Universally Composable Auditable Surveillance

Valerie Fetzer[1,4], Michael Klooß[2]*, Jörn Müller-Quade[1,4], Markus Raiber[1,4], and Andy Rupp[3,4]

[1] Karlsruhe Institute of Technology, Germany
{valerie.fetzer, joern.mueller-quade, markus.raiber}@kit.edu
[2] Aalto University, Finland
michael.klooss@aalto.fi
[3] University of Luxembourg, Luxembourg
andy.rupp@uni.lu
[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.

* 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: Π_{AS} UC-realizes $\mathcal{F}_{\mathrm{AS}}$ . . . 91
F Security Proof: Π_{AD} UC/YOSO-realizes $\mathcal{F}_{\mathrm{AD}}$ . . . 113
G Security Proof: Π_{ASTE} UC-realizes $\mathcal{F}_{\mathrm{ASTE}}$ . . . 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].[5] 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, $\mathcal{F}_{\mathrm{AS}}$. The functionality $\mathcal{F}_{\mathrm{AS}}$ 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 Π_{AS} which UC-realizes $\mathcal{F}_{\mathrm{AS}}$. It uses several cryptographic building blocks like commitments, signatures and zero-knowledge proofs and is based on an ideal functionality for auditable decryption, $\mathcal{F}_{\mathrm{AD}}$, 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 $\mathcal{F}_{\mathrm{AD}}$ is also modeled in the UC framework to enable a flexible use of this building block and we provide a protocol Π_{AD} that UC-realizes it. We cast our protocol Π_{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, $\mathcal{F}_{\mathrm{ASTE}}$, for Auditably Sender-Traceable Encryption (ASTE). This functionality demonstrates the usefulness of $\mathcal{F}_{\mathrm{AS}}$ as a building block. Effectively, $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathcal{F}_{\mathrm{AS}}$, 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 $\mathcal{F}_{\mathrm{AD}}$ for auditable decryption, which may be of independent interest. We stress however, that the realization of $\mathcal{F}_{\mathrm{AS}}$ is not in any way restricted to this setting.
3. We demonstrate the applicability of $\mathcal{F}_{\mathrm{AS}}$ by building $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathcal{F}_{\mathrm{AS}}$ 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 $\mathcal{F}_{\mathrm{AS}}$ 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, $\mathcal{F}_{\mathrm{AS}}$ 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 $(\mathit{uid}, \mathit{secret})$, *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 $\mathcal{F}_{\mathrm{ASTE}}$, 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 $\mathcal{F}_{\mathrm{ASTE}}$, 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

Paper	Lawful-only[1]	Retro-spective[2]	Silent[3]	No EWE needed[4]	Public Statistics[5]	UC-secure[6]	Flexible Framework[7]
[44], [27]	no	no	yes	yes	no	no	no
[22]	no	yes	yes	yes	no	no	yes
[11]	no	yes	no	yes	yes	yes	no
[47]	no	yes	yes	yes	yes	no	partially[8]
[38]	yes	yes	no	yes	no	no	partially[8]
[34]	yes	yes	yes	no	yes	yes	no
Our System	yes	yes	yes	yes	yes	yes	yes

[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.

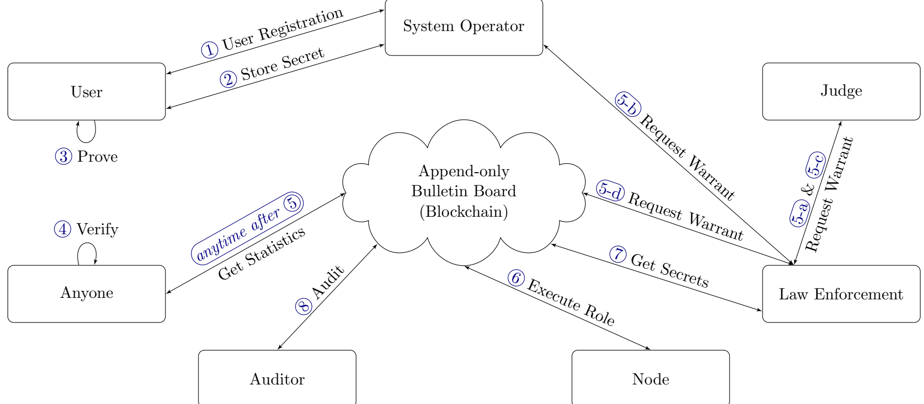

Fig. 1: High Level System Overview. The numbers represent the intended order of task execution.

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 $\mathcal{F}_{\text{AS}}$ to an anonymous message transfer service and instantiate it with the protocols Π_{AS} and Π_{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 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.[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[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.[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.

[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.

[11] This security property holds for a user colluding with other users and blockchain nodes but not one colluding with SO or LE.

[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 $\mathcal{F}_{\mathrm{AS}}$, 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 $\mathcal{F}_{\mathrm{AS}}$ in Figs. 2 and 3. Note

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
W^{pub}	The information about the warrant that should be known to the public

Table 2: Variables used in $\mathcal{F}_{\text{AS}}$

that the functionality implicitly checks the *pid* of calling parties: If the functionality description states "Input P: $(\ldots)$" 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 $\mathcal{F}_{\text{AS}}$ is a trustworthy incorruptible entity, we do not need to encrypt it and directly store the secret inside $\mathcal{F}_{\text{AS}}$. $\mathcal{F}_{\text{AS}}$ first checks that the user is registered in the system and has not yet stored a secret for the current period. Then, $\mathcal{F}_{\text{AS}}$ 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 $\mathcal{F}_{\text{AS}}$. For an example on how to use $\mathcal{F}_{\text{AS}}$ 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. $\mathcal{F}_{\text{AS}}$ 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. $\mathcal{F}_{\text{AS}}$ 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.

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

System Parameters:

- f_t — Transparency function. Outputs public information of a warrant. Interface is $W^{\mathrm{pub}} \leftarrow f_t(W)$.
- f_p — Policy Function. Checks whether a given warrant is allowed by system policy. Interface is $\{0,1\} \leftarrow f_p(W)$.
- $\mathcal{S}$ — Space of secrets
- $\mathcal{R}$ — an NP relation for statements about stored secrets: Contains $(stmt_R, wit_R) \in \mathcal{R}$ pairs where $stmt_R$ is a statement and wit_R is a witness for that statement
- System pids: pid_{SO}, pid_{J}, pid_{AU}

Functionality State:

- $\mathsf{L_I}$: List of initialized parties (initially empty). Contains (pid) entries.
- $\mathsf{L_U}$: List of registered users (initially empty). Contains (pid, uid) pairs.
- $\mathsf{L_S}$: List of stored secrets (initially empty). Contains $(uid_i, vper_i, secret_i)$ entries.
- L_W: List of warrants that were requested by LE from J (initially empty). Entries are of the form (W, b), where b is a bit that states whether the warrant was granted or denied by the judge.
- L_π: List of proofs (initially empty). Entries are of the form $(stmt_R, wit_R, \pi, b)$, where the bit b states whether the relation is fulfilled or not.

