \text{(ORDERED)} \)
    4. Find \((pid, t_2, (\text{JudgeKey}, vk_J))\) in \text{ORDERED}, otherwise abort
    5. For each entry \((pid, t, (id, W_{pub}, W_{enc}, \pi_{\widetilde{W}}))\) in \text{ORDERED}:
       (a) Assemble ZK-Statement:
           \[ stmt_{\widetilde{W}} := (ek_{AU}, vk_J, W_{enc}, W_{pub}) \]
       (b) Verify Proof: If \( \text{NIZK Verify}(crs_{SS}^{rk}, stmt_{\widetilde{W}}, \pi_{\widetilde{W}}) = 0 \), skip the following steps and proceed to next entry
       (c) Decrypt \( W_{enc}: W \leftarrow \text{PKE Dec}(sk_{AU}, W_{enc}) \)
       (d) If \( W \notin L_{Warrants}, \) store \( W \) in \( L_{Warrants} \)
Behavior of Nodes:

– Upon being activated for the first time, send (Register) to \( G_{\text{CLOCK}} \).

– Upon receiving a message (Generate, \( R, ek_R, dk_R, vk_R, sk_R \)) from \( F_{\text{BCRA}} \), store \( (R, ek_R, dk_R, vk_R, sk_R) \) in \( \text{L}_{\text{Roles}} \).

– Upon receiving input (ExecuteRole):
  1. Send \( (\text{Clock-Read}) \) to \( G_{\text{CLOCK}} \) and receive \( (t_{\text{Now}}) \).
  2. Set \( cnum := \left\lfloor \frac{t_{\text{Now}}}{t_{\text{com}}} \right\rfloor \).
  3. Send \( (\text{READ}) \) to \( F_{\text{BCRA}} \) and receive \( (\text{ORDERED}) \).
  4. Find \( (\text{pid}, t, (\text{JudgeKey}, vk_J)) \) in \( \text{ORDERED} \), otherwise abort
  5. Find \( (\text{pid}, t, (\text{AuditorKey}, ek_AU)) \) in \( \text{ORDERED} \), otherwise abort
  6. For each role in \( \text{L}_{\text{Roles}} \), check if it should be executed this round, if yes execute it.
  7. After having executed all roles for this round, send (Clock-Update) to \( G_{\text{CLOCK}} \).

– Executing Role KeyGen1:
  1. Retrieve \( sk_R \) associated with this role \( R_i \) from \( \text{L}_{\text{Roles}} \).
  2. Initialize empty list \( \text{L}_{\text{CommitteePK}}^{\text{next}} \).
  3. For each role \( R_j \) in the next committee
    (a) Find \( (\text{roleassign}, t, (\text{Generate}, R_j, ek_{R_j}, vk_{R_j})) \) in \( \text{ORDERED} \)
    (b) Insert \( ek_{R_j} \) into \( \text{L}_{\text{CommitteePK}}^{\text{next}} \).
  4. Generate key share: \( s_i \leftarrow Z_p \).
  5. Share secret key: choose a random degree \( t \) polynomial \( F(x) = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_t * x^t \) with \( F(0) = s_i \)
  6. For each role \( R_j \) in the next committee:
    (a) Set \( sh_j : = F(j) \)
    (b) Generate ciphertext \( ct_j : = \text{PKE.Enc(ek}_{R_j}, sh_j; r_j) \)
  7. Assemble ZK-Witness: \( \text{wit}_{KG} : = (s_i, \{r_j\}_{j \in [n]}, F) \)
  8. Assemble ZK-Statement:
    \( \text{stmt}_{KG} : = (\{ ct_j, ek_{R_j} \}_{j \in [n]} ) \)
  9. Compute Proof:
    \( \pi_{KG} : = \text{NIZK.Prove(crs}^{ss}_{sk}, \text{stmt}_{KG}, \text{wit}_{KG}, \text{RAD}^{KG}) \)
  10. Prepare message: \( \text{msg} : = (\text{KEYGen1}, (\{ ct_j \}_{j \in [n]} ), \pi_{KG}) \)
  11. Sign message: \( \sigma : = \Sigma.\text{Sign}(sk_R, \text{msg}) \)
  12. Remove the entry for this role from \( \text{L}_{\text{Roles}} \).
  13. Delete all state except \( \text{msg}, \sigma \).
  14. Send \( (\text{POST}, (R_i, \text{msg}, \sigma)) \) to \( F_{\text{BCRA}} \).

– Executing Role KeyGen2:
  1. Retrieve \( sk_R \) associated with this role \( R_i \) from \( \text{L}_{\text{Roles}} \).
  2. Initialize empty lists \( \text{L}_{\text{CommitteePK}}^{\text{next}}, \text{L}_{\text{CommitteePK}}^{\text{current}}, \text{L}_{\text{CommitteeVK}} \).
  3. For each role \( R_j \) in the next committee
    (a) Find \( (\text{roleassign}, t, (\text{Generate}, R_j, ek_{R_j}, vk_{R_j})) \) in \( \text{ORDERED} \)
Executing Role Handover:

4. For each role $R_j$ in the current committee:
   (a) Find $(\text{roleassign}, t, (\text{GENERATE}, R_j, \text{ek}_R, \text{vk}_R))$ in ORDERED
   (b) Insert $\text{ek}_R$ into $\text{LCommitteePK}^{\text{current}}$

5. For each role $R_j$ in the previous committee:
   (a) Find $(\text{roleassign}, t, (\text{GENERATE}, R_j, \text{ek}_R, \text{vk}_R))$ in ORDERED
   (b) Insert $\text{vk}_R$ into $\text{LCommitteeVK}^{\text{current}}$

6. For each entry $(\text{pid}, t, (R_j, (\text{KEYGEN1}, \{\{ct_j\}_{j \in [n]}, \pi_{KG}\}), \sigma))$ in ORDERED
   (a) Retrieve $\text{vk}_R$ from $\text{LCommitteeVK}^{\text{current}}$
   (b) Check if $\Sigma \text{Vfy}(\text{vk}_R, (\text{KEYGEN1}, \{\{ct_j\}_{j \in [n]}, \pi_{KG}\}), \sigma) = 1$, otherwise skip this entry.
   (c) Assemble ZK-Statement:
       $$\text{stmt}_{KG} := (\{ct_j, \text{ek}_R\}_{j \in [n]})$$
   (d) If $\text{NIZK.Verify}((\text{crs}_{2\delta}, \text{stmt}_{KG}, \pi_{KG}, \mathcal{R}_{AD}^{\text{crs}_{2\delta}})) \neq 1$ skip this entry.
   (e) Add $ct_t$ to $L_{\text{Qual}}$ (the one addressed to the currently executing role)

7. Sort $L_{\text{Qual}}$ lexicographically

8. Obtain the first $t+1$ entries $ct^1_t, \ldots, ct^{t+1}_t$ from $L_{\text{Qual}}$

9. Decrypt $ct^i_t$ as $sh_k \leftarrow \text{PKE.Dec}(\text{dk}_R, ct^i_t)$

10. Set $sk_i = \sum_{k=1}^{t+1} sh_k$ and $pk_i = \text{TPKE.Sk2Pk}(sk_i)$

11. Share secret key: choose a random degree $t$ polynomial $F(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_t \cdot x^t$ with $F(0) = sk_i$

12. For each role $R_j$ in the next committee:
   (a) Set $sh_j := F(j)$
   (b) Generate ciphertext $ct_j \leftarrow \text{PKE_Enc(ek}_R, sh_j; r_j)$

13. Assemble ZK-Witness: $\text{wit}_{KG} := (sk_i, \{r_j\}_{j \in [n]}, \text{dk}_R, F)$

14. Assemble ZK-Statement:
    $$\text{stmt}_{KG} := (\{pk_i, \{ct^i_j\}_{i \in [t+1]}, \{ct_j, \text{ek}_R\}_{j \in [n]}, \text{ek}_R\})$$

15. Compute Proof:
    $$\pi_{KG} \leftarrow \text{NIZK.Prove}((\text{crs}_{2\delta}, \text{stmt}_{KG}, \text{wit}_{KG}, \mathcal{R}_{AD}^{\text{crs}_{2\delta}}))$$

16. Prepare message: $msg := (\text{KEYGEN2}, \{\{pk_i, \{ct_j\}_{j \in [n]}, \pi_{KG}\})$

17. Sign message: $\sigma \leftarrow \Sigma . \text{Sign}(sk_R, msg)$

18. Remove the entry for this role from $L_{\text{Roles}}$

19. Delete all state except $msg, \sigma$

20. Send $(\text{POST}, (R_i, msg, \sigma))$ to $\mathcal{F}_{\text{BCRA}}$

- Executing Role Handover:

1. Retrieve $\text{dk}_R, \text{sk}_R$ associated with this role $R_i$ from $L_{\text{Roles}}$

2. Initialize empty lists $L_{\text{CommitteePK}^{\text{next}}}$, $L_{\text{CommitteePK}^{\text{current}}}$, $L_{\text{CommitteeVK}^{\text{current}}}$, $L_{\text{Shares}}$ and $L_{\text{Requests}}$

3. For each role $R_j$ in the next committee
   (a) Find $(\text{roleassign}, t, (\text{GENERATE}, R_j, \text{ek}_R, \text{vk}_R))$ in ORDERED
   (b) Insert $\text{ek}_R$ into $L_{\text{CommitteePK}^{\text{next}}}$
4. For each role $R_i$ in the current committee:
   (a) Find $(\text{roleassign}, t_i, (\text{GENERATE}, R_j, ek_R, vk_R))$ in $\text{current}$
   (b) Insert $ek_R$ into $L_{\text{CommitteePK}}$
5. For each role $R_j$ in the previous two committees:
   (a) Find $(\text{roleassign}, t_i, (\text{GENERATE}, R_j, ek_R, vk_R))$ in $\text{current}$
   (b) Insert $vk_R$ into $L_{\text{CommitteeVK}}$
6. For each entry $(\text{pid}, t, (R_j, msg, \sigma))$ with $msg := (\text{KeyShare}, (vk_i, \{ct_j\}, \pi_{KS}))$
      for the previous committee in $\text{current}$
   (a) Retrieve $vk_R$ from $L_{\text{CommitteeVK}}$
   (b) Check $\Sigma.Vfy(vk_R, msg, \sigma) = 1$, otherwise ignore this message
   (c) Assemble ZK-Statement:
       $\text{stmt}_{KS} := (vk_i, \{ct_j, ek_R\}_{j\in[n]})$
   (d) Check NIZK.$\text{Verify}(\text{crs}_{sk}^{SS}, \text{stmt}_{KS}, \pi_{KS}, \mathcal{R}_{KS}^{AD}) = 1$, otherwise ignore this message
   (e) Add $\{ct_j\}_{j\in[n]}$ to $L_{\text{Qual}}$
7. Sort $L_{\text{Qual}}$ lexicographically
8. For each entry $(\text{pid}, t, (R_j, msg, \sigma))$ with $msg := (\text{KeyShare}, (vk_i, \{ct_k\}, \pi_{KS}))$
      for the current committee in $\text{current}$:
   (a) Retrieve $vk_R$ from $L_{\text{CommitteeVK}}$
   (b) Check if $\Sigma.Vfy(vk_R, msg, \sigma) = 1$, otherwise skip this entry.
   (c) Retrieve from $L_{\text{Qual}}$ from the first $t+1$ entries the ciphertext $ct_j$ for $R_j$ each, as $L_{\text{KCom}} := \{ct_1, \ldots, ct_{t+1}\}$
   (d) Assemble ZK-Statement:
       $\text{stmt}_{KS} := (pk_j, \{ct_k\}_{k\in[n]}, L_{\text{KCom}})$
   (e) Check if NIZK.$\text{Verify}(\text{crs}_{sk}^{SS}, \text{stmt}_{KS}, \pi_{KS}, \mathcal{R}_{KS}^{AD}) = 1$, otherwise skip this entry.
   (f) Add $ct_i$ (the one addressed to the currently executing role) to $L_{\text{Qual}}$
9. Sort $L_{\text{Qual}}$ lexicographically
10. Reconstruct and reshare secret key share:
    (a) Obtain the first $t+1$ entries $ct_1^t, \ldots, ct_{t+1}^t$ from $L_{\text{Qual}}$
    (b) Decrypt $ct_i^t$ as $sh_k \leftarrow \text{PKE}.\text{Dec}(dk_R, ct_i^t)$
    (c) Lagrange-interpolate $G(x)$ from $sh_1, \ldots, sh_{t+1}$ to obtain secret key share $sk_i := G(i)$ and set $vk_i = \text{TPKE}.5k2\text{PKE}(sk_i)$
    (d) Choose a random degree $t$ polynomial $F(x) = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_t * x^t$ with $F(0) = sk_i$
    (e) For each role $R_j$ in the next committee:
        i. Generate ciphertext $ct_j \leftarrow \text{PKE}.\text{Enc}(ek_R, F(j); r_j)$
    (f) Assemble ZK-Witness:
        $\text{wit}_{KS} := (dk_R, sk_i, \{r_j\}_{j\in[n]}, F, G)$
    (g) Assemble ZK-Statement:
        $\text{stmt}_{KS} := (ek_R, \{ct_j\}_{j\in[t+1]} \{ct_j, ek_R\}_{j\in[n]}, vk_i)$
C Auditably Sender-Traceable Encryption (ASTE)

In this section, we give the full description of the Auditably Sender-Traceable Encryption Transfer ideal functionality \( F_{\text{ASTE}} \) and the protocol \( H_{\text{ASTE}} \).

C.1 Overview

The Auditably Sender-Traceable Message Transfer (ASTE) functionality \( F_{\text{ASTE}} \) is a toy example with a two-fold purpose. Firstly, it demonstrates the applicability of \( F_{\text{AS}} \). Secondly, the techniques for using zero-knowledge together with PROM-based NINCE can be used in other settings. Together, this “toy example” provides a simple yet general explanation for how to build applications on top of \( F_{\text{AS}} \).

The \( F_{\text{ASTE}} \) functionality is similar to a PKE. It allows users (participants in the auditable surveillance system) to encrypt confidential messages to each other, by simply encrypting a message to the recipient. Note that \( F_{\text{ASTE}} \) does not
model a secure channel but only provides (CCA-secure) encryption. In particular, if the ciphertext are sent in an anonymous way, the recipient does not learn the identity of the sender. This allows to implement, for example, an anonymous confidential feedback channel. Clearly, this is also possible with a standard PKE scheme. However, unlike in a standard PKE scheme, \( F_{\text{ASTE}} \) guarantees that an identity of a system participant is attached to every ciphertext. On the one hand, the recipient only learns the message \( m \), and is unable to deanonymize the party which generated the ciphertext. On the other hand, law enforcement learns nothing about the message \( m \), but is able to deanonymize ciphertexts. The construction can be extended to leak also the message \( m \) to law enforcement, or, indeed, leak any a efficient function \( f \) of sender identity, recipient identity and message \( m \).

To prevent perpetual surveillance of a user, the system operates in “validity periods”. When the period changes, all users must run an update (i.e. refresh their key material) to continue to use the system. As a result, law enforcement must specify which user should be surveilled in which period. As in \( F_{\text{AS}} \), statistics about the surveillance are published and an audit trail is accessible for the auditor.

On a more technical level, \( F_{\text{ASTE}} \) is very similar to \( F_{\text{AS}} \). It does not offer store secret, but instead has update, which refreshes the user’s escrow secret key to the current validity period (if necessary). Moreover, instead of get secrets, law enforcement can use prepare access which, given a valid warrant \( W \) is a preparation step (intuitively, the escrow secrets are learnt). By running execute access on ciphertext \( c \) of a user \( U \) with uid \( uid \) which is surveilled for period \( vper \), law-enforcement learns whether \( c \) belongs to \((uid, vper)\), thus deanonymizing the subject of surveillance \( U \) during period \( vper \). Finally, users can use the encryption scheme via encrypt message and decrypt ciphertext.

The protocol \( \Pi_{\text{ASTE}} \) reflects the close relation of \( F_{\text{ASTE}} \) to \( F_{\text{AS}} \): Indeed, except for the tasks encrypt message, decrypt message and execute data, \( \Pi_{\text{ASTE}} \) can delegate most of the work to \( F_{\text{AS}} \), and only needs to do a little bookkeeping and consistency checks. For example, in update, a fresh escrow secret is obtained by running store secret. To implement encryption with trapdoor access, \( \Pi_{\text{ASTE}} \) relies on a special “encryption scheme” \( \text{PKE}_{\text{AS}} \) where:

- \text{PKE}\(_{\text{AS}}\).\text{Gen} generates a long-term public key \( pk \) which is also used as the user identity \( uid \).
- \text{PKE}\(_{\text{AS}}\).\text{Enc} relies on two strong primitives:
  - It uses non-interactive non-committing encryption (NINCE) in the programmable ROM (PROM). This allows to deal with the “adaptive corruption” of user secret keys (of some period) due to surveillance by law enforcement. As discussed in Section 5.3, it is likely that NINCE is required to instantiate \( F_{\text{ASTE}} \), unless one allows highly interactive and/or high communication during execute access. As it is well-known that NINCE is impossible in the standard model \([48]\), relying on the PROM is both natural and necessary.
- It uses the NIZK proofs provided by $\mathcal{F}_{\text{AS}}$. All ciphertexts carry a consistency proof which is checked by an honest recipient. In particular, if $\text{PKE}_{\text{AS}}.\text{Dec}_{\text{RE}}(\text{crs}, \text{sk}, c, \text{vper}) \neq \bot$, then the consistency proof holds. Moreover, if the consistency proof holds, then $\text{PKE}_{\text{AS}}.\text{Dec}_{\text{LE}}(\text{crs}, \text{usk}, c, \text{vper})$ returns $\text{uid}$ which correctly deanonymizes the user (with secret key $\text{sk}$). Importantly, the consistency proof also demonstrates possession of the (long-term) secret key $\text{sk}$ to $\text{pk}$, thus it is impossible to send messages in the name of another (honest) user.

- There are two decryption methods: The recipient uses $\text{PKE}_{\text{AS}}.\text{Dec}_{\text{RE}}$ to decrypt the message $m$. Law enforcement uses $\text{PKE}_{\text{AS}}.\text{Dec}_{\text{LE}}$ to deanonymize the user, i.e. it tests if the ciphertext belongs to an identity $\text{uid}$.

To model the validity period, $\mathcal{F}_{\text{ASTE}}$ (and consequently $\Pi_{\text{ASTE}}$) rely on the global clock functionality $\mathcal{G}_{\text{CLOCK}}$.

### C.2 Auditably Sender-Traceable Encryption

In this section, we formally define the ideal functionality for Auditably Sender-Traceable Encryption and discuss certain design choices. For the writing conventions used see Appendix B.1.

<table>
<thead>
<tr>
<th><strong>$\mathcal{F}_{\text{ASTE}}$</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Parameters:</strong></td>
</tr>
<tr>
<td>- $f_t$, $f_p$: Transparency and policy function as in $\mathcal{F}_{\text{AS}}$.</td>
</tr>
<tr>
<td>- $\text{poly}(\lambda)$: length of $\text{uid}$.</td>
</tr>
<tr>
<td>- $\text{leak}(m, \text{uid}, \text{vper})$: Leakage to adversary when a message is sent, usually $\text{leak}(m, \text{uid}, \text{vper}) = (</td>
</tr>
<tr>
<td>- $\text{System pids: } \text{pid}<em>{\text{SO}}, \text{pid}</em>{\text{J}}, \text{pid}<em>{\text{AU}}, \text{pid}</em>{\text{LE}}$</td>
</tr>
<tr>
<td><strong>Functionality State:</strong></td>
</tr>
<tr>
<td>- $\mathcal{L}<em>I$: Identical to $\mathcal{F}</em>{\text{AS}}$. (List of initialized parties (initially empty). Contains $\text{pid}$ entries.)</td>
</tr>
<tr>
<td>- $\mathcal{L}<em>U$: Identical to $\mathcal{F}</em>{\text{AS}}$. (List of registered users (initially empty). Contains $\text{pid}$, $\text{uid}$ pairs.)</td>
</tr>
<tr>
<td>- $\mathcal{L}_{\text{vper}}$: List of tuples $(\text{uid}, \text{vper})$ to keep track of which users have updated in which periods. (Initially empty.)</td>
</tr>
<tr>
<td>- $\mathcal{L}<em>{\text{msgs}}$: List of all messages sent by honest users of $\mathcal{F}</em>{\text{ASTE}}$. Contains tuples of the form $(c, m, \text{uid}, \text{uvper})$.</td>
</tr>
<tr>
<td>- $\mathcal{L}<em>W$: Identical to $\mathcal{F}</em>{\text{AS}}$.</td>
</tr>
<tr>
<td>- $\mathcal{L}_{W-\text{cached}}$: List of $(\text{uid}_i, \text{vper}_i)$ tuples tracking access rights of LE. (Initially empty.)</td>
</tr>
<tr>
<td>- $\text{ureg}_i \in {0, 1}$ is 1 if user $U_i$ received output of registration. Else 0.</td>
</tr>
<tr>
<td>- $\text{uvper}_i$: (Most recent) validity period for which user $U_i$ has registered (all initialized to $\bot$).</td>
</tr>
</tbody>
</table>
Init:
– This task is identical to \textbf{Init} in $F_{AS}$, except that $(\text{Init}, P)$ is leaked to the adversary when party $P$ inputs $(\text{Init}, P)$:
– $F_{ASTE}$ sends $(\text{Register}, sid)$ to $G_{CLOCK}$.

\textbf{Party Init}:
– This task is identical to \textbf{Party Init} in $F_{AS}$.

\textbf{User Registration}:
– Input $U_i$: $(\text{Register})$
– Input $SO$: $(\text{Register})$
– Note: $U_i$ will abort any other task (except \textbf{Party Init}) unless $ureg_i = 1$.
1. As soon as $U_i$ gave input, send $(\text{Register}, U_i)$ to the adversary. Wait for $(\text{Register}, U_i, uid_i)$ from the adversary.
2. If $(\_, uid_i) \in L_U$ abort.
3. Store $(U_i, uid_i)$ in $L_U$.
4. SO: Generate delayed output to $(\text{Registered}, uid_i)$.
5. $U$: Generate delayed output to $(\text{Registered}, uid_i)$, and when delivered, set $ureg_i = 1$.

\textbf{Next Period}:
– Input $SO$: $(\text{NextPeriod})$
– Behavior: $F_{ASTE}$ sends $(\text{Clock-Update}, sid_C)$ to $G_{CLOCK}$.

\textbf{Update}:
– Input $U_i$: $(\text{Update})$
– Input $SO$: $\emptyset$
– Behavior:
  • $F_{ASTE}$ queries $(\text{Clock-Read}, sid_C)$ to $G_{CLOCK}$ and receives response $(\text{Clock-Read}, sid_C, vper)$.
  • If $uvper_i = vper$, immediately output $(\text{UpdateDone}, vper)$ to $U_i$. Else continue.
  • Send $(\text{Update}, U_i, vper)$ to the adversary and wait for $(\text{Ok})$ from the adversary.
  • $F_{ASTE}$ queries $(\text{Clock-Read}, sid_C)$ to $G_{CLOCK}$ and receives response $(\text{Clock-Read}, sid_C, vper')$.
  • If $vper \neq vper'$ abort this task.
  • Generate delayed output $(\text{UpdateDone}, vper)$ to SO.
  • Generate delayed output $(\text{UpdateDone}, vper)$ to $U_i$, and, when delivered, set $uvper_i := vper$ and add $(U_i, uvper_i)$ to $L_{uvper}$.

\textbf{Encrypt message}:
– Input $U_i$: $(\text{Encrypt}, U_j, m)$
– Behavior:
  1. Copy $uvper_i$ and use this copy in the rest of this task. (Protect against concurrent calls to \textbf{update}.)
  2. If $uvper_i = \bot$, abort.
3. If $U_j$ is corrupted, let $m' = m$, else let $m' = \perp$.
4. If $(uid_i, uvper_i) \in L_W \text{-cached}$, let $uid' = uid$, else let $uid' = \perp$.
5. Send $(\text{ENCRYPT}, U_j, \text{leak}(m, uid_i, uvper_i), m', uid')$ to the adversary.
6. Once $(\text{Ciphertext}, U_j, c)$ is received from the adversary, add $(c, m, uid_i, uvper_i)$ to $L_{msgs}$.
7. Generate immediate output $(\text{Ciphertext}, c)$ to $U_i$.

**Decrypt Ciphertext:**
- Input $U_j$: $(\text{Decrypt}, c)$
- Behavior:
  1. If $(c, \cdot, \cdot, \cdot) \notin L_{msgs}$ then
     - Send urgent request $(\text{SIMJECTMSG}, c)$ to the adversary and wait for immediate response $(\text{INJECTMSG}, c, m, vper)$.
     - Run inject message for input $(\text{INJECTMSG}, c, m, vper)$.
  2. Find $(c, m, uid_i, uvper_i)$ in $L_{msgs}$ and let $m = \perp$ if no entry was found.
  3. $\mathcal{FASTE}$ sends $(\text{CLOCK-READ}, sid_C)$ to $\mathcal{GCLOCK}$ and receives response $(\text{CLOCK-READ}, sid_C, vper)$.
  4. If $uvper \neq vper$ let $m = \perp$.
  5. Generate immediate output $(\text{Plaintext}, m)$ to $U_j$.

**Inject Message:**
- Note: This is an auxiliary “task”. It is only available as a subroutine within other tasks.
- Input: $(\text{INJECTMSG}, c, m, uid, vper)$
  1. If $(m, uid, vper) = (\perp, \perp, \perp)$, skip the rest.
  2. If $\exists U: (U, uid) \in L_u$ and $U$ is not corrupted, abort.
  3. If SO is honest and $(uid, vper) \notin L_{uvper}$, abort.
  4. Store $(c, m, uid, vper)$ in $L_{msgs}$.

**Request Warrant:**
- This task is identical to Request Warrant in $\mathcal{F}_{AS}$.

**Prepare Access:**
- Input LE: $(\text{ACCESSPREP}, W)$
- Behavior:
  1. Send $(\text{ACCESSLEAK})$ to the adversary.
  2. If no entry $(W, 1)$ exists in $L_W$, abort. (Warrant not granted)
  3. For all $(uid', vper') \in W$, if $(uid', vper') \in L_{uvper}$ store $(uid', vper') \in L_{W \text{-cached}}$.
  4. Generate delayed output $(\text{ACCESSPREPDONE})$ for LE.

**Execute Access:**
- Input LE: $(\text{ACCESSEXEC}, c, (uid, vper))$
- Behavior:
  1. If $(uid, vper) \notin L_{W \text{-cached}}$, return $\perp$. 

75
2. If \((c, v, \cdot, \cdot) \notin L_{msgs}\) then:
   (a) Send urgent request \((\text{SimAccessInject}, c)\) to the adversary and 
   wait for immediate response \((\text{InjectMsg}, c, m, uid, vper)\).
   (b) Run \text{inject message} for input \((\text{InjectMsg}, c, m, uid, vper)\).

3. If there exist \(m', uid', vper'\) such that \((c, m', uid', vper') \in L_{msgs}\) and 
   \(uid = uid'\), then set \(uid' = uid\), else \(uid' = \perp\).

4. Output \((\text{AccessExecDone}, uid')\) immediately.

Get Statistics:
– This task is identical to \text{Get Statistics} in \(\mathcal{F}_{AS}\).

Audit:
– This task is identical to \text{Audit} in \(\mathcal{F}_{AS}\).

Remark 4 (Handling of leakage). We handle leakages in \text{encrypt message} and 
\text{prepare access} in two different ways: In \text{encrypt message}, we added an 
explicit leak of \(m\) resp. \(uid\) to the adversary, if the receiver \(U_j\) is corrupted 
resp. the sender \(U_i\) in the period \(uvper_i\) under surveillance. This simplifies the 
simulator description, but is not strictly necessary:

– To learn the message \(m\), the simulator could call \text{decrypt ciphertext} on 
behalf of \(U_j\).
– To learn the \(uid\) if the sender is under surveillance, the simulator may trial- 
decrypt the ciphertexts \(c\) w.r.t. all tuples \((uid_k, uvper_k) \in L_{W-cached}\) for honest 
users \(U_k\).

Indeed, in \text{prepare access}, we do not add an explicit leak of all affected cipher- 
texts, and let simulator use the trial-decryption strategy.

Remark 5 (Law-enforcement access). The interface and behavior of law enforce- 
ment’s data access requires some clarifications.

1. The access task is split into \text{prepare access} and \text{execute access}. This 
corresponds to the idea of first publishing (a redacted version of) the warrant 
and obtaining some escrow secrets (usually interactively), and then using 
these escrow secrets to identify (and decrypt) affected ciphertexts. Indeed, 
\text{execute access} is an offline procedure in our modelling, since the adversary 
cannot delay its output in any way.\(^{24}\)

2. In real-world implementations, in particular in \(\Pi_{ASTE}\), honestly generated 
ciphertexts of corrupted parties which are never received by honest parties 
can still be decrypted be LE. To model this, \(\mathcal{F}_{ASTE}\) leaks a non-honest 
ciphertext (in \((\text{SimAccessInject}, c')\)) in \text{execute access}, and then runs 
\text{inject message} to inject a suitable tuple into \(L_{msgs}\), if necessary. Observe

\(^{24}\)Relaxing the model to online access is possible, but depending on its exact nature, 
can incur additional leakage (e.g. information about the used warrant, ciphertext 
size, etc.) and allows to delay the output. On the other hand, it may circumvent the 
necessity of strong (setup) assumptions, like the PROM, see Section 5.3.
that even messages encrypted to public keys which are not registered in
the system exists may still decrypt for LE. Thus, \textit{inject message} does
not require a recipient. This modelling artefact leaks some information
of the supposedly offline decryption queries to the ideal adversary, but any
c \in L_{\text{msgs}} (in particular, ciphertexts en- or decrypted by honest users) is
accessed “silently”.

3. Intuitively, a warrant \( W = \{(uid_i, vper_i, meta_i)\}, \) empowers law en-
forcement to extract information from \textit{any valid} ciphertexts corresponding to
some \((uid_i, vper_i)\) in \( W \). Validity of ciphertext makes no sense in the ideal
functionality, but, intuitively, any ciphertext which a \textit{decrypt ciphertext}
or \textit{inject message} “accepts”, i.e. which ends up in \( L_{\text{msgs}} \) is valid.

4. The metadata in a warrant does not affect the possibility of law-enforcement
access, and is meant only for the judge, the auditor and public statistics.

Remark 6 (System integrity). The honesty of the system operator SO plays a
central role in the system’s integrity. A corrupted SO can circumvent any re-
striction w.r.t. registration, and consequently thwart deanonymization. This is
inherited from \( F_{\text{AS}} \), because the NIZK provided by \( F_{\text{AS}} \) does not ensure that
user secrets have been deposited in case of malicious parties (cooperating with
SO). Note however, that in the real world, SO would violate the law or con-
tractual obligation by such actions, and is therefore strongly incentivized to not
misbehave.

C.3 The Protocol \( \Pi_{\text{ASTE}} \)

In this section, we define the protocol for realizing \( F_{\text{ASTE}} \). The high-level idea
is the following:

Setup As setup, a CRS (via \( F_{\text{CRS}} \)) for a commitment key, a modified bulletin
board (\( F_{\text{BB}} \)) to find public keys of parties and allow basic synchronization,
and the auditable surveillance functionality \( F_{\text{AS}} \) are used. Moreover, the
global clock \( G_{\text{CLOCK}} \) also models the advancing validity periods. (The clock
is “controlled” solely by SO in our protocol, while all parties read the time.)

\textbf{Passthrough tasks} Except perhaps for some smaller adaptation, \textit{Init, Party}
\textit{Init, RequestWarrant, Get Statistics} and \textit{Audit} basically pass their
inputs through to the \( F_{\text{AS}} \) hybrid functionality.

\textbf{Next Period} SO simply advances the clock. (SO registered with \( G_{\text{CLOCK}} \) dur-
ing \textit{Init}.)

\textbf{User Registration} In this task, the user generates a keypair through \((pk, sk) \leftarrow
PKE_{\text{AS}}, \text{Gen}(1^\lambda)\). The public key \( pk \) is also the \( uid \). The user tries to register
its \( uid \) with \( F_{\text{BB}} \). Note that after registration, the user retrieves its own \( uid \),

\footnote{It can be enforced that only ciphertexts to public keys which are in the system are
valid (and therefore decryptable). For example, by only accepting publics key which
are signed by SO, and adapting the protocol accordingly. Since corrupt user can use
secure communication anyway, there is no clear benefit to this, and therefore \( F_{\text{ASTE}}
\) (and \( \Pi_{\text{ASTE}} \)) does not enforce it.}

77
to check whether the registration succeeded.\textsuperscript{26} (\mathcal{F}_{BB} does not respond (with success or failure) to registration queries, so this is the only way to be sure.)

**Update** The user \( U_i \) first retrieves the current period \( v_{\text{per}} \), and then sends an update request to \( SO \). \( SO \) retrieves the current period itself (so that neither party can cheat the other w.r.t. the period), and they both engage in \textit{store secret} from \( \mathcal{F}_{AS} \). The resulting secret is the user’s escrow secret \( SK_i \) and remembers the period in \( uv_{\text{per}} := v_{\text{per}} \).

**Encrypt Message** This is merely an application of \( \mathcal{PKE}_{AS} \).\textsuperscript{Enc.}

**Decrypt Ciphertext** The receiver first retrieves the current time \( v_{\text{per}} \), and then decrypts the ciphertext via \( \mathcal{PKE}_{AS} \).\textsuperscript{Dec} \((\text{crs}, sk_j, c, v_{\text{per}})\). Using the current time ensures the surveillability of any received messages in the respective validity periods.

**Prepare Access** This is effectively a pass-through to \textit{get secrets} in \( \mathcal{F}_{AS} \). After receiving \( (\text{GotSecrets, ...}) \), \( LE \) stores these escrow secrets (in \( L_{\text{W-cached}} \)) for later use.

**Execute Access** \( LE \) uses a stored escrow secret \( (\text{uid}, v_{\text{per}}, SK_i) \) from \( L_{\text{W-cached}} \) to check via \( \mathcal{PKE}_{AS} \).\textsuperscript{Dec} \((SK_i, c, v_{\text{per}})\) whether a ciphertext \( c \) decrypts to \( \text{uid} \) when trying \( (SK_i, v_{\text{per}}) \). If so, the sender \( \text{uid} \) was deanonymized, otherwise it returns \( \perp \).

The encryption subroutine \( \mathcal{PKE}_{AS} \) is described in Section 5.2. It is formally described in Appendix C.4

\textbf{Table II\textsubscript{ASTE}}

<table>
<thead>
<tr>
<th>The protocol II\textsubscript{ASTE} uses hybrid functionalities ( \mathcal{F}<em>{CRS}, \mathcal{F}</em>{BB}, \mathcal{F}<em>{AS}, \mathcal{F}</em>{RO}, ) and global functionality ( \mathcal{G}_{CLOCK} ).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Parameters</strong></td>
</tr>
<tr>
<td>- ( f_t, f_p ): Transparency and policy function as in ( \mathcal{F}_{AS} ).</td>
</tr>
<tr>
<td>- System pids: ( \text{pid}<em>{SO}, \text{pid}<em>1, \text{pid}</em>{AU}, \text{pid}</em>{LE} )</td>
</tr>
<tr>
<td><strong>Hybrid functionalities</strong></td>
</tr>
<tr>
<td>- ( \mathcal{F}<em>{CRS} ): The distribution of ( \mathcal{F}</em>{CRS} ) is ( \text{crs}<em>{\text{ASTE}} = \text{crs}</em>{\text{com}} ), where ( \text{crs}_{\text{com}} \leftarrow \text{COM.Setup}(1^\lambda) ).</td>
</tr>
<tr>
<td>- ( \mathcal{F}<em>{AS} ): The secret key space ( S ) of ( \mathcal{F}</em>{AS} ) is defined as the user secret key space ( \text{SKE}<em>{\text{NCE}} ), where ( \text{SKE}</em>{\text{NCE}} ) is the scheme used in ( \mathcal{PKE}<em>{AS} ). The relation ( \mathcal{R} ) is set to ( \mathcal{R}</em>{\text{PKEAS}} ) or some NP-complete relation. Other system parameters of ( \mathcal{F}_{AS} ) are set as in II\textsubscript{AS} above.</td>
</tr>
<tr>
<td>- ( \mathcal{F}<em>{BB}, \mathcal{F}</em>{RO}, \mathcal{G}_{CLOCK} ) have no parameterization.</td>
</tr>
<tr>
<td><strong>States of the Parties:</strong></td>
</tr>
<tr>
<td>- User, SO and LE: Cache ( crs ) from ( \mathcal{F}_{CRS} ).</td>
</tr>
</tbody>
</table>

\textsuperscript{26} For simplicity, the user will never generate another keypair, and thus, be unable to register if the adversary decides to be “rushing” and register some user with the same \( \text{uid} \) before. This attack can be repeated even if the user generates a fresh keypair. Also, the user which “stole” the identity (and hence public key) cannot create valid ciphertexts, because this requires knowledge of the corresponding secret key.
User $U_i$:
- User long-term key: $(pk_i, sk_i)$, where $uid_i = pk_i$ is the user identify.
- User secret (with period): $(usk_i, vper_i)$, where $usk_i$ is the current user secret, and $vper_i$ the current user validity period.
- $ureg_i \in \{0, 1\}$ indicating whether $U_i$ completed registration (initially 0).
- Public-key caches: $uid_j = pk_j$ (once retrieved from $F_{BB}$).

LE: Cache $LW_{cached}$ with entries $(uid_i, vper_i, usk_i)$.

SO, J and AU only need to interact with $F_{AS}$.

Init:
- Input SO: $\langle$Init, SO$\rangle$
- Input J: $\langle$Init, J$\rangle$
- Input AU: $\langle$Init, AU$\rangle$
- Note: For all parties:
  - This task can only be invoked once.
  - Ignore all other messages until this task has finished.
  - Behavior
    1. SO: Send $\langle$Register$\rangle$ to $G_{CLOCK}$.
    2. SO: Send $\langle$value$\rangle$ to $F_{CRS}$ and cache the result as $crs$.
    3. SO: Send $\langle$Init, SO$\rangle$ to $F_{AS}$.
    4. $P \in \{J, AU\}$: Send $\langle$Init, $P$$\rangle$ to $F_{AS}$.
    5. SO, J, AU: Upon receiving $\langle$InitFinished$\rangle$, output $\langle$InitFinished$\rangle$.

Party Init:
- This is merely a proxy to Party Init from $F_{AS}$.

User Registration:
- Input $U_i$: $\langle$Register$\rangle$
- Input SO: $\langle$Register$\rangle$
- Note: Until $ureg_i = 1$, $U_i$ aborts all other tasks (except Party Init) and ignores their messages.
- Behavior
  1. $U_i$: Send $\langle$value$\rangle$ to $F_{CRS}$ and cache the result as $crs$.
  2. $U_i$: If $pk_i \neq \bot$, let $uid_i = pk_i$ and skip to step 4.
  3. Compute and store $(pk_i, sk_i) \leftarrow PKE_{AS}.Gen(1^\lambda)$.
  4. $U_i$: Let $uid_i = pk_i$.
  5. $U_i$: Send $\langle$Register, $uid_i, uid_i$$\rangle$ to $F_{BB}$.
  6. $U_i$: Send $\langle$Retrieve, $uid$$\rangle$ to $F_{BB}$. If the response is not $\langle$Retrieve, $U_i$, $uid$, $uid$$\rangle$, abort. ($uid$ was taken.)
  7. $U_i$: Send $\langle$Ready$\rangle$ to SO.
  8. $U_i$: Send $\langle$Register, $uid_i$$\rangle$ to $F_{AS}$.
  9. SO: Upon receiving $\langle$Ready$\rangle$ from $U_i$, send $\langle$Retrieve, $U_i$$\rangle$ to $F_{BB}$ and receive $\langle$Retrieve, $U_i$, $uid_i$, $uid'_i$$\rangle$. If $uid_i \neq uid'_i$, abort.
  10. SO: Call $F_{AS}$ with input $\langle$Register, $uid_i$$\rangle$. 

79
11. SO: Receive and propagate output (Registered, uid) (or propagate an abort) from $\mathcal{F}_{AS}$.

12. $U_i$: Receive output (Registered) (or propagate an abort) from $\mathcal{F}_{AS}$.

13. $U_i$: Set $\text{ureg}_i := 1$ and output (Registered).

**Next Period:**
- Input SO: (NextPeriod)
- Behavior: SO sends (Clock-Update) to $\mathcal{G}_{\text{CLOCK}}$.

**Update:**
- Input $U_i$: (Update)
- Input SO: $\emptyset$
- Behavior:
  1. $U_i$: Query (Clock-Read) to $\mathcal{G}_{\text{CLOCK}}$ and receive (Clock-Read, vper).
  2. $U_i$: If $uvper_i = vper$, $U_i$ output (Updatedone, vper). Else continue.
  3. $U_i$: Send (Update, uid$_i$, vper) to SO.
  4. SO: Upon receiving (Update, uid, vper) from $U_i$:
  5. SO: Query (Clock-Read) to $\mathcal{G}_{\text{CLOCK}}$ and receive (Clock-Read, vper').
     If $vper' \neq vper$, abort.
  6. SO: Send (StoreSecret, vper) to $\mathcal{F}_{AS}$.
  7. $U_i$: Upon receiving (SecretStored, usk'$_i$), update $(usk_i, uvper_i) := (usk'_i, vper)$. 
  8. $U_i$: Generate output (Updatedone, vper).
  9. SO: Upon receiving (SecretStored, uid$_i$), generate output (Updatedone, vper).

**Encrypt Message:**
- Input $U_i$: (Encrypt, $U_j$, m)
- Behavior:
  1. Locally copy $(usk_i, uvper_i)$ and use this copy.
  2. If $uvper_i = \perp$, abort.
  3. If $pk_j$ is not cached, send (Retrieve, $U_j$) to $\mathcal{F}_{\text{BB}}$ and later receive (Retrieve, $U_j$, uid$_j$, uid'$_j$). If $uid_j \neq uid'_j$ or $uid = \perp$, abort. Else cache $pk_j = uid_j$.
  4. $c \leftarrow \text{PKE}_{\mathcal{A}}.\text{Enc}(\text{crs}, pk_j, m, (sk_i, usk_i, uid_i, uvper_i))$
  5. Output (Ciphertext, c).

**Decrypt Ciphertext:**
- Input $U_j$: (Decrypt, c)
- Behavior:
  1. Send (Clock-Read) to $\mathcal{G}_{\text{CLOCK}}$ and receive (Clock-Read, vper).
  2. Compute $m = \text{PKE}_{\mathcal{A}}.\text{Dec}_{\text{RE}}(\text{crs}, sk_j, c, vper)$ and output (Plaintext, m).

**Request Warrant:**
- This is merely a proxy to Request Warrant from $\mathcal{F}_{AS}$.

**Prepare Access:**
- Input LE: (AccessPrep, W)

80
Behavior:
1. Parse $W = (W_1, \ldots, W_v)$ and $W_i = (uid_i, vper_i, meta_i)$.
2. Send ($\text{GetSecrets}, W$) to $F_{\text{AS}}$ and receive response ($\text{GotSecrets}$, $(usk_1, \ldots, usk_v)$) (or an abort happens). For $i = 1, \ldots, v$, if $usk_i \neq \perp$ add $(uid_i, vper_i, usk_i)$ to $L_{W-\text{cached}}$.
3. Output ($\text{AccessPrepDone}$)

Execute Access:
– Input LE: ($\text{AccessExec}, c, (uid, vper)$)
– Behavior:
  1. Find $usk$ with $(uid, vper, usk)$ in $L_{W-\text{cached}}$. If none found, abort.
  2. Let $uid' = \text{PKE}_{\text{AS}}.\text{Dec}_E(\text{crs}, usk_i, c)$. (Note that $uid'$ may be $\perp$.)
  3. Output ($\text{AccessExecDone}, uid'$).

Get Statistics:
– This is merely a proxy to $\text{Get Statistics}$ from $F_{\text{AS}}$.

Audit:
– This is merely a proxy to $\text{Audit}$ from $F_{\text{AS}}$.

In Appendix G we show the following theorem.

**Theorem 3.** Suppose $\Sigma'$ is a EUF-CMA-secure signature scheme, COM is a statistically binding and computationally hiding commitment scheme, $\text{PKE}_{\text{NICE}}$ is a strong NINCE PKE scheme and IND-CCA secure and key-committing, $\text{SKE}_{\text{NICE}}$ is a strong NINCE SKE scheme and IND-CCA secure and key-committing. Moreover, suppose all schemes are perfectly correct. Let $\text{leak}(m, uid, vper) = (|m|, vper)$ as the system parameter in $F_{\text{ASTE}}$, and let $f_p, f_t$ be policy and transparency functions. Then $\Pi_{\text{ASTE}}$ UC-realizes $F_{\text{ASTE}}$ in the $\{F_{\text{CRS}}, F_{\text{BB}}, F_{\text{AS}}, G_{\text{CLOCK}}\}$-hybrid model.

C.4 Definition of Algorithms for $\text{PKE}_{\text{AS}}$

In this section, we define the subroutines which implement the encryption scheme used in $\Pi_{\text{ASTE}}$, called $\text{PKE}_{\text{AS}}$. To simplify presentation, we use a notation where $\text{PKE}_{\text{AS}}$ is given (oracle) access to both a random oracle (RO) and an “anonymously identity-bound” non-interactive zero-knowledge (NIZK) proofs system. Both will be realized by hybrid functionalities in $\Pi_{\text{ASTE}}$. Namely, as $\Pi_{\text{ASTE}}$ is in the $\{F_{\text{CRS}}, F_{\text{BB}}, F_{\text{AS}}, G_{\text{CLOCK}}\}$-hybrid model, $F_{\text{RO}}$ will be used to implement the random oracle(s), and $F_{\text{AS}}$ provides the “anonymously identity-bound” NIZK prove and verify interface. For simplicity, we simply write $\text{RO}(m)$ for the protocol: $\text{RO}(x)$ to a random oracle query (instead of “send ($\text{Hash}, x$) to $F_{\text{RO}}$, receive ($\text{HashConfirm}, m, x'$)”), and likewise, we write $\text{NIZK}_{\text{AS}}$ for the “anonymously identity-bound” NIZK, i.e. run $\pi \leftarrow \text{NIZK}_{\text{AS}}.\text{Prove}((usk, uid, vper), stmt, wit)$ for a prove request, and $\text{NIZK}_{\text{AS}}.\text{Verify}(vper, stmt, \pi)$ for a verification request. See Remark 7 for the full specification of the shorthand notation.

**Remark 7 (Shorthand notation in $\text{PKE}_{\text{AS}}$).** We write $\text{RO}(m)$ for the protocol:
1. Send \((\text{Hash}, m)\) to \(F_{RO}\). 
2. Immediately receive \((\text{HashConfirm}, m, h)\) and output \(h\).

For \(\text{NIZK}_{\text{AS}}\), statement resp. witness have the form \(\text{stmt}_R = (\text{vper, stmt})\) resp. \(\text{wit}_R = (\text{uid, usk, wit})\). We add an explicit witness relation \(R\) to statement and verify as a short-hand to indicate that, if necessary, an (NP-)reduction is used for witness and statement to prove relation \(R\) (if it does not coincide with the relation used in \(F_{\text{AS}}\)). We write \(\text{NIZK}_{\text{AS}}.\text{Prove}((\text{usk, uid, vper}), \text{stmt}_R, \text{wit}_R, R)\) for the protocol:

1. Let \(\text{stmt}_R = (\text{vper, stmt})\) and \(\text{wit}_R = (\text{uid, usk, wit})\). (If necessary, apply (NP-)reduction.)
2. Send \(\text{Prove}((\text{usk, uid, vper}), \text{stmt}_R, \text{wit}_R, R)\) to \(F_{\text{AS}}\).
3. Immediately receive \((\text{Proof}, \text{stmt}_R, \text{wit}_R, \pi)\) and output \(\pi\).

We write \(\text{NIZK}_{\text{AS}}.\text{Verify}((\text{stmt}_R, \pi, R)\) for the protocol:

1. Send \((\text{VERIFY, stmt}_R)\) to \(F_{\text{AS}}\). (If necessary, apply (NP-)reduction.)
2. Immediately receive \((\text{Verification}, \text{stmt}_R, \pi, b)\) and output \(b\).

We also write \(\text{NIZK}_{\text{AS}}.\text{Sim}(\text{stmt}_R, R)\) for the proof simulation subroutine in the \(F_{\text{AS}}\)-hybrid model. This means:

1. The simulator sends \((\text{Prove}, \text{stmt}_R)\) to the (dummy) adversary (i.e. the environment).
2. The simulator immediately receives \((\text{Proof}, \pi)\) and the subroutine outputs \(\pi\).

Note that the (dummy) adversary (i.e. the environment) must immediately answer the proof-string request \((\text{Prove}, \text{stmt}_R)\), this request contains no information on \(\text{wit}_R\), and the immediate response \((\text{Proof}, \pi)\) must be an accepting proof. Thus, exactly corresponds to simulation of proofs. Moreover, from the adversary’s perspective, the ideal NIZK in the (hybrid) \(F_{\text{AS}}\)-functionality always verifies the relation w.r.t. a witness. Thus, the resulting (idealized) NIZK has the properties of a straight-line simulation-extractable NIZK. We write \(\text{NIZK}_{\text{AS}}.\text{Ext}(\text{stmt}_R, \pi)\) for the witness to \((\text{stmt}_R, \pi)\) (assuming \(\text{NIZK}_{\text{AS}}.\text{Verify}(\text{stmt}_R, \pi) = 1\)).

**Construction 1.** The construction of \(\text{PKE}_{\text{AS}}\) is given as pseudocode for the algorithms \(\text{PKE}_{\text{AS}}.\text{Gen}, \text{PKE}_{\text{AS}}.\text{Enc}, \text{PKE}_{\text{AS}}.\text{Dec}_{\text{RE}}, \text{PKE}_{\text{AS}}.\text{Dec}_{\text{LE}}, \text{PKE}_{\text{AS}}.\text{Sim}, \text{PKE}_{\text{AS}}.\text{Expln}_{\text{RE}}, \text{PKE}_{\text{AS}}.\text{Expln}_{\text{LE}}\). We let \(t(\lambda) = 2\lambda\). The relation \(R_{\text{PKE}_{\text{AS}}}\) for the NIZK proofs is stated once in \(\text{PKE}_{\text{AS}}.\text{Enc}\). The message space is \(\{0,1\}^\text{poly}(\lambda)\). The space \(\text{SKE}_{\text{NCE}}.K\) is \(\{0,1\}^\text{poly}(\lambda)\) for some \(\text{poly}\). The space \(\text{UTD}\) is \(\{0,1\}^\text{poly}(\lambda)\) for some \(\text{poly}\); in our case, \(\text{UTD}\) is the same as public keys to \(\text{PKE}_{\text{NCE}}\) (i.e. public keys to \(\text{PKE}_{\text{AS}}\)).

\(^{27}\) In principle, the message space may be \(\{0,1\}^*\), however, then all building blocks (in particular additive secret sharing, commitments, and statement and witness size in the zero-knowledge proof) would have to deal with varying message length. For simplicity, we therefore fix the length.
PKE_{AS}, which is w.l.o.g. \( \{0,1\}^{\text{poly}(\lambda)} \) for some poly. We assume that these spaces are groups with group operation denoted “+” (so that we can use additive secret sharing); for example, by viewing \( \{0,1\}^n \) as \( \mathbb{Z}_n \) or using XOR.

Remark 8. While it is possible to optimize the construction of PKE_{AS} in many ways, we are not aware of any technique to avoid the rather costly cut-and-choose steps. Therefore, instead of optimizing for size or efficiency, we have optimized PKE_{AS} for clarity. The ciphertext size can, presumably, be reduced by a (small) constant factor without compromising security. Also for simplicity, PKE_{AS} includes the recipients public key in a ciphertext, as this allows to define and use PKE_{AS}.Vfy independent of the receiver’s key.

Remark 9. While we do not make this explicit in the description of PKE_{AS}, we assume that PKE_{AS} and its subroutines SKE_{NCE}, PKE_{NCE} use independent random oracles RO_{PKE_{AS}}, RO_{SKE_{NCE}} and RO_{PKE_{NCE}}. This is achieved by the usual domain separation techniques. In the specification of PKE_{AS}, we will write RO for RO_{PKE_{AS}} and assume RO_{PKE_{AS}}: \( \{0,1\}^* \rightarrow \{0,1\}^{\ell(\lambda)} \) to keep visual noise low.

Remark 10. Key generation for PKE_{AS}, i.e. PKE_{AS}.Gen(1^\lambda), is independent of the (global) commitment key \( \text{ck} \) and the ROM. This complies with our requirements for encryption schemes (and allows proving statements about PKE_{AS}.Gen with zero-knowledge proofs for NP).

\[
PKE_{AS}.Gen(1^\lambda) \\
1: / / For simplicity, sk equals the random coins. \\
2: (pk, sk) \leftarrow PKE_{NCE}.Gen(1^\lambda; sk)
\]
PKEAS.Enc$(ck, pk_R, m, (sk_S, usk, uid, vper))$

1: \text{// Prepare cut-and-choose encryption for LE } 
2: \text{for } i = 1, \ldots, \ell(\lambda) \text{ } 
3: \quad m_{i,0} + m_{i,1} = m \text{ } \quad \text{// Additive secret shares } 
4: \quad \text{usk}_{i,0} + \text{usk}_{i,1} = \text{usk} 
5: \quad \text{uid}_{i,0} + \text{uid}_{i,1} = \text{uid} 
6: \quad \text{for } b \in \{0, 1\} 
7: \quad c_{i,b}^R = \text{PKE}_{\text{NSCE}}.\text{Enc}(pk_R, m_{i,b}; r_{i,b}^R) 
8: \quad c_{i,b}^L = \text{SK}_{\text{NSCE}}.\text{Enc}(\text{usk}_{i,b}, \text{uid}_{i,b}; r_{i,b}^L) 
9: \quad w_{i,b} = (m_{i,b}, \text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^R, r_{i,b}^L) 
10: \quad \text{com}_{i,b} = \text{COM}.\text{Com}(ck, w_{i,b}, r_{i,b}^\text{com}) 
11: \quad d_{i,b} = (w_{i,b}, r_{i,b}^\text{com}) 
12: \quad \text{// Consistency proof for commitments and ctxts. } 
13: \quad \text{stmt} = (\text{vper}, pk_R, \text{com}_{i,b}, c_{i,b}^R, c_{i,b}^L, 0) 
14: \quad \text{wit} = (\text{usk}, \text{uid}, (d_{i,b}, i, b)) 
15: \quad \pi_{\text{con}} = \text{NIZK}_{\text{AS}}.\text{Prove}(\text{usk}, \text{uid}, \text{vper}, \text{stmt}, \text{wit}, R_{\text{PKEAS}}) \text{ for relation } R_{\text{PKEAS}} = \{(\text{stmt}, \text{wit}) \mid 
16: \quad (\text{uid}, sk_S) = \text{PKE}_{\text{NSCE}}.\text{Gen}(1^\lambda; sk_S) 
17: \quad \text{com}_{i,b} = \text{COM}.\text{Com}(ck, m_{i,b}, \text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^R, r_{i,b}^L, r_{i,b}^\text{com}) \text{ for all } i, b 
18: \quad \forall i: m = m_{i,0} + m_{i,1} 
19: \quad \forall i: \text{usk} = \text{usk}_{i,0} + \text{usk}_{i,1} 
20: \quad \forall i: \text{uid} = \text{uid}_{i,0} + \text{uid}_{i,1} 
21: \quad \} 
22: \quad \gamma = \text{RO}(\text{stmt}, \pi_{\text{con}}) \in \{0, 1\}^\ell 
23: \quad \pi_{\text{cut}} = (d_{i,b}, \gamma) 
24: \quad \pi = (\pi_{\text{cut}}, \pi_{\text{con}}) 
25: \quad \text{return } (pk_R, (\text{com}_{i,b}, c_{i,b}^R, c_{i,b}^L, 1, b, \pi)).$

PKEAS.Vfy$(ck, c, vper)$

1: \text{parse } c = (pk, ((\text{com}_{i,b}, c_{i,b}^R, c_{i,b}^L), 1, b), 
2: \quad ((d_{i,\gamma}, i, \pi_{\text{con}})) 
3: \quad \text{stmt} = (vper, pk_R, \text{com}_{i,b}, c_{i,b}^R, c_{i,b}^L, 1, b) 
4: \quad \gamma = \text{RO}(\text{stmt}, \pi_{\text{con}}) \in \{0, 1\}^\ell 
5: \quad \text{if } \text{NIZK}_{\text{AS}}.\text{Verify}(\text{stmt}, \pi_{\text{con}}, R_{\text{PKEAS}}) = 0 \text{ then} 
6: \quad \text{return } 0 
7: \quad \text{// Check cut-and-choose proof } 
8: \quad \text{for } i = 1, \ldots, \ell(\lambda) 
9: \quad b = \gamma_i 
10: \quad \text{parse } d_{i,b} = (w_{i,b}, r_{i,b}^\text{com}) 
11: \quad \text{parse } w_{i,b} = (m_{i,b}, \text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^R, r_{i,b}^L) 
12: \quad \text{check } \text{com}_{i,b} \text{ is } \text{COM}.\text{Com}(ck, w_{i,b}, r_{i,b}^\text{com}) 
13: \quad \text{check } c_{i,b}^R \text{ is } \text{PKE}_{\text{NSCE}}.\text{Enc}(pk, m_{i,b}, r_{i,b}^R) 
14: \quad \text{check } c_{i,b}^L \text{ is } \text{SK}_{\text{NSCE}}.\text{Enc}(\text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^L) 
15: \quad \text{return } 0 \text{ if any check fails } 
16: \quad \text{return } 1.
PKEAS-DecRE\(\left(ck, \ sk, \ c, \ vper\right)\)

1: if PKE\(\text{AS. Vfy}\left(ck, c, vper\right) = 0\) then return \(⊥\)
2: \(\gamma = \) \(\text{parse }\ c = \left(\pk, \left(\com_{i,b}, \ct^{\text{RE}}_{i,b}, \ct^{\text{LE}}_{i,b} \right)_{i,b}, \left((\delta_{i,\gamma_{i}}), \pi_{\text{con}}\right)\right)\)
3: if \(\pk\) does not belong to \(\sk\) then return \(⊥\)
4: \(\text{stmt} = \left(vper, \pk_{R}, \left(\com_{i,b}, \ct^{\text{RE}}_{i,b}, \ct^{\text{LE}}_{i,b} \right)_{i,b}\right)\)
5: // Reconstruct \(m\) from secret-shares
6: for \(i = 1, \ldots, \ell(\lambda)\)
7: \(\text{for } b \in \{0, 1\}\)
8: \(m_{i,b} = \text{PKE}_{\text{NCE. Dec}}(\sk, \ct^{\text{RE}}_{i,b})\)
9: \(m_{i} = m_{i,0} + m_{i,1} \quad \text{if one share is } ⊥\)
10: if \(m\) is absolute majority of \(m_{i}\) then
11: return \(m\)
12: else return \(⊥\)

PKEAS-DecLE\(\left(ck, \ usk, \ c, \ vper\right)\)

1: if PKE\(\text{AS. Vfy}\left(ck, c, vper\right) = 0\) then return \(⊥\)
2: \(\text{parse }\ c = \left(\pk, \left(\com_{i,b}, \ct^{\text{RE}}_{i,b}, \ct^{\text{LE}}_{i,b} \right)_{i,b}, \left((\delta_{i,\gamma_{i}}), \pi_{\text{con}}\right)\right)\)
3: \(\gamma = \text{RO}(\text{stmt}, \pi_{\text{con}}) \in \{0, 1\}^{\ell}\)
4: // PKE\(\text{AS. Vfy}\left(ck, c, vper\right) \neq ⊥\) ensures NIZK\(\text{AS. Verify}(\text{stmt}, \pi_{\text{con}}, \mathcal{R}_{\text{PKEAS}}) = 1\).
5: // Extract possible uid from cut-and-choose proof
6: \(\text{for } i = 1, \ldots, \ell(\lambda)\)
7: \(b = \gamma_{i}\)
8: \(\text{parse }\ delta_{i,b} = \left(w_{i,b}, \r_{\text{com}}_{i,b}\right)\)
9: \(\text{parse }\ w_{i,b} = \left(m_{i,b}, \usk_{i,b}, \uid_{i,b}, \delta^{\text{RE}}_{i,b}, \delta^{\text{LE}}_{i,b}\right)\)
10: \(\usk_{i,1-b} = \usk_{i,b} + \usk\)
11: \(\uid_{i,1-b} = \text{SKE}_{\text{NCE. Dec}}(\usk_{i,1-b}, \ct^{\text{LE}}_{i,1-b})\)
12: \(\uid_{i} = \uid_{i,0} + \uid_{i,1} \quad \text{if one is } ⊥\)
13: if \(\uid\) is absolute majority of \(\uid_{i}\) then
14: return \(\uid\)
15: \(\text{else return } ⊥\)
PKEAS\texttt{.Sim}(ck, pk, vper)
1: \hspace{1em} // Prepare cut-and-choose encryption
2: \hspace{1em} for \(i = 1, \ldots, \ell(\lambda)\)
3: \hspace{1em} \(m_{i,0}, m_{i,1} \leftarrow \{0, 1\}^{\text{poly}}\) \hspace{1em} // Simulate random shares
4: \hspace{1em} usk_{i,0}, usk_{i,1} \leftarrow \text{SKE}\text{NCE.K}
5: \hspace{1em} uid_{i,0}, uid_{i,1} \leftarrow \text{UID}
6: \hspace{1em} for \(b \in \{0, 1\}\) \hspace{0.5em} \(d_{i,b} = (w_{i,b}, r_{i,b}^\text{comp})\)
7: \hspace{1em} \(w_{i,b} = (m_{i,b}, \text{usk}_{i,b}, \text{uid}_{i,b}, r_{i,b}^\text{RE}, r_{i,b}^\text{LE})\)
8: \hspace{1em} \(\text{com}_{i,b} = \text{COM}.\text{Com}(ck, w_{i,b}, r_{i,b}^\text{comp})\)
9: \hspace{1em} \(\text{com}_{i,b} = \text{COM}.\text{Com}(ck, (\text{usk}_{i,b}, \text{uid}_{i,b}); r_{i,b}^\text{comp})\)
10: \hspace{1em} \(ct_{i,b}^\text{RE} = \text{PKE}\text{NCE}.\text{Sim}(1^\lambda, pk)\) \hspace{1em} // Simulate encryptions
11: \hspace{1em} \(ct_{i,b}^\text{LE} = \text{SKE}\text{NCE}.\text{Sim}(1^\lambda)\)
12: \hspace{1em} // Non-interactively compute challenge \(\gamma\)
13: \hspace{1em} \(\text{stmt} = (vper, pk_R, (\text{com}_{i,b}, ct_{i,b}^\text{RE}, ct_{i,b}^\text{LE})_{i,b})\)
14: \hspace{1em} \(\gamma = \text{RO}(\text{stmt}, \pi\text{con}) \in \{0, 1\}^\ell\)
15: \hspace{1em} // Simulate the consistency proof.
16: \hspace{1em} \(\pi\text{cut} = (d_{i,\gamma_i})_i\)
17: \hspace{1em} return \((pk, ((\text{com}_{i,b}, ct_{i,b})_{i,b}, ct_R, (\pi\text{cut}, \pi\text{con})))\).

PKEAS\texttt{.Expln}_{\text{RE}}(ck, sk, c, m, vper)
1: \hspace{1em} parse \(c = (pk, ((\text{com}_{i,b}, ct_{i,b}^\text{RE}, ct_{i,b}^\text{LE})_{i,b}); (d_{i,\gamma_i})_i, \pi\text{con})\)
2: \hspace{1em} \(\gamma = \text{RO}(vper, pk, ((\text{com}_{i,b}, ct_{i,b}^\text{RE}, ct_{i,b}^\text{LE})_{i,b})_i)\)
3: \hspace{1em} // Explain all unopened \(ct_{i,b}^\text{RE}\)
4: \hspace{1em} for \(i = 1, \ldots, \ell(\lambda)\)
5: \hspace{1em} \(b = 1 - \gamma_i\)
6: \hspace{1em} \text{PKE}\text{NCE}.\text{Expln}(sk, ct_{i,b}^\text{RE}, m - m_{i,b})

PKEAS\texttt{.Expln}_{\text{LE}}(ck, usk, c, uid, vper)
1: \hspace{1em} parse \(c = (pk, ((\text{com}_{i,b}, ct_{i,b}^\text{RE}, ct_{i,b}^\text{LE})_{i,b}); (d_{i,\gamma_i})_i, \pi\text{con})\)
2: \hspace{1em} \(\gamma = \text{RO}(vper, pk, ((\text{com}_{i,b}, ct_{i,b}^\text{RE}, ct_{i,b}^\text{LE})_{i,b})_i)\)
3: \hspace{1em} // Explain all unopened \(ct_{i,b}^\text{LE}\)
4: \hspace{1em} for \(i = 1, \ldots, \ell(\lambda)\)
5: \hspace{1em} \(b = 1 - \gamma_i\)
6: \hspace{1em} \text{SKE}\text{NCE}.\text{Expln}(usk - \text{usk}_{i,b}, ct_{i,b}^\text{LE}, uid - uid_{i,b})

86
C.5 On Efficiency and the Necessity of (NI)NCE

One may argue that, due to the use of the cut-and-choose zero-knowledge proof of knowledge in PKE_{AS}, it is unnecessarily or impractically inefficient. However, as noted before, this is a consequence of factors which seem impossible (or at least, very hard) to avoid.