System Init:

- Input SO & J & AU: $(\textsc{Init}, \mathrm{SO/J/AU})$
  - 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 $\mathsf{L_I}$, $\mathsf{L_U}$, $\mathsf{L_S}$, L_W, and L_π
  2. Send $(\textsc{Init})$ to the adversary
  3. Add pid_{SO} and pid_{J} to $\mathsf{L_I}$
- Output SO & J & AU: $(\textsc{InitFinished})$

Party Init:

- Input some Party P: $(\textsc{PInit})$
- Behavior:
  1. If $pid \notin \mathsf{L_I}$, store pid in $\mathsf{L_I}$
  2. Send $(\textsc{PInit})$ to the adversary
- Output to P: $(\textsc{PInitFinished})$

User Registration:

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

Store Secret:

- Input U: $(\textsc{StoreSecret}, uid, vper)$
- Input SO: $(\textsc{StoreSecret}, vper')$
- Behavior:
  1. If SO is corrupted (and U is honest): As soon as U gave input, send $(\textsc{StoreSecret}, vper)$ to the adversary and then wait for input from SO
  2. If $vper \neq vper'$, abort
  3. Check if user is registered: If $(pid, uid) \notin \mathsf{L_U}$, abort.
  4. Check if user already registered a secret for the current validity period: If there exists an entry $(uid, vper, \cdot)$ in the list of stored secrets $\mathsf{L_S}$, abort
  5. Generate secret: $secret \xleftarrow{r} \mathcal{S}$
  6. Store $(uid, vper, secret)$ in the list of stored secrets $\mathsf{L_S}$
- Output U: $(\textsc{SecretStored}, secret)$
- Output SO: $(\textsc{SecretStored}, uid)$

Request Warrant:

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

Fig. 2: The ideal functionality $\mathcal{F}_{\mathrm{AS}}$

Functionality $\mathcal{F}_{\text{AS}}$ (continued)

Get Secrets:
- Input LE: (GetSecrets, W)
- Behavior:
  1. Check if warrant was granted by Judge: If no entry $(W, 1)$ exists in L_W, abort. (Warrant either not requested or not granted)
  2. Parse warrant: $(W_1, \ldots, W_v) \leftarrow W$ and $(uid_i, vper_i, meta_i) \leftarrow W_i$
  3. For each i from 1 to v:
     (a) Check if secret is stored: Get $secret_i$ for which an entry $(uid_i, vper_i, secret_i)$ exists in the list of stored secrets L_S. If none exists, set $secret_i = \bot$.
  4. If LE is honest: Send (GetSecrets) to the adversary
- Output LE: (GotSecrets, $(secret_1, \ldots, secret_v)$)

Get Statistics:
- Input some Party P: (GetStatistics)
- Behavior:
  1. Send (GetStatistics) to the adversary
  2. Initialize empty list $\mathsf{L}_{\mathsf{Stats}}$
  3. For every $(W, b) \in \mathsf{L}_W$ with $b = 1$:
     (a) Apply transparency function to warrant: $W^{\text{pub}} \leftarrow f_t(W)$
     (b) Append W^{pub} to $\mathsf{L}_{\mathsf{Stats}}$
- Output to P: (GotStatistics, $\mathsf{L}_{\mathsf{Stats}}$)

Audit:
- Input AU: (AuditRequest)
- Behavior:
  1. Send (AuditRequest) to the adversary
  2. Initialize empty list L_{AU}
  3. For every $(W, b) \in \mathsf{L}_W$ with $b = 1$, append W to L_{AU}
- Output to AU: (AuditAnswer, L_{AU})

Prove:
- Input U: (Prove, $stmt_R$, wit_R)
- Behavior:
  1. Parse $(uid, secret, wit') \coloneqq wit_R$
  2. Parse $(vper, stmt') \coloneqq stmt_R$
  3. Let pid_{U} be the pid of the party calling this task
  4. If $\mathcal{R}(stmt_R, wit_R) \neq 1$ or $(pid_{\text{U}}, uid) \notin \mathsf{L}_\mathsf{U}$ or $(uid, vper, secret) \notin \mathsf{L}_{\mathsf{Secrets}}$, then abort.
  5. Send urgent request (Prove, $stmt_R$) to the adversary.
  6. Wait for immediate response (Proof, π) from the adversary
  7. Store $(stmt_R, wit_R, \pi, 1)$ in L_π
- Output U (immediate): (Proof, $stmt_R$, wit_R, π)

Verify:
- Input some party P: (Verify, $stmt_R$, π)
- Behavior:
  1. If $pid \notin \mathsf{L}_\mathsf{I}$, abort (Verifier did not initialize)
  2. For $(stmt_R, \pi)$, check if there exists $(stmt_R, wit_R, \pi, b) \in \mathsf{L}_\pi$
  3. If entry exists
     (a) (Immediately) output (Verification, $stmt_R, \pi, b$) to P
  4. If entry does not exist
     (a) Send urgent request (Verify, $stmt_R, \pi$) to the adversary
     (b) Wait for immediate response (witness, wit_R) from the adversary
     (c) Set $b \leftarrow \mathcal{R}(stmt_R, wit_R)$
     (d) Only if SO is honest:
        i. Parse $(uid, secret, wit') \coloneqq wit_R$
        ii. Parse $(vper, stmt') \coloneqq stmt_R$
        iii. If $(uid, vper, secret) \notin \mathsf{L}_{\mathsf{Secrets}}$, set $b = 0$ (User has not stored a secret)
     (e) Store $(stmt_R, wit_R, \pi, b)$ in L_π
     (f) Prepare the output message (Verification, $stmt_R, \pi, b$)
- Output to P (immediate): (Verification, $stmt_R, \pi, b$)

Fig. 3: The ideal functionality $\mathcal{F}_{\text{AS}}$ (continued)

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 $\mathcal{F}_{\text{AS}}$. The Prove task allows a prover to generate a proof π for a statement $stmt_R$ in some NP-relation $\mathcal{R}$, 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 $\mathcal{F}_{\mathrm{AS}}$ (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 Π_{AS} for Realizing $\mathcal{F}_{\mathrm{AS}}$

The functionality $\mathcal{F}_{\mathrm{AS}}$ 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 Π_{AS} *in the real world* that achieves the same security guarantees as $\mathcal{F}_{\mathrm{AS}}$. We later prove that our constructed protocol Π_{AS} UC-realizes $\mathcal{F}_{\mathrm{AS}}$ in the $\{\mathcal{F}_{\mathrm{AD}}, \mathcal{F}_{\mathrm{CRS}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{G}_{\mathrm{CLOCK}}\}$-hybrid model. As a setup assumption, we use the well-known functionality $\mathcal{F}_{\mathrm{CRS}}$ 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 $\mathcal{F}_{\mathrm{BB}}$, where any party can register a single (uid, v) pair associated with its identity, where the uids 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 $\mathcal{F}_{\mathrm{AD}}$, 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 $\mathcal{F}_{\mathrm{AD}}$ 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 $\mathcal{F}_{\mathrm{AD}}$ utilizes a global clock functionality $\mathcal{G}_{\mathrm{CLOCK}}$ (from [5]) to model time.

We now elaborate on some of the core techniques we use in Π_{AS} and refer for the complete description of Π_{AS} to Appendix B.2 and for the used hybrid functionalities (except $\mathcal{F}_{\mathrm{AD}}$) to Appendix B.1. The hybrid functionality $\mathcal{F}_{\mathrm{AD}}$ 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 $\mathcal{F}_{\mathrm{AD}}$ together. The functionality $\mathcal{F}_{\mathrm{AD}}$ is then used to provide all parties with access to the needed public keys.