NINCE In a nutshell, NINCE-secure SKE (or PKE) allows to simulate an arbitrary number of ciphertexts without knowledge of the encrypted message, and upon a “corruption” query, output a secret key which correctly decrypts all simulated ciphertexts. By an information-theoretic argument, NINCE is impossible in the plain model, and even in the non-programmable random oracle model [48] (NPROM). Thus, in this context, a programmable ROM is not an excessively strong assumption. For NCE, encryption and decryption is allowed to be interactive, and there are simple constructions in the plain model (assuming secure erasure).

NINCE and F_{ASTE} and F_{AS} Since we do not consider adaptive corruption, there is no immediate relation between F_{ASTE} and NINCE. However, there is an informal connection which suggests that implementing F_{ASTE} based on F_{AS} requires similar assumptions and techniques as NINCE. Namely, if F_{AS} is used to store escrow keys, and non-interactive encryption and decryption algorithms are used, as is done in Π_{ASTE}, then the request of the escrow keys behaves very similar to adaptive corruption of the key. In particular, this suggests that realizing F_{ASTE} with a non-interactive implementation of Execute Access is impossible without strong setups (like the PROM). As such setups are “black-box”, combining them with zero-knowledge is non-trivial. Although expensive, cut-and-choose techniques are (to the best of our knowledge) the only known approach.

Avoiding NINCE in the PROM As noted above, avoiding PROM-based NINCE and cut-and-choose constructions seems hard. Except, by an interactive protocol for the Execute Access task; however, this is undesirable in practice.

C.6 Efficiency estimate of PKE_{AS}

As the efficiency of ASTE is depends mostly on PKE_{AS}, we give a rough estimate here, which clarifies the rough order of magnitude.

Clearly, we require $2\ell(\lambda)$ encryptions with PKE_{NCE} and SKE_{NCE}, where $\ell(\lambda) = 2\lambda$ is the minimal choice for $\lambda$ bits of security.\footnote{If $k$ of the cut-and-choose ciphertexts are “bad”, e.g., don’t share the message $m$ extracted from $\pi_{con}$, then the probability to get a bad challenge, failing to detect the inconsistency, is $2^{-k}$. We need $k \geq \ell(\lambda)/2$ (i.e., the majority of shares is bad) for decryption of PKE_{AS} to fail/be inconsistent with $\pi_{con}$. For $\lambda$ bits of security, we thus impose $2^{-\ell(\lambda)-\ell(\lambda)/2} = 2^{-\ell(\lambda)/2} \leq 2^{-\lambda}$, hence $\ell(\lambda) \geq 2\lambda$. With this, about $2^\lambda$ tries are required to get a constant success chance.} With space-efficient public key encryption schemes based on DLOG, the size of public key encryptions is two
group elements of size $2\lambda$ each. For secret key encryptions, the size is (at least) plaintext length, which is at least $2\lambda$ bits for $uid$ (as a PKE public key). This means at least $4\lambda \cdot 4\lambda$ bits for public key (resp. $4\lambda \cdot 2\lambda$ secret key) ciphertexts, so at least $24\lambda^2$ bits for the ciphertexts in the cut-and-choose setting alone. For $\lambda = 128$, this means $24 \cdot 2^{16}$ bits are a lower bound on the size, and using, e.g., DDH-based public key encryption schemes, at least $8\lambda$ group exponentiations are required.

For the cut-and-choose proof, there are many options and trade-offs. Our definition of $\text{PKE}_{\text{AS}}$ is optimized for clarity, and thus we chose to make separate $com_{i,b}$. However, succinct commitments with succinct proofs of opening are an option as well. Similarly, the proof system for $\pi_{\text{con}}$ can be succinct. This is roughly a trade-off between size and computational efficiency of the sender. For the receiver, depending on the chosen proof system, a larger or smaller amount of computation is incurred. Thus, it is not straightforward to find a suitable estimate here.

We give rough estimates of (lower) bounds for $\lambda = 128$ per encryption (i.e. run of $\text{PKE}_{\text{AS}, \text{Enc}}$):

- At least $1536$ kiB are required per ciphertext.
- For succinct (and efficient) zero-knowledge proofs and commitments, the total size should be around $1536$ kiB to $2$ MiB, depending on the chosen proof systems, but the computational overhead is likely rather large (say $10 \times$—$1000 \times$) for current proof systems.
- For practically efficient (but large) zero-knowledge proofs based on $\Sigma$-protocols and ElGamal commitments:
  - At least additionally $8\lambda$ group exponentiations are required for computing commitments and the same amount for the proofs. This leads to at least $24\lambda$ overall, roughly $3000$ for $\lambda = 128$.
  - Ciphertext size increases by at least $4\lambda \cdot 4\lambda$ for the commitments and twice that for proofs. Total ciphertext size is at least $(24 + 16 + 32)\lambda^2$, which is more than $4.5$ MiB for $\lambda = 128$.

Overall, $\text{PKE}_{\text{AS}}$ encryption is not unusably inefficient nor are the ciphertexts unusably large. But even our (very) optimistic lower bounds show that the costs incurred by the cut-and-choose proof seem to make the scheme rather costly. This leaves open the question for a truly practical $\text{PKE}_{\text{AS}}$ alternative (which presumably requires new techniques for zero-knowledge-compatible NINCE).

---

29 This is a very optimistic assumption. Many approaches may incur a far larger overhead if secret key and message spaces of encryption schemes are unfavorable for standard $\Sigma$-protocol-based zero-knowledge proofs. Handling via (NP-)reductions would inflating proof size and computational costs. Techniques for small overhead exist in special groups, e.g., given bilinear maps, but the resulting group elements are larger than (our assumed) $2\lambda$ bits.
D Discussion of Design Decisions

In this section we want to elaborate on some decisions made during the design of the whole system.

A first question would be why we used the threshold crypto approach (secret-sharing a single decryption key and encrypting the user’s escrow secrets with the corresponding public key) instead of directly secret-sharing the user escrow secrets with the blockchain committee. Since there is more literature related to the secret-sharing approach (e.g., [33, 8]) this approach might seem more natural at first glance. But a big drawback of this approach is scalability. Since there very well may be millions of escrow secrets (one for each user and validity period), the workload for the committee is immense since they need to hand-over all those secrets. While there has been effort to speed up the hand-over process [20], the workload for the committee remains high. This raises the question on how to incentivize parties to apply for a committee position, which is an entirely different research direction.

One idea to reduce the workload for committee members would be to not have one committee that handles all secrets, but multiple committees that each handle one (or a constant number of) secrets. While this reduces the workload per committee member, it significantly increases the number of nodes needed for all the committees. If not enough nodes are available to fill the millions of committee member positions, some nodes will have to apply for multiple committees – and thus their workload increases again. Despite the glaring disadvantage of workload, the multiple-committee approach has one advantage: if an attacker were to succeed in corrupting the majority of one of the committees, only the secrets of that committee would be at risk, not all secrets.

With our threshold crypto approach, the committee only stores one item (the decryption key), so the hand-over phase should be more efficient than the secret-sharing approach.

One disadvantage of our current approach is that if an attacker manages to compromise a majority of the blockchain committee, it can reconstruct the threshold decryption key. If that attacker then also colludes (or is) the system operator, the attacker would be able to compute all escrowed user secrets. To prevent that, the threshold encryption keypair would need to be changed periodically, which would lead to a significant efficiency decrease. Also, then the question arises how law enforcement would be able to receive decrypted user secrets that were encrypted with a previous threshold key. The committee could keep copies of all previous keypairs to be able to decrypt ciphertexts created during previous key epochs. This would lead to a forward secure but not backward secure system.

Another question is why we put a lot of trust on the system operator. As explained in Section 2.3 we implicitly have to trust SO in some aspects, although SO is modeled as corruptible in our system. In particular, we assume that SO honestly stores the partial secrets and ciphertexts and releases them upon LE’s request because if SO cheats at this step, we can only detect it but not prevent it. We believe this assumption to be reasonable since, to earn revenue, SO has
an intrinsic motivation to keep the system running for a long time. However, it would be possible to remove that trust assumption. The user could encrypt the ciphertext containing the escrow secret again, this time under LE’s public encryption key. The resulting ciphertext would then directly be sent to and stored by LE. While this approach removes some trust from SO, it has two disadvantages: 1. LE is now directly involved in every Store Secret task and thus possibly in millions of interactions each validity period (e.g., month). 2. This directly limits the role of LE to be played by a single party (or multiple parties sharing a single encryption keypair). In our system we wanted to minimize the involvement of LE (in our system LE’s workload only scales with the number of surveilled users, not with the number of existing users) and to keep the possibility open to expand the system to multiple law enforcement agencies (each with their own key pair). For these two reasons, we decided to put the additional trust in SO.

This second reason is also the reason why there is no fixed LE keypair in our system (neither in $F_{AS}$ nor in $F_{AD}$): We wanted the system to be easily expandable to multiple LE keys. Therefore, every time LE requests the decryption of a partial secret on the blockchain, it can specify a public key under which the result will be encrypted. When the role of LE is played by multiple parties, each party can specify its own key pair there.

Having the idea of multiple LE parties in mind, the question arises why we limit our system to a single auditor party with a single and fixed encryption key pair. Would it not be preferable to be able to selectively determine a different auditor for each warrant request? We deliberately chose to stick to a single auditor key pair to keep the system from being overly complex. Nonetheless, the system is designed in a way so that it is easy to extend it to multiple auditor parties (for example, there might be a separate AU party for each LE party).

A note on GDPR’s “right to be forgotten”. We also consider user rights, such as the “right to be forgotten” (Article 17 of the European General Data Protection Regulation (GDPR)), which enables users to request the deletion of all their personal data. Publicly accessible within our system are only ciphertexts that are stored on the blockchain and thus cannot simply be deleted, as blockchains are write-only. There are two types of ciphertexts.

Firstly, there are ciphertexts containing a secret-share of the user’s escrow secret. The other secret-share of this secret is held by the system operator, who has the ability to delete its share upon receiving a deletion request from the user. Without the system operator’s share, it is not possible to reconstruct the user’s escrow secret.

Secondly, court orders encrypted under the auditor’s key are stored on the blockchain. To avoid granting infinite access to these orders, our system can be expanded to incorporate annually rotating auditor keys. Then, upon the expiration of the retention obligation, the auditor’s decryption key for the warrants pertaining to that specific year can simply be deleted.
The concept of deleting an encryption key instead of deleting the ciphertext is commonly known as “crypto-shredding”. This approach is widespread in the IT security community, e.g., in HDDs, SSDs, file systems and databases.

E Security Proof: $\Pi_{AS}$ UC-realizes $\mathcal{F}_{AS}$

Here we prove that the protocol $\Pi_{AS}$ from Appendix B.2 UC-realizes the functionality $\mathcal{F}_{AS}$ from Section 3. In particular, we prove the following theorem.

**Theorem 1.** $\Pi_{AS}$ UC-realizes $\mathcal{F}_{AS}$ in the $\{\mathcal{F}_{AD}, \mathcal{F}_{CRS}, \mathcal{F}_{BB}, \mathcal{G}_{CLOCK}\}$-hybrid model under the assumptions that COM is a (computationally) hiding, (statistically) binding and (dual-mode) extractable and equivocable commitment scheme, $\Sigma$ is a EUF-CMA secure signature scheme, NIZK is a straight-line simulation-extractable non-interactive zero-knowledge proof system, and TPKE is IND-CPA secure against all PPT-adversaries $\mathcal{A}$ who statically corrupts either (1) a subset of the users, (2) $LE$ and a subset of the users, (3) $SO$ and a subset of the users, or (4) $SO$, $LE$ and a subset of the users.

To simplify the security proof a bit, we split it into two cases, depending on whether $SO$ is honest or corrupted. We call the first and second corruption scenario (where $SO$ is honest) **System Security** and the third and fourth corruption scenario (where $SO$ is corrupted) **User Security** and prove them separately.

E.1 System Security

In the **System Security** case, we handle the following corruptions:

- $U$ can be statically corrupted, so some $U$ are corrupted and some $U$ are honest.
- $SO$ is honest here
- $LE$ can be statically corrupted
- $J$ and $AU$ are always honest

We now first describe a simulator $S$ for those corruptions and then proceed to prove the security. The simulator uses the following writing conventions:

- If $\mathcal{F}_{AS}$ wants to deliver outputs to honest parties, delay them until the simulator explicitly allows them.
- Deny any other calls until the System Init task has been successfully completed
- Hybrid functionalities $\mathcal{F}_{CRS}$ and $\mathcal{F}_{BB}$ are simulated honestly. We assume in the following that whenever calls to those functionalities are made, $S$ simulates them honestly.
- Hybrid functionality $\mathcal{F}_{AD}$ is mostly simulated honestly, except the tasks Request Decryption, Get Statistics and Handover (for details see the paragraph “simulation of $\mathcal{F}_{AD}$”)

The simulator $S$ for System Security is defined as follows.
Simulator for System Security

State of the Simulator

- **Setup**
  - TPKE keypair: \((\text{pk}, \text{sk})\)
  - SO Signing keypair: \((\text{vk}_{SO}, \text{sk}_{SO})\)
  - J Signing keypair: \((\text{vk}_J, \text{sk}_J)\)
  - Common reference strings \((\text{crs}^A_{zk}, \text{crs}_{com})\)
  - Trapdoor \(td_{com}\) for extracting commitments
  - Trapdoor \(td_{NIZK}\) for simulating and extracting NIZKs

\[ L_{\text{sys-sec}} \]
- \(L_{\text{sys-sec}}^{\text{Registered}}\) with \((uid, \sigma_{\text{reg}}, \text{com}_{uid}, \text{decom}_{uid})\) entries for **corrupted** users. Filled in the task User Registration. This is the same list that SO keeps in \(P_{AS}\).
- \(L_{\text{sys-sec}}^{\text{HonestU}}\) with \((pid, uid, \text{vk}_U, \text{sk}_U)\) entries for **honest** users. Filled in the task User Registration.
- \(L_{\text{sys-sec}}^{\text{StoreSecret}}\) with \((uid, \text{vper}, \text{secret}, \text{com}_{\text{sec}}, \sigma_{\text{sec}}, \sec_2, ct, \sigma_U)\) entries for **corrupted** users. Filled during StoreSecret.

- If LE is corrupted:
  - The list \(L_{\text{sys-sec}}^{\text{Warrants}}\) with \((W, 1, \tilde{W}, \sigma_{\tilde{W}})\) entries. Stored in Request Warrant (like J in \(P_{AS}\))
  - The list \(L_{\text{sys-sec}}^{\text{HonestSecrets}}\) stores \((uid_i, \text{vper}_i, \sec_{i, 1})\) entries for **honest** users. Filled during Request Warrant and used in \(F_{AD}.\text{RequestDecryption}\)
  - The list \(L_{\text{sys-sec}}^{\text{ReqNotReady}}\) temporarily stores \((ct_i, \text{sec}_{2,i})\) entries for **honest** users. Filled during the simulation of \(F_{AD}.\text{RequestDecryption}\) and cleared during the simulation of \(F_{AD}.\text{Handover}\).

- If LE is honest:
  - The lists \(L_{\text{sys-sec}}^{\text{Warrants}}, L_{\text{sys-sec}}^{\text{Requests}}, L_{\text{sys-sec}}^{\text{ReadyRequests}}, L_{\text{sys-sec}}^{\text{Stats}}\) in \(F_{AD}\) are not used. Instead, manage a separate list for statistics: \(L_{\text{sys-sec}}^{\text{Stats}}\) with \(f_t(W)\) entries.
  - This list is filled during RequestWarrant.
  - List for simulation of \(F_{BB}\): \(L_{BB}\) (initially empty).

**System Setup** \((F_{CRS})\):

- \((\text{crs}^A_{zk}, td_{NIZK}) \leftarrow \text{NIZK.Setup}(1^\lambda)\)
- \((\text{crs}_{com}, td_{com}^{\text{Ext}}) \leftarrow \text{COM.Setup}^{\text{Ext}}(1^\lambda)\)
- Store all generated values

**System Setup** \((F_{BB})\):

- Create empty list \(L_{BB}\).
System Setup ($\mathcal{F}_{AD}$):
- Execute this during the System Init task
- Create signing keypair for SO: ($vk_{SO}, sk_{SO}$) $\leftarrow \Sigma.\text{Gen}(1^\lambda)$ and store it
- Create signing keypair for J: ($vk_J, sk_J$) $\leftarrow \Sigma.\text{Gen}(1^\lambda)$ and store it
- Honestly simulate the Init task of $\mathcal{F}_{AD}$ with the generated public keys as inputs. Store all generated variables and lists. Leak all necessary information to the adversary.

System Setup ($\mathcal{F}_{AS}$):
- Choose the same system parameters for $\mathcal{F}_{AS}$ as $\Pi_{AS}$

Simulation of $\mathcal{F}_{CRS}$
Corrupt P can issue calls to $\mathcal{F}_{CRS}$ whenever they want, not just at the intended points in the protocol. We specify here once how all calls from P to $\mathcal{F}_{CRS}$ to all tasks are handled and ignore calls to $\mathcal{F}_{CRS}$ for most tasks in the following.
$S$ answers all calls honestly.

Simulation of $\mathcal{F}_{BB}$
Corrupted U can issue calls to $\mathcal{F}_{BB}$ whenever they want, not just at the intended points in the protocol. We specify here once how all calls from P to $\mathcal{F}_{BB}$ to all tasks are handled and ignore calls to $\mathcal{F}_{BB}$ in the following.
$S$ answers all calls honestly.

Simulation of $\mathcal{F}_{AD}$
Corrupt P can issue calls to $\mathcal{F}_{AD}$ whenever they want, not just at the intended points in the protocol. We specify here once how all calls from P to $\mathcal{F}_{AD}$ to all tasks are handled and ignore calls to $\mathcal{F}_{AD}$ for most tasks in the following.
$S$ simulates $\mathcal{F}_{AD}$ honestly (including taking track of all internal lists and leaking information to the adversary), except for the following tasks:

- Request Decryption
  1. Retrieve $\left(\text{REQUEST}, \tilde{W}, \sigma_{\tilde{W}}\right)$ as input from LE
  2. Check Warrant Signature: $b \leftarrow \Sigma.Vfy(vk_J, \tilde{W}, \sigma_{\tilde{W}})$
  3. If $b = 0$, abort (Warrant not valid)
  4. Find the entry $(W, 1, \tilde{W}, \sigma_{\tilde{W}}) \in \text{sys-sec Warrants}$ containing $\tilde{W}$. If none exists, abort (warrant not granted)
  5. Call $\mathcal{F}_{AS}$ in the name of LE with input $(\text{GetSecrets}, W)$ and get output $(\text{GotSecrets}, (secret_1, \ldots, secret_v))$ for LE
  6. Parse enhanced warrant: $(\tilde{W}_1, \ldots, \tilde{W}_v) \leftarrow \tilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \tilde{W}_i$
7. For each $i$ from 1 to $v$:
   (a) Check if there is an entry $(\cdot, \text{uid}_i, \cdot, \cdot) \in L_{\text{sys-sec}}^\text{HonestU}$. If yes (case: user is honest) go to step 7b, if not (case: user is corrupted) go to step 7f.
   (b) For $\text{uid}_i$, retrieve $(\text{uid}_i, \text{vper}_i, \text{sec}_{1,i})$ from $L_{\text{sys-sec}}^\text{HonestSecrets}$
   (c) Compute $\text{sec}_{2,i}$ such that $\text{sec}_{1,i} \oplus \text{sec}_{2,i} = \text{secret}_i$
   (d) Add $(ct_i, \text{sec}_{2,i})$ to $L_{\text{sys-sec}}^\text{ReqNotReady}$
   (e) Then go to step 8
   (f) Append $ct_i$ to $L_{\text{Requests}}$ of $\mathcal{F}_\text{AD}$
   (g) Then go to step 8

8. Send $\text{(REQUEST, pid}_{\text{LE}}, f_t(W), |\tilde{W}|, v) \rightarrow \mathcal{A}$

9. When $\mathcal{A}$ allows to deliver output, store $\tilde{W}$ in list of warrants $L_{\text{Warrants}}$ of $\mathcal{F}_\text{AD}$

10. Output (REQUEST) to LE

- **Get Statistics**
  - If LE is corrupted:,
    1. Simulate task honestly
  - If LE is honest:
    1. Send (GETSTATISTICS) to the adversary
    2. Output (GETSTATISTICS, $L_{\text{sys-sec}}^\text{Stats}$)

- **Handover**
  Simulate honestly, but additionally execute the following steps at the end:
  - For each item $(ct_i, \text{sec}_{2,i})$ in $L_{\text{sys-sec}}^\text{ReqNotReady}$, append $(ct_i, \text{sec}_{2,i})$ to $L_{\text{sys-sec}}^\text{Pending}$
  - Clear $L_{\text{sys-sec}}^\text{ReqNotReady}$

**System Init (honest SO, honest J, honest AU)**

1. Upon receiving the leak (INIT) from $\mathcal{F}_\text{AS}$
   (a) Initialize $\mathcal{F}_\text{AD}$ as described above
   (b) Allow $\mathcal{F}_\text{AS}$ to deliver output to all parties

**Party Init (honest P)**

1. Upon receiving the leak (PINIT) from $\mathcal{F}_\text{AS}$
   (a) Send (GETSKEYS) and (GETPK) to the adversary
   (b) Allow $\mathcal{F}_\text{AS}$ to deliver output to all parties

**Party Init (corrupted P)**

This is handled by simulation of $\mathcal{F}_\text{AD}$.

**User Registration (honest U, honest SO)**

1. Upon receiving the leak (REGISTER, pid, uid) from $\mathcal{F}_\text{AS}$
   (a) Send (OK) to $\mathcal{F}_\text{AS}$
2. When $F_{AS}$ wants to deliver output,
   (a) Generate keypair for the user: $(vk_U, sk_U) \leftarrow \Sigma, \text{Gen}(1^\lambda)$
   (b) Store $(pid, uid, vk_U, sk_U)$ in list of honest users $L_{\text{sys-sec}}^{\text{honestU}}$
   (c) Honestly simulate Register and Retrieve calls to $F_{BB}$ including leaks to the adversary
   (d) Send $(\text{GetSKeys})$ and $(\text{GetPK})$ to the adversary
   (e) Allow $F_{AS}$ to deliver output

User Registration (corrupted U, honest SO)
1. Upon receiving the message $(uid)$ from U with $pid = pid_U'$
   (a) If there exists an entry $(uid, \cdot, \cdot, \cdot)$ in $L_{\text{sys-sec}}^{\text{Registered}}$, let $S$ abort in the name of SO (User already successfully registered)
   (b) Call $F_{AS}$ in the name of U with $(\text{Register}, uid)$ under the pid $pid_U'$.
      i. After receiving $(\text{Register}, pid_U', uid)$ from $F_{AS}$, send $(\text{Ok})$ to $F_{AS}$
      ii. If $F_{AS}$ aborts, $S$ aborts as well.
2. Upon receiving output $(\text{Registered})$ from $F_{AS}$ to U
   (a) Note that receiving this output implies that $uid = uid'$, where $uid'$ is the secret SO input.
   (b) Report message $(\text{UID\_OK})$ from SO to U
3. Upon receiving $(\text{Ok})$ from U to SO
   (a) Honestly simulate call $(\text{Retrieve}, uid)$ to $F_{BB}$ including leaks to the adversary
   (b) Receive answer $(\text{Retrieve}, pid_U'^*, uid, vk_U'^*)$ from $F_{BB}$. If $pid_U'^* \neq pid_U'$, abort (UID already taken by another user or not registered).
   (c) $(\text{com}_{uid}, \text{decom}_{uid}) = \text{COM}.\text{Com}(\text{crs}_{\text{com}}, uid)$
   (d) $\sigma_{\text{reg}} = \Sigma, \text{Sign}(sk_{SO}, \text{com}_{uid})$
   (e) Store $(uid, \sigma_{\text{reg}}, \text{com}_{uid}, \text{decom}_{uid})$ in $L_{\text{sys-sec}}^{\text{Registered}}$
   (f) Report message $(\sigma_{\text{reg}}, \text{com}_{uid}, \text{decom}_{uid})$ from SO to U
   (g) Allow $F_{AS}$ to deliver output to SO

Store Secret (honest U, honest SO)
Nothing to do, since $F_{AS}.\text{StoreSecret}$ does not leak anything, $H_{AS}.\text{StoreSecret}$ has no calls to subfunctionalities and all involved parties are honest.

Store Secret (corrupted U, honest SO)
1. Upon receiving $(uid, \sigma_{\text{reg}}, \text{com}_{uid}, \text{decom}_{uid}, \text{com}_{sec2}, \text{com}_{vper}, \text{decom}_{vper})$ from U to SO:
   (a) Extract Commitment $\text{com}_{vper}$ to get $vper: vper \leftarrow \text{COM}.\text{Extract}(t_{\text{com}}^{\text{ext}}, \text{com}_{vper})$
   (b) Call $F_{AS}$ with input $(\text{STORESECRET}, uid, vper)$ in the name of U
   (c) If $F_{AS}$ aborts, let $S$ abort as well (in the name of SO)
2. Upon receiving (SECRETSTORED, secret) as output from $F_{AS}$ to U
   (a) Extract Commitment $com_{sec2}$ to get $sec_2$: $sec_2 \leftarrow COM.\hat{\text{Extract}}(t_{com}^{ext},$ $com_{sec2})$
   (b) Calculate $sec_1$ such that $secret = sec_1 \oplus sec_2 \in S$
   (c) Upon receiving $(ct, com_{sec}, \pi_{as}, \sigma_U, \sigma_{sec_1}, \text{com}_{sec_1}, \text{decom}_{sec_1})$ from U to SO:
      (a) Compute signature $\sigma_{sec} = \Sigma.\text{Sign}(sk_{SO}, (\text{com}_{uid}, \text{com}_{sec}, \text{com}_{oper}))$
      (b) Report message $(\sigma_{sec})$ from SO to U
      (c) Store $(uid, vper, secret, sec_1, \text{com}_{sec_2}, \text{decom}_{sec_1}, \sigma_{sec_1}, sec_2, ct, \sigma_U)$ in $L_{sys}^{\text{sec}}$
      (d) Allow $F_{AS}$ to deliver output to SO

Request Warrant (corrupted LE, honest J, honest SO)

1. Upon receiving the message $(W)$ from LE to J
   (a) Call $F_{AS}$ in the name of LE with input (REQUESTWARRANT, W)
   (b) Allow $F_{AS}$ to deliver output to J
2. Case 1: If then $F_{AS}$ delivers the output (REQUESTWARRANT, 0) to LE:
   (a) Report message $(0, \bot)$ from J to LE
      Note that this concluded the simulation for this task in this case
3. Case 2: If then $F_{AS}$ leaks (REQUESTWARRANT, $f_i(W), |\tilde{W}|, v)$:
   (a) Set $b := 1$
   (b) Set $\sigma_W = \Sigma.\text{Sign}(sk_{J}, W)$
   (c) Store $(W, b, \bot, \bot)$ in $L_{Warrants}$
   (d) Report message $(b, \sigma_W)$ from J to LE
4. Upon receiving the message $(W, \sigma'_W)$ from LE to SO
   (a) Verify Signature: If $\sigma_W \neq \sigma'_W$, abort in the name of SO
   (b) Parse warrant: $(W_1, \ldots, W_v) \leftarrow W$ and $(uid_i, vper_i, meta_i) \leftarrow W_i$
   (c) Create empty lists $L_{C}^{sys-sec}$ and $L_{MagReqW}^{sys-sec}$
   (d) For each $i$ from 1 to $v$:
      i. Check if there is an entry $(i, uid_i, \cdot, \cdot) \in L_{C}^{sys-sec \text{ HonestU}}$. If yes (case: user is honest) go to step 4(d)i, if not (case: user is corrupted) go to step 4(d)x
      ii. Fake ciphertext $ct_i \leftarrow \text{TPKE.Enc}(pk_0)$ and append $ct_i$ to $L_{C}^{sys-sec \text{ HonestU}}$
      iii. For $uid_i$, retrieve entry $(pid_i, uid_i, vk_U, sk_U) \in L_{sys-sec \text{ HonestU}}$
      iv. Create signature $\sigma_{U,i} \leftarrow \Sigma.\text{Sign}(sk_U, (ct_i, uid_i, vper_i))$
      v. Draw random $sec_{1,i} \leftarrow S$ and commit to it: $(com_{sec_1,i}, \text{decom}_{sec_1,i}) \leftarrow COM.\text{Com}(crs_{com}, sec_{1,i})$
      vi. Create signature $\sigma_{sec_1,i} \leftarrow \Sigma.\text{Sign}(sk_U, com_{sec_1,i})$
      vii. Append $(\sigma_{U,i}, sec_{1,i}, com_{sec_1,i}, \text{decom}_{sec_1,i}, \sigma_{sec_1,i})$ to $L_{MagReqW}^{sys-sec \text{ HonestSecrets}}$
      viii. Append $(uid_i, vper_i, sec_{1,i})$ to $L_{MagReqW}^{sys-sec \text{ HonestSecrets}}$
      ix. Then go to step 4e
3. When A

5. Upon receiving the message (from

2. Upon receiving the leak (REQUESTWARRANT, f_t(W), |W|, v) from F_{AS}: 

x. For uid_i retrieve the entry (uid_i, vper_i, secret_i, sec_1, com_{sec_1}, \{\text{decom}_1, \sigma_{sec_1}\}, \sigma_{sec_2}, ct_i, \sigma_{U,i}) from L_{MagSec}. If none exists, abort in the name of SO (User has not stored a secret)

xi. Append ct_i to L_{MagSec} and (\{\text{decom}_1, \sigma_{sec_1}\}, \sigma_{sec_1}, ct_i) to L_{MagReqW}

xii. Then go to step 4e

(e) Send \(W, 1, \ldots, v\) to each \(W_i\)

(f) Report message (\(W, 1, \ldots, v\)) from SO to LE

5. Upon receiving the message (\(W, \{\sigma_{U,i}, \sigma_{sec_1}, \text{com}_{sec_1}, \sigma_{sec_1}, \sigma_{sec_1}\}\}_{i = 1}^{v}\) from LE to J

(a) Build \(W\) out of \(\hat{W}\) by deleting \(ct_i\) from each \(\hat{W}_i\)

(b) If there is no entry (W, 1, \ldots, 1) in L_{Warrants} abort in the name of J

(c) Parse enhanced warrant: (\(\hat{W}_1, \ldots, \hat{W}_v\)) \leftarrow W and (uid_i, vper_i, meta_i, ct_i) \leftarrow \hat{W}_i

(d) For i from 1 to v:
   - For uid_i, retrieve entry (\(\text{pid}_U, uid_i, v\sigma_{U,i}\)) in L_{BB}
   - Check signatures: If \(\Sigma.Vfy(v\sigma_{U,i}, ct_i, uid_i, \sigma_{U,i}) = 0\) or \(\Sigma.Vfy(v\sigma_{U,i}, \text{com}_{sec_1}, \sigma_{sec_1}) = 0\), abort in the name of J
   - Check commitment: If \(\text{COM.Open}(\text{crs}_{com}, \text{com}_{sec_1}, \sigma_{sec_1}, sec_{1,i}) = 0\), abort in the name of J

(e) Sign enhanced warrant: \(\sigma_{\hat{W}} = \Sigma:\text{Sign}(sk_{U,i}, \hat{W})\)

(f) Change the entry (W, 1, \ldots, 1) in L_{MagSec} to (W, 1, \hat{W}, \sigma_{\hat{W}})

(g) Report message (\(\sigma_{\hat{W}}\)) from J to LE

6. When the adversary allows \(F_{AD}\) to deliver output in \(F_{AD}.\text{RequestDecryption}\)

(a) Finish simulation of \(F_{AD}.\text{RequestDecryption}\)

(b) Send (OK) to \(F_{AS}\)

(c) Receive output \(\text{WarrantAnswer}, b\) from \(F_{AS}\)

**Request Warrant (honest LE, honest J, honest SO)**

1. Upon being notified that \(F_{AS}\) wants to deliver output to J, allow the output

2. Upon receiving the leak \(\text{REQUESTWARRANT}, f_t(W), |W|, v\) from \(F_{AS}\):

(a) Send (REQUEST, \text{pid}_{LE}, f_t(W), |W|, v) to \(A\)

3. When \(A\) allows to deliver output in \(F_{AD}.\text{RequestDecryption}\)

(a) Store \(f_t(W)\) in L_{Stats} for managing statistics in \(F_{AD}\)

(b) Send (OK) to \(F_{AS}\)

(c) Allow \(F_{AS}\) to deliver output to LE
<table>
<thead>
<tr>
<th>Get Secrets (corrupted LE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is handled by simulation of $F_{AD}$.</td>
</tr>
<tr>
<td>Get Secrets (honest LE)</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>1. Upon receiving the leak (GetSecrets) from $F_{AS}$</td>
</tr>
<tr>
<td>(a) Leak (Retrieve) to $A$</td>
</tr>
<tr>
<td>(b) Allow $F_{AS}$ to deliver output</td>
</tr>
<tr>
<td>Get Statistics (corrupted P)</td>
</tr>
<tr>
<td>This is handled by simulation of $F_{AD}$.</td>
</tr>
<tr>
<td>Get Statistics (honest P)</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>1. Upon receiving the leak (GetStatistics) from $F_{AS}$</td>
</tr>
<tr>
<td>(a) Send (GetStatistics) to $A$ and wait for okay from $A$</td>
</tr>
<tr>
<td>(b) Allow $F_{AS}$ to deliver output</td>
</tr>
<tr>
<td>Audit (honest AU)</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>1. Upon receiving the leak (AuditRequest) from $F_{AS}$</td>
</tr>
<tr>
<td>(a) Send (AuditRequest) to $A$ and wait for okay from $A$</td>
</tr>
<tr>
<td>(b) Allow $F_{AS}$ to deliver output</td>
</tr>
<tr>
<td>Prove (corrupted U)</td>
</tr>
<tr>
<td>Nothing to do since this is a completely local task.</td>
</tr>
<tr>
<td>Prove (honest U)</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>1. Upon receiving the message (Prove, stmt) from $F_{AS}$</td>
</tr>
<tr>
<td>(a) Simulate proof: $\pi \leftarrow \text{NIZK.Sim}(td_{\text{NIZK}}, stmt, R_{2k})$</td>
</tr>
<tr>
<td>(b) Send (Proof, $\pi$) to $F_{AS}$</td>
</tr>
<tr>
<td>Verify (corrupted P)</td>
</tr>
<tr>
<td>Nothing to do since this is a completely local task.</td>
</tr>
<tr>
<td>Verify (honest P)</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>1. Upon receiving the message (Verify, stmt, $\pi$) from $F_{AS}$</td>
</tr>
<tr>
<td>(a) Extract Witness: $\text{wit}<em>R \leftarrow \text{NIZK.Ext}(td</em>{\text{NIZK}}, stmt, \pi, R_{2k})$</td>
</tr>
<tr>
<td>(b) Send (witness, $\text{wit}<em>R$) to $F</em>{AS}$</td>
</tr>
</tbody>
</table>

We now prove the following theorem:

**Theorem 5 (System Security).** $\Pi_{AS}$ UC-realizes $F_{AS}$ in the $\{F_{AD}, F_{CRS}, F_{BB}, \varphi_{\text{CLOCK}}\}$-hybrid model under the assumptions that

- COM is a (computationally) hiding, (statistically) binding and extractable commitment scheme
- $\Sigma$ is a EUF-CMA secure signature scheme

98
NIZK is a straight-line simulation-extractable non-interactive zero-knowledge proof system.

TPKE is IND-CPA secure against all PPT-adversaries \( A \) who statically corrupts one of the following sets

1. a subset of the users
2. LE and a subset of the users

To prove Theorem 5, we proceed in a series of games, where \( F_{AS}^{i+1} \) behaves the same as \( F_{AS}^i \) except for the stated changes and \( S^{i+1} \) behaves the same as \( S^i \) except for the stated changes.

**Game 0.** Game 0 is equivalent to the real experiment. That is,

\[
H_0 := \text{Exp}^{\pi_{AS}}_{\text{CRS}, \text{BB}, \text{CLOCK}, S^0, Z}
\]

with \( S^0 \) being the dummy adversary. This means that all parties execute the real protocol.

**Proof Sketch: Game 0 \( \approx \) Game 1.**

\( S^1 \) executes the same code as the real parties on the same inputs (that have been forwarded by \( F_{AS}^1 \)). Thus, Game 0 and Game 1 are identical from the environment’s view.

**Game 1.** In this game, the simulator \( S^1 \) takes control over \( F_{CRS}, F_{BB} \) and \( F_{AD} \), but executes them honestly. Additionally, \( F_{AS}^1 \) is introduced, and all honest parties are replaced by dummy parties that forward their inputs to \( F_{AS}^1 \) and when receiving output from \( F_{AS}^1 \) forward it to the environment. \( F_{AS}^1 \), upon receiving any message from a dummy party forwards it to the simulator \( S^1 \) and asks it for output. \( S^1 \) executes the real protocol for all honest parties (using the inputs received from \( F_{AS}^1 \)) and instructs \( F_{AS}^1 \) to deliver the resulting outputs. Note that \( S^1 \) simulates \( F_{BB} \) like \( S \) does.

**Proof Sketch: Game 1 \( \approx \) Game 2.**

Indistinguishability follows from the extractability of \( \text{COM} \) and the simulation-extractability of \( \text{NIZK} \).

**Game 2.** During setup of \( F_{AD} \), \( S^3 \) is asked for the keypairs for SO, J and AU. \( S^3 \) generates the keypairs honestly and stores them.

**Proof Sketch: Game 2 \( \approx \) Game 3.**

\( S^3 \) runs the same key generation algorithm as the real protocol does for honest parties, therefore \( Z \)’s view remains unchanged.

**Game 4.** \( S^4 \) keeps track of additional information. For each honest \( U \), it stores \((uid, vper, sec)\) in the list \( L^{\text{HonestSecrets}} \) at the end of the Store Secret task. For each corrupted \( U \), it stores \((uid, vper, \bot, sec_1, com_{sec_1}, decom_{sec_1}, \sigma_{sec_1}, sec_2, \ldots)\).
\(ct, \sigma_U\) in the list \(L_{\text{sys-sec}}\) at the end of the Store Secret task. Note that normally instead of \(\bot\) the user’s secret should be stored in the list. But at this point we do not know the secret yet.

**Proof Sketch:** Game 3 \(\approx\) Game 4.

\(S^4\) only keeps track of more information (that the honest parties acquire during the real protocol), nothing to distinguish for \(Z\).

**Game 5.** In this game, \(F^5_{\text{AS}}\) additionally handles the tasks System Init and Party Init the same way \(F_{\text{AS}}\) does, and \(S^5\) handles these task the same way \(S\) does.

**Proof Sketch:** Game 4 \(\approx\) Game 5.

Since SO, J and AU are honest, the games are identical for the System Init task.

**Party Init for honest Parties:** \(Z\)’s view remains identical, since \(S^5\) only needs to leak \((\text{GetSKeys})\) and \((\text{GetPK})\) to \(A\) to emulate the call to \(F^5_{\text{AD}}\).

**Party Init for corrupted Parties:** Only the simulation of hybrid functionalities needs to be handled, which are already simulated honestly.

**Game 6.** In this game, \(F^6_{\text{AS}}\) additionally handles the Audit task the same way \(F_{\text{AS}}\) does, and \(S^6\) handles this task the same way \(S\) does. Note that since the auditor AU is assumed to be honest, simulation only requires a simulated \((\text{AuditRequest})\) to \(F^6_{\text{BCRA}}\).

**Proof Sketch:** Game 5 \(\approx\) Game 6.

Since AU is honest, these games are identical.

**Game 7.** In this game, \(F^7_{\text{AS}}\) additionally handles the User Registration task the same way \(F_{\text{AS}}\) does, and \(S^7\) handles this task the same way \(S\) does.

**Proof Sketch:** Game 6 \(\approx\) Game 7.

For honest U: \(Z\)’s view remains identical, since \(S^7\) only needs to leak \((\text{GetSKeys})\) and \((\text{GetPK})\) to \(A\) to honestly emulate the calls to \(F^7_{\text{AD}}\). Note that additionally some information about honest users \((\text{pid}, \text{uid}, \text{keys})\) are stored in a designated list \(L_{\text{sys-sec}}\).

For corrupted U: Since \(S\) essentially simulates SO in the User Registration task honestly, the hybrids are indistinguishable for \(Z\). Note that \(S\) keeps track of the list of registered users like SO does in \(H_{\text{AS}}\).

**Game 8.** If LE is honest, \(F^8_{\text{AS}}\) additionally handles the Request Warrant task the same way \(F_{\text{AS}}\) does for honest LE, and \(S^8\) handles this task the same way \(S\) does for honest LE. Additionally, \(S^8\) simulates \(F^8_{\text{AD}}\) as \(S\) does for the Get Statistics task in \(F^8_{\text{AD}}\). Note that if LE is corrupted, this game changes nothing.

**Proof Sketch:** Game 7 \(\approx\) Game 8.

\(F^8_{\text{AS}}\) sends \(\left\{ \text{REQUEST WARRANT}, f_t(W), |\tilde{W}|, v \right\}\) to \(S^8\). \(S^8\) then simulates the execution of \(F^8_{\text{AD}}, \text{RequestDecryption}\) by sending \(\left\{ \text{REQUEST}, pid_{LE}, f_t(W), |\tilde{W}|, v \right\}\) to \(A\), which yields an indistinguishable execution of the Request Warrant task. But now the Get Statistics task has to be adapted as well, since we now do not store a warrant in \(L_{\text{Warrants}}\) (since we do not actually know the warrant). This is fixed by directly storing \(f_t(W)\) in a separate list \(L_{\text{sys-sec}}\) during Request Warrant and using this list in \(F^8_{\text{AD}}, \text{GetStatistics}\) to return the same statistics as the real game does. The task \(F^8_{\text{AD}}, \text{GetStatistics}\) now behaves like \(S\) does.
since there we only changed the behavior for honest LE and the task is executed honestly for corrupted LE.

Game 9. If LE is corrupted, $F_{AS}^9$ additionally handles the Request Warrant task the same way $F_{AS}$ does for corrupted LE, and $S^9$ handles this task the same way $S$ does for corrupted LE. Additionally, $S^9$ simulates $F_{AD}$ as $S$ does for the Request Decryption and Handover tasks in $F_{AD}$. Note that if LE is honest, this game changes nothing. Also note that now $S^9$ simulates $F_{AD}$ as $S$ does (for every task and every corruption scenario).

Proof Sketch: Game 8 $\approx$ Game 9.

$S^9$ receives the necessary information from LE to call $F_{AS}^9$ with the correct input. After receiving output from $F_{AS}^9$, $S^9$ mostly executes the real protocol for J and SO. For corrupted users, the information needed to send the correct messages can be extracted from the list $L_{sys-sec}^{StoreSecret}$ that was filled during Store Secret (for corrupted users). For honest users, the strategy is a bit different, since the list $L_{sys-sec}^{StoreSecret}$ has no entries for honest users. For honest users, $S^9$ creates a fake ciphertext (by encrypting 0) and then honestly creates the remainder of the information needed to send the correct messages. Since TPKE is IND-CPA-secure, this change can not be detected. If the signature scheme $\Sigma$ is EUF-CMA-secure and the commitment scheme $COM$ is binding, $S^9$ aborts in the same cases as SO/J would do in $\Pi_{AS}$. Note that $S^9$ stores the granted warrants in $L_{sys-sec}^{Warrants}$ during the Request Warrant task.

Now, the simulation of $F_{AD}.\text{RequestDecryption}$ needs to be handled as well. Since $\Sigma$ is EUF-CMA-secure, $S^9$ aborts in the same cases as the real $F_{AD}$ would. $S^9$ has access to the list $L_{sys-sec}^{Warrants}$ that is filled during Request Warrant (which needs to be called before this task). Using $L_{sys-sec}^{Warrants}$, $S^9$ can call $F_{AS}$ with the correct inputs and gets the corresponding outputs. Next, the secrets need to be prepared for retrieval. Here we again need to distinguish between secrets of honest and corrupted users. For corrupted users, we already know the correct ciphertext (from the input of this task) and we can just append the ciphertext to $L_{sys-sec}^{Requests}$ of $F_{AD}$ like an honest $F_{AD}$. The remainder of the secret retrieval (for corrupted users) is then handled by the honest simulation of $F_{AD}$. For honest users, we retrieve the entry $(uid, vper, sec1)$ from $L_{sys-sec}^{HonestSecrets}$ and use the output of $F_{AS}^9$ to calculate the secret $sec2$ (in the clear) that needs to be retrieved. We then store this secret $sec2$ directly to a special list $L_{sys-sec}^{ReqNotReady}$. Then the $F_{AD}.\text{Handover}$ task has to be adapted as well. It now additionally moves all entries from $L_{sys-sec}^{ReqNotReady}$ to $L_{sys-sec}^{Pending}$. This ensures that now for honest users the correct secrets can be retrieved as well. Note that for honest users, the ciphertext encrypting the user's secret in Store Secret is not used during retrieval anymore.

Game 10. In this game, $F_{AS}^{10}$ additionally handles the tasks Get Secrets and Get Statistics the same way $F_{AS}$ does, and $S^{10}$ handles this task the same way $S$ does.

Proof Sketch: Game 9 $\approx$ Game 10.

Get Secrets: For honest LE, $Z$'s view remains identical, since $S^{10}$ only needs to leak (RETRIEVE) to $A$ to emulate the $F_{AD}.\text{RetrieveSecret}$ task. For corrupted
LE, only the simulation of $F_{AS}$ RetrieveSecret needs to be handled, which is already simulated like the final simulator does and returns the correct value.

Get Statistics: For an honest party sending that message, $S^{10}$ only needs to send (GetStatistics) to $A$. For a corrupted party, this game is identical to Game 9 since this task is already handled by the simulation of $F_{AD}$.

Game 11. In this game, $F^{11}_{AS}$ additionally handles the Store Secret task the same way $F_{AS}$ does, and $S^{11}$ handles this task the same way $S$ does.

Proof Sketch: Game 10 $\approx$ Game 11.

For corrupted $U$: $S^{11}$ can obtain the required inputs to call $F^{11}_{AS}$ from the message $(uid, \sigma_{reg}, comuid, decomuid, comsec_2, compr_vper, decomvper)$ from $U$ by extracting $vper$ from $com_{vper}$. Since COM is extractable, this is possible with overwhelming probability and allows $S^{11}$ to call $F^{11}_{AS}$ with the correct input. Using the output $(SecretStoredProcedure, secret)$ of $F^{11}_{AS}$ and by extracting $sec_2$ from $com_{sec_2}$ (which is again possible with overwhelming probability since COM is extractable), $S^{11}$ can calculate the correct $sec_1$ and then honestly execute the remainder of the protocol for $SO$.

For honest $U$: nothing to do in this case.

Game 12. In this game, $F^{12}_{AS}$ additionally handles the Prove and Verify tasks the same way $F_{AS}$ does, and $S^{12}$ handles this task the same way $S$ does.

Proof Sketch: Game 11 $\approx$ Game 12.

Since NIZK is a simulation extractable zero-knowledge proof system, $Z$ can not distinguish between the outputs it gets from $F^{12}_{AS}$ and the outputs it gets from $F_{AS}$.

Note that $S^{12}$ equals the final simulator $S$. Since we showed that Game $i$ is indistinguishable from Game $i + 1$ for $i \in \{0, \ldots, 11\}$, it follows that Game 0 (the real protocol) is indistinguishable from Game 12 (ideal execution) and thus Theorem 5 follows.

$\square$

E.2 User Security

In the User Security case, we handle the following corruptions:

- $U$ can be statically corrupted, so some $U$ are corrupted and some $U$ are honest.
- $SO$ is corrupted here (static corruption)
- LE can be corrupted here (static corruption)
- J and AU are always honest

We now first describe a simulator $S$ for those corruptions and then proceed to prove the security. The simulator uses the following writing conventions:

- If $F_{AS}$ wants to deliver outputs to honest parties, delay them until the simulator explicitly allows them.
- Deny any other calls until the System Init task has been successfully completed
- Hybrid functionalities $F_{CRS}, F_{BB}$ are simulated honestly. We assume in the following that whenever calls to those functionalities are made, $S$ simulates them honestly.

102
Hybrid functionality $\mathcal{F}_{\text{AD}}$ is mostly simulated honestly, except the tasks Request Decryption and Handover (for details see the paragraph "simulation of $\mathcal{F}_{\text{AD}}"."

The simulator $\mathcal{S}$ for System Security is defined as follows.

<table>
<thead>
<tr>
<th>Simulator for User Security</th>
</tr>
</thead>
</table>

**State of the Simulator**

- **Setup:**
  - TPKE keypair: $(pk, sk)$
  - Verification key of SO: $\mathfrak{v}_S$
  - J Signing keypair: $(\mathfrak{v}_j, sk_j)$
  - Common reference strings $(\text{crs}^\text{AS}, \text{crs}^\text{eq})$
  - Trapdoor $\text{td}^\text{NIZK}$ for simulating and extracting proofs
  - Trapdoor $\text{td}^\text{eq}$ for equivocating commitments
- Stored in User Registration:
  - $\mathcal{L}_{\text{user-sec}}$ Registered with $(\mathit{pid}_U, \mathit{uid}, \mathit{com}_{\mathit{uid}}, \mathit{decom}_{\mathit{uid}}, \mathit{\sigma}_{\mathit{reg}}, \mathfrak{v}_U, sk_U)$ entries. Note that only honest users are in that list.
- Stored in Request Warrant:
  - $(W, b)$ — list of already requested preliminary warrants
  - $(\tilde{W}, \sigma_{\tilde{W}})$ — list of stored enhanced warrants
- Stored for simulation of $\mathcal{F}_{\text{AD}}$:
  - $\mathcal{L}_\text{user-sec}^\text{Secrets}$ with $(\mathit{uid}, \mathit{vper}, \mathit{sec}_1, \mathit{ct})$ entries. Entries are stored in $\text{AS.StoreSecret}$ (for honest $U$) and used in $\mathcal{F}_{\text{AD}}$.RequestSecret.
  - Only for corrupted LE: The list $\mathcal{L}_\text{user-sec}^\text{ReqNotReady}$ temporarily stores $(\mathit{ct}_i, \mathit{sec}_{2,i})$ entries (for honest $U$). Filled during the simulation of $\mathcal{F}_{\text{AD}}$.RequestDecryption and cleared during the simulation of $\mathcal{F}_{\text{AD}}$.Handover.
  - Everything else that is stored inside $\mathcal{F}_{\text{AD}}$
- List for simulation of $\mathcal{F}_{\text{BB}}$:
  - $\mathcal{L}_{\text{BB}}$ (initially empty). Entries are of the form $(\mathit{pid}_U, \mathit{uid}, \mathfrak{v}_U)$

**System Setup ($\mathcal{F}_{\text{CRS}}$):**

- $(\text{crs}^\text{AS}, \text{td}^\text{NIZK}) \leftarrow \text{NIZK.Setup}(1^\lambda)$
- $(\text{crs}^\text{eq}, \text{td}^\text{eq}) \leftarrow \text{COM.Setup}^\text{Equiv}(1^\lambda)$
- Store all values

**System Setup ($\mathcal{F}_{\text{BB}}$):**

- Create empty list $\mathcal{L}_{\text{BB}}$

**System Setup ($\mathcal{F}_{\text{AD}}$):**

- See simulation of Init-Task
System Setup ($F_{AS}$):

– Choose the same system parameters for $F_{AS}$ as $H_{AS}$

Simulation of $F_{CRS}$

Corrupted $P$ can issue calls to $F_{CRS}$ whenever they want, not just at the intended points in the protocol. We specify here once how all calls from $P$ to $F_{CRS}$ to all tasks are handled and ignore calls to $F_{CRS}$ for most tasks in the following.

$S$ answers all calls honestly.

Simulation of $F_{BB}$

Corrupted $P$ can issue calls to $F_{BB}$ whenever they want, not just at the intended points in the protocol. We specify here once how all calls from $P$ to $F_{BB}$ to all tasks are handled and ignore calls to $F_{BB}$ in the following.

$S$ answers all calls honestly.

Simulation of $F_{AD}$

Corrupted $P$ can issue calls to $F_{AD}$ whenever they want, not just at the intended points in the protocol. We specify here once how all calls from $P$ to $F_{AD}$ to all tasks are handled and ignore calls to $F_{AD}$ for most tasks in the following.

$S$ simulates $F_{AD}$ honestly (including taking track of all internal lists and leaking information to the adversary), except for the following tasks:

– Request Decryption

1. Receive $\left(\text{REQUEST, } \bar{W}, \sigma_{\bar{W}}\right)$ as input from LE
2. Check Warrant Signature: $b \leftarrow \Sigma.Vfy(vk_j, \bar{W}, \sigma_{\bar{W}})$
3. If $b = 0$, abort (Warrant not valid)
4. Build preliminary warrant $W$ out of $\bar{W}$ by deleting $ct_j$ from each $\bar{W}_j$
5. Call $F_{AS}$ in the name of LE with input $(\text{GETSECRETS, } W)$
6. If $F_{AS}$ aborts, $S$ lets $F_{AD}$ abort as well (Warrant either not requested or not granted)
7. Receive output $(\text{GOTSECRETS, } (\text{secret}_1, ..., \text{secret}_v))$ from $F_{AS}$ to LE
8. Parse warrant: $\left(\bar{W}_1, ..., \bar{W}_v\right) \leftarrow \bar{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \bar{W}_i$
9. For $i$ from 1 to $v$:
   (a) Check if there is an entry $(uid_i, vper_i, ..., ct_i)$ in $L^{user-sec}_{\text{Secrets}}$. If yes (case: user is honest) go to step 9b, if not (case: user is corrupted) go to step 9f.
(b) Retrieve the entry \((uid, vper, ct)\) from \(L_{\text{user-sec}}\).
(c) Compute \(sec_2, i\) such that \(sec_1, i \oplus sec_2, i = secret_i\).
(d) Add \((ct, sec_2, i)\) to \(L_{\text{ReqNotReady}}\).
(e) Then go to step 10.
(f) Append \(ct\) to \(L_{\text{Requests}}\).
(g) Then go to step 10.

10. Send \((\text{REQUEST}, pid_{LE}, f_t(W), |\overline{W}|, v)\) to \(A\).
11. When \(A\) allows to deliver output, store \(\overline{W}\) in list of warrants \(L_{\text{Warrants}}\) of \(F_{\text{AD}}\).
12. Give output \((\text{REQUEST})\) to \(LE\).

**Handover**

Simulate honestly, but additionally execute the following steps at the end:
- For each item \((ct, sec_2, i)\) in \(L_{\text{user-sec}}\), append \((ct, sec_2, i)\) to \(L_{\text{Pending}}\).
- Clear \(L_{\text{user-sec}}\).

**System Init (corrupted SO, honest J, honest AU):**

1. Upon receiving \((\text{INIT}, SO, \nu_{SO})\) from SO to \(F_{\text{AD}}\):
   (a) Create signing keypair for J: \((\nu_{J}, sk_{J}) \leftarrow \Sigma.\text{Gen}(1^\lambda)\) and store it.
   (b) Store \(\nu_{SO}\).
   (c) Honestly simulate the Init-Task of \(F_{\text{AD}}\) with the following inputs:
      \(J \rightarrow F_{\text{AD}}\): \((\text{INIT}, J, \nu_{J})\)
      \(AU \rightarrow F_{\text{AD}}\): \((\text{INIT}, AU)\)
      \(SO \rightarrow F_{\text{AD}}\): \((\text{INIT}, SO, \nu_{SO})\)
   (d) During simulation of the Init-Task, store all generated variables and lists and leak all necessary information to the adversary.
   (e) Call \(F_{\text{AS}}\) in the name of SO with input \((\text{INIT}, SO)\).
2. Upon receiving output \((\text{INITFinished})\) from \(F_{\text{AS}}\) to SO:
   (a) Report message \((\text{INITFinished})\) from \(F_{\text{AD}}\) to SO.
   (b) Send \((\text{GETSKEYS})\) and \((\text{GETPK})\) to the adversary.
   (c) Allow \(F_{\text{AS}}\) to deliver output to all honest parties.

**Party Init (honest P):**

1. Upon receiving the leak \((\text{PINIT})\) from \(F_{\text{AS}}\):
   (a) Send \((\text{GETSKEYS})\) and \((\text{GETPK})\) to the adversary.
   (b) Allow \(F_{\text{AS}}\) to deliver output to all parties.
Party Init (corrupted P)
This is handled by simulation of \( F_{AD} \).

User Registration (honest U, corrupted SO):
1. Upon receiving the leak \((\text{Register}, \text{pid}, \text{uid})\) from \( F_{AS} \)
   (a) Report message \((\text{uid})\) from U to SO
2. Upon receiving the message \((\text{uid}_{\text{OK}})\) from SO to U
   (a) Call \( F_{AS} \) in the name of SO with input \((\text{Register}, \text{uid})\) and send \((\text{OK})\) to \( F_{AS} \)
   (b) If \( F_{AS} \) aborts, let S abort as well (User already registered or UID already taken)
   (c) Generate keypair for U: \((\text{vk}_U, \text{sk}_U) \leftarrow \Sigma.\text{Gen}(1^\lambda)\)
   (d) Honestly simulate U’s Register and Retrieve calls to \( F_{BB} \) with the correct inputs
   (e) When \( F_{AS} \) wants to deliver output to U, delay it and report message \((\text{OK})\) from U to SO
3. Upon receiving the message \((\sigma_{\text{reg}}, \text{com}_{\text{uid}}, \text{decom}_{\text{uid}})\) from SO to U
   (a) Send \((\text{GetSKeys})\) and \((\text{GetPK})\) to the adversary
   (b) Verify Commitment: If \( \Sigma.\text{Open}(\text{crs}_{\text{com}}, \text{com}_{\text{uid}}, \text{decom}_{\text{uid}}, \text{uid}) = 0 \), abort.
   (c) Verify Signature: If \( \Sigma.\text{Vfy}(\text{vk}_SO, \sigma_{\text{reg}}, \text{com}_{\text{uid}}) = 0 \), abort.
   (d) Store \((\text{pid}_U, \text{uid}, \text{com}_{\text{uid}}, \text{decom}_{\text{uid}}, \sigma_{\text{reg}}, \text{vk}_U, \text{sk}_U)\) in \( L_{\text{user-sec Registered}} \)
   (e) Allow \( F_{AS} \) to deliver output to U

User Registration (corrupted U, corrupted SO):
This is handled by simulation of \( F_{AD} \).

Store Secret (honest U, corrupted SO):
1. Upon receiving the leak \((\text{STORESECRET}, \text{vper})\) from \( F_{AS} \)
   (a) Call \( F_{AS} \) in the name of SO with input \((\text{STORESECRET}, \text{vper})\)
   (b) If \( F_{AS} \) aborts, abort as well (Either wrong validity period or user is not registered yet or user has already stored a secret for the current validity period)
   (c) Retrieve output \((\text{SECRETSTORED}, \text{uid})\) from \( F_{AS} \) to SO, but delay output to U
   (d) For \text{uid}, retrieve entry \((\text{pid}_U, \text{uid}, \text{com}_{\text{uid}}, \text{decom}_{\text{uid}}, \sigma_{\text{reg}}, \text{vk}_U, \text{sk}_U)\) from \( L_{\text{user-sec Registered}} \)
   (e) Create fake commitment on \( sec_2 \): \((\text{cem}_{sec_2}, \text{eq}_{sec_2}) \leftarrow \text{COM.SimCom}(td_{\text{com}})\)
   (f) Create real commitment on \( vper \): \((\text{com}_{vper}, \text{decom}_{vper}) \leftarrow \text{COM.Com}(\text{crs}_{\text{com}}, vper)\)
   (g) Report message \((\text{uid}, \sigma_{\text{reg}}, \text{com}_{\text{uid}}, \text{decom}_{\text{uid}}, \text{cem}_{sec_2}, \text{com}_{vper}, \text{decom}_{vper})\) from U to SO
2. Upon receiving the message \((sec_1)\) from SO to U
   (a) Create fake commitment on secret: \((com_{sec}, eq_s) \leftarrow COM.\textit{SimCom}(td^e_{com})\)
   (b) Fake ciphertext \(ct \leftarrow \text{TPKE.Enc}(pk, 0)\)
   (c) Honestly commit and sign \(sec_1\): \((com_{sec_2}, \text{decom}_{sec_1}) \leftarrow \text{COM.Com}(crs_{com_\text{sec}_1})\) and \(\sigma_{sec_1} \leftarrow \Sigma.\textit{Sign}(sk_U, com_{sec_1})\)
   (d) Create signature \(\sigma_U \leftarrow \Sigma.\textit{Sign}(sk_U, (ct, uid, vper))\)
   (e) \(\text{stmt}_{ss} := (sec_1, com_{sec}, \text{decom}_{sec_2}, pk, ct, crs_{com})\)
   (f) Fake proof: \(\overline{\pi}_{ss} \leftarrow \text{NIZK.Sim}(td_{\text{NIZK}}, \text{stmt}_{ss}, \mathcal{R}^*_{ss})\)
   (g) Report message \((ct, com_{sec}, \overline{\pi}_{ss}, \sigma_U, \sigma_{sec_2}, \text{decom}_{sec_1})\) from U to SO

3. Upon receiving the message \((\sigma_{ss})\) from SO to U
   (a) Verify signature: If \(\Sigma.\textit{Vfy}(vk_{SO}, (com_{uid}, com_{sec}, com_{vper}), \sigma_{ss}) = 0\), let \(S\) abort in the name of U
   (b) Store \((uid, vper, sec_1, ct)\) in \(L^{\text{ver-sc}}\)
   (c) Allow \(F_{AS}\) to deliver output to U

\textbf{Store Secret (corrupted U, corrupted SO):}
Nothing to do, since \(\Pi_{AS}.\text{StoreSecret}\) has no calls to subfunctionalities and all involved parties are corrupted.