User Registration. Users create a signing keypair $(\mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}})$ during registration. To ensure at most one account per user, we use an idealized bulletin board $\mathcal{F}_{\mathrm{BB}}$, where any party can register a single $(uid, \mathsf{vk}_{\mathrm{U}})$ pair associated with its identity, where the *uid*s 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 $\mathcal{F}_{\mathrm{AD}}$, 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 $\mathcal{F}_{\mathrm{AD}}$ 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 $\underset{\smile}{sec_1}$.[13] Afterwards, LE provides J with the ciphertexts *ct* to get a signature on $\widetilde{W}$, 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.[14] LE can now utilize the hybrid functionality $\mathcal{F}_{\text{AD}}$ to request the decryption of all ciphertexts corresponding to the warrant $\widetilde{W}$.

Get Secrets. After $\mathcal{F}_{\text{AD}}$ processed the decryption request, LE can retrieve the partial secrets sec_2 for a warrant from $\mathcal{F}_{\text{AD}}$. Of course, $\mathcal{F}_{\text{AD}}$ verified the validity of all requests and partial secrets for invalid requests can not 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 $\mathcal{F}_{\text{AD}}$ knows all requested[15] warrants, $\mathcal{F}_{\text{AD}}$ 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 π 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 Π_{AS} UC-realizes $\mathcal{F}_{\text{AS}}$. In particular, we show the following theorem.

Theorem 1. *Π_{AS} UC-realizes $\mathcal{F}_{\text{AS}}$ in the $\{\mathcal{F}_{\text{AD}}, \mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BB}}, \mathcal{G}_{\text{CLOCK}}\}$-hybrid model under the assumptions that* COM *is a (computationally) hiding, (statistically) binding and (dual-mode) extractable and equivocable commitment scheme, Σ 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.*

4.2 Decrypting Secrets with $\mathcal{F}_{\text{AD}}$

We now want to briefly describe our ideal auditable decryption functionality $\mathcal{F}_{\text{AD}}$. 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 $\mathcal{G}_{\text{CLOCK}}$ 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 $\mathcal{G}_{\text{CLOCK}}$. $\mathcal{F}_{\text{AD}}$ starts by creating an encryption keypair and registering with the global clock $\mathcal{G}_{\text{CLOCK}}$. In Π_{AS} SO and J each create a signing keypair and pass their public verification keys to $\mathcal{F}_{\text{AD}}$, where they are stored. There are also tasks to provide the parties in Π_{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.

[14] 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.

[15] 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 Π_{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 Π_{AD} for Realizing $\mathcal{F}_{\text{AD}}$

We cast our protocol Π_{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 $\mathcal{G}_{\text{CLOCK}}$.
2. $\mathcal{F}_{\text{BCRA}}$ 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 $\mathcal{F}_{\text{BCRA}}$ 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,[16] 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 $\mathcal{F}_{\text{BCRA}}$ 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 $\mathcal{F}_{\text{AD}}$ in the $\{\mathcal{F}_{\text{BCRA}}, \mathcal{F}_{\text{CRS}}, \mathcal{G}_{\text{CLOCK}}\}$-hybrid model.

Init, Get Tasks and Interaction with $\mathcal{G}_{\text{CLOCK}}$. During initialization, AU creates an encryption keypair and SO, J and AU post their public keys to the ledger $\mathcal{F}_{\text{BCRA}}$. The common reference string is obtained by querying $\mathcal{F}_{\text{CRS}}$, the other Get Tasks are handled by reading from the ledger $\mathcal{F}_{\text{BCRA}}$. All honest nodes N register with the clock $\mathcal{G}_{\text{CLOCK}}$ upon first activation. Additionally, the distributed key generation protocol from [23] is run by the first roles assigned through $\mathcal{F}_{\text{BCRA}}$, 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 $\widetilde{W}$, instead of simply posting the warrant to $\mathcal{F}_{\text{BCRA}}$, LE instead posts an encryption W^{enc} of the warrant $\widetilde{W}$ under AU's public key, the output W^{pub} of the transparency function and a NIZK-proof that it knows a valid signature under J's public key on $\widetilde{W}$ and both W^{enc} and W^{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 $\widehat{ct}$ 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.

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

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

Init: Let POSTED be the empty set, let ORDERED be the empty sequence.

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

Order: On input (ORDER, pid, m) from the adversary where some $(pid, t, m) \in \text{POSTED}$ and no $(pid, t', m) \in \text{ORDERED}$, append (pid, t_{Now}, m) to ORDERED.

Read: On input (READ) from $pid \in \text{MACHINE}$, leak (READ) to the adversary. For each entry $(pid, t, m) \in \text{POSTED}$ for which no $(pid, t', m) \in \text{ORDERED}$ and $t = t_{\text{Now}} - \delta_{\text{Post}}$, append (pid, t_{Now}, m) to ORDERED. Then output (ORDERED) to pid.

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

- For $i \in (1, \ldots, |S|)$:
  1. Mark M_i as corrupted
  2. Set $m = (\text{NEXTCOMMITTEE}, R, \mathsf{ek}_i, \mathsf{vk}_i)$, add $(\texttt{roleassign}, t_{\text{Now}}, m)$[a] to POSTED and leak m to the adversary
- For $i \in (|S|+1, \ldots, n)$:
  1. Sample $\left(\mathsf{ek}_{R_i}, \mathsf{dk}_{R_i}\right) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
  2. Sample $\left(\mathsf{vk}_{R_i}, \mathsf{sk}_{R_i}\right) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  3. Sample a uniformly random $M_{R_i} \in \text{MACHINE}$
  4. Set $m = \left(\text{NEXTCOMMITTEE}, R, \mathsf{ek}_{R_i}, \mathsf{vk}_{R_i}\right)$, add $(\texttt{roleassign}, t_{\text{Now}}, m)$ to POSTED and leak m to the adversary

When $(\texttt{roleassign}, t', m)$ is later added to ORDERED output $\left(\text{GENERATE}, R, \mathsf{ek}_{R_i}, \mathsf{dk}_{R_i}, \mathsf{vk}_{R_i}, \mathsf{sk}_{R_i}\right)$ to M_{R_i} for $i \in (|S|+1, \ldots, n)$

Forward Security: When M_R becomes corrupted, output $(\text{GENERATE}, R, \mathsf{ek}_R, \mathsf{dk}_R, \mathsf{vk}_R, \mathsf{sk}_R)$ to the adversary if $(\text{NEXTCOMMITTEE}, R, \mathsf{ek}_R, \mathsf{vk}_R) \notin \text{ORDERED}$.

[a] `roleassign` is a special pid used to represent role-assignment messages

Fig. 4: Blockchain with role assignment functionality loosely based on [28]

After the responses have been posted, LE again reads the content of $\mathcal{F}_{\text{BCRA}}$, decrypts all responses using its decryption key, and combines the partial decryptions ct^* to obtain the secret for each ciphertext ct.

Role Execute. Nodes N read the content of $\mathcal{F}_{\text{BCRA}}$ at least once per round (this is ensured by them only sending an update-message to $\mathcal{G}_{\text{CLOCK}}$ after having done so). Afterwards, they check if they were assigned a role in the current round. If this is the case, they proceed as follows by parsing the content of the ledger:

- They gather all required encryption/verification keys for relevant previous and the next committee
- They gather all messages with key shares of the threshold decryption key
- They gather all requests for decryption

After gathering all relevant messages, they fulfill their role as committee member of the current round. For all messages gathered from the ledger, they validate the signature and accompanying proofs and ignore the message if they are invalid.