\textbf{Request Warrant (corrupted LE, honest J, corrupted SO):}
1. Upon receiving the message \((W)\) from LE to J
   (a) Call \(F_{AS}\) in the name of LE with input \((\text{REQUEST WARRANT}, W)\)
   (b) Allow \(F_{AS}\) to deliver output to J
2. Upon receiving the message \((\text{REQUEST WARRANT}, W, b)\) from \(F_{AS}\):
   (a) If \(b = 0\), set \(\sigma_W = \bot\). Else, set \(\sigma_W = \Sigma.\textit{Sign}(sk_J, W)\)
   (b) Store \((W, b)\) in list of already requested preliminary warrants
   (c) Report message \((b, \sigma_W)\) from J to LE
3. Upon receiving the message \((\overline{W}, \{\sigma_{U,i}, sec_{1,i}, com_{sec_{1,i}}, \text{decom}_{sec_{1,i}}, \sigma_{sec_{1,i}}\}_{i \in \{1, \ldots, v\}})\)
   from LE to J
   (a) Build preliminary warrant \(W''\) out of \(\overline{W}\) by deleting \(ct_i\) from each \(\overline{W}_i\)
   (b) If there is no entry \((W'', 1)\) in the list of already processed preliminary warrants, abort in the name of J
   (c) Check if \((\overline{W}, \cdot)\) is already in list of stored enhanced warrants. If yes, abort in the name of J (Warrant already processed).
   (d) Parse enhanced warrant: \((\overline{W}_1, \ldots, \overline{W}_v) \leftarrow \overline{W}\) and \((uid_i, vper_i, meta_i, ct_i) \leftarrow \overline{W}_i\)
   (e) For \(i\) from 1 to \(v\):
      - For \(uid_i\), honestly simulate a call to \(F_{BB}\) in the name of J with \((\text{RETRIEVE}, uid_i)\) and get answer \((\text{RETRIEVE}, pid_{U,i}, uid_i, vk_{U,i})\)
      - Check signatures: If \(\Sigma.\textit{Vfy}(vk_{U,i}, (ct_i, uid_i, vper_i), \sigma_{U,i}) = 0\) or \(\Sigma.\textit{Vfy}(\nu_{U,i}, com_{sec_{1,i}}, \sigma_{sec_{1,i}}) = 0\), abort in the name of J
      - Check commitment: If \(\text{COM.Open}(crs_{com_1}, com_{sec_{1,i}}, \text{decom}_{sec_{1,i}}, sec_{1,i}) = 0\), abort in the name of J
(f) Sign enhanced warrant: $\sigma_{\tilde{W}} = \Sigma \cdot \text{Sign}(sk_J, \tilde{W})$

(g) Store ($\tilde{W}, \sigma_{\tilde{W}}$) in list of stored enhanced warrants

(h) Report message ($\sigma_{\tilde{W}}$) from J to LE

4. When the adversary allows $F_{AD}$ to deliver output in $F_{AD}$.RequestDecryption

   (a) Finish simulation of $F_{AD}$.RequestDecryption (like described above)

   (b) Send (OK) to $F_{AS}$

   (c) Receive output (WarrantAnswer, b) from $F_{AS}$

Request Warrant (honest LE, honest J, corrupted SO):

1. Upon $F_{AS}$ asking to deliver output to J, allow the output

2. Upon receiving the leak ($\text{REQUESTWARRANT}, W, b$) from $F_{AS}$

   (a) If $b = 0$, set $\sigma_W = \bot$. Else, set $\sigma_W = \Sigma \cdot \text{Sign}(sk_J, W)$

   (b) Store ($W, \sigma_W$) in list of already requested preliminary warrants

   (c) If $b = 0$, directly go to step 4b

   (d) Send (GETSKEYS) to the adversary

   (e) Report message ($W, \sigma_W$) from LE to SO

3. Upon receiving the message ($\tilde{W}, \{\sigma_{U,i}, \text{sec}_{\text{com}}, \text{com}_{\text{sec}}, i, \text{decom}_{\text{sec}}, i, \sigma_{\text{sec}}, i\}_{i \in \{1, \ldots, v\}}$)

   from SO to LE

   (a) Parse enhanced warrant: ($\tilde{W}_1, \ldots, \tilde{W}_v$) $\leftarrow \tilde{W}$ and ($uid_i, vper_i, meta_i, 
       ct_i$) $\leftarrow \tilde{W}_i$

   (b) Build preliminary warrant $W'$ out of $\tilde{W}$ by deleting $ct_i$ from each $\tilde{W}_i$

   (c) If $W' \neq W$, abort in the name of LE. (SO sent wrong $\tilde{W}$)

   (d) For $i$ from 1 to $v$:

      i. For $uid_i$, honestly simulate a call to $F_{BB}$ in the name of LE with
         ($\text{RETRIEVE}, uid_i$) and get answer ($\text{RETRIEVE}, pid_{U,i}, uid_i, vk_{U,i}$)

      ii. Check signatures: If $\Sigma \cdot \text{Vfy}(vk_{U,i}, (ct_i, uid_i, vper_i), \sigma_{U,i}) = 0$ or $\Sigma \cdot \text{Vfy}(vk_{U,i},
         \text{com}_{\text{sec}}, i, \text{sec}_{i, i}) = 0$, abort in the name of LE

      iii. Check commitment: If $\text{COM.Open}(\text{crs}_{\text{com}}, \text{com}_{\text{sec}}, i, \text{decom}_{\text{sec}}, i, \text{sec}_{i, i}) = 0$, abort in the name of LE

      iv. For $uid_i$, honestly simulate a call to $F_{BB}$ in the name of J with
         ($\text{RETRIEVE}, uid_i$)

   (e) Sign enhanced warrant: $\sigma_{\tilde{W}} = \Sigma \cdot \text{Sign}(sk_J, \tilde{W})$

   (f) Store ($\tilde{W}, \sigma_{\tilde{W}}$) in list of stored enhanced warrants

   (g) Simulate call to $F_{AD}$.RequestDecryption by sending ($\text{REQUEST}, pid_{LE},
       f_i(W), |\tilde{W}|, v$) to $A$

4. When the adversary allows $F_{AD}$ to deliver output in $F_{AD}$.RequestDecryption

   (a) Store $\tilde{W}$ in list of warrants $L_{\text{Warrants}}$ of $F_{AD}$

   (b) Send (OK) to $F_{AS}$

   (c) Allow $F_{AS}$ to deliver output to LE
Get Secrets (honest LE):
1. Upon receiving the leak (GetSecrets) from \( F_{AS} \)
   (a) Leak (Retrieve) to \( A \)
   (b) Allow \( F_{AS} \) to deliver output

Get Secrets (corrupted LE):
This is handled by simulation of \( F_{AD} \).

Get Statistics (honest P):
1. Upon receiving the leak (GetStatistics) from \( F_{AS} \)
   (a) Leak (GetStatistics) to \( A \) and wait for okay from \( A \)
   (b) Allow \( F_{AS} \) to deliver output

Get Statistics (corrupted P):
This is handled by simulation of \( F_{AD} \).

Audit (honest AU):
1. Upon receiving the leak (AuditRequest) from \( F_{AS} \)
   (a) Leak (AuditRequest) to \( A \) and wait for okay from \( A \)
   (b) Allow \( F_{AS} \) to deliver output

Prove (corrupted U)
Nothing to do since this is a completely local task.

Prove (honest U)
1. Upon receiving the message (Prove, stmt\(_R\)) from \( F_{AS} \)
   (a) Simulate proof: \( \pi \leftarrow \text{NIZK.Sim}(td_{NIZK}, stmt_{R}, R_{AS}^{AS}) \)
   (b) Send (Proof, \( \pi \)) to \( F_{AS} \)

Verify (corrupted P)
Nothing to do since this is a completely local task.

Verify (honest P)
1. Upon receiving the message (Verify, stmt\(_R\), \( \pi \)) from \( F_{AS} \)
   (a) Extract Witness: \( \text{wit}_R \leftarrow \text{NIZK.Ext}(td_{NIZK}, stmt_{R}, \pi, R_{AS}^{AS}) \)
   (b) Send (witness, \( \text{wit}_R \)) to \( F_{AS} \)

We now prove the following theorem:

**Theorem 6 (User Security).** \( \Pi_{AS} \) UC-realizes \( F_{AS} \) in the \( \{F_{AD}, F_{CRS}, F_{BB}, G_{CLOCK}\} \)-hybrid model under the assumptions that

- \( \text{COM} \) is a (computationally) hiding, (statistically) binding and equivocal commitment scheme
- \( \Sigma \) is a EUF-CMA secure signature scheme
- **NIZK** is a straight-line simulation-extractable non-interactive zero-knowledge proof system
- **TPKE** is IND-CPA secure

against all PPT-adversaries \( A \) who statically corrupts one of the following sets

1. SO and a subset of the users
2. SO, LE and a subset of the users

To prove Theorem 6, we proceed in a series of games, where \( F_{AS}^{i+1} \) behaves the same as \( F_{AS}^i \) except for the stated changes and \( S^{i+1} \) behaves the same as \( S^i \) except for the stated changes.

**Game 0.** Game 0 is equivalent to the real experiment. That is, 
\[
H_0 := \text{Exp}_{\overline{\text{AD}}}^{\overline{\text{AS}}, \overline{\text{CRS}}, \overline{\text{BB}}, \overline{\text{CLOCK}}, S^0, Z}
\]
with \( S^0 \) being the dummy adversary. This means that all parties execute the real protocol.

**Game 1.** In this game, the simulator \( S^1 \) takes control over \( F_{\text{CRS}}, F_{\text{BB}}, F_{\text{AS}} \), but executes them honestly. Additionally, \( F_{\text{AS}}^1 \) is introduced, and all honest parties are replaced by dummy parties that forward their inputs to \( F_{\text{AS}}^1 \) and when receiving output from \( F_{\text{AS}}^1 \) forward it to the environment. \( F_{\text{AS}}^1 \) upon receiving any message from a dummy party forwards it to the simulator \( S^1 \) and asks it for output. \( S^1 \) executes the real protocol for all honest parties (using the inputs received from \( F_{\text{AS}}^1 \)) and instructs \( F_{\text{AS}}^1 \) to deliver the resulting outputs. Note that \( S^1 \) simulates \( F_{\text{BB}} \) and \( F_{\text{CRS}} \) like \( S \) does.

Proof Sketch: Game 0 \( \approx \) Game 1.

**Game 2.** In this game, \( S^2 \) generates the common reference string for the commitment scheme with an equivocation trapdoor \( td_{\text{com}}^{\text{eq}} \) as \((\text{crs}_{\text{com}}, td_{\text{com}}^{\text{eq}}) \leftarrow \text{COM.Setup}^{\text{Ext}}(1^\lambda)\). \( S^2 \) also generates the common reference string for the non-interactive proof system with a trapdoor for simulating and extracting proofs, i.e., \((\text{crs}_{\text{zk}}^A, td_{\text{NIZK}}) \leftarrow \text{NIZK.Setup}(1^\lambda)\). Thus, \( S^2 \) simulates \( F_{\text{CRS}} \) the same way \( S \) does.

Proof Sketch: Game 1 \( \approx \) Game 2.

Indistinguishability follows from the equivocality of \( \text{COM} \) and the simulation-extractability of \( \text{NIZK} \).

**Game 3.** During setup of \( F_{\text{AD}} \), \( S^3 \) is asked for the keypairs for J and AU. \( S^3 \) generates the keypairs honestly and stores them.

Proof Sketch: Game 2 \( \approx \) Game 3.

\( S^3 \) runs the same key generation algorithm as the real protocol does for honest parties, therefore \( Z \)’s view remains unchanged.

**Game 4.** \( S^4 \) keeps track of additional information. For J it stores a list of \((W, b)\) elements and a list of \((W, \sigma_W)\) elements in the Request Warrant task. For each honest U, it stores \((\text{pid}_U, \text{uid}_U, \text{com}_uid, \text{decom}_uid, \sigma_{\text{reg}}, \text{vk}_U, \text{sk}_U)\) in the
list \text{User−sec\_Registered} at the end of the User Registration task and \((uid, vper, secI, ct)\) in
the list \text{User−sec\_Secrets} at the end of the Store Secret task.

Proof Sketch: Game 3 \simeq Game 4.

\(S^4\) only keeps track of more information (that the honest parties acquire during
the real protocol), nothing to distinguish for \(Z\).

Game 5. In this game, \(F^5_{AS}\) additionally handles the tasks System Init and
Party Init the same way \(F_{AS}\) does, and \(S^5\) handles these task the same way \(S\)
does.

Proof Sketch: Game 4 \simeq Game 5.

\(S^5\) simulates the System Init task honestly, therefore \(Z\)'s view remains un-
changed.
Party Init for honest Parties: \(Z\)'s view remains identical, since \(S^5\) only needs to
leak \((\text{GETSKEYS})\) and \((\text{GETPK})\) to \(A\) to emulate the call to \(F_{AD}\).
Party Init for corrupted Parties: Only the simulation of hybrid functionalities
needs to be handled, which are already simulated honestly.

Game 6. In this game, \(F^6_{AS}\) additionally handles the Audit task the same
way \(F_{AS}\) does, and \(S^6\) handles this task the same way \(S\) does. Note that since
the auditor \(AU\) is assumed to be honest, simulation only requires a simulated
\((\text{AUDITREQUEST})\) to \(F_{BCRA}\).

Proof Sketch: Game 5 \simeq Game 6.

Since \(AU\) is honest, these games are identical.

Game 7. In this game, \(F^7_{AS}\) additionally handles the User Registration task
the same way \(F_{AS}\) does, and \(S^7\) handles this task the same way \(S\) does.

Proof Sketch: Game 6 \simeq Game 7.

For honest \(U\), \(S^7\) learns the \(uid\) the environment has chosen for the user and
can thus invoke \(F_{AS}\) with the correct input. Apart from that, \(S^7\) plays the role
of the user honestly. If the signature scheme \(\Sigma\) is EUF-CMA-secure and the
commitment scheme \(COM\) is binding, \(S^7\) aborts in the same cases as \(U\) would
do in \(\Pi_{AS}\).

For corrupted \(U\), this game is identical to Game 6 since this task is already
handled by the simulation of \(F_{AD}\).

Game 8. If \(LE\) is honest, \(F^8_{AS}\) additionally handles the Request Warrant task
the same way \(F_{AS}\) does for honest \(LE\), and \(S^8\) handles this task the same way
\(S\) does for honest \(LE\). Note that if \(LE\) is corrupted, this game changes nothing.

Proof Sketch: Game 7 \simeq Game 8.

\(S^8\) essentially simulates the real protocol for honest parties (only the call to \(F_{AD}\)
is simulated instead of executed honestly), therefore \(Z\) can not distinguish those
games. The simulated call to \(F_{AD}\) is indistinguishable since \(S^8\) leaks the correct
information to the adversary. Note that if the signature scheme \(\Sigma\) is EUF-CMA-
secure and the commitment scheme \(COM\) is binding, \(S^8\) aborts in the same cases
as \(LE\) or \(J\) would do in \(\Pi_{AS}\).

Game 9. If \(LE\) is corrupted, \(F^9_{AS}\) additionally handles the Request Warrant task
the same way \(F_{AS}\) does for corrupted \(LE\), and \(S^9\) handles this task the same
way \(S\) does for corrupted \(LE\). Additionally, \(S^9\) simulates \(F_{AD}\) as \(S\) does for the
Request Decryption and Handover tasks in \(F_{AD}\). Note that if \(LE\) is honest, this
game changes nothing. Also note that now $S^9$ simulates $F_{AD}$ as $S$ does (for every task and every corruption scenario).

**Proof Sketch: Game 8 \(\approx\) Game 9.**

In Request Warrant, $S^9$ receives the necessary information from LE to call $F^9_{AS}$ with the correct input. After receiving output from $F^9_{AS}$, $S^9$ essentially executes the real protocol for $J$ and $SO$. Now, the simulation of $F_{AD}$.RequestDecryption needs to be handled as well. Since $\Sigma$ is EUF-CMA-secure, $S^9$ aborts in the same cases as the real $F_{AD}$ would. $S^9$ can call $F^9_{AS}$ with the correct inputs and gets the corresponding outputs. Next, the secrets need to be prepared for retrieval. Here we need to distinguish between secrets of honest and corrupted users. For corrupted users, we already know the correct ciphertext (from the input of this task) and we can just append the ciphertext to $L_{Requests}$ of $F_{AD}$ like an honest $F_{AD}$. The remainder of the secret retrieval (for corrupted users) is then handled by the honest simulation of $F_{AD}$. For honest users, we retrieve the entry $(uid, vper, sec_1, ct)$ from $L_{Secrets}$ and use the output of $F^9_{AS}$ to calculate the secret $sec_2$ (in the clear) that needs to be retrieved. We then store this secret directly to a special list $L_{user-sec ReqNotReady}$. Then, the $F_{AD}$.Handover task has to be adapted as well. It now additionally moves all entries from $L_{user-sec ReqNotReady}$ to $L_{Pending Requests}$. This ensures that now for honest users the correct secrets can be retrieved as well. Note that for honest users, the ciphertext encrypting the user's secret in Store Secret is not used during retrieval anymore.

**Game 10.** In this game, $F^{10}_{AS}$ additionally handles the tasks Get Secrets and Get Statistics the same way $F_{AS}$ does, and $S^{10}$ handles these tasks the same way $S$ does.

**Proof Sketch: Game 9 \(\approx\) Game 10.**

For a corrupted party calling those tasks, this game is identical to Game 9 since those tasks are already handled by the simulation of $F_{AD}$. For honest parties, $S^{10}$ only needs to leak the same messages to the adversary as $F_{AD}$ would do. All the information $S^{10}$ needs for that are provided by leaks from $F_{AS}$.

**Game 11.** In this game, $F^{11}_{AS}$ additionally handles the Store Secret task the same way $F_{AS}$ does, and $S^{11}$ handles this task the same way $S$ does.

**Proof Sketch: Game 10 \(\approx\) Game 11.**

For corrupted U: nothing to do in this case. For honest U: With the message $(STORESECRET, vper)$ $S^{11}$ is able to invoke $F^{11}_{AS}$ with the correct inputs for SO. Instead of computing real commitments on $sec_2$ and secret, $S^{11}$ computes equivocal commitments on them. Since COM is hiding and equivocable, $Z$ can not differentiate between the real and the simulated commitments. Also, $S^{11}$ simulates the proof $\pi_{ss}$ instead of computing a real proof for honest users. Due to the zero-knowledge property of NIZK, $Z$ can not detect this. $S^{11}$ also creates a fake ciphertext (by encrypting 0) which is not detected by $Z$ since TPKE is IND-CPA-secure and we do not use this ciphertext during retrieval anymore. Apart from these changes, $S^{11}$ simulates the role of the user honestly.

**Game 12.** In this game, $F^{12}_{AS}$ additionally handles the Prove and Verify tasks the same way $F_{AS}$ does, and $S^{12}$ handles this task the same way $S$ does.
Proof Sketch: Game 11 \(\approx\) Game 12.

For corrupted parties invoking this task, there is nothing to do for \(S^{12}\), since both tasks are local tasks. So let us assume the calling parties are honest in the following. Since \(\text{NIZK}\) is a simulation extractable zero-knowledge proof system, \(Z\) can not distinguish between the outputs it gets from \(F_{AS}^{12}\) and the outputs it gets from \(F_{AS}\). Note that in the \(F_{AS}\cdot\text{Verify}\) task, it may be the case that no corresponding entry exists in \(L_{\text{Secrets}}\). But since SO is corrupted, the additional check in \(F_{AS}\cdot\text{Verify}\) that involves \(L_{\text{Secrets}}\) is skipped and the correct output is given.

Note that \(S^{12}\) equals the final simulator \(S\). Since we showed that Game \(i\) is indistinguishable from Game \(i + 1\) for \(i \in \{0, \ldots, 11\}\), it follows that Game 0 (the real protocol) is indistinguishable from Game 12 (ideal execution) and thus Theorem 6 follows.

Proof (Proof of Theorem 1). The theorem is directly implied by Theorem 5 and Theorem 6.

\[\square\]

\[\square\]

F Security Proof: \(\Pi_{\text{AD}}\) UC/YOSO-realizes \(F_{\text{AD}}\)

Here we prove that the protocol \(\Pi_{\text{AD}}\) from Appendix B.4 UC-realizes the functionality \(F_{\text{AD}}\) from Appendix B.3 in the YOSO model. In particular, we prove the following theorem:

**Theorem 2.** If \(\text{NIZK}\) is a straight-line simulation-extractable non-interactive zero-knowledge proof system, \(\Sigma\) is an EUF-CMA secure signature scheme, the PKE scheme used by LE and AU is an IND-CPA secure public key encryption scheme, the PKE scheme that is a parameter of \(F_{\text{BCRA}}\) is a RIND-SO secure public key encryption scheme, and TPKE is an IND-CPA secure, randomizable and binding \((t,n)\)-threshold PKE, then \(\Pi_{\text{AD}}\) UC-realizes \(F_{\text{AD}}\) in the \(\{F_{\text{BCRA}},F_{\text{CRS}},G_{\text{CLOCK}}\}\)-hybrid model with respect to adversaries \(A\) that may statically corrupt SO and/or LE as well as mobile adaptively corrupt at most a fraction \(\frac{t}{n} - \epsilon\) of nodes \(N\).

First, note that \(\Pi_{\text{AD}}\) is YOSO when considering SO, LE, J and AU as input/output roles: Nodes \(N_i\) only send one (batch of) outgoing message(s) and do not retain state after doing so. Thus, we can simplify our corruption treatment in the following security proof to static corruptions.

We first describe the simulator \(S\) and then proceed to prove indistinguishability between the real and ideal worlds. Again, we use the convention that when \(F_{\text{AD}}\) wants to deliver output to an honest party, the adversary is notified about this (including the relevant task name, but not the other output) and output is only delivered after the adversary allows to do so.

The Simulator \(S\) is defined as follows.

<table>
<thead>
<tr>
<th>Simulator (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State of the Simulator</td>
</tr>
</tbody>
</table>
Setup
- SO verification key \( \mathsf{vk}_S \)
- J verification key \( \mathsf{vk}_J \)
- AU encryption keypair \( (\mathsf{ek}_A, \mathsf{sk}_A) \)
- Functionality public key \( \mathsf{pk} \)
- Common reference string \( \mathsf{crs}_{SS} \)
- Trapdoor \( \mathsf{td} \) for extracting and simulating zero-knowledge proofs

Lists to keep track of simulation
- \( L_{\mathsf{C}} \) to store \((C, \mathsf{nym})\) pairs (where \( C \leftarrow \text{PKE.Enc}(\mathsf{ek}, \mathsf{nym}) \))
- \( L_{\mathsf{secrets}} \) to store shares of retrieved secrets
- \( L_{\mathsf{Committee}^i} \): List of public keys of the committee with number \( i \). Entries are of the form \( (\mathsf{ek}_j, \mathsf{vk}_j) \).
- \( L_{\mathsf{Committee}^i} \): List of keys for the committee with number \( i \). Entries are of the form \( (\mathsf{ek}_j, \mathsf{dk}_j, \mathsf{vk}_j, \mathsf{sk}_j) \).
- For each committee a set \( \mathcal{H} \) of honest roles, a set \( \mathcal{B} \) of corrupted roles and a set \( \mathcal{SH} \) of corruptible roles.

State of \( \mathcal{F}_{\mathsf{BCRA}} \):
- Set \( \mathsf{Posted} \), initialized to \( \emptyset \)
- Sequence \( \mathsf{Ordered} \), initialized to \( () \)

Simulation of \( \mathcal{F}_{\mathsf{CRS}} \)
A corrupted party \( P \) can issue calls to \( \mathcal{F}_{\mathsf{CRS}} \) whenever it wants. Since simulation of \( \mathcal{F}_{\mathsf{CRS}} \) is straightforward, we specify here once how all calls from corrupted \( P \) to \( \mathcal{F}_{\mathsf{CRS}} \) are handled and omit calls to \( \mathcal{F}_{\mathsf{CRS}} \) in the following description of the simulator.

- Upon receiving (\text{value}) from corrupted \( P \) to \( \mathcal{F}_{\mathsf{CRS}} \):
  1. If this is the first time such a message is received:
     a. Generate and store \((\mathsf{crs}_{SS}^{(i)}, \mathsf{td}) \leftarrow \text{NIZK.Setup}()\)
  2. Report message \( \mathsf{crs}_{SS}^{(i)} \) from \( \mathcal{F}_{\mathsf{CRS}} \) to \( P \)

Simulation of \( \mathcal{F}_{\mathsf{BCRA}} \)
- Upon receiving (\text{Post}), (\text{Order}) or (\text{Read}) from corrupted \( P \) simulate \( \mathcal{F}_{\mathsf{BCRA}} \) honestly
- Upon receiving (\text{NextCommittee}, \( R, n, S \)) with \(|S| < cn \) containing entries of the form \((M_i \in \text{MACHINE}, (\mathsf{ek}_i, \mathsf{vk}_i))\) from the adversary
  1. Let \( j \) be the id of the next committee. If \( L_{\mathsf{Committee}^j} \neq \emptyset \), ignore this message.
  2. Otherwise, send (\text{Clock-Read}) to \( G_{\text{CLOCK}} \) and receive \( t_{\text{Now}} \)
- For \( i \in \{1, \ldots, |S|\} \):
  a. Mark \( M_i \) as corrupted
  b. Add \((R_i, (\mathsf{ek}_{R_i}, \perp, \mathsf{vk}_{R_i}, \perp)) \) to \( L_{\mathsf{Committee}^j} \)
  c. Set \( m = (\text{NextCommittee}, \mathcal{R}, \mathsf{ek}_i, \mathsf{vk}_i) \), add \((\text{roleassign}, t_{\text{Now}}, m) \) to \( \mathsf{Posted} \) and send \( m \) to the adversary on behalf of \( \mathcal{F}_{\mathsf{BCRA}} \)
For $i \in ([S]+1, \ldots, n)$:
(a) Sample $(ek_R, dk_R) \leftarrow \text{PKE.Gen}(1^\lambda)$
(b) Sample $(vk_R, sk_R) \leftarrow \Sigma.\text{Gen}(1^\lambda)$
(c) Add $(R_i, (ek_R, dk_R, vk_R, sk_R))$ to $L_{\text{Committee}}^j$
(d) Set $m = (\text{NextCommittee}, R_i, ek_R, vk_R)$, add $(\text{roleassign}, t_{\text{Now}}, m)$ to $\text{Posted}$ and send $m$ to the adversary on behalf of $\mathcal{F}_{\text{BCRA}}$

\text{Init (honest SO)}
-Upon receiving $(\text{Init}, \text{SO}, vk_{\text{SO}}, pk)$ from $\mathcal{F}_{\text{AD}}$
  1. Store $pk$ as the functionality public key
  2. Send $(\text{CLOCK-Read})$ to $\mathcal{G}_{\text{CLOCK}}$ and receive $t_{\text{Now}}$
  3. Set $m := (\text{OperatorKey}, vk_{\text{SO}})$
  4. Add $(\text{pid}_{\text{SO}}, t_{\text{Now}}, m)$ to $\text{Posted}$ and send $(\text{Post}, \text{pid}_{\text{SO}}, m)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  5. Allow $\mathcal{F}_{\text{AD}}$ to deliver output

\text{Init (corrupted SO)}
-Upon receiving $(\text{Post}, \text{OperatorKey}, vk_{\text{SO}})$ from $\text{SO}$ to $\mathcal{F}_{\text{BCRA}}$
  1. Send $(\text{CLOCK-Read})$ to $\mathcal{G}_{\text{CLOCK}}$ and receive $t_{\text{Now}}$
  2. Set $m := (\text{OperatorKey}, vk_{\text{SO}})$
  3. Add $(\text{pid}_{\text{SO}}, t_{\text{Now}}, m)$ to $\text{Posted}$ and send $(\text{Post}, \text{pid}_{\text{SO}}, m)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  4. Send $(\text{Init}, \text{SO}, vk_{\text{SO}})$ to $\mathcal{F}_{\text{AD}}$ on behalf of $\text{SO}$
  5. Upon receiving $(\text{Init}, \text{SO}, vk_{\text{SO}}, pk)$ from $\mathcal{F}_{\text{AD}}$, store $pk$ and allow $\mathcal{F}_{\text{AD}}$ to deliver output

\text{Init (honest J)}
- Upon receiving $(\text{Init}, J, vk_J)$ from $\mathcal{F}_{\text{AD}}$
  1. Send $(\text{CLOCK-Read})$ to $\mathcal{G}_{\text{CLOCK}}$ and receive $t_{\text{Now}}$
  2. Set $m := (\text{JudgeKey}, vk_J)$
  3. Add $(\text{pid}_J, t_{\text{Now}}, m)$ to $\text{Posted}$ and send $(\text{Post}, \text{pid}_J, m)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  4. Allow $\mathcal{F}_{\text{AD}}$ to deliver output
Init (honest AU)
- Upon receiving (INIT, AU) from $\mathcal{F}_{\text{AD}}$
  1. Send (CLOCK-READ) to $G_{\text{CLOCK}}$ and receive $t_{\text{Now}}$
  2. Generate $(ek_{\text{AU}}, sk_{\text{AU}}) \leftarrow \text{PKE.Gen}(1^\lambda)$ and store $(ek_{\text{AU}}, sk_{\text{AU}})$
  3. Set $m := (\text{AuditorKey}, ek_{\text{AU}})$
  4. Add $(pid_{\text{AU}}, t_{\text{Now}}, m)$ to $\text{POSTED}$ and send $(\text{POST}, pid_{\text{AU}}, m)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  5. Allow $\mathcal{F}_{\text{AD}}$ to deliver output

Get Public Key (honest P)
- Upon receiving (GetPK) from $\mathcal{F}_{\text{AD}}$
  1. Send (READ) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  2. Allow $\mathcal{F}_{\text{AD}}$ to deliver output

Get Public Key (corrupted P)
- This is handled by simulation of key generation roles and $\mathcal{F}_{\text{BCRA}}$

Get System Keys (honest P)
- Upon receiving (GetSKeys) from $\mathcal{F}_{\text{AD}}$
  1. Send (READ) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  2. Allow $\mathcal{F}_{\text{AD}}$ to deliver output

Get System Keys (corrupted P)
- This is handled by simulation of $\mathcal{F}_{\text{BCRA}}$

Request Decryption (honest LE)
- Upon receiving (REQUEST, $pid$, $W_{\text{pub}}^{\text{enc}}$, len, $v$) from $\mathcal{F}_{\text{AD}}$
  1. Send (CLOCK-READ) to $G_{\text{CLOCK}}$ and receive $t_{\text{Now}}$
  2. If $sk_{LE} = \bot$ set $(ek_{LE}, sk_{LE}) \leftarrow \text{PKE.Gen}(1^\lambda)$
  3. Send (READ) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  4. Generate $W_{\text{enc}} \leftarrow \text{PKE.Enc}(ek_{\text{AU}}, 0_{\text{len}})$
  5. For $i = 1, \ldots, v$ do:
    a) $ct_i \leftarrow \text{TPKE.Enc}(pk, 0)$
  6. $stmt_{\overline{W}} := (ek_{\text{AU}}, vk_{J}, W_{\text{enc}}^{\text{enc}}, W_{\text{pub}}^{\text{enc}}, \{ct_i\}_{i \in [v]})$
  7. $\pi_{\overline{W}} \leftarrow \text{NIZK.Sim}(crs_{sk}, stmt_{\overline{W}}, td)$
  8. Set $msg := (\text{REQUEST}, W_{\text{pub}}^{\text{enc}}, W_{\text{enc}}^{\text{enc}}, \{ct_i\}_{i \in [v]}, \pi_{\overline{W}}, ek_{LE})$, add $(pid, t_{\text{Now}}, msg)$ to $\text{POSTED}$ and send $(\text{POST}, pid, msg)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$
  9. Allow $\mathcal{F}_{\text{AD}}$ to deliver output
Request Decryption (corrupted LE)

- Upon receiving \((\text{POST}, (\text{REQUEST}, W^{\text{pub}}, W^{\text{enc}}, \{ct_i\}_{i \in [n]}, \pi_{\tilde{W}}, ek_{\text{LE}}))\) from LE to \(F_{\text{BCRA}}\)
  1. \(\text{stmt}_{\tilde{W}} := (ek_{\text{AU}}, vk_{\text{J}}, W^{\text{enc}}, W^{\text{pub}}, \{ct_i\}_{i \in [n]})\)
  2. If \(\text{NIZK.Verify}(\text{crs}_{ssk}, \text{stmt}, \pi_{\tilde{W}}) \neq 1\) ignore this message
  3. Otherwise, extract \(\text{wit}_{\tilde{W}} := (\tilde{W}, r, \sigma_{\tilde{W}}, \{ct_i, r_i\}_{i \in [n]}) \leftarrow \text{NIZK.Ext}(\text{crs}, \pi_{\tilde{W}}, td)\)
  4. Send \((\text{REQUEST}, \tilde{W}, \sigma_{\tilde{W}})\) to \(F_{\text{AD}}\) on behalf of LE

Retrieve Secret (honest LE)

- Upon receiving \((\text{RETRIEVE})\) from \(F_{\text{AD}}\)
  1. Send \((\text{READ})\) to \(A\) on behalf of \(F_{\text{BCRA}}\)
  2. Allow \(F_{\text{AD}}\) to deliver output

Retrieve Secret (corrupted LE)

- This is handled during simulation of handover and \(F_{\text{BCRA}}\)

Get Statistics (honest P)

- Upon receiving \((\text{GETSTATISTICS})\) from \(F_{\text{AD}}\)
  1. Send \((\text{READ})\) to \(A\) on behalf of \(F_{\text{BCRA}}\)
  2. Allow \(F_{\text{AD}}\) to deliver output

Get Statistics (corrupted P)

- This is handled by simulation of \(F_{\text{BCRA}}\)

Auditor Audit (honest AU)

- Upon receiving \((\text{AUDITREQUEST})\) from \(F_{\text{AD}}\)
  1. Send \((\text{READ})\) to \(A\)
  2. Allow \(F_{\text{AD}}\) to deliver output

RoleExecute (honest N) For each committee, after \(F_{\text{BCRA}}\) delivered all secret keys, the simulator fills the three sets \(H, B, SH\) as follows:

- All roles that are already corrupted are assigned to \(B\)
- \(t - |B|\) random roles that are not corrupted are assigned to \(SH\)
- The remaining roles are assigned to \(H\)
It holds that $SH \cap B = \emptyset$ and $H \cap (SH \cup B) = \emptyset$ and $H \cup SH \cup B$ contains all roles of that committee. Whenever the adversary corrupts a party that is assigned a role in $H$ and has not executed yet, the simulator picks a random role $R_i \in SH$ instead, sets $SH := SH \setminus \{R_i\}$, $B := B \cup \{R_i\}$ and returns the state of role $R_i$ instead (which consists of that roles secret keys). When the adversary corrupts a party that is assigned a role $R_i$ in $SH$, the simulator sets $SH := SH \setminus \{R_i\}$, $B := B \cup \{R_i\}$ and returns the state of that role.