They gather all resharings of the threshold decryption key that were made by the previous committee and addressed to the current role and combine them to

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 $\widehat{ct}$ 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[17] 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 $\mathcal{F}_{\mathrm{BCRA}}$.

Get Statistics and Audit. Obtaining statistics is achieved by reading the content of $\mathcal{F}_{\mathrm{BCRA}}$ and gathering all W^{pub} accompanied by valid NIZK-proofs. Similarly, the audit is performed by AU reading the content of $\mathcal{F}_{\mathrm{BCRA}}$ and decrypting all W^{enc} accompanied by valid NIZK-proofs.

Security. In Appendix F we show that our protocol $\mathit{\Pi}_{\mathrm{AD}}$ UC-realizes $\mathcal{F}_{\mathrm{AD}}$. In particular, we show the following theorem.

Theorem 2. *If* NIZK *is a straight-line simulation-extractable non-interactive zero-knowledge proof system,* Σ *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* $\mathcal{F}_{\mathrm{BCRA}}$ *is a RIND-SO secure public key encryption scheme, and* TPKE *is an IND-CPA secure, randomizable and binding* (t, n)*-threshold PKE, then* $\mathit{\Pi}_{\mathrm{AD}}$ *UC-realizes* $\mathcal{F}_{\mathrm{AD}}$ *in the* $\{\mathcal{F}_{\mathrm{BCRA}}, \mathcal{F}_{\mathrm{CRS}}, \mathcal{G}_{\mathrm{CLOCK}}\}$*-hybrid model with respect to adversaries* $\mathcal{A}$ *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 $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathcal{F}_{\text{AS}}$. The construction and techniques developed for $\mathcal{F}_{\text{ASTE}}$, 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 $\mathcal{F}_{\text{ASTE}}$

We briefly describe the capabilities, i.e., the tasks, offered by $\mathcal{F}_{\text{ASTE}}$. The full description is in Appendix C.2. Basically, $\mathcal{F}_{\text{ASTE}}$ 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 $\mathcal{F}_{\text{AS}}$ with minimal changes to System Init to formally handle update periods (i.e., $\mathcal{G}_{\text{CLOCK}}$).

Register, Update, and Next Period. Similar to $\mathcal{F}_{\text{AS}}$, 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 $\mathcal{F}_{\text{AS}}$.

Execute Access. In this task, LE can check whether a ciphertext c was generated by the user with identity *uid* during period *vper*, where LE is supposed to have previously prepared access for a warrant which affects $(\mathit{uid}, \mathit{vper})$.

5.2 The Protocol Π_{ASTE}

To realize $\mathcal{F}_{\mathrm{ASTE}}$, we build a protocol Π_{ASTE} (whose full description is in Appendix C.3). Firstly, we work in the $\{\mathcal{F}_{\mathrm{CRS}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{F}_{\mathrm{AS}}, \mathcal{G}_{\mathrm{CLOCK}}\}$-hybrid model. Thus, we can rely on $\mathcal{F}_{\mathrm{AS}}$ 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, $\mathsf{PKE}_{\mathsf{AS}}$. In the protocol, the user identity *uid* will be set to a $\mathsf{PKE}_{\mathsf{AS}}$ public key.

The scheme $\mathsf{PKE}_{\mathsf{AS}}$ 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 $\mathsf{pk}_R = uid_R$).
- 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 $\mathcal{F}_{\mathrm{AS}}$, which additionally ensure that the tuple $(uid, usk, vper)$ is in $\mathsf{L}_{\mathsf{Secrets}}$, 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 $\mathsf{PKE}_{\mathsf{AS}}$*.* 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 $\mathsf{PKE}_{\mathsf{NCE}}$ and $\mathsf{SKE}_{\mathsf{NCE}}$ be such schemes. Intuitively, we use $\mathsf{PKE}_{\mathsf{NCE}}$ to encrypt the message for the receiver, and $\mathsf{SKE}_{\mathsf{NCE}}$ 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 RO has no "code", so it is impossible to prove (correct) encryption of a message for $\mathsf{PKE}_{\mathsf{NCE}}$ and $\mathsf{SKE}_{\mathsf{NCE}}$ 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 $\mathsf{PKE}_{\mathsf{NCE}}$ and $\mathsf{SKE}_{\mathsf{NCE}}$.[18]

Remark 2. For simplicity and concreteness, we only prove knowledge of the tuple $t = (m, uid, usk, vper)$ in $\mathsf{PKE}_{\mathsf{AS}}$, 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.)

$m_{i,0/1}$ 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 $\mathsf{PKE_{AS}}$ are given in Fig. 5; the two latter algorithms depend on a public commitment key ck. Now, we explain the idea behind the proofs $(\pi_{\mathsf{cut}}, \pi_{\mathsf{con}})$ in $\mathsf{PKE_{AS}.Enc}$ and $\mathsf{PKE_{AS}.Vfy}$ (see Fig. 5). The values m, uid, usk are first secret-shared, then used in encryptions $ct^{RE}_{i,b}$, $ct^{LE}_{i,b}$. 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 π_{con} (realized by $\mathcal{F}_{\mathrm{AS}}$) allows to recover the committed values in the security proof. Moreover, π_{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 π_{cut} is the cut-and-choose part which ensures that enough $\mathsf{PKE_{NCE}}$ resp. $\mathsf{SKE_{NCE}}$ ciphertexts encrypt m' under pk_R resp. uid' under usk'. For this, π_{cut} forces the encryptor to open a randomly chosen share $\gamma_i \in \{0,1\}$ of $ct^{RE}_{i,b}$, $ct^{RE}_{i,b}$ and $com_{i,b}$ for each $i = 1, \ldots, \ell(\lambda)$, where the challenge $\gamma = \mathsf{RO}(stmt, \pi_{\mathsf{con}})$ is derived following the Fiat–Shamir paradigm. Note that if both for $\gamma_i = 0$ and $\gamma_i = 1$ can be opened, then π_{con} ensures that the value reconstructed from the shares is m' (resp. uid', usk'), unless the binding property of COM is broken. Assuming unconditionally binding COM, the latter cannot happen. Then, by a standard argument, if less than the majority of ciphertext shares reconstruct m' (resp. uid', usk'), the probability to succeed in π_{cut} is about $2^{-\ell(\lambda)/2} = 2^{-\lambda}$. Consequently, if π_{con} and π_{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 $ct^{RE}_{i.b}$, computing m_i, and then picking the majority value m of m_i (or $\perp$ if none exists) agrees with overwhelming probability with the extracted value m' of π_{con}. Unsurprisingly, $\mathsf{PKE_{AS}.Dec_{RE}}$ will do just that.