- Upon receiving $(\text{ExecuteRole}, \text{pid}, N)$ from $F_{AD}$
  1. Send $(\text{Read})$ to $A$ on behalf of $F_{BCRA}$
  2. Send $(\text{CLOCK-Read})$ to $G_{CLOCK}$ and receive $t_{\text{Now}}$
  3. If $t_{\text{last}} + t_{\text{com}} \leq t_{\text{Now}}$ execute $\text{PrepareExecution}$
  4. If this node is scheduled to execute a role $R_i$ this round, execute the respective task $\text{KeyGen1}$, $\text{KeyGen2}$ or $\text{Handover}$ below.

- $\text{PrepareExecution}$
  1. Set $t_{\text{last}} = t_{\text{Now}}$
  2. Initialize empty lists $L_{\text{CommitteePK}}$ and $L_{\text{CommitteeVK}}$
  3. For each role $R_j$ in the next committee
     (a) Find $(\text{roleassign}, t, (\text{GENERATE}, R_j, ek_R, vk_R))$ in ORDERED
     (b) Insert $ek_R$ into $L_{\text{next CommitteePK}}$
  4. For each role $R_j$ in the current committee
     (a) Find $(\text{roleassign}, t, (\text{GENERATE}, R_j, ek_R, vk_R))$ in ORDERED
     (b) Insert $ek_R$ into $L_{\text{CommitteePK}}$
  5. For each role $R_j$ in the previous two committees
     (a) Find $(\text{roleassign}, t, (\text{GENERATE}, R_j, ek_R, vk_R))$ in ORDERED
     (b) Insert $vk_R$ into $L_{\text{CommitteeVK}}$
  6. Assign the roles of the current committee to $H, SH, B$ as described above
  7. If KeyGen1 has been executed, but KeyGen2 not, do the following:
     (a) For each entry $(\text{pid}, t, (R_j, (\text{KEYGEN1}, \left(\{ct_j\}_{j \in [n]}, \pi_{KG}\}), \sigma))$ in ORDERED
        i. Retrieve $vk_R$ from $L_{\text{CommitteeVK}}$
        ii. Check if $\Sigma. \text{Vfy}(vk_R, (\text{KEYGEN1}, \left(\{ct_j\}_{j \in [n]}, \pi_{KG}\}), \sigma) = 1$, otherwise skip this entry.
        iii. Assemble ZK-Statement:
            $\text{stmt}_{KG} := (\{ct_j, ek_{R_j}\}_{j \in [n]})$
        iv. If $\text{NIZK.Verify}(\text{crs}_{zk}^{SS}, \text{stmt}_{KG}, \pi_{KG}, R_{KG1}^{AD}) \neq 1$ skip this entry
        v. Add $ct_i$ to $L_{\text{Qual}}$ (the one addressed to the currently executing role)
(b) Sort \(L_{\text{Qual}}\) lexicographically

(c) For each of the first \(t+1\) entries in \(L_{\text{Qual}}\) do the following:

i. Decrypt the ciphertexts for all roles in \(\mathcal{H}\) (by assumption of \(n > 2t + 1\) this results in at least \(t+1\) shares) and use the resulting shares to interpolate all \(t+1\) polynomials \(G_1, \ldots, G_{t+1}\)

ii. For each role \(R_j \in \mathcal{SH} \cup \mathcal{B}\) compute \(sk_j := \sum_{k=1}^{t+1} G_k(j)\) and \(pk_j := \text{TPKE}.\text{Sk2Pk}(sk_j)\)

iii. For each role \(R_j \in \mathcal{H}\) set \(pk_j := pk_{\lambda_j,0} \cdot \prod_{k \in \mathcal{SH} \cup \mathcal{B}} pk_{\lambda_{j,k}}^k\), where \(\lambda_{i,j}\) are appropriate Lagrange coefficients.

8. If key generation already finished, do the following:

(a) For each entry \((\text{pid}, t, (R_j, \text{msg}, \sigma))\) with \(\text{msg} := (\text{KeyShare}, (vk_i, \{ct_j\}, \pi_{KS}))\) for the previous committee in ORDERED

i. Retrieve \(\text{pk}_{R_j}\) from \(L_{\text{CommitteeVK}}\)

ii. Check \(\Sigma.\text{Vfy}(\text{pk}_{R_j}, \text{msg}, \sigma) = 1\), otherwise ignore this message

iii. Assemble ZK-Statement:

\[\text{stmt}_{KS} := \text{ct}_j \in [n], \{ek_{R_j}\}\]

iv. Check NIZK.\text{Verify}(\text{crs}_{SS}, \text{stmt}_{KS}, \pi_{KS}, \mathcal{R}_{KS}) = 1\), otherwise ignore this message

v. Add \(\{\text{ct}_j\} \in [n]\) to \(L_{\text{ct}}\)

(b) Sort \(L_{\text{ct}}\) lexicographically

(c) For each entry \((\text{pid}, t, (R_j, \text{msg}, \sigma))\) with \(\text{msg} := (\text{KeyShare}, (vk_i, \{ct_j\}, \pi_{KS}))\) for the current committee in ORDERED:

i. Retrieve \(\text{vk}_{R_j}\) from \(L_{\text{CommitteeVK}}\)

ii. Check if \(\Sigma.\text{Vfy}(\text{vk}_{R_j}, (\text{KeyShare}, \text{msg}, \sigma)) = 1\), otherwise skip this entry.

iii. Retrieve from \(L_{\text{ct}}\) from the first \(t+1\) entries the ciphertext \(ct_j\) for \(R_j\) each, as \(L_{\text{KCom}} := \{ct_1, \ldots, ct_n\}\)

iv. Assemble ZK-Statement:

\[\text{stmt}_{KS} := \{\text{ct}_j\} \in [n], L_{\text{KCom}}\]

v. Check if NIZK.\text{Verify}(\text{crs}_{SS}, \text{stmt}_{KS}, \pi_{KS}, \mathcal{R}_{KS}) = 1\), otherwise skip this entry.

vi. Add \(ct_i\) (the one addressed to the currently executing role) to \(L_{\text{ct}}\)

(d) Sort \(L_{\text{ct}}\) lexicographically

(e) For each of the first \(t+1\) entries in \(L_{\text{ct}}\) do the following:

i. Decrypt the ciphertexts for all roles in \(\mathcal{H}\) (by assumption of \(n > 2t + 1\) this results in at least \(t+1\) shares) and use the resulting shares to interpolate all \(t+1\) polynomials \(G_1, \ldots, G_{t+1}\)

ii. For each role \(R_j \in \mathcal{SH} \cup \mathcal{B}\) compute \(sk_j := \sum_{k=1}^{t+1} G_k(j)\) and \(pk_j := \text{TPKE}.\text{Sk2Pk}(sk_j)\)

iii. For each role \(R_j \in \mathcal{H}\) set \(vk_j := pk_{\lambda_j,0} \cdot \prod_{k \in \mathcal{SH} \cup \mathcal{B}} pk_{\lambda_{j,k}}^k\), where \(\lambda_{i,j}\) are appropriate Lagrange coefficients.
(f) For each entry \((\text{pid}, t, (\text{REQUEST}, W^{\text{pub}}, W^{\text{enc}}, \{\overset{\_}{c}_i\}_{i \in [v]}, \pi^{\_}_W, \text{ek}_{\text{LE}}))\) in ORDERED
   i. Assemble ZK-Statement:
      \[\overset{\_}{\text{stmt}}_W := (\text{ek}_{\text{AU}}, \overset{\_}{\text{vk}}_j, W^{\text{enc}}, W^{\text{pub}}, \{\overset{\_}{c}_i\}_{i \in [v]})\]
   ii. If NIZK.Verify\((\overset{\_}{\text{crs}}_sk, \overset{\_}{\text{stmt}}_W, \pi^{\_}_W, R^{\text{AD}}) \neq 1\), ignore this message
   iii. For each \(i \in [v]\):
      A. Store \((\text{ek}_{\text{LE}}, ct_i, \bot)\) in \(L_{\text{Requests}}\)

(g) If LE is corrupted:
   i. Send \((\text{RETRIEVE})\) to \(\mathcal{F}_{\text{AD}}\) and receive \((\text{RETRIEVE}, \overset{\_}{\text{Pending}})_{\text{Requests}}\)
   ii. For each entry \((\text{ek}_{\text{LE}}, ct_i, \bot)\) in \(L_{\text{Requests}}\):
      A. Find \((ct_i, \text{secret}_i)\) in \(L_{\text{Pending}}\)
      B. For each role \(R_j\) in \(\mathcal{H}\), compute \(\overset{\_}{\text{ct}}_i^j \leftarrow \text{TPKE}.T\text{Dec}(sk_j, ct_i)\), where \(sk_j\) was computed earlier
      C. Run \(\{\overset{\_}{\text{ct}}_i^k\}_{k \in H} \leftarrow \text{TPKE}.\text{SimTDec}(pk, ct_i, \text{secret}_i, \{\overset{\_}{c}_i\}_{i \in [\mathcal{H}]})\)
      D. Update the entry in \(L_{\text{Requests}}\) to \((\text{ek}_{\text{LE}}, ct_i, \{\overset{\_}{c}_i\}_{k \in H})\)

- KeyGen1
  1. Retrieve the respective \(dk_R, sk_R\)
  2. Generate key share: \(s_i \leftarrow \mathbb{Z}_p\)
  3. Share secret key: choose a random degree \(t\) polynomial \(F(x) = a_0 + a_1*x + a_2*x^2 + \ldots + a_t*x^t\) with \(F(0) = s_i\)
  4. For each role \(R_j\) in the next committee:
     (a) Set \(sh_j := F(j)\)
     (b) Generate ciphertext \(ct_j \leftarrow \text{PKE}.\text{Enc}(ek_{R_j}, sh_j, r_j)\)
  5. Assemble ZK-Witness: \(wit_{KG} := \{s_i, \{r_j\}_{j \in [n]}, F\}\)
  6. Assemble ZK-Statement:
     \[\text{stmt}_{KG} := (\{ct_j, \text{ek}_{R_j}\}_{j \in [n]}\]
  7. Compute Proof:
     \(\pi_{KG} \leftarrow \text{NIZK}.\text{Prove}(\overset{\_}{\text{crs}}_{sk}, \text{stmt}_{KG}, \text{wit}_{KG}, R^{\text{AD}})\)
  8. Prepare message: \(msg := (\text{KeyGen1}, (\{ct_j\}_{j \in [n]}, \pi_{KG}))\)
  9. Sign message: \(\sigma \leftarrow \Sigma.\text{Sign}(sk_R, msg)\)
  10. Add \((\text{pid}, t, \text{Now}, (R_i, msg, \sigma))\) to \(\text{POSTed}\) and send \((\text{POST, pid, (R_i, msg, \sigma)})\) to \(\mathcal{A}\) on behalf of \(\mathcal{F}_{\text{BCRA}}\)

- KeyGen2
  1. If \(R_i \in \mathcal{H}\) execute this role honestly
  2. Otherwise \((R_i \notin \mathcal{H})\) do the following:
     (a) Choose random degree-\(t\) polynomial \(F(x)\)
     (b) For each role \(R_j\) in the next committee:
        i. Set \(sh_j := F(j)\)
        ii. Generate ciphertext \(ct_j \leftarrow \text{PKE}.\text{Enc}(ek_{R_i}, sh_j, r_j)\)

120
(c) Retrieve pk, computed during preparation
(d) Assemble ZK-Statement:
   \[ \text{stmt}_{KG} := (pk_i, \{ct^k\}_{k \in [t+1]}, \{ct_j, ek_{R_j}\}_{j \in [n]}, ek_R) \]
(e) Simulate Proof:
   \( \pi_{KG} \leftarrow \text{NIZK.Sim}(crs_{SS}^{SS}, \text{stmt}_{KG}, td, R_{AD}^{KG2}) \)
(f) Prepare message: \( msg := (\text{KEYGEN2}, (pk_i, \{ct_j\}_{j \in [n]}, \pi_{KG})) \)
(g) Sign message: \( \sigma \leftarrow \Sigma.\text{Sign}(sk_R, msg) \)
(h) Add \((\text{pid, tNow, } (R_i, msg, \sigma))\) to \(\text{POSTed}\) and send \((\text{POST, pid, } (R_i, msg, \sigma))\) to \(A\) on behalf of \(\mathcal{F}_{BCRA}\)

Handover
1. If \(R_i \in \mathcal{SH}\) execute this role honestly
2. Otherwise \((R_i \in \mathcal{H})\) do the following:
   (a) Reshare secret key share:
      i. Retrieve \(vk_i\) and the first \(t + 1\) entries of \(L_{\text{Qual}}^i\) as \(\{ct^k\}_{k \in [t+1]}\)
         computed during preparation
      ii. Choose a random degree \(t\) polynomial \(F(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_t x^t\)
      iii. For each role \(R_j\) in the next committee:
         A. Generate ciphertext \(ct_j \leftarrow \text{PKE.Enc}(ek_{R_j}, F(j); r_j)\)
      iv. Assemble ZK-Statement:
         \(\text{stmt}_{KS} := (ek_{R_j}, \{ct^k\}_{k \in [t+1]}, \{ct_j, ek_{R_j}\}_{j \in [n]}, vk_i)\)
   v. Simulate Proof:
      \( \pi_{KS} \leftarrow \text{NIZK.Sim}(crs_{SS}^{SS}, \text{stmt}_{KS}, td, R_{AD}^{KS}) \)
   vi. Add \(msg := (\text{KEYSHARE}, (vk_i, \{ct_j\}_{j \in [n]}, \pi_{KS})\) to \(L_M\)
(b) For each entry \((ek_{LE}, ct, L_{\text{Decryptions}})\) in \(L_{\text{Requests}}\):
   • If LE is honest:
      i. Partially decrypt \(ct\) to \(ct^* \leftarrow \text{TPKE.TDec}(sk_i, ct)\)
      ii. Assemble ZK-Statement:
         \(\text{stmt}_{Dec} := (vk_i, ct, ct^*)\)
      iii. Simulate Proof:
         \( \pi_{Dec} \leftarrow \text{NIZK.Sim}(crs_{SS}^{SS}, \text{stmt}_{Dec}, td, R_{AD}^{Dec}) \)
   iv. Generate response ciphertext: \(msg \leftarrow \text{PKE.Enc}(ek_{LE}, (0|ct, 0|ct^*, 0|\pi_{Dec}))\)
   v. Add \((\text{REQUEST}, msg)\) to \(L_M\)
   • If LE is corrupted:
      i. Retrieve \(ct^*\) for this role from \(L_{\text{Decryptions}}\)
      ii. Assemble ZK-Statement:
         \(\text{stmt}_{Dec} := (vk_i, ct, ct^*)\)
      iii. Simulate Proof:
         \( \pi_{Dec} \leftarrow \text{NIZK.Sim}(crs_{SS}^{SS}, \text{stmt}_{Dec}, td, R_{AD}^{Dec}) \)
   iv. Encrypt answer: \(msg \leftarrow \text{PKE.Enc}(ek_{LE}, (ct, ct^*, \pi_{Dec}))\)
   v. Add \((\text{REQUEST}, msg)\) to \(L_M\)

121
To prove Theorem 2, we proceed in a series of games, where \( F_{AD}^{i+1} \) behaves the same as \( F_{AD}^i \) except for the stated changes and \( S^{i+1} \) behaves the same as \( S^i \) except for the stated changes.

**Game 0.** Hybrid 0 is equivalent to the real experiment. That is,

\[
H_0 := \text{Exp}\text{r}_{\text{CRS},F_{BCRA},G_{\text{CLOCK}},S^0,Z}
\]

with \( S^0 \) being the dummy adversary. This means that all parties execute the real protocol.

**Game 1.** In this game, the simulator \( S^1 \) takes control over \( F_{CRS} \) and \( F_{BCRA} \), but executes them honestly. Additionally, \( F_{AD}^1 \) is introduced, and all honest parties are replaced by dummy parties that forward their inputs to \( F_{AD}^1 \) and when receiving output from \( F_{AD}^1 \) forward it to the environment. \( F_{AD}^1 \) upon receiving any message from a dummy party forwards it to the simulator \( S^1 \) and asks it for output. \( S^1 \) executes the real protocol for all honest parties (using the inputs received from \( F_{AD}^1 \)) and instructs \( F_{AD}^1 \) to deliver the resulting outputs.

**Proof Sketch:** Game 0 \( \approx \) Game 1.

\( S^1 \) executes the same code as the real parties on the same inputs (that have been forwarded by \( F_{AD}^1 \)). Thus, Game 0 and Game 1 are identical from the environments view.

**Game 2.** In this game, \( S^2 \) generates the common reference string for the zero-knowledge proof system with an extraction and simulation trapdoor \( td \) as \((\text{crs}_{SS_{zk}},td) \leftarrow \text{NIZK.Setup}(1^\lambda)\).

**Proof Sketch:** Game 1 \( \approx \) Game 2.

Indistinguishability follows from the simulation extractability of \text{NIZK}.

**Game 3.** In this game, \( F_{AD}^3 \) now handles (INIT,…) messages the same way \( F_{AD} \) does, except that during SO init it also sends the generated secret key \( sk \) to \( S^3 \), and \( S^3 \) handles this task the same way \( S \) does, except that it also stores the received secret key \( sk \). Note that \( S^3 \) still chooses the outputs for (GETPK) and can thus still return the public encryption key \( pk' \) generated during the protocol.

**Proof Sketch:** Game 2 \( \approx \) Game 3.

If SO is honest, \( S^3 \) receives the message (INIT, SO, \( \text{vk}_{SO}, pk \)) from \( F_{AD}^3 \) and can thus perfectly simulate the real protocol for this task. If SO is corrupted, \( S^3 \) can wait for a message (POST, OperatorKey, \( \text{vk}_{SO} \)) from SO to \( F_{BCRA} \) and then
send (Init, SO, vk_{SO}) to F^3_AD on behalf of SO. Since both J and AU are assumed to be honest, S^3 receives the respective messages from F^3_AD and can perfectly simulate the real protocol for the respective task.

Game 4. In this game, F^3_AD now handles (GetSKs) the same way F_AD does, and S^4 handles this task the same way S does.

Proof Sketch: Game 3 \approx Game 4.
For corrupted parties, this is perfectly simulated by executing F_BCRA. For honest parties, F^3_AD outputs the keys if they are available via F_BCRA. Thus, these games are statistically indistinguishable.

Game 5. In this game, after simulation of F_BCRA resulted in all secret keys for the next committee being delivered, S^5 fills the three sets H, B, SH as follows:
- All roles that are already corrupted are assigned to B
- t − |B| random roles that are not corrupted are assigned to SH
- The remaining roles are assigned to H

It holds that SH \cap B = \emptyset and H \cap (SH \cup B) = \emptyset and H \cup SH \cup B contains all roles of that committee.

Proof Sketch: Game 4 \approx Game 5.
This change is only syntactical.

Game 6. In this game, whenever the adversary corrupts a party that is assigned a role in H and has not executed yet, the simulator picks a random role R_i \in SH instead, sets SH := SH \setminus \{R_i\}, B := B \cup \{R_i\} and returns the state of role R_i instead (which consists of that role's secret keys). When the adversary corrupts a party that is assigned a role R in SH, the simulator sets SH := SH \setminus \{R\}, B := B \cup \{R\} and returns the state of that role.

Proof Sketch: Game 5 \approx Game 6.
Since all roles in a committee have the same description, and the assignment of a role to a node only becomes visible after it executed, these games are indistinguishable.

Game 7. In this game, the simulator uses the shares of uncorrupted roles to reconstruct the secret key sk' associated with the public encryption key pk' generated during the key generation protocol. Additionally, S^7 performs the steps listed under PrepareExecution in S, except that in step 8g) instead of querying F_AD for decrypted secrets it uses sk' to decrypt ciphertexts ct_i to secret_i.

Proof Sketch: Game 6 \approx Game 7.
This change only affects the internal view of the simulator.

Game 8. This is the same as the previous game, except that during KeyGen2 and Handover the simulation trapdoor td is used to generate the zero-knowledge proofs.

Proof Sketch: Game 7 \approx Game 8.
Indistinguishability follows from the zero-knowledge property of NIZK.

Game 9. In this game, if LE is corrupted, then when processing requests for decryption while executing roles in H, S^9 uses \hat{ct} computed during PrepareExecution instead of executing TPKE.TDec. (This corresponds to step 2b), LE corrupted in S)
Proof Sketch: Game 8 $\approx$ Game 9.

If LE is honest, these two games are the same. Otherwise, $\hat{c}_t$ has been computed to ensure that TPKE outputs the correct plaintext. Since honest roles delete all internal state before sending their message, these two games are indistinguishable.

**Game 10.** In this game, if LE is honest, then when processing requests for decryption while executing roles in $H$, $S^{10}$ encrypts (the appropriate amount of) zeroes instead. (This corresponds to step 2b), LE honest in $S$)

Proof Sketch: Game 9 $\approx$ Game 10.

If LE is corrupted, these two games are the same. Otherwise, indistinguishability follows from IND-CPA security of PKE.

**Game 11.** In this game, $S^{11}$ simulates the distributed key generation protocol to output $pk$ (received from $F_{11}^{AD}$ during SO Init) as resulting public encryption key. This is achieved by following the steps listed under KeyGen2 and Handover in $S$. Additionally, $sk$ (received from $F_{11}^{AD}$ during SO Init) is used during PrepareExecution to decrypt ciphertexts. $F_{11}^{AD}$ now handles (EXECUTEROLE) messages the same way $F_{AD}$ does.

Proof Sketch: Game 10 $\approx$ Game 11.

Since by assumption at most $t$ roles in each committee are corrupted, the shares received by corrupted roles are statistically close to those in Game 10. Since honest parties delete all internal state before sending their message, corrupting them after they executed does not reveal anything. Thus, only ciphertexts addressed to uncorrupted roles differentiate these two games, and RIND-SO security of PKE ensures indistinguishability.

**Game 12.** In this game, $F_{12}^{AD}$ now handles (GETPK) the same way $F_{AD}$ does, and $S^{12}$ handles this task the same way $S$ does.

Proof Sketch: Game 11 $\approx$ Game 12.

Since the previous games already ensured that the same public key $pk$ is used in both $F_{AD}$ and the protocol simulation, these two games are perfectly indistinguishable.

**Game 13.** In this game, when receiving $\left(\text{REQUEST}, \ W, \sigma_{\bar{W}}\right)$ messages, $F_{AD}^{13}$ checks if the signature is valid. If the signature is valid, it forwards $\left(\text{REQUEST}, \ W\right)$ to $S^{13}$ (but not the signature $\sigma_{\bar{W}}$!), otherwise it ignores the message. $S^{13}$ follows the protocol honestly except for the NIZK proof, where it uses the simulation trapdoor to generate $\pi_{\bar{W}} \leftarrow \text{NIZK.Sim}(\text{crs}_{sk}, \text{stmt}_{\bar{W}}, t_d, R_{AD}^d)$ without knowing the witness.

Proof Sketch: Game 12 $\approx$ Game 13.

If LE is corrupted, these two games are identical. If LE is honest, then indistinguishability follows from zero-knowledge of NIZK.

**Game 14.** In this game, when receiving $\left(\text{POST}, \ \text{(REQUEST, } W_{\text{pub}}, W_{\text{enc}}, \{ct_i\}_{i \in [v]}, \pi_{\bar{W}}, e_k_{\text{LE}})\right)$ from LE to $F_{BCRA}$, if the zero-knowledge proof verifies, $S^{14}$ uses the trapdoor $td$ to extract a witness and aborts the simulation of this fails.

Proof Sketch: Game 13 $\approx$ Game 14.

This game only differs from Game 13 if the abort happens. Due to the simulation
extractability of NIZK, this only happens with negligible probability, and thus the games are indistinguishable.

Game 15. In this game, when receiving (Post, (Request, \( W^{\text{pub}}, W^{\text{enc}}, \{ct_i\}_{i\in[v]}, \pi_{\tilde{W}}, \text{ek}_{\text{LE}} \)) from LE to \( F_{\text{BCRA}} \), \( S^{15} \) sends \((\text{Request}, \tilde{W}, \sigma_{\tilde{W}})\) to \( F^{15}_{\text{AD}} \) in the name of LE. \( F^{15}_{\text{AD}} \) now fills the lists \( L_{\text{Warrants}}, L_{\text{Requests}} \) the same way \( F_{\text{AD}} \) does, and as a consequence also fills the lists \( L_{\text{Pending Requests}} \) and \( L_{\text{Ready Requests}} \) during Handover.

Proof Sketch: Game 14 \( \approx \) Game 15. This only affects the internal view of \( F^{15}_{\text{AD}} \), but no output depends on that view (yet). Thus, these games are perfectly indistinguishable.

Game 16. In this game, \( F^{16}_{\text{AD}} \) handles (Retrieve) messages the same way \( F_{\text{AD}} \) does, and \( S^{16} \) handles this task the same way \( S \) does.

Proof Sketch: Game 15 \( \approx \) Game 16. If LE is corrupted, this changes nothing. If LE is honest, the list \( L_{\text{Ready Requests}} \) has been correctly filled in Game 15. Since by assumption, at least \( t+1 \) roles in each committee are not corrupted, every ciphertext that \( F_{\text{AD}} \) decrypts also results in at least \( t+1 \) decryption shares in the real protocol. As such, the same set of decryptions is output in both games and indistinguishability follows.

Game 17. This game is the same as Game 16 except that \( F^{17}_{\text{AD}} \) now stores \( \tilde{W} \) in \( L_{\text{Warrants}} \) when \( S^{17} \) instructs it to deliver output while handling (Request) messages.

Proof Sketch: Game 16 \( \approx \) Game 17. This change is only syntactical.

Game 18. In this game, \( F^{18}_{\text{AD}} \) additionally handles (AuditRequest) messages the same way \( F_{\text{AD}} \) does, and \( S^{18} \) handles this task the same way \( S \) does. Note that since the auditor AU is assumed to be honest, simulation only requires a simulated (Read) to \( F_{\text{BCRA}} \).

Proof Sketch: Game 17 \( \approx \) Game 18. If LE is honest, these games are identical, since \( S^{17} \) generated entries in ORDERED with valid proofs for exactly the warrants that \( F^{18}_{\text{AD}} \) has in \( L_{\text{Warrants}} \), and simulation of AU is perfect. If LE is corrupted, then these games are also identical: In Game 17 AU outputs exactly those warrants stored in ORDERED for which the zero-knowledge proof is valid. For exactly those warrants, \( S^{18} \) sends (Request,\ldots) messages to \( F^{18}_{\text{AD}} \), resulting in them being in \( L_{\text{Warrants}} \). Thus, AU outputs the same set of warrants in Game 18 as in Game 17.

Game 19. This game is the same as Game 18 except that \( S^{19} \) computes \( W^{\text{enc}} \leftarrow \text{PKE}(\text{ek}_{\text{AU}}, 0^{|\tilde{W}|}) \) instead of encrypting \( \tilde{W} \).

Proof Sketch: Game 18 \( \approx \) Game 19. If LE is corrupted, these two games are again identical. If LE is honest, indistinguishability follows directly from the IND-CPA security of PKE.

Game 20. In this game, \( F^{20}_{\text{AD}} \) no longer sends the secret decryption key \( sk \) to \( S^{20} \) during Init. Instead of using \( sk \) to decrypt ciphertexts during \textbf{PrepareExecute} handling, \( S^{20} \) now queries \( F^{20}_{\text{AD}} \) for the decryption as described in \( S \).
Proof Sketch: Game 19 $\approx$ Game 20.

The only place $sk$ was used was to decrypt ciphertexts in step 8g) of Prepare-Execute. Thus, if LE is honest, these games are the same. If LE is corrupted, these two games only differ if there is an entry $(ek_{LE}, ct_i, \perp)$ in $L_{Requests}$ for which no matching entry $(ct_i, secret_i)$ exists in $L_{Pending_{Requests}}$ returned by $F_{AD}^{20}$. This can not be the case: Only ciphertexts $ct_i$ corresponding to valid zero-knowledge proofs are inserted into $L_{Requests}$. But for each valid zero-knowledge proof, the extracted warrant (and thus all listed ciphertexts) was send as input to $F_{AD}^{20}$, and thus a valid decryption is returned. Therefore, these games are indistinguishable.

Game 21. In this game, $F_{AD}^{21}$ handles (REQUEST) messages the same way as $F_{AD}$, and $S_{21}$ handles this task the same way as $S$.

Proof Sketch: Game 20 $\approx$ Game 21.

For corrupted LE, this changes nothing. For honest LE, $S^{21}$ no longer receives $\tilde{W}$ but only $f_t(W), |\tilde{W}|, v$, i.e. the output of the transparency function, the length of the warrant and the number of listed ciphertexts. $\tilde{W}$ was used to compute the transparency function and obtain the listed ciphertexts before, since encryption of the warrant was already replaced by an encryption of zeroes in Game 19. Since the output of the transparency function was passed by $F_{AD}^{21}$, the only missing thing are the listed ciphertexts, which the simulator replaces by ciphertexts of zeroes. By assumption, at most $t$ roles in each committee are corrupted, and up to $t$ decryption shares obtained by the environment in this way reveal no information about the plaintexts. Note that the decryption of these ciphertexts has not been used by $S^{21}$ since Game 10, except for parties in $SH$, for which it is ensured that there are at most $t$ parties in $SH \cup B$. Thus, indistinguishability follows from IND-CPA security of TPKE.

Game 22. In this game, $F_{AD}^{22}$ additionally handles (GETSTATISTICS) messages the same way $F_{AD}$ does, and $S^{22}$ handles this task the same way $S$ does.

Proof Sketch: Game 21 $\approx$ Game 22.

These games are identical for the same reason as above, the set of warrants in ORDERED with valid proofs is the same as the set of warrants in $F_{AD}^{18}$.

Note that Game 22 is the same as the ideal world. Since we showed that Game $i$ is indistinguishable from Game $i + 1$ for $i \in \{0, \ldots, 21\}$, it follows that Game 0 (the real protocol) is indistinguishable from Game 22 (ideal execution) and thus Theorem 2 follows.

G Security Proof: $\Pi_{ASTE}$ UC-realizes $F_{ASTE}$

In this section, we prove that the protocol $\Pi_{ASTE}$ from Appendix C.3 UC-realizes the functionality $F_{ASTE}$ from Appendix C.2.

G.1 Simulator

In this section, we describe the inner workings of the simulator for $\Pi_{ASTE}$. Apart from message sending and law enforcement access, most other tasks are trivial to simulate. They are basically taken care of by $F_{AS}$, due to $H_{ASTE}$ working
the $F_{\text{ASTE}}$-hybrid model, and the simulator merely implements the bookkeeping of $F_{\text{ASTE}}$. To simplify the description of the simulator, we assume w.l.o.g. that we deal with the dummy adversary.

We use the phrase “the simulator executes the honest protocol” for some party $P$ to express, that the simulator follows the instructions of party $P$ honestly, while at the same time honestly emulating the hybrid functionalities and the adversary (if necessary, e.g. to see when a (delayed) message arrives).

Simulator overview The simulation of most tasks is easy since $F_{\text{ASTE}}$ together with $PKE_{\text{AS}}$ do all the hard work. Indeed, Init. Party Init, Next Period, User Registration, Update, Request Warrant, Get Statistics and Audit are handled by running the real protocol essentially unmodified. The interesting cases are Encrypt Message, Decrypt Ciphertext, Prepare Access and Execute Access. For these cases, ciphertexts of honest users must be simulated, and ciphertexts of corrupted users must be extracted (in particular, the $uid$). This is handled by $PKE_{\text{AS}}$. We implicitly use that the simulator can program $F_{\text{RO}}$ and simulate and extract NIZK proofs of $F_{\text{ASTE}}$.

Notation We use the notation introduced for $PKE_{\text{AS}}$, see Remark 7. Like $F_{\text{ASTE}}$ (and $H_{\text{ASTE}}$), $S$ internally uses lists, e.g. $L_{\text{uvper}}, L_{\text{msgs}}, L_{W}, L_{W-\text{cached}}$, and variables $ureg_i, uski_i, uid_i$, and so on. To distinguish the variables of $F_{\text{ASTE}}$ and $S$, if not clear from the context, we write $F_{\text{ASTE}}.L_{\text{uvper}}$ for $F_{\text{ASTE}}$. Variables without extra specification (e.g. $L_{\text{uvper}}$) belong to the simulator.