Observe that $\mathsf{PKE_{AS}.Vfy}$ 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 $\mathsf{PKE_{AS}.Dec_{RE}}$ and $\mathsf{PKE_{AS}.Dec_{LE}}$, which work as follows:

- $\mathsf{PKE_{AS}.Dec_{RE}}$ decrypts all $ct^{RE}_{i,b}$ 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.
- $\mathsf{PKE_{AS}.Dec_{LE}}$ 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) $ct^{LE}_{i,b}$ 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 π_{con} coincide with the actual results of decryption. In particular, if $\mathsf{PKE_{AS}.Vfy}$ 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 $\mathcal{F}_{\mathrm{AS}}$-hybrid functionality. Thus, LE ob-

$\mathsf{PKE_{AS}.Enc}(ck, \mathsf{pk}_R, m, (\mathsf{sk}_S, usk, uid, vper))$

// Prepare cut-and-choose encryption for LE

for $i = 1, \ldots, \ell(\lambda)$

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

$usk_{i,0} + usk_{i,1} = usk$

$uid_{i,0} + uid_{i,1} = uid$

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

$ct^{RE}_{i,b} = \mathsf{PKE_{NCE}.Enc}(\mathsf{pk}_R, m_{i,b}; r^{RE}_{i,b})$

$ct^{LE}_{i,b} = \mathsf{SKE_{NCE}.Enc}(usk_{i,b}, uid_{i,b}; r^{LE}_{i,b})$

$w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, r^{RE}_{i,b}, r^{LE}_{i,b})$

$com_{i,b} = \mathsf{COM.Com}(ck, w_{i,b}; r^{com}_{i,b})$

$d_{i,b} = (w_{i,b}, r^{com}_{i,b})$

// Consistency proof for commitments and ctxts.

$stmt = (vper, \mathsf{pk}_R, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b})$

$wit = (\sigma_{vper}, usk, uid, (d_{i,b})_{i,b})$

$\pi_{\mathsf{con}} = \mathsf{NIZK_{AS}.Prove}((usk, uid, vper),$
$stmt, wit, \mathcal{R}_{\mathsf{PKEAS}})$ for relation

$\mathcal{R}_{\mathsf{PKEAS}} = \{(stmt, wit) \mid$

$(uid, \mathsf{sk}_S) = \mathsf{PKE_{NCE}.Gen}(1^\lambda; \mathsf{sk}_S)$

$com_{i,b} = \mathsf{COM.Com}(ck, (m_{i,b}, usk_{i,b},$
$uid_{i,b}, r^{RE}_{i,b}, r^{LE}_{i,b}); r^{com}_{i,b})$ for all i, b

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

$\forall i\colon usk = usk_{i,0} + usk_{i,1}$

$\forall i\colon uid = uid_{i,0} + uid_{i,1}$

$\}$

// Cut-and-choose: Query challenge γ

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

$\pi_{\mathsf{cut}} = (d_{i,\gamma_i})_i$

$\pi = (\pi_{\mathsf{cut}}, \pi_{\mathsf{con}})$

return $(\mathsf{pk}_R, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b}, \pi)$.

$\mathsf{PKE_{AS}.Gen}(1^\lambda)$

1 : // For simplicity, sk equals the random coins.

2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE_{NCE}.Gen}(1^\lambda; \mathsf{sk})$

$\mathsf{PKE_{AS}.Vfy}(ck, c, vper)$

parse $c = (\mathsf{pk},$
$((com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b}),$
$((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$

$stmt = (vper, \mathsf{pk}_R,$
$(com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b})$

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

if $\mathsf{NIZK_{AS}.Verify}(stmt, \pi_{\mathsf{con}}, \mathcal{R}_{\mathsf{PKEAS}}) = 0$

then return 0

// Check cut-and-choose proof

for $i = 1, \ldots, \ell(\lambda)$

$b = \gamma_i$

parse $d_{i,b} = (w_{i,b}, r^{com}_{i,b})$

parse $w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b},$
$r^{RE}_{i,b}, r^{LE}_{i,b})$

if $com_{i,b} = \mathsf{COM.Com}(ck, w_{i,b}; r^{com}_{i,b})$

or $ct^{RE}_{i,b} = \mathsf{PKE_{NCE}.Enc}(\mathsf{pk}, m_{i,b}; r^{RE}_{i,b})$

or $ct^{LE}_{i,b} = \mathsf{SKE_{NCE}.Enc}(usk_{i,b}, uid_{i,b}; r^{LE}_{i,b})$

then return 0

return 1

Fig. 5: Encryption and ciphertext verification subroutines of $\mathsf{PKE_{AS}}$.

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 $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}$ 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 $\mathcal{F}_{\mathrm{ASTE}}$. 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].[19] 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 1.5MiB for the ciphertexts, without accounting for π_{cut} and π_{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 $\mathit{\Pi}_{\mathrm{ASTE}}$, since ciphertext validity is publicly verifiable.[20] 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 $\mathsf{PKE}_{\mathsf{AS}}$ 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.[21] Also, [34] is limited to the messaging application scenario, while we also specify an auditable surveillance functionality $\mathcal{F}_{\mathrm{AS}}$ that can be used for many applications, e.g., for $\mathcal{F}_{\mathrm{ASTE}}$.

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 $\mathcal{F}_{\mathrm{AS}}$ 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 $\mathcal{F}_{\mathrm{BCRA}}$ hybrid functionality are still novel and experimental [8, 14, 20], so the actual guarantees of such a realization may differ from our assumptions. For $\mathsf{PKE}_{\mathsf{AS}}$, 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.

[21] While the ideal functionality in [34] technically only supplies some *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

[1] Harold Abelson, Ross J. Anderson, Steven M. Bellovin, et al. "Keys under doormats: mandating insecurity by requiring government access to all data and communications". In: *J. Cybersecur.* 1.1 (2015), pp. 69–79. DOI: 10.1093/cybsec/tyv009. URL: https://doi.org/10.1093/cybsec/tyv009.

[2] Ghada Arfaoui, Olivier Blazy, Xavier Bultel, et al. "How to (Legally) Keep Secrets from Mobile Operators". In: *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part I.* Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12972. Lecture Notes in Computer Science. Springer, 2021, pp. 23–43. DOI: 10.1007/978-3-030-88418-5_2. URL: https://doi.org/10.1007/978-3-030-88418-5_2.

[3] Michael Backes, Jan Camenisch, and Dieter Sommer. "Anonymous yet accountable access control". In: *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society, WPES 2005, Alexandria, VA, USA, November 7, 2005.* Ed. by Vijay Atluri, Sabrina De Capitani di Vimercati, and Roger Dingledine. ACM, 2005, pp. 40–46. DOI: 10.1145/1102199.1102208. URL: https://doi.org/10.1145/1102199.1102208.

[4] Michael Backes and Dennis Hofheinz. "How to Break and Repair a Universally Composable Signature Functionality". In: *ISC.* Vol. 3225. Lecture Notes in Computer Science. Springer, 2004, pp. 61–72.

[5] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. *Bitcoin as a Transaction Ledger: A Composable Treatment.* Cryptology ePrint Archive, Report 2017/149. https://ia.cr/2017/149. 2017.

[6] Adam M. Bates, Kevin R. B. Butler, Micah Sherr, et al. "Accountable Wiretapping -or- I know they can hear you now". In: *NDSS 2012.* The Internet Society, Feb. 2012.

[7] Mihir Bellare and Ronald L. Rivest. "Translucent Cryptography - An Alternative to Key Escrow, and Its Implementation via Fractional Oblivious Transfer". In: *Journal of Cryptology* 12.2 (Mar. 1999), pp. 117–139. DOI: 10.1007/PL00003819.

[8] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, et al. "Can a Public Blockchain Keep a Secret?" In: *TCC 2020, Part I.* Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12550. LNCS. Springer, Heidelberg, Nov. 2020, pp. 260–290. DOI: 10.1007/978-3-030-64375-1_10.

[9] Zvika Brakerski, Pedro Branco, Nico Döttling, Sanjam Garg, and Giulio Malavolta. "Constant Ciphertext-Rate Non-committing Encryption from Standard Assumptions". In: *TCC 2020, Part I*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12550. LNCS. Springer, Heidelberg, Nov. 2020, pp. 58–87. DOI: 10.1007/978-3-030-64375-1_3.

[10] Ernest F. Brickell, Peter Gemmell, and David W. Kravitz. "Trustee-based Tracing Extensions to Anonymous Cash and the Making of Anonymous Change". In: *6th SODA*. Ed. by Kenneth L. Clarkson. ACM-SIAM, Jan. 1995, pp. 457–466.

[11] Joakim Brorsson, Bernardo David, Lorenzo Gentile, Elena Pagnin, and Paul Stankovski Wagner. "PAPR: Publicly Auditable Privacy Revocation forÂ Anonymous Credentials". In: *Topics in Cryptology – CT-RSA 2023*. Ed. by Mike Rosulek. Cham: Springer International Publishing, 2023, pp. 163–190.

[12] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. "The Wonderful World of Global Random Oracles". In: *EUROCRYPT 2018, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. LNCS. Springer, Heidelberg, 2018, pp. 280–312. DOI: 10.1007/978-3-319-78381-9_11.

[13] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. "Universal Composition with Responsive Environments". In: *ASIACRYPT 2016, Part II*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10032. LNCS. Springer, Heidelberg, Dec. 2016, pp. 807–840. DOI: 10.1007/978-3-662-53890-6_27.

[14] Matteo Campanelli, Bernardo David, Hamidreza Khoshakhlagh, Anders Konring, and Jesper Buus Nielsen. *Encryption to the Future: A Paradigm for Sending Secret Messages to Future (Anonymous) Committees*. Cryptology ePrint Archive, Report 2021/1423. https://eprint.iacr.org/2021/1423. 2021.

[15] Matteo Campanelli, Bernardo David, Hamidreza Khoshakhlagh, Anders Konring, and Jesper Buus Nielsen. "Encryption to the Future". In: *Advances in Cryptology – ASIACRYPT 2022*. Ed. by Shweta Agrawal and Dongdai Lin. Cham: Springer Nature Switzerland, 2022, pp. 151–180.

[16] Ran Canetti. "Universally Composable Security". In: *Journal of the ACM* 67.5 (Sept. 2020), 28:1–28:94. DOI: 10.1145/3402457. URL: https://doi.org/10.1145/3402457.

[17] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Report 2000/067. https://ia.cr/2000/067. 2000.

[18] Ran Canetti. "Universally Composable Signature, Certification, and Authentication". In: *CSFW*. IEEE Computer Society, 2004, p. 219.

[19] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. LNCS. Springer, Heidelberg, Aug. 2001, pp. 19–40. DOI: 10.1007/3-540-44647-8_2.

[20] Ignacio Cascudo, Bernardo David, Lydia Garms, and Anders Konring. *YOLO YOSO: Fast and Simple Encryption and Secret Sharing in the YOSO Model*. Cryptology ePrint Archive, Report 2022/242. https://ia.cr/2022/242. 2022.

[21] Council of the European Union. *Council Resolution on Encryption – Security through encryption and security despite encryption*. https://data.consilium.europa.eu/doc/document/ST-13084-2020-REV-1/en/pdf. 2020.

[22] Vanesa Daza, Abida Haque, Alessandra Scafuro, Alexandros Zacharakis, and Arantxa Zapico. "Mutual Accountability Layer: Accountable Anonymity Within Accountable Trust". In: *Cyber Security, Cryptology, and Machine Learning - 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 - July 1,*

2022, Proceedings. Ed. by Shlomi Dolev, Jonathan Katz, and Amnon Meisels. Vol. 13301. Lecture Notes in Computer Science. Springer, 2022, pp. 318–336. DOI: 10.1007/978-3-031-07689-3_24. URL: https://doi.org/10.1007/978-3-031-07689-3_24.

[23] Andreas Erwig, Sebastian Faust, and Siavash Riahi. *Large-Scale Non-Interactive Threshold Cryptosystems Through Anonymity*. Cryptology ePrint Archive, Report 2021/1290. https://eprint.iacr.org/2021/1290. 2021.

[24] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner. “Practical Accountability of Secret Processes”. In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, Aug. 2018, pp. 657–674.

[25] Eiichiro Fujisaki and Tatsuaki Okamoto. “Secure Integration of Asymmetric and Symmetric Encryption Schemes”. In: *Journal of Cryptology* 26.1 (Jan. 2013), pp. 80–101. DOI: 10.1007/s00145-011-9114-1.

[26] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. “On the Implausibility of Differing-Inputs Obfuscation and Extractable Witness Encryption with Auxiliary Input”. In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 518–535. DOI: 10.1007/978-3-662-44371-2_29.

[27] Christina Garman, Matthew Green, and Ian Miers. “Accountable Privacy for Decentralized Anonymous Payments”. In: *FC 2016*. Ed. by Jens Grossklags and Bart Preneel. Vol. 9603. LNCS. Springer, Heidelberg, Feb. 2016, pp. 81–98.

[28] Craig Gentry, Shai Halevi, Hugo Krawczyk, et al. “YOSO: You Only Speak Once - Secure MPC with Stateless Ephemeral Roles”. In: *CRYPTO 2021, Part II*. Ed. by Tal Malkin and Chris Peikert. Vol. 12826. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 64–93. DOI: 10.1007/978-3-030-84245-1_3.

[29] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. “Practical Non-interactive Publicly Verifiable Secret Sharing with Thousands of Parties”. In: *EUROCRYPT 2022, Part I*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13275. LNCS. Springer, Heidelberg, 2022, pp. 458–487. DOI: 10.1007/978-3-031-06944-4_16.

[30] Craig Gentry, Allison B. Lewko, and Brent Waters. “Witness Encryption from Instance Independent Assumptions”. In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 426–443. DOI: 10.1007/978-3-662-44371-2_24.

[31] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. “How to Run Turing Machines on Encrypted Data”. In: *CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. LNCS. Springer, Heidelberg, Aug. 2013, pp. 536–553. DOI: 10.1007/978-3-642-40084-1_30.

[32] Shafi Goldwasser and Sunoo Park. “Public Accountability vs. Secret Laws: Can They Coexist?: A Cryptographic Proposal”. In: *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society, Dallas, TX, USA, October 30 - November 3, 2017*. Ed. by Bhavani M. Thuraisingham and Adam J. Lee. ACM, 2017, pp. 99–110. DOI: 10.1145/3139550.3139565. URL: https://doi.org/10.1145/3139550.3139565.

[33] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. “Storing and Retrieving Secrets on a Blockchain”. In: *Public-Key Cryptography – PKC 2022*. Ed. by Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe. Cham: Springer International Publishing, 2022, pp. 252–282.

[34] Matthew Green, Gabriel Kaptchuk, and Gijs Van Laer. "Abuse Resistant Law Enforcement Access Systems". In: *EUROCRYPT 2021, Part III*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12698. LNCS. Springer, Heidelberg, Oct. 2021, pp. 553–583. DOI: 10.1007/978-3-030-77883-5_19.