High level simulation strategy The simulator keeps its “shadow copies” or lists and variables of $F_{\text{ASTE}}$ in perfect synchronization with those of $F_{\text{ASTE}}$, to the extent possible. Clearly, $L_{\text{msgs}}$ cannot equal $F_{\text{ASTE}}.L_{\text{msgs}}$, since $S$ cannot know all messages or $uids$ in $L_{\text{msgs}}$; hence they are filled in with placeholders (for simplicity, $\perp$). These placeholders are later, if necessary, filled in, e.g. if due to surveillance of a user the $uids$ to ciphertexts become known.

Most simulation steps are quite straightforward, thanks to $F_{\text{ASTE}}$ which provides a powerful and easy-to-use basis. The steps where simulation requires most care (besides ensuring synchronization of variables with $F_{\text{ASTE}}$) are the tasks involving $PKE_{\text{AS}}$, which are encrypt message, decrypt ciphertext, prepare access and execute access. We give a high-level overview.

In encrypt message, the simulator generates ciphertexts for honest parties by running $PKE_{\text{AS}}.\text{Sim}$, i.e. these ciphertexts have no message $m$ or $uid$ associated with them (yet), and thus in $L_{\text{msgs}}$ the respective components are set to $\perp$ (whereas in $F_{\text{ASTE}}.L_{\text{msgs}}$: $m$ and $uid$ are filled in). In case $m$ or $uid$ is leaked to $S$ (because the receiver is corrupted or $U_i$ is under surveillance), $S$ runs $PKE_{\text{AS}}.\text{Expln}_{\text{msg}}$ or $PKE_{\text{AS}}.\text{Expln}_{\text{LE}}$ to ensure consistency under decryption, and also updates the entry in $L_{\text{msgs}}$. When an honest user decrypts a ciphertext $c$ which is part of $F_{\text{ASTE}}.L_{\text{msgs}}$, $S$ learns nothing (and does nothing). If $(c, \cdot, \cdot, \cdot) \notin F_{\text{ASTE}}.L_{\text{msgs}}$, then $F_{\text{ASTE}}$ requests $(\text{SimInjectMsg}, c)$ of $S$. To respond, $S$ extract the NIZK to learn $(m, uid_i)$ from $c$. 

127
For simulating prepare access, the main difficulty is to ensure that for corrupted LE the ciphertext contents of surveillable ciphertexts are always correct. To this end, all affected ciphertexts are first identified (via execute access) and their uids are explained (via PKE_{AS,ExplnLE}). The simulation of execute access for (honest) LE need only answer inject message queries for ciphertexts not in $F_{ASTE}^Lmsgs$. Again, the contents of the ciphertext are obtained by extracting the NIZK consistency proof.

The detailed description of $S$ follows. We will consider the simulation strategy task-by-task, and treat different corruption cases separately. This is possible since we assume static corruption, and since the simulation is otherwise indifferent of the corruption pattern. Moreover, we take care of any messages (of relevance) to the hybrid functionality $F_{AS}$ during the description of the simulator, as each message is uniquely associated with a single task.

**Setup of (hybrid) functionalities** Unless noted otherwise, the simulator emulates queries to hybrid functionalities honestly. Of course, the simulator exploits its control by programming the functionalities to some extent (e.g. simulation for NIZK_{AS}, programming $F_\text{RO}$) and exploiting their extractability (i.e. $S$ learns every query to a hybrid functionality). We are explicit about any deviations from honest emulation.

**Trivially simulatable tasks** In all these tasks, $S$ does the necessary bookkeeping so all lists remain synchronized with $F_{ASTE}$. Otherwise, simulation is trivial.

*Party Init* The simulation is trivial, since this task is identical to party init in $F_{AS}$ and the protocol merely proxies to $F_{AS}$.

*Request Warrant* The simulation is trivial, since this task is identical to request warrant in $F_{AS}$ and the protocol merely proxies to $F_{AS}$.

*Get statistics* The simulation is trivial, since this task is identical to get statistics in $F_{AS}$ and the protocol merely proxies to $F_{AS}$.

*Audit* The simulation is trivial, since this task is identical to audit in $F_{AS}$ and the protocol merely proxies to $F_{AS}$.

*Next Period* For honest SO, the simulator does nothing. In fact, $F_{ASTE}$ (and $G_{CLOCK}$) do not explicitly notify the adversary of this task, so this is only indirectly observable (by reading the clock). For corrupted SO, the simulator sends (NEXTPERIOD) to $F_{ASTE}$ whenever SO intends to send (CLOCK-UPDATE) to $G_{CLOCK}$ and SO is currently registered with $G_{CLOCK}$. (Note that the registration state can be checked by sending (IS-REGISTERED, sidC) to $G_{CLOCK}$.)
**Init** Upon receiving \((\text{Init}, P)\) for \(P \in \{J, \text{AU}\}\) from \(\mathcal{F}_{\text{ASTE}}\), \(S\) executes the honest protocol for \(P\) (i.e. sends \((\text{Init})\) to \(\mathcal{F}_{\text{AS}}\) and forwards the output, once it is delivered). Since \(J\) and \(\text{AU}\) are assumed uncorruptable, this completes their simulation.

For honest \(SO\): Upon receiving \((\text{Init}, SO)\), the simulator executes the honest protocol for \(SO\).

For corrupted \(SO\): When \(S\) receives \((\text{Init}, SO)\) in place of \(\mathcal{F}_{\text{AS}}\) from \(SO\), it sends \((\text{Init}, SO)\) to \(\mathcal{F}_{\text{ASTE}}\) on behalf of \(SO\). When the simulated \(\mathcal{F}_{\text{AS}}\) would deliver output \((\text{InitFinished})\) to \(\text{AU}\) (resp. \(J\)), \(S\) allows the output of \((\text{InitFinished})\) for \(\mathcal{F}_{\text{ASTE}}\) to \(\text{AU}\) (resp. \(J\)).

**User Registration** We consider the corruption cases separately. Note that an honest user first writes to \(\mathcal{F}_{\text{BB}}\) and then checks if the write to \(\mathcal{F}_{\text{BB}}\) was successful. This ensures that it only continues the protocol if \(\mathcal{F}_{\text{BB}}\) contains the correct entry. Also note that the leaks to the dummy adversary (as part of simulating \(\mathcal{F}_{\text{AS}}\)) are straightforward to simulate in all cases. If \(U\) is honest, the \(uid\) is chosen by the simulator.

*Case 1: Honest \(U_i\) and honest \(SO\)* Upon receiving \((\text{Register}, U_i)\) from \(\mathcal{F}_{\text{ASTE}}\), the simulator essentially honestly executes the protocol on behalf of both parties, with following additional steps:

1. Let \((\text{pk}_i, \text{sk}_i)\) be the keypair generated for \(U_i\). When \(S\) on behalf of \(\mathcal{F}_{\text{BB}}\) emulates a response \((\text{Retrieve}, U_i, uid_i, uid_i)\) to \(U_i\), i.e. once \(U_i\) knows it has secured its unique \(uid_i\) on \(\mathcal{F}_{\text{BB}}\), \(S\) sends \((\text{Register}, U_i, uid_i)\) to \(\mathcal{F}_{\text{ASTE}}\) (i.e. registers \(U_i\) with that \(uid_i\), as \(uid_i\) is now locked).
2. \(Sim\) also emulates \(\mathcal{F}_{\text{AS}}\) honestly. (The necessary inputs and leakage are trivially known to \(S\).)
3. When a simulated party (i.e. \(U_i\) or \(SO\)) generates output in the protocol, \(S\) allows \(\mathcal{F}_{\text{ASTE}}\) to deliver this party’s output. When \(S\) allows \(\mathcal{F}_{\text{ASTE}}\) to deliver output to \(U_i\), it sets \(\text{ureg}_i := 1\).

*Case 2: Honest \(U_i\) and corrupted \(SO\)* Upon receiving \((\text{Register}, U_i)\) from \(\mathcal{F}_{\text{ASTE}}\), the simulator essentially honestly executes the protocol on behalf of \(U_i\), with the same (additional) steps in Items 1 and 2 and\(^30\) 3 as in Case 1.

Moreover, when \(S\) (in place of \(\mathcal{F}_{\text{AS}}\)) receives \((\text{Register}, uid)\) from \(SO\), it sends \((\text{Register})\) to \(\mathcal{F}_{\text{ASTE}}\) on behalf of \(SO\).

*Case 3: Corrupted \(U_i\) and honest \(SO\)* Upon receiving \((\text{Ready})\) from \(U_i\), the simulator essentially honestly executes the protocol on behalf of \(SO\), with following additional step: After simulating a response \((\text{Retrieve}, U_i, uid, uid')\) of \(\mathcal{F}_{\text{BB}}\) for the query made by \(SO\) in this protocol run, if \(uid' = uid\), \(S\) sends \((\text{Register}, U_i, uid)\) to \(\mathcal{F}_{\text{ASTE}}\). (At this point, \(uid_i\) for \(U_i\) is locked.)

\(^{30}\) Since \(SO\) is corrupted (and of its output has no effect in \(\mathcal{F}_{\text{ASTE}}\)), we can immediately allow it or delay it forever without affecting the simulation.
All other steps are simulated as in the actual protocol. Once $S$ (in place of $F_{AS}$) delivers output (Registered) to $U_i$, set $ureg_i := 1$. As in step 3 above, once SO would generate (accepting) output, $S$ allows $F_{ASTE}$ to deliver output to SO.

**Case 4: Corrupted U and corrupted SO** There is nothing to do for the simulator.

**Update** We consider the corruption cases separately. Note that for honest $U_i$, if $uuper_i$ equals the current time, both the real and the ideal protocol immediately finish without any visible (network) interaction. So there is nothing to simulate (and it’s perfectly simulated). As noted before, $S$ also keeps track of (a simulated) $uuper_i$ for corrupted users.

Note that for honest users, **update** sends $(\text{Update}, U_i, \text{vper})$ to $S$ and $uid_i$ is known (through $F_{BB}$), thus, $S$ can simulate the leaks to the environment in **store secret** of $F_{AS}$ trivially.

**Case 1: Honest U and honest SO** Upon receiving leakage $(\text{Update}, U_i, \text{vper})$, the simulator acts as follows:

- $S$ acts for $U_i$ resp. SO exactly as in the protocol.
- When $U_i$ (resp. SO) send $(\text{STORESECRET}, uid_i, uuper_i)$ (resp. $(\text{STORESECRET}, uuper_i)$) to $F_{AS}$ (whichever sends first), send $(\text{Ok})$ to $F_{ASTE}$.
- $S$ runs the remaining protocol and allows $F_{ASTE}$ to deliver output once a simulated party generates output. Note that, when delivering output for $U_i$, $S$ also sets $uuper_i := \text{vper}$, where $\text{vper}$ is the agreed upon period.
- As noted above, the leakage $(\text{STORESECRET}, \text{vper})$ from $F_{AS}$ to the dummy adversary is trivial to honestly simulate.

**Case 2: Honest U and corrupted SO** Upon receiving leakage $(\text{Update}, U_i, \text{vper})$, the simulator honestly executes the protocol on behalf of $U_i$.\(^{31}\) Note again that, when $S$ would generate protocol output for $U_i$, it allows $F_{ASTE}$ to deliver output and sets $uuper_i := \text{vper}$, where $\text{vper}$ is the agreed upon period. (Observe that $S$ implements $F_{AS}$, and thus learns the inputs of SO to $F_{AS}$ (if any). Thus simulating possible aborts is straightforward.)

**Case 3: Corrupted U and honest SO** Upon receiving $(\text{Update}, uid, \text{vper})$ by $U_i$ on behalf of SO, the simulator honestly executes the protocol in the name of SO. Suppose SO calls (the simulated) $F_{AS}$ with $(\text{STORESECRET}, \text{vper})$. Once output $(\text{SECRETSTORED}, \cdot)$ is delivered to $U_i$ by $S$ in the name of $F_{AS}$. Then set $uuper_i := \text{vper}$.

**Case 4: Corrupted U and corrupted SO** There is nothing to do for the simulator.

\(^{31}\) Note that the (dummy) adversary now receives $(\text{STORESECRET}, uuper_i)$ from the simulated $F_{AS}$. But as noted, $S$ can trivially simulate the leakage.
Encrypt Message For encryption, we only need to consider honest users. When an honest user $U_i$ invokes $F_{\text{ASTE}}$ with $(\text{ENCRYPT}, U_j, m)$, then $S$ receives leakage $(\text{ENCRYPT}, U_j, (|m|, \text{usper}), m', \text{uid}')$ from $F_{\text{ASTE}}$. (Recall that $\text{leak}(m, \text{uid}, \text{usper}) = (|m|, \text{usper})$ in Theorem 3, and that $m' = \bot$ if the receiver $U_j$ is honest and $m' = m$ otherwise.) Then $S$ proceeds as follows:

1. Emulate the real protocol up to the computation of $c$. (In particular, wait (if necessary) until the (dummy) adversary allows output (\text{RETRIEVE}, \ldots) of $F_{\text{IB}}$.)
2. Simulate the ciphertext $c$ by running $\text{PKE}_{\text{AS}}.\text{Sim}(\text{crs}, \text{pk}_j, \text{usper}_i)$.
3. If $m' \neq \bot$, then $U_j$ is corrupted and $S$ immediately runs $\text{PKE}_{\text{AS}}.\text{ExtractMsg}(\text{crs}, \text{pk}_j, c, m, \text{usper}_i)$ on the simulated ciphertext. (In this case $m' = m$, i.e. $S$ learns $m$.)
4. If $\text{uid}' \neq \bot$, then $(\text{uid}_i, \text{usper}_i) \in L_{\text{W-cached}}$ and $S$ runs $\text{PKE}_{\text{AS}}.\text{ExtractMsg}(\text{crs}, \text{uski}, c, \text{uid}_i, \text{usper}_i)$ immediately on the simulated ciphertext. (In this case, $\text{uid}' = \text{uid}_i$, i.e. $S$ learns $U_i$)
5. Store $(c, m', \text{uid}', \text{usper}_i)$ in $L_{\text{msg}}$.
6. Send $(\text{CRYPTTEXT}, U_j, c)$ to $F_{\text{ASTE}}$.

Extract-and-inject ciphertext $c$ In the tasks decrypt ciphertext, prepare access and execute access, the ideal functionality $F_{\text{ASTE}}$ may send an urgent query $(\text{SIMINJECTMsg}, c)$ or $(\text{SIMACCESSINJECT}, c)$ to the adversary to make sense of some ciphertext which is not (yet) contained in $L_{\text{msg}}$. The simulator handles the query as follows:

1. If the NIZK proof $\pi$ of consistency of $c$ is not accepting, then $S$ sends $(\text{INJECTMsg}, c, \bot, \bot, \bot)$ to $F_{\text{ASTE}}$ (and skips the rest).
2. $S$ extracts the NIZK proof $\pi_{\text{cons}}$ (from $c$) to obtain $(m, \text{uid})$. Since the NIZK of $F_{\text{ASTE}}$ is ideal, extraction always succeeds.
3. Add $(c, m, \text{uid}, \text{usper})$ to $L_{\text{msg}}$.
4. Send $(\text{INJECTMsg}, c, m, \text{uid}, \text{usper})$ to $F_{\text{ASTE}}$.

Decrypt Ciphertext For decryption, we only need to consider honest users. Moreover, if $(c, \cdot, \cdots) \in L_{\text{msg}}$, then $S$ learns nothing about the call, as it is local and immediately finishes. (The simulation — doing nothing — is perfect.) Thus, $S$ only handles the case where $(c, \cdot, \cdots) \notin L_{\text{msg}}$. Namely, when $S$ receives $(\text{SIMINJECTMsg}, c)$, then $S$ acts as follows:

1. Let $m' = \text{PKE}_{\text{AS}}.\text{Dec}(\text{crs}, \text{sk}_j, c, \text{usper})$.
2. If $m' = \bot$, send $(\text{INJECTMsg}, \bot, \bot, \bot)$ to $F_{\text{ASTE}}$.
3. Else, run extract-and-inject ciphertext $c$ (as defined above).

For future reference, note that whenever $S$ adds $(c, m, \text{uid}_i, \text{usper}_i)$ to $L_{\text{msg}}$, it is ensured that $F_{\text{ASTE}}$ adds it to $F_{\text{ASTE}} \cdot L_{\text{msg}}$ as well. If $SO$ is corrupted, this is trivial. If $SO$ is honest, $(\text{uid}, \text{usper}) \in L_{\text{usper}}$ is checked during the inject message task in $F_{\text{ASTE}}$, and this check will succeed, since for $(\text{uid}, \text{usper}) \notin L_{\text{usper}}$, it is impossible to generate a (fake) NIZK proof (and thus ciphertext $c$ associated with fake $(\text{uid}, \text{usper})$).
**Prepare Access** We consider simulation of honest and corrupt LE separately. The main difference is, that honest LE only fills $L_{\text{w-cached}}$ at this step, so the simulator has not much to do. On the other hand, a corrupted LE learns $(\text{uid}_i, \text{vper}_i, \text{ask}_i)$ tuples covered by the warrant. The simulator must now use $\text{PKE}_{\text{AS}}.\text{Explain}_{\text{LE}}$ to fix the $\text{uids}$ in the simulated ciphertexts.

**Honest LE** Upon receiving $(\text{Access Leak})$ from $F_{\text{ASTE}}$, $S$ simulates $H_{\text{ASTE}}$ for LE by simulating $F_{\text{AS}}$ on input $(\text{Get Secrets}, W)$ (for unknown $W = \{W_i\}_{i=1}^n$ with $W_i = (\text{uid}_i, \text{vper}_i, \text{meta}_i)$). Note that get secrets (only) leaks $(\text{Get Secrets})$ to the adversary, and then allows the adversary to delay the output. If $F_{\text{ASTE}}$ aborts, $S$ instructs $F_{\text{AS}}$ to abort. Once the adversary allows to deliver output, $S$ allows $F_{\text{ASTE}}$ to deliver output.

**Corrupted LE** Upon receiving $(\text{Get Secrets}, W)$ from LE (in place of $F_{\text{AS}}$), $S$ proceeds as follows: Let $W = \{(\text{uid}_i, \text{vper}_i, \text{meta}_i)\}_{i=1}^n$. Send $(\text{Access Prep}, W)$ to $F_{\text{ASTE}}$ on behalf of LE. If $F_{\text{ASTE}}$ aborts, $S$ instructs $F_{\text{AS}}$ to abort. Else, receive $(\text{Access Leak})$. Before sending $(\text{Get Secrets}, \ldots)$ to LE, do the following:

Allow $F_{\text{ASTE}}$ the output of $(\text{Access Prep Done})$ (to simulated LE, i.e. $S$). This ensures that execute access now decrypts all ciphertexts affected by $W$. Now, $S$ can run a trial-decryption process and explain all affected ciphertexts. For each $(\text{uid}_j, \text{vper}_j, \cdot) \in W$:

- If $(\text{uid}_j, \text{vper}_j) \in L_{\text{vper}}$, add $(\text{uid}_j, \text{vper}_j)$ to $L_{\text{w-cached}}$. Else, skip to next $(\text{uid}, \text{vper})$ pair.
- For each ciphertext with undefined $\text{uid}$, i.e. for all $(c, m, \perp, \text{vper}) \in L_{\text{msgs}}$, $S$ sends $(\text{Access Exec}, c, (\text{uid}_j, \text{vper}_j))$ to $F_{\text{ASTE}}$, and immediately\(^{32}\) receives $(\text{Access Exec Done}, \text{uid})$ from $F_{\text{ASTE}}$. If $\text{uid} \neq \perp$, then
  - If there is no honest $U_i$ with $(U_i, \text{uid}) \in L_U$ and $\text{vper}_i = 1$ then abort the entire simulation with output inconsistent. Else we have $\text{uid}_i = \text{uid}$.
  - Explain $c$ via $\text{PKE}_{\text{AS}}.\text{Explain}_{\text{LE}}(\text{crs}, \text{ask}_i, c, \text{uid}_i, \text{vper}_i)$.
  - Update $(c, m, \perp, \text{vper})$ to $(c, m, \text{uid}, \text{vper})$ in $L_{\text{msgs}}$.

After this, $S$ continues to simulate $F_{\text{AS}}$ honestly. Note that at this point, any simulated ciphertext which would be accessible with warrant $W$ has its $\text{uid}$ explained, and thus decrypts correctly.

**Execute Access** Since execute access is a completely local task, there is nothing to do for corrupted LE. For honest LE the situation is very similar to decrypt ciphertext. The only side-effect observed by the simulator are message injection requests $(\text{Sim Access Inject}, c)$. These are handled as follows:

1. Let $\text{uid} = \text{PKE}_{\text{AS}}.\text{Dec}_{\text{LE}}(\text{crs}, \text{sk}_i, c, \text{vper})$.
2. If $\text{uid} = \perp$, send $(\text{Inject Msg}, \perp, \perp, \perp)$ to $F_{\text{ASTE}}$.
3. Else, run extract-and-inject ciphertext $c$ (as defined above).

\(^{32}\) There will never be a $(\text{Sim Access Inject}, c)$ request, since $c$ was simulated, thus $(c, m, \text{uid}, \text{vper}) \in F_{\text{ASTE}}.L_{\text{msgs}}$. 

132
G.2 Simulator overview

<table>
<thead>
<tr>
<th>Simulator S</th>
</tr>
</thead>
</table>

The simulator has access to the global functionality \( G_{\text{CLOCK}} \). It implements the hybrid functionalities \( F_{\text{CRS}}, F_{\text{BB}}, F_{\text{AS}}, F_{\text{RO}} \).

**Simulation of \( F_{\text{CRS}} \)** \( S \) simulates \( F_{\text{CRS}} \) honestly.

**Simulation of \( F_{\text{BB}} \)** \( S \) simulates \( F_{\text{BB}} \) honestly, but it observes (adversarial) messages.

**Simulation of \( F_{\text{RO}} \)** \( S \) simulates \( F_{\text{RO}} \) honestly, but it also programs \( F_{\text{RO}} \) and observes (adversarial) queries. Programmability and extractability of \( F_{\text{RO}} \) are used as part of the NINCE encryption subroutines, namely \( \text{SKE}_{\text{NCE}} \) in \( \text{PKE}_{\text{AS}} \).

**Simulation of \( F_{\text{AS}} \)** \( S \) simulates \( F_{\text{AS}} \) honestly, but observes (adversarial) messages.

**States of the Parties:**

- User, SO and LE: Cache \( crs \) from \( F_{\text{CRS}} \).
- User \( U_i \):
  - User long-term key: \( (pk_i, sk_i) \), where \( uid_i = pk_i \) is the user identify.
  - User secret (with period): \( (usk_i, uvper_i) \), where \( usk_i \) is the current user secret, and \( oper_i \) the current user validity period.
  - \( ureg_i \in \{0, 1\} \) indicating whether \( U_i \) completed registration (initially 0).
- Public-key caches: \( uid_j = pk_j \) (once retrieved from \( F_{\text{BB}} \)).
- LE: Cache \( L_{\text{W,cached}} \) with entries \( (uid_i, uvper_i, usk_i) \).
- SO, J and AU only need to interact with \( F_{\text{AS}} \).

**Init:**

- This is merely a proxy to **Init** from \( F_{\text{AS}} \). For concreteness, we specify this proxy simulation, but leave other proxies to the reader.

**Init (honest J or AU):**

1. Upon receiving \( (\text{Init}, P) \) for \( P \in \{J, AU\} \) from \( F_{\text{ASTE}} \), leak \( (\text{Init}, P) \) in place of \( F_{\text{AS}} \) to the adversary.
2. When the adversary allows delivery of output \( (\text{InitFinished}) \) to \( P \), allow \( F_{\text{ASTE}} \) delivery of \( (\text{InitFinished}) \) to \( P \).
Init (honest SO or corrupt SO):
1. Upon receiving (Init, SO) from $\mathcal{F}_{\text{ASTE}}$, leak (Init, SO) in place of $\mathcal{F}_{\text{AS}}$ to the adversary.
2. When the adversary allows delivery of output (InitFinished) to SO, allow $\mathcal{F}_{\text{ASTE}}$ delivery of (InitFinished) to SO.

Init (corrupt SO):
1. Upon receiving (Init, SO) from SO in place of $\mathcal{F}_{\text{AS}}$, leak (Init, SO) in place of $\mathcal{F}_{\text{AS}}$ to the adversary.
2. When the adversary allows delivery of output (InitFinished) to SO, allow $\mathcal{F}_{\text{ASTE}}$ delivery of (InitFinished) to SO.

Party Init:
- This is merely a proxy to Party Init from $\mathcal{F}_{\text{AS}}$ (analogous to Init).

User Registration (Honest $U_i$ and honest SO):
- Upon receiving (Register, $U_i$) from $\mathcal{F}_{\text{ASTE}}$, $\mathcal{S}$ emulates $\Pi_{\text{ASTE}}$ for $U_i$:
  1. Send (value) to $\mathcal{F}_{\text{CRS}}$ and cache the result as $\text{crs}$.
  2. Generate and store $(\text{pk}_i, \text{sk}_i) \leftarrow \text{PKE}_{\text{AS}}\text{.Gen}(1^\lambda)$ (unless already stored).
  3. Send (Register, $uid_i$, $uid_i$) to $\mathcal{F}_{\text{BB}}$.
  4. Send (Retrieve, $uid$) to $\mathcal{F}_{\text{BB}}$ and abort if the (eventually delivered) response is not (Retrieve, $U_i$, $uid$, $uid$).
  5. Send (Ready) to SO.
  6. Send (Register, $uid_i$) to $\mathcal{F}_{\text{AS}}$.
  7. Send (Retrieve, $uid$) to $\mathcal{F}_{\text{BB}}$.
  8. Upon output (Registered) from $\mathcal{F}_{\text{AS}}$ to $U_i$, set $\text{ureg}_i := 1$. Moreover, $\mathcal{S}$ allows $\mathcal{F}_{\text{ASTE}}$ output for $U_i$.
- For SO, $\mathcal{S}$ emulates $\Pi_{\text{ASTE}}$ as well:
  1. Upon receiving (Ready) from $U_i$, send (Retrieve, $U_i$) to $\mathcal{F}_{\text{BB}}$ and receive (Retrieve, $U_i$, $uid_i$, $uid'_i$). If $uid_i \neq uid'_i$, abort.
  2. Call $\mathcal{F}_{\text{AS}}$ with input (Register, $uid_i$).
  3. Upon output (Registered, $uid$) from $\mathcal{F}_{\text{AS}}$ to SO, $\mathcal{S}$ allows $\mathcal{F}_{\text{ASTE}}$ output for SO.

User Registration (Honest $U_i$ and corrupted SO):
- For honest $U_i$, the honest simulation is unchanged.
- For SO, $\mathcal{S}$ acts as follows:
  1. Upon receiving (Register, $uid_i$) on behalf of $\mathcal{F}_{\text{AS}}$, $\mathcal{S}$ sends (Register) in place of SO (for this session) to $\mathcal{F}_{\text{ASTE}}$.
  2. Upon output (Registered, $uid$) from $\mathcal{F}_{\text{AS}}$ to SO, $\mathcal{S}$ allows $\mathcal{F}_{\text{ASTE}}$ output for SO.
User Registration (Corrupted $U_i$ and honest SO):

– For honest SO, $S$ executes the honest simulation of SO with following change:
  1. Upon receiving (READ) from $U_i$, $S$ sends (REGISTER) to $F_{ASTE}$ on behalf of $U_i$.
  2. $S$ receives (REGISTER, $U_i$) from $F_{ASTE}$, but delays its response, until:
     3. After delivering (RETRIEVE, $U_i$, uid, $uid'$) to the simulated SO on behalf of (the simulated) $F_{BB}$ and if $uid = uid'$, then $S$ sends (REGISTER, $U_i$, $uid'$) to $F_{ASTE}$. Moreover, $S$ allows output of (REGISTERED, $uid_i$) to $U_i$ and sets its simulated $ureg_i := 1$, completing the registration of $U_i$.

User Registration (Corrupted $U_i$ and honest SO):

– There is nothing to do for $S$.

Next Period:

– There is nothing to do for $S$.

Update (Honest $U_i$ and honest SO):

– Upon receiving (UPDATE, $U_i$, $vper$) from $F_{ASTE}$, $S$ emulates $\Pi_{ASTE}$ for $U_i$:
  1. Query (CLOCK-READ) to $G_{CLOCK}$ and receive (CLOCK-READ, $vper$).
  2. If $uvper = vper$, $U_i$ output (UPDATEREAD, $vper$). Else continue.
  3. Send (UPDATE, $uid_i$, $vper$) to SO.
  4. Send (STORESECRET, $uid_i$, $vper$) to (the simulated) $F_{AS}$.
  5. Upon receiving (SECRETSTORED, $usk'_i$), and update $(usk_i, uvper_i) := (usk'_i, vper)$. Moreover, $S$ allows output of (UPDATEREAD, $vper$) by SO to $F_{ASTE}$.

– For SO, $S$ emulates $\Pi_{ASTE}$ as well:
  1. Upon receiving (UPDATE, $uid$, $vper$) from $U_i$:
  2. Query (CLOCK-READ) to $G_{CLOCK}$ and receive (CLOCK-READ, $vper'$).
     If $vper' \neq vper$, abort.
  3. Send (STORESECRET, $vper$) to $F_{AS}$.
  4. Upon receiving (SECRETSTORED, $uid_i$), the simulator $S$ allows output of (UPDATEREAD, $vper$) to SO by $F_{ASTE}$.

– $S$ observes the behaviour of the $F_{AS}$ calls (in this session) and acts as follows:
  • When $U_i$ (respectively SO) send (STORESECRET, $uid_i$, $uvper_i$) (respectively (STORESECRET, $uvper_i$)) to $F_{AS}$ (whichever sends first), send (OK) to $F_{ASTE}$.
Update (Honest \(U_i\) and corrupt SO):
- Simulation for honest \(U_i\) is unchanged.
- For corrupt SO, no additional special handling required. (Note that \(F_{ASTE}\) still receives (OK) from the simulator \(S\) once (simulated) \(U_i\) or SO send (STORESECRET, \(uvper_i\)) for this session to \(F_{AS}\)).

Update (Corrupt \(U_i\) and honest SO):
- Simulation for honest SO is unchanged.
- For corrupt \(U_i\), no special handling is required. (Note again that \(F_{ASTE}\) still receives (OK) from the simulator \(S\) once \(U_i\) or (simulated) SO send (STORESECRET, \(uid_i, uvper_i\)) or (STORESECRET, \(uvper_i\)) for this session to \(F_{AS}\)).

Update (Corrupt \(U_i\) and honest SO):
- There is nothing to do for \(S\).

Encrypt Message (Honest \(U_i\)):
- Upon receiving (Ciphertext, \(U_j,\text{leak}, m', uid'\)), emulate the honest encryption, but with simulated ciphertexts. First, proceed as honest \(U_i\):
  1. Locally copy (\(usk_i, uvper_i\)) and use this copy.
  2. If \(uvper_i = \perp\), abort.
  3. If \(pk_j\) is not cached, send (RETRIEVE, \(U_j\)) to (the simulated) \(F_{BB}\) and later receive (RETRIEVE, \(U_j, uid_j, uid'_j\)). If \(uid_j \neq uid'_j\) or \(uid = \perp\), abort. Else cache \(pk_j = uid_j\).
- Now, simulate the ciphertext and keep \(S.L_{msgs}\) up-to-date.
  4. Simulate the ciphertext \(c\) as \(c \leftarrow \text{PKE}_{AS,\text{Sim}}(crs, pk_j, uvper_i)\).
  5. If \(m' \neq \perp\), then \(U_j\) is corrupted and the simulator \(S\) immediately runs \(\text{PKE}_{AS,\text{Expln}}(crs, pk_j, c, m, uvper_i)\) on the simulated ciphertext. (In this case \(m' = m\), i.e. \(S\) learns \(m\).)
  6. If \(uid' \neq \perp\), then \((uid_i, uvper_i) \in L_{W-cached}\) and \(S\) immediately runs \(\text{PKE}_{AS,\text{Expln}}(crs, usk_i, c, uid_i, uvper_i)\) on the simulated ciphertext. (In this case, \(uid' = uid_i\), i.e. \(S\) learns \(U_i\).)
  7. Store \((c, m', uid', uvper_i)\) in \(L_{msgs}\).
  8. Send (Ciphertext, \(U_i, c\)) to \(F_{ASTE}\):
     - Note that \(S\) programs \(F_{RO}\) during the \(\text{PKE}_{AS}\) simulation and explanation steps.