[35] Encryption Working Group. *Moving the Encryption Policy Conversation Forward*. Tech. rep. Carnegie Endowment for International Peace, 2019.

[36] Carmit Hazay, Arpita Patra, and Bogdan Warinschi. "Selective Opening Security for Receivers". In: *ASIACRYPT 2015, Part I*. Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9452. LNCS. Springer, Heidelberg, 2015, pp. 443–469. DOI: 10.1007/978-3-662-48797-6_19.

[37] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation". In: *TCC 2017, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. LNCS. Springer, Heidelberg, Nov. 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12.

[38] Stanislaw Jarecki and Vitaly Shmatikov. "Handcuffing Big Brother: an Abuse-Resilient Transaction Escrow Scheme". In: *EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, Heidelberg, May 2004, pp. 590–608. DOI: 10.1007/978-3-540-24676-3_35.

[39] Daniel Jost and Ueli Maurer. "Overcoming Impossibility Results in Composable Security Using Interval-Wise Guarantees". In: *CRYPTO 2020, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. LNCS. Springer, Heidelberg, Aug. 2020, pp. 33–62. DOI: 10.1007/978-3-030-56784-2_2.

[40] Dafna Kidron and Yehuda Lindell. "Impossibility Results for Universal Composability in Public-Key Models and with Fixed Inputs". In: *Journal of Cryptology* 24.3 (July 2011), pp. 517–544. DOI: 10.1007/s00145-010-9069-7.

[41] Joe Kilian and Frank Thomson Leighton. "Fair Cryptosystems, Revisited: A Rigorous Approach to Key-Escrow (Extended Abstract)". In: *CRYPTO'95*. Ed. by Don Coppersmith. Vol. 963. LNCS. Springer, Heidelberg, Aug. 1995, pp. 208–221. DOI: 10.1007/3-540-44750-4_17.

[42] Joshua A. Kroll, Edward W. Felten, and Dan Boneh. *Secure protocols for accountable warrant execution*. https://www.jkroll.com/papers/warrant_paper.pdf. 2014.

[43] Joshua A. Kroll, Joe Zimmerman, David J. Wu, et al. *Accountable Cryptographic Access Control*. https://www.cs.yale.edu/homes/jf/kroll-paper.pdf. 2018.

[44] Dennis Kügler and Holger Vogt. "Offline Payments with Auditable Tracing". In: *FC 2002*. Ed. by Matt Blaze. Vol. 2357. LNCS. Springer, Heidelberg, Mar. 2003, pp. 269–281.

[45] Kaoru Kurosawa and Jun Furukawa. "Universally Composable Undeniable Signature". In: *ICALP (2)*. Vol. 5126. Lecture Notes in Computer Science. Springer, 2008, pp. 524–535.

[46] Xiao Liang and Omkant Pandey. "Towards a Unified Approach to Black-Box Constructions of Zero-Knowledge Proofs". In: *CRYPTO 2021, Part IV*. Ed. by Tal Malkin and Chris Peikert. Vol. 12828. LNCS. Virtual Event: Springer, Heidelberg, Aug. 2021, pp. 34–64. DOI: 10.1007/978-3-030-84259-8_2.

[47] Jia Liu, Mark Dermot Ryan, and Liqun Chen. "Balancing Societal Security and Individual Privacy: Accountable Escrow System". In: *CSF 2014 Computer Security Foundations Symposium*. Ed. by Anupam Datta and Cedric Fournet. IEEE Computer Society Press, 2014, pp. 427–440. DOI: 10.1109/CSF.2014.37.

[48] Jesper Buus Nielsen. "Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case". In: *CRYPTO 2002*. Ed.

by Moti Yung. Vol. 2442. LNCS. Springer, Heidelberg, Aug. 2002, pp. 111–126. DOI: 10.1007/3-540-45708-9_8.
[49] Official Journal of the European Communities. *Council Resolution on on the lawful interception of telecommunications*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31996G1104&from=EN. 1995.
[50] Official Journal of the European Communities. *Directive (EU) 2018/843 of the European Parliament and of the Council amending Directive (EU) 2015/849 on the prevention of the use of the financial system for the purposes of money laundering or terrorist financing, and amending Directives 2009/138/EC and 2013/36/EU*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31996G1104&from=EN. 2018.
[51] Pascal Paillier and Moti Yung. "Self-Escrowed Public-Key Infrastructures". In: *ICISC 99*. Ed. by JooSeok Song. Vol. 1787. LNCS. Springer, Heidelberg, Dec. 2000, pp. 257–268.
[52] Gaurav Panwar, Roopa Vishwanathan, Satyajayant Misra, and Austin Bos. "SAMPL: Scalable Auditability of Monitoring Processes using Public Ledgers". In: *ACM CCS 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, Nov. 2019, pp. 2249–2266. DOI: 10.1145/3319535.3354219.
[53] Giuseppe Persiano, Duong Hieu Phan, and Moti Yung. "Anamorphic Encryption: Private Communication Against a Dictator". In: *EUROCRYPT 2022, Part II*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13276. LNCS. Springer, Heidelberg, 2022, pp. 34–63. DOI: 10.1007/978-3-031-07085-3_2.
[54] Mike Rosulek. "Must You Know the Code of f to Securely Compute f?" In: *CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. LNCS. Springer, Heidelberg, Aug. 2012, pp. 87–104. DOI: 10.1007/978-3-642-32009-5_7.
[55] Stefan Savage. "Lawful Device Access without Mass Surveillance Risk: A Technical Design Discussion". In: *ACM CCS 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM Press, Oct. 2018, pp. 1761–1774. DOI: 10.1145/3243734.3243758.
[56] Alessandra Scafuro. "Break-glass Encryption". In: *PKC 2019, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. LNCS. Springer, Heidelberg, Apr. 2019, pp. 34–62. DOI: 10.1007/978-3-030-17259-6_2.
[57] Sacha Servan-Schreiber and Archer Wheeler. "Judge, Jury & Encryptioner: Exceptional Access with a Fixed Social Cost". In: *CoRR* abs/1912.05620 (2019). URL: http://arxiv.org/abs/1912.05620.
[58] Charles V. Wright and Mayank Varia. "Crypto Crumple Zones: Enabling Limited Access without Mass Surveillance". In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 288–306. DOI: 10.1109/EuroSP.2018.00028. URL: https://doi.org/10.1109/EuroSP.2018.00028.
[59] Yusuke Yoshida, Fuyuki Kitagawa, Keita Xagawa, and Keisuke Tanaka. "Non-committing Encryption with Constant Ciphertext Expansion from Standard Assumptions". In: *ASIACRYPT 2020, Part II*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12492. LNCS. Springer, Heidelberg, Dec. 2020, pp. 36–65. DOI: 10.1007/978-3-030-64834-3_2.
[60] Adam Young and Moti Yung. "Auto-Recoverable Auto-Certifiable Cryptosystems". In: *EUROCRYPT'98*. Ed. by Kaisa Nyberg. Vol. 1403. LNCS. Springer, Heidelberg, 1998, pp. 17–31. DOI: 10.1007/BFb0054114.