Decrypt Ciphertext (Honest \(U_i\)):
- This is handled as part of Simulator Injected Messages.
Simulator Injected Messages:
- Upon receiving (SIMINJECTMSG, c) or (SIMACCESSINJECT, c) from $\mathcal{F}_{\text{ASTE}}$:
  1. If the NIZK proof $\pi$ of consistency of $c$ does not verify, then $\mathcal{S}$ sends
     (INJECTMSG, c, ⊥, ⊥, ⊥) to $\mathcal{F}_{\text{ASTE}}$ (and skips the rest).
  2. $\mathcal{S}$ verifies extracts the NIZK proof $\pi_{\text{con}}$ (from $c$) to obtain $(m, \text{uid})$.
      Since the NIZK of $\mathcal{F}_{\text{ASTE}}$ is ideal, extraction always succeeds.
  3. Add $(c, m, \text{uid}, \text{uvper})$ to $\mathcal{S}.l_{\text{msgs}}$.
  4. Send (INJECTMSG, c, m, uid, uvper) to $\mathcal{F}_{\text{ASTE}}$.

Request Warrant:
- This is merely a proxy to Request Warrant from $\mathcal{F}_{\text{AS}}$.

Prepare Access (Honest LE):
- Upon receiving (ACCESSLEAK):
  1. Simulate sending (GETSECRETS, $W$) to $\mathcal{F}_{\text{AS}}$, that is, send (GETSECRETS) to the adversary on behalf of $\mathcal{F}_{\text{AS}}$.

Prepare Access (Corrupted LE):
- Upon receiving (GETSECRETS, $W$) from LE in place of $\mathcal{F}_{\text{AS}}$:
  1. Send (ACCESSPREP, $W$) to $\mathcal{F}_{\text{ASTE}}$ on behalf of LE.
  2. If $\mathcal{F}_{\text{ASTE}}$ aborts, $\mathcal{S}$ lets (its simulation of) $\mathcal{F}_{\text{AS}}$ to abort as well.
  3. Receive and ignore (ACCESSLEAK) from $\mathcal{F}_{\text{ASTE}}$.
  4. Allow $\mathcal{F}_{\text{ASTE}}$ the output of (ACCESSPREPDONE) (to LE).
  5. Retroactively explain all affected ciphertext using PKE$_{\text{AS}}$.Expln$_{\text{LE}}$ and the Execute Access task of $\mathcal{F}_{\text{ASTE}}$, as defined in Appendix G.1.

Execute Access:
- This is handled as part of Simulator Injected Messages.

Get Statistics:
- This is merely a proxy to Get Statistics from $\mathcal{F}_{\text{AS}}$.

Audit:
- This is merely a proxy to Audit from $\mathcal{F}_{\text{AS}}$.

G.3 Proof of Theorem 3 (Sketch)
In this section, we sketch the proof for Theorem 3, which we state here again.

Theorem 3. Suppose $\Sigma'$ is a EUF-CMA-secure signature scheme, COM is a statistically binding and computationally hiding commitment scheme, PKE$_{\text{NCE}}$ is
a strong NINCE PKE scheme and IND-CCA secure and key-committing. SKE_{NCE}
is a strong NINCE SKE scheme and IND-CCA secure and key-committing. Moreover, suppose all schemes are perfectly correct. Let \text{leak}(m, \text{uid}, \text{wper}) = (|m|, \text{wper}) as the system parameter in \mathcal{F}_{ASTE}, and let \text{fp}, \text{ft} be policy and transparency functions. Then \Pi_{ASTE} UC-realizes \mathcal{F}_{ASTE} in the \{\mathcal{F}_{CRS}, \mathcal{F}_{BB}, \mathcal{F}_{AS}, \mathcal{G}_{CLOCK}\}-hybrid model.

We prove that the real protocol execution and the simulation are indistinguishable using several game hops, starting with the real protocol. We do bookkeeping with the same variables as the simulator and \mathcal{F}_{ASTE} use. To refer to a variable of \mathcal{F}_{ASTE}, we write, for example, \mathcal{F}_{ASTE}.Lmsgs, to distinguish it from other variables maintained by the simulator. For a variable v which is kept by honest parties, or by the simulator in the name of honest and corrupted parties or hybrid functionalities, we simply write v, e.g. ureg, or wper, for (honest or corrupt) user U_i; seldom, we write \mathcal{F}_{AS}.L_U to emphasize that the variable is part of the (simulated) hybrid functionality \mathcal{F}_{AS}. We introduce also certain invariants, which are useful to see that S uses \mathcal{F}_{ASTE} correctly. Moreover, we will gradually ensure that lists and variables of S and \mathcal{F}_{ASTE} “coincide” (to the extent possible).

**Game 0.** This is the real protocol execution.

**Game 1.** We introduce the simulator’s additional variables, namely the lists L_U, L_wper, Lmsgs, etc., but also the variables ureg, wper, (and sk, usk, etc.) both for honest parties and corrupted parties. They are initialized as in \mathcal{F}_{ASTE}. To gradually turn the real world into the simulation, we imagine access to the functionality \mathcal{F}_{ASTE} (for now, simulated by the game itself). Moreover, we treat honest parties like dummy parties, since we can freely see their input to subroutines, and completely modify the behaviour of subroutines, so as to gradually replace the real protocol by interaction with \mathcal{F}_{ASTE} and S.

This change is only conceptual.

**Game 2 (Simulate Init).** For Init, we replace P \in \{AU, J\} by dummy parties which interact with \mathcal{F}_{ASTE} instead of the protocol. In turn we simulate their protocol messages with the simulator. For honest SO, we do the same. For corrupted SO, we also apply the simulation strategy (to correctly interact for \mathcal{F}_{ASTE}).

Since simulation and real protocol are effectively equivalent (namely, up to some protocol steps, they are just calls to \mathcal{F}_{AS} and \mathcal{G}_{CLOCK}), these changes are only conceptual.

**Game 3 (Simulate Party Init).** We let honest parties call \mathcal{F}_{ASTE} instead of running the protocol for party init, run the simulator for this task. Since, simulation is trivial, it is perfectly indistinguishable. Now, the lists L_i and \mathcal{F}_{ASTE}.L_i evidently always coincide.

**Game 4 (Simulate user registration).** We let honest parties call \mathcal{F}_{ASTE} instead of running the protocol for user registration, and run the simulator for this task. We track the variable ureg, exactly as in the simulation. Moreover, we link these with respective user registration task of the imagined \mathcal{F}_{ASTE}.

Concretely, the following 4 cases affect ureg:
– Case 1 (Honest $U_i$ and honest SO): $S$ runs the protocol for $U_i$ and SO. It sets $ureg_i = 1$ when the real protocol outputs (REGISTERED) to $U_i$.

– Case 2 (Honest $U_i$ and corrupted SO): $S$ runs the protocol for $U_i$. It sets $ureg_i = 1$ when the real protocol outputs (REGISTERED) to $U_i$.

– Case 3 (Corrupted $U_i$ and honest SO): $S$ runs the protocol for SO. It sets $ureg_i = 1$ after $F_{AS}$ outputs (REGISTERED) to $U_i$.

– Case 4 (Corrupted $U_i$ and SO): The corrupted parties act unchanged.

By definition of $S$, we could in all cases simply execute $S$ with $F_{ASTE}$ to simulate the real protocol. Indeed, $S$ allows to deliver (REGISTERED) to $U_i$ and SO (immediately) after the real protocol generates output (REGISTERED).

All in all, we may assume the simulation strategy is applied to the user registration task. We see that $ureg_i$ and $F_{ASTE}.ureg_i$ always coincide. The change is only conceptual.

**Game 5 (Simulate update)**. We let honest parties call $F_{ASTE}$ instead of running the protocol for update, and run the simulator for this task. Now, we track $uvper_i$ and $L_{uvper}$. Moreover, we link these with respective update task of the imagined $F_{ASTE}$. As in Game 4, we consider 4 cases of corruption, which affect how the variables $uvper_i$ and $L_{uvper}$ are changed. Let $vper$ be the agreed upon $vper$ (which is the response of $G_{CLOCK}$ to the honest party (or parties)).

– Case 1 (Honest $U_i$ and honest SO): When the real protocol generates output (UPDATEDONE, $vper$) to $U_i$, set $uvper_i = vper$ and add ($uid_i, uvper_i$) to $L_{uvper}$.

– Case 2 (Honest $U_i$ and corrupted SO): Run the protocol for $U_i$. When the real protocol outputs (UPDATEDONE, $vper$) to $U_i$, set $uvper_i = vper$ and add ($uid_i, uvper_i$) to $L_{uvper}$.

– Case 3 (Corrupted $U_i$ and honest SO): Run the protocol for SO. When $F_{AS}$ outputs (SECRETSTORED, $usk$) to $U_i$, set $uvper_i = vper$ and add ($uid_i, uvper_i$) to $L_{uvper}$.

– Case 4 (Corrupted $U_i$ and SO): Let the corrupt parties act unchanged.

As in Game 4, by definition of $S$, we could in all cases simply execute $S$ with $F_{ASTE}$ to simulate the real protocol. Indeed, $S$ allows deliver output to $U_i$ and/or SO right after the real protocol generates output.

The change is indistinguishable. We see that $uvper_i$ (resp. $L_{uvper}$) and $F_{ASTE}.uvper_i$ (resp. $F_{ASTE}.L_{uvper}$) always coincide.

**Game 6 (Simulate (but immediately explain) honest ciphertexts)**. In this game, we replace the computation of honest ciphertexts by simulations, using $PKE_{AS}.Sim$, but immediately explain their contents using $PKE_{AS}.Expln_{RE}$ and $PKE_{AS}.Expln_{LE}$ afterwards. Thus, these ciphertexts still correctly decrypt given the secret keys. To keep track of simulated ciphertexts, the game adds $c$ to $L_{ctsim}$.

This change is indistinguishable by a straightforward reduction to straight-line simulation-extractability of $NIZK_{AS}$, and strong NINCE of $PKE_{AS}$ (or rather, strong NINCE of $PKE_{NCE}$ and $SKE_{NCE}$).

**Game 7 (Consistency of ciphertexts proofs and decryptions)**. In this game, we ensure that any non-simulated ciphertexts $c$ which did not decrypt to $\bot$ by an
honest user U, or honest LE has a consistent NIZK proof. For any $c \notin L_{\text{claim}}$ and which is part of honest party’s ciphertext verification or decryption run (either PKE$_{\text{AS Vfy}}$, or PKE$_{\text{AS DecRE}}$ or PKE$_{\text{AS DecLE}}$) with period $v_{\text{per}}$, we add following abort to the game: Assemble the statement $stämt_c$ for $c$ w.r.t. validity period $v_{\text{per}}$; let $π$ be the proof (contained in $c$). Let NIZK$_{\text{AS Ext}}(stämt_c, π) = \text{wit}$ be the extracted witness, which contains user secret $usk$, user id $uid$, and message $m$:

Invariant 1 (Consistency of PKE$_{\text{AS DecRE}}$): If PKE$_{\text{AS DecRE}}(crs, sk_c, c, v_{\text{per}}) \neq m$, abort with output inconsistent. (Here honest U decrypts.)

Invariant 2 (Consistency of PKE$_{\text{AS DecLE}}$): If PKE$_{\text{AS DecLE}}(crs, usk_c, c, v_{\text{per}}) \neq uid$, abort with output inconsistent. (Here honest LE decrypts.)

Lemma 3. Let $c = (pk, \ldots)$ be a ciphertext which was not generated in the name of an honest user, i.e. $c \notin L_{\text{com}}$. Let $stämt_c$ be the assembled statement for $c$ w.r.t. validity period $v_{\text{per}}$ and proof $π$. Then probability that PKE$_{\text{AS Vfy}}(crs, c, v_{\text{per}}) = 1$, but the extracted witness NIZK$_{\text{AS Ext}}(stämt_c, π) = \text{wit}$ contains user secret $usk$, user id $uid$ or message $m$ with PKE$_{\text{AS DecRE}}(crs, sk_c, c, v_{\text{per}}) \neq m$ or PKE$_{\text{AS DecLE}}(crs, usk_c, c, v_{\text{per}}) \neq uid$ is negligible.

By Lemma 3, the probability that Game 7 outputs inconsistent (which is the only visible change that is made) is negligible. Thus, this change is indistinguishable.

Proof (Proof sketch for Lemma 3). By assumption, the commitment scheme COM is statistically binding. Thus, with overwhelming probability the commitment key $ck$, is perfectly binding, and every commitment contains unique values. We will assume this in the following. Moreover, we show the claim for LE, i.e. PKE$_{\text{AS DecLE}}(crs, usk_c, c, v_{\text{per}}) \neq uid$. The claim for PKE$_{\text{AS DecRE}}(crs, sk_c, c, v_{\text{per}}) \neq m$ follows analogously.

Let $c = (pk, (com_{i,b}, ct^R_{i,b}, ct^L_{i,b})_{i,b}, ((d_{i,\gamma_1}), (π_{\text{con}})))$ and $stämt = (v_{\text{per}}, pk, (com_{i,b}, ct^R_{i,b}, ct^L_{i,b})_{i,b})$, and $γ = \text{RO}(stämt, π_{\text{con}})$.

First, note that using NIZK$_{\text{AS Ext}}(stämt, π)$, we can extract $m$, $uid$, $usk$ from $π$. This is possible, since $c$ was not simulated (i.e. $c \notin L_{\text{claim}}$), and a failure of extraction would break straight-line simulation-extractability (which holds information-theoretically in $F_{\text{AS}}$). Let $w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, γ^R_{i,b}, \gamma^L_{i,b})$ be the unique values contained in $com_{i,b}$. Thus, with overwhelming probability we extract a witness and $uid = uid_{i,0} + uid_{i,1}$ resp. $usk = usk_{i,0} + usk_{i,1}$ holds for all $i = 1, \ldots, ℓ(λ)$.

Now check (by recomputing) whether $\text{SKE}_{\text{NCE}}(usk_{i,b}, uid_{i,b}; r^R_{i,b}) = ct_{i,b}$ for all $i \in \{1, \ldots, ℓ\}$, $b \in \{0, 1\}$. Suppose that for $k$ choices of $i$, at least one share does not satisfy the check, i.e. $\text{SKE}_{\text{NCE}}(usk_{i,b}, uid_{i,b}; r^R_{i,b}) \neq ct_{i,b}$. Then with probability at most $2^{-k}$, the challenge $γ = \text{RO}(stämt, π_{\text{con}}) \in \{0, 1\}^ℓ$ avoids all of them. Thus, with probability $1 - 2^{-k}$, the cut-and-choose instance will be rejected, and hence the probability that a cut-and-choose instance with $k \leq ℓ(λ)/2$ inconsistent indices is not rejected is at most $2^{-(ℓ(λ)/2)}$. Consequently, if at most $Q(λ)$ queries to the random oracle are made (by all parties), the probability that a bad cut-and-choose instance, where only a minority of indices
i have consistent shares, is not rejected is at most \( Q(\lambda) \cdot 2^{1-\ell(\lambda)/2} = \text{neg}(\lambda) \) for polynomial \( Q \). Finally, note that by correctness of \( \text{SKENCE} \), for any index \( i \) for which the ciphertexts \( c_{i,b} \) are consistent, the decryption \( c_{b,\cdot}^{LE} \) with usk \( i,b \) yields the encrypted (and extracted) message \( uid_{i,b} \). Thus, LE can correctly decrypt whenever \( k > \ell(\lambda)/2 \).

**Game 8** (*Ciphertexts as unique handles for honest users.*). In this game, we introduce following additional abort:

**Invariant 3 (Uniqueness of \( c \)):** For any \( c \in L_{ctsim} \), there is only one entry \((c, m, uid, wper)\) in \( L_{msgs} \). More precisely, upon any call to \( \text{PAE}_{AS},Vfy \) (e.g., as part of \( \text{PAE}_{AS},\text{Dec}_{LE} \) or \( \text{PAE}_{AS},\text{Dec}_{RE} \)) succeeds for another \( wper' \neq wper \) or output another \( m \) or \( uid \) than the one contained in \( L_{msgs} \), abort with inconsistent.

The uniqueness of \( c \) allows \( c \) to serve as a unique identifier, which connects incomplete entries \((c, *, *, wper)\) in \( L_{msgs} \) with complete entries \((c, m, uid, wper)\) in \( F_{ASTE} \cdot L_{msgs} \). This will later be useful, to ensure that \( L_{msgs} \) and \( F_{ASTE} \cdot L_{msgs} \) coincide for honest ciphertexts, up to unknown \( m \) and/or \( uid \) for simulated ciphertexts, and prevent any other entry for ciphertext \( c \) in \( L_{msgs} \) and \( F_{ASTE} \cdot L_{msgs} \).

We sketch the proof: Let \( c \in L_{ctsim} \). Observe that honestly generated and simulated ciphertexts have high entropy, thus, the probability that a collision occurs is negligible for honest generation or simulation. Firstly, suppose that \( c \in L_{ctsim} \) and \( \text{PAE}_{AS},Vfy \) accepts for two periods \( wper \neq wper' \). Since the associated statements \( stmt \neq stmt' \) are also inequal, by simulation-extractability (and since only one proof was simulated), we can extract the witness. However, the probability that this happens is negligible, as it is easily reduced to some hardness assumption, e.g. the hiding property of the commitment scheme used in \( \text{PAE}_{AS} \) cut-and-choose. Thus, the accepting \( wper \) is unique, as claimed. Secondly, we show that \( m \) and \( uid \) are fixed for \( c \). If \( c \) were to decrypt via \( \text{PAE}_{AS},\text{Dec}_{RE} \) under more than one user’s \( sk \), or via \( \text{PAE}_{AS},\text{Dec}_{LE} \) under more than one \( usk \), then the key-committing property of \( \text{SKENCE} \) or \( \text{SKENCE} \) would be broken. Thus, this also happens with negligible probability. As decryption is deterministic, \( c \) decrypts under (at most) one key uniquely to \( m \) resp. \( uid \). Overall, \( c \) uniquely identifies \((m, uid, wper)\) as claimed.

**Game 9** (*Consistency checks on uid (for honest U).*) We add additional checks to those of Game 10. Roughly, the game aborts if an honest user’s \( usk \) is extracted from any ciphertext.\(^{33}\) More precisely, let \((c, m, uid, vper, usk)\) be (part of) the (extracted) information during verification or decryption of \( c \) w.r.t. period \( vper \), then the game runs following check:

**Invariant 4:** If \( \text{PAE}_{AS},Vfy(\text{crs}, c, vper) = 1 \) and \( \text{PAE}_{AS},\text{Dec}_{LE}(\text{crs}, usk, c, vper) = uid \) and \((U, uid) \in L_{U} \) for honest \( U \), then abort with inconsistent. (Recall that \( c \notin L_{ctsim} \))

\(^{33}\) Note that extraction is never used on ciphertexts in \( L_{ctsim} \). Hence, such a ciphertext would “frame” an honest user.
The probability that such an abort happens is negligible, as it can be reduced to the IND-CCA security of $\text{PKE}_{\text{NCE}}$ and key-committing of $\text{SKE}_{\text{NCE}}$. More concretely, if this happens for some user $U_i$, then (from the extracted NIZK proof) the user’s long term secret key $sk_j$ is recovered, since the NIZK proves (in particular) knowledge of $sk$ with $(uid, sk) = \text{PKE}_{\text{AS}}.\text{Gen}(1^\lambda; sk)$. Moreover, since $\text{SKE}_{\text{NCE}}$ is key-committing, it is hard to find $usk \neq usk'$ with $\text{PKE}_{\text{AS}}.\text{Dec}_{\text{LE}}(crs, usk', c, vper) \neq \bot$.

Game 10 (Consistency checks on uid for honest SO). We add additional checks to those of Game 7 if SO is honest. Namely, after (successful) extraction, add the following invariants, where $(c, m, uid, vper)$ is (part of) the (extracted) information during verification or decryption of $c$ w.r.t. period $vper$:

**Invariant 5 (Period consistency):** In the above situation, if $(uid, vper) \notin L_{vper}$, abort with invalid period. (If SO is honest, an update in period $vper$ is required to produce valid ciphertexts w.r.t. $vper$.)

**Invariant 6 (Registration consistency):** In the above situation, if $(., uid) \notin L_U$, abort with invalid period. (If SO is honest, a user must be registered to produce any valid ciphertexts.)

These invariants are immediately implied by the guarantees of NIZK$_{\text{AS}}$, or rather, the guarantees of the Verify task of $\mathcal{F}_{\text{AS}}$. Indeed, as long as SO is honest, Verify checks invariant 5 and returns 0 if it is not satisfied. Moreover, for honest SO, invariant 5 immediately implies invariant 6, since unregistered users cannot have completed update (in any period).

Game 11 (Almost simulate encrypt message). In this game, we use the simulation to handle encrypt message for an honest user $U_i$. However, as in Game 6, all ciphertext are still explained immediately after their simulation. Observe that the respective tuples are added to the lists $L_{\text{msgs}}$ (resp. $\mathcal{F}_{\text{ASTE}}.L_{\text{msgs}}$), are exactly as they would be by $S$ (resp. $\mathcal{F}_{\text{ASTE}}$). (But not everything related to $L_{\text{msgs}}$ is handled as in (or by) $\mathcal{F}_{\text{ASTE}}$. This will take some more steps.)

This change is merely conceptual and thus perfectly indistinguishable.

Game 12 (Simulate decrypt message). In this game, we use the simulation to handle decrypt message for an honest user $U_j$. The lists $L_{\text{msgs}}$ and $\mathcal{F}_{\text{ASTE}}.L_{\text{msgs}}$ remain synchronized.

This change is perfectly indistinguishable. Indeed, only if the ciphertext which $U_j$ decrypts is not part of $\mathcal{F}_{\text{ASTE}}.L_{\text{msgs}}$, will there be any change, namely, the $(\text{SIMInjectMsg}, c)$ request. We argue indistinguishability as follows:

**Honest SO:** In this case, invariants 1–6 ensure that simulation and real protocol are indistinguishable. More concretely: By invariants 1–4, we already verify whether $\text{PKE}_{\text{AS}}.\text{Dec}_{\text{LE}}$ and extracted NIZK proof agree on $c$ (and abort the entire game otherwise). Thus, the only possible difference would be if inject message fails to inject since the provided $(c, m, uid, vper)$ fails the check that $(uid, vper) \in L_{vper}$ or $uid$ belong to an honest user. This is excluded by invariants 5–6 Thus, there is no change in behaviour.

**Corrupted SO:** In this case, invariants 1–4 still ensure that simulation can find proper inputs to the request $(\text{SIMInjectMsg}, c)$. Invariants 5 and 6 do
not apply for corrupted SO, but the inject message subroutine does not
enforce them in this case either, it only checks that uid does not belong to
an honest user, which is still ensured by invariant 4.

Game 13 (Simulating request warrant and switching to \(F_{\text{ASTE-L_W}}\)). In
this game, we simulate the request warrant task in \(F_{\text{ASTE}}\); and switch all
uses of \(L_W\) (of the hybrid functionality \(F_{\text{AS}}\)) to uses of \(F_{\text{ASTE-L_W}}\). This works
without any problem, since the request warrant task of \(F_{\text{ASTE}}\) is merely a
proxy to the task of \(F_{\text{AS}}\). Thus, corrupted parties are handled as usual. For
honest parties, we switch their interface to \(F_{\text{ASTE}}\) and plug in the simulation
(exactly as in Game 2). Clearly, \(L_W\) and \(F_{\text{ASTE-L_W}}\) always coincide. Moreover,
the change is perfectly indistinguishable.

Game 14 (Simulate get statistics). Since get statistics in \(F_{\text{ASTE}}\) is
merely a proxy to \(F_{\text{AS}}\), simulation is trivial and perfectly indistinguishable,
since we have replaced the use of \(L_{\text{uperv}}\) with \(F_{\text{ASTE-L_{uperv}}}\) globally (in Game 13).

Game 15 (Simulate audit). Since audit in \(F_{\text{ASTE}}\) is merely a proxy to
\(F_{\text{AS}}\), simulation is trivial and perfectly indistinguishable, since we have replaced
the use of \(L_W\) with \(F_{\text{ASTE-L_W}}\) globally (in Game 13).

Game 16 (Almost simulate prepare access). In this game, we use the
simulator for prepare access, except that still all ciphertext are fully explained
to begin with (by Game 6). Observe that, since \(L_{\text{uperv}}\) and \(F_{\text{ASTE-L_{uperv}}}\) coincide
by Game 5, also \(L_{W\text{-cached}}\) and \(F_{\text{ASTE-L_{W-cached}}}\) coincide.

This change is indistinguishable. To see this, we distinguish two cases:

Honest LE In this case, the simulation is perfect and only requires a little book-
keeping between \(F_{\text{ASTE}}\) and the (simulated) \(F_{\text{AS}}\).

Corrupted LE The simulation acts whenever it receives (GetSecrets, \(W\)) in
place of \(F_{\text{AS}}\). Namely, the simulator sends (AccessPrep, \(W\)) to \(F_{\text{ASTE}}\) on behalf of LE,
and receives (AccessLeak) or \(F_{\text{ASTE}}\) aborts (in which case \(S\)
causes \(F_{\text{AS}}\) to abort). Recall that simulation must ensure that \(L_{W\text{-cached}}\) and
\(F_{\text{ASTE-L_{W-cached}}}\) coincide, and the above strategy ensures exactly that. Before
(GotSecrets, \ldots) would be delivered to LE, \(S\) does the following: \(S\) allows
\(F_{\text{ASTE}}\) to deliver output. Now, the simulator runs execute access on all simul-
ated ciphertexts in \(L_{\text{msgs}}\) (i.e. the subset \(L_{\text{ctsim}}\)) with all possible \((uid, vper)\)
where \((uid, vper) \in W\). If a ciphertext \(c\) is deanonymized with \((uid, vper)\), \(S\)
runs \(PKE_{\text{AS}} \cdot \text{Expln}_{\text{LE}}(crs, usk, c, uid, vper)\) and updates \((c, m, uid, vper)\) in \(L_{\text{msgs}}\).

Observe that, in the current game, the calls to \(PKE_{\text{AS}} \cdot \text{Expln}_{\text{LE}}\) do nothing,
because all (honest) ciphertexts are already explained, cf. Game 6. However, this
will change in future games, hence it is important to explain all yet unexplained
ciphertexts during simulation of execute access. Also observe that, with this
game’s changes, \(L_{W\text{-cached}}\) and \(F_{\text{ASTE-L_{W-cached}}}\) always coincide. This change is
perfectly indistinguishable and only conceptual.

Game 17 (Simulate execute access). In this game, we use the simulation
to handle execute access. The lists \(L_{\text{msgs}}\) and \(F_{\text{ASTE-L_{msgs}}}\) continue to always
coincide after this change.
This change is perfectly indistinguishable. Indeed, it is only bookkeeping. This is only relevant for honest LE. For \((\text{AccessExec}, c, (\text{uid}, \text{vper}))\), the check \((\text{uid}, \text{vper}) \in \text{L}_W\text{-cached} \text{ and } (\text{uid}, \text{vper}) \in \text{F}_{\text{ASTE}}\text{-L}_W\text{-cached}\) are identical, since \(\text{L}_W\text{-cached} \text{ and } \text{F}_{\text{ASTE}}\text{-L}_W\text{-cached}\) coincide (by Game 16). The task is local, except if \(c \notin \text{L}_\text{msgs}\), thus it suffices to argue that an occurrence of \((\text{InjectMsg}, c)\) does not result in a difference between the output of the protocol \text{execute access}\ and the output of \text{F}_{\text{ASTE}}.\ This is almost identical to the argument for \text{decrypt message} in Game 12 (for corrupted SO). Indeed, by invariants 1–4, no difference will occur (or the game would have aborted).

\textbf{Remark 11.} At this point, the game handles all lists, in particular \(\text{L}_\text{msgs}\) and \(\text{F}_{\text{ASTE}}\text{-L}_\text{msgs}\), exactly as \(S\) (resp. \(\text{F}_{\text{ASTE}}\)). The only difference compared to a full simulation is the ciphertext handling in \text{encrypt message} and \text{prepare access}. For \text{encrypt message}, Game 11 still uses \(\text{PKE}_{\text{AS}}\text{-ExPln}_{\text{RE}}\) and \(\text{PKE}_{\text{AS}}\text{-ExPln}_{\text{LE}}\) immediately after simulating a ciphertext, and this has not been removed. For \text{prepare access}, Game 16, the call to \(\text{PKE}_{\text{AS}}\text{-ExPln}_{\text{LE}}\) is skipped (as all ciphertext are already explained). We complete the simulation of these task in the following steps.

\textbf{Game 18 (Simulate honest ciphertext uids).} In this game, we complete the simulation of \text{prepare access}. Instead of running \(\text{PKE}_{\text{AS}}\text{-ExPln}_{\text{LE}}\) on a honestly generated ciphertexts immediately after their simulation (in \text{encrypt message}), it is run only when necessary.

This change is indistinguishable. To see this, consider two cases:

\begin{itemize}
  \item \textbf{Case 1 (Honest LE).} There is no difference in this case.
  \item \textbf{Case 2 (Corrupt LE).} Whenever \(\text{F}_{\text{AS}}\) releases secrets due to a warrant, the game programs all (not yet programmed) affected honest ciphertexts, as introduced in Game 16. Whether an honest ciphertext is affected can be seen from \(\text{F}_{\text{ASTE}}\text{-L}_\text{msgs}\) directly, but since \(S\) has no access to \(\text{F}_{\text{ASTE}}\text{-L}_\text{msgs}\), it uses the roundabout way of repeatedly running trial calls to \text{execute access} until all possible cases are exhausted. Moreover, the case of \((\text{SimAccessInject}, c)\) is handled correctly. Furthermore, fresh honest encryptions during \text{encrypt message} are always explained immediately by \(S\) if they lie in a surveilled epoch, i.e. if \((\text{uid}_i, \text{vper}_i) \in \text{L}_W\text{-cached}\).
\end{itemize}

The indistinguishability in Case 2 reduces to strong NINCE-security of \(\text{PKE}_{\text{AS}}\). The last remaining difference in ciphertext handling is addressed next.

\textbf{Game 19 (Simulate honest ciphertext m).} In this game, we complete the simulation of \text{encrypt message}. Instead running \(\text{PKE}_{\text{AS}}\text{-ExPln}_{\text{RE}}\) on a honestly generated ciphertexts immediately after their simulation, it is run only when necessary. That is, it is run only when the receiver is corrupted.

This change indistinguishable, as it reduces to the strong NINCE-security of \(\text{PKE}_{\text{AS}}\).

\textbf{Game 20 (Tying up loose ends).} At this point, the game handles everything pertaining ciphertexts (in particular filling the list \(\text{L}_\text{msgs}\)) by simply running \(S\). In other words, the simulator is used everywhere in the game. The only
required change to remove superfluous aborts (such as inconsistent), which never occur in $S$. Since the probability of these aborts is negligible, this change is indistinguishable. Altogether, this proves our claimed indistinguishability of real execution (Game 0) and ideal execution (Game 20).

Remark 12 (Overview of invariants in terms of $F_{\text{ASTE}}$). Besides invariants 1–6 introduced in the games, one can also view invariants in terms of the variables kept for the simulator and for $F_{\text{ASTE}}$. Over the course of several games, it is ensured step by step that these “invariants” hold.

- $\text{ureg}_i$ and $F_{\text{ASTE}}$: $\text{ureg}_i$ always coincide.
- $\text{vper}_i$ (resp. $L_{\text{vper}}$) and $F_{\text{ASTE}}$: $\text{vper}_i$ (resp. $F_{\text{ASTE}}$: $L_{\text{vper}}$) always coincide.
- $\text{ureg}_i = 0 \implies L_{\text{vper}}$ contains no $\text{uid}_i$, assuming honest SO. (Without registration, users cannot run update.)
- $(\text{uid}_i, \text{vper}) \notin L_{\text{vper}} \implies$ no messages in $L_{\text{msgs}}$ from $U_i$ in period $\text{vper}$, assuming honest SO. (Users can only encrypt in periods for which they have run update.)
- If $U_i$ is not corrupted, then no extracted (thus non-simulated) ciphertexts has the $\text{uid}$ of a an honest user.
- $c$ is a unique identifier in $L_{\text{msgs}}$ for “honest” ciphertexts.
- If $(c, m', \text{uid}', \text{vper}) \in L_{\text{msgs}}$ then $\exists m, \text{uid}, \text{vper}: (c, m, \text{uid}, \text{vper}) \in L_{\text{msgs}}$ where $m' = m$ (resp. $\text{uid}' = \text{uid}$) or $m' = \bot$ (resp. $\text{uid}' = \bot$).
- Any simulated ciphertext $c$ is explained only when necessary, i.e. corruption of receiver or corrupted LE and surveillance of the user in the respective period of $c$ (once $\text{usk}$’s are “released” via the (simulated) hybrid functionality $F_{\text{AS}}$).