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* $(\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *of* PPT *algorithms such that*

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

For PKE schemes in the ROM, $\mathsf{PKE.Enc}$ *and* $\mathsf{PKE.Dec}$ *(but not* $\mathsf{PKE.Gen}$*) 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.* $\mathsf{SKE.Gen}$ *outputs only* sk*, and* $\mathsf{SKE.Enc}$ *takes* sk *as input.*

Remark 3. We explicitly forbid key generation algorithms ($\mathsf{PKE.Gen}$ and $\mathsf{SKE.Gen}$) 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$-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* $(\mathsf{sk}_1, \mathsf{sk}_2, c)$ *such that* $\mathsf{sk}_1 \neq \mathsf{sk}_2$ *but* $\mathsf{SKE.Dec}(\mathsf{sk}_1, c) \neq \bot$ *and* $\mathsf{SKE.Dec}(\mathsf{sk}_2, c) \neq \bot$*.*

Let PKE *be a perfectly correct PKE scheme. We say* PKE *is* (public-)key-committing*, if it is hard to find a tuple* $((\mathsf{pk}_1, \mathsf{sk}_1), (\mathsf{pk}_2, \mathsf{sk}_2), c)$ *such that* $(\mathsf{pk}_i, \mathsf{sk}_i) \in \mathrm{Supp}\big(\mathsf{PKE.Gen}(1^\lambda)\big)$ *and* $\mathsf{pk}_1 \neq \mathsf{pk}_2$ *but* $\mathsf{PKE.Dec}(\mathsf{sk}_1, c) \neq \bot$ *and* $\mathsf{PKE.Dec}(\mathsf{sk}_2, c) \neq \bot$*.*

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

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

$\mathsf{Exp}^{\mathsf{s\text{-}nince\text{-}real}}_{\mathsf{PKE},\mathcal{S},\mathcal{A}}(1^\lambda)$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$

// *$\mathcal{A}$ gets access to encryption oracle.*

$\mathsf{state}_\mathcal{A} \leftarrow \mathcal{A}^{\mathsf{PKE.Enc}(\mathsf{pk},\cdot)}(1^\lambda, \mathsf{pk})$

$b \leftarrow \mathcal{A}(\mathsf{state}_\mathcal{A}, \mathsf{sk})$

return b

$\mathsf{Exp}^{\mathsf{s\text{-}nince\text{-}ideal}}_{\mathsf{PKE},\mathcal{S},\mathcal{A}}(1^\lambda)$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$

// *$\mathcal{A}$ gets access to simulation oracle.*

$\mathsf{state}_\mathcal{A} \leftarrow \mathcal{A}^{\mathcal{S}(1^\lambda, \mathsf{pk}, |\cdot|)}(\mathsf{pk})$

// *Let $m_1, \ldots, m_n$ be the queries of $\mathcal{A}$.*

// *Let $\mathsf{state}_\mathcal{S}$ be the state of $\mathcal{S}$*

$\mathsf{Expln}(\mathsf{state}_\mathcal{S}, \mathsf{sk}, m_1, \ldots, m_n)$ // *Program* RO

$b \leftarrow \mathcal{A}(\mathsf{state}_\mathcal{A}, \mathsf{sk})$

return b

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

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

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

We define strong NINCE *for SKE schemes analogously, except that* pk *is not given to* $\mathcal{A}$ *and* $\mathcal{S}$*.*

A.2 Instantiations of NINCE

We construct a public-key NINCE scheme $\mathsf{PKE}_{\mathsf{NCE}}$ with message space $\{0,1\}^{\mathsf{poly}(\lambda)}$ from any secure PKE scheme with superpolynomial and uniformly sampleable

message space $\mathcal{M}$, randomness space $\mathcal{R}$, and three independent random oracles $\mathsf{RO}_I\colon \{0,1\}^* \to \{0,1\}^\lambda$, $\mathsf{RO}_M\colon \mathcal{M} \to \{0,1\}^{\mathsf{poly}(\lambda)}$, and $\mathsf{RO}_R\colon \mathcal{M} \to \mathcal{R}$. Recall that any number of random oracles can be obtained from a single random oracle $\mathsf{RO}\colon \{0,1\}^* \to \{0,1\}^\lambda$ by domain separation, e.g. by using $\mathsf{RO}_I(x) = \mathsf{RO}(0,x)$, $\mathsf{RO}_M(x) = \mathsf{RO}(1,x)$, $\mathsf{RO}_R(x) = \mathsf{RO}(2,x)$.[22] Moreover, we make the simplifying assumption, that the secret key sk contains the public key pk, and thus decryption can use pk. Any PKE scheme is trivially modified to satisfy this property without affecting its security.

$\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Gen}(1^\lambda)$

$(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PKE}.\mathsf{Gen}(1^\lambda)$

$\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Enc}(\mathsf{pk}, m)$

$esk \leftarrow \mathcal{M}$ // ephemeral secret
$c_1 = \mathsf{PKE}.\mathsf{Enc}(\mathsf{pk}, esk; \mathsf{RO}_R(esk))$
$c_2 = \mathsf{RO}_M(esk) \oplus m$
$t = \mathsf{RO}_I(\mathsf{pk}, esk, c_1, c_2, m)$
return (c_1, c_2, t)

$\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Dec}(\mathsf{sk}, c)$

parse $(c_1, c_2, t) = c$
$esk \leftarrow \mathsf{PKE}.\mathsf{Dec}(\mathsf{sk}, c_1; \mathsf{RO}_R(esk))$
$m = \mathsf{RO}_M(esk) \oplus c_2$
$t' = \mathsf{RO}_I(\mathsf{pk}, esk, c_1, c_2, m)$
if $t = t'$ **return** m
else return $\perp$

$\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Sim}(\mathsf{pk})$

$esk \leftarrow \mathcal{M}$
$c_1 = \mathsf{PKE}.\mathsf{Enc}(\mathsf{pk}, esk; \mathsf{RO}_R(esk))$
$c_2 \xleftarrow{r} \{0,1\}^{\mathsf{poly}(\lambda)}$
$t \leftarrow \{0,1\}^\lambda$
return (c_1, c_2, t)

$\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Expln}(\mathsf{sk}, c, m)$

parse $(c_1, c_2, t) = c$
$esk \leftarrow \mathsf{PKE}.\mathsf{Dec}(\mathsf{sk}, c_1)$ // $\neq \perp$
program $\mathsf{RO}_M(esk) \coloneqq c_2 \oplus m$
program $\mathsf{RO}_I(\mathsf{pk}, esk, c_1, c_2, m) \coloneqq t$

The construction of $\mathsf{PKE}_{\mathsf{NCE}}$ 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 is 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* $\mathsf{PKE}_{\mathsf{NCE}} = (\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Gen}, \mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Enc}, \mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Dec}, \mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Sim}, \mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Expln})$ *defined in Appendix A.2 is IND-CCA-secure, strong NINCE-secure, and perfectly correct.*

The proof is standard.

[22] Additional transformations are necessary if $\mathcal{R}, \mathcal{M} \neq \{0,1\}^\lambda$ or $\mathsf{poly}(\lambda) \neq \lambda$. 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 $\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Expln}$ works, or it was queried before. Since sk is only available to $\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Expln}$, but not $\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Sim}$, 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 $\mathsf{PKE}_{\mathsf{NCE}}$ is only a variant of the Fujisaki–Okamoto transformation [25, 37].

Key-committment of $\mathsf{PKE}_{\mathsf{NCE}}$ 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 $\{0,1\}^{\mathsf{poly}(\lambda)}$.

$\underline{\mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Gen}(1^\lambda)}$
$\mathsf{sk} \leftarrow \{0,1\}^\lambda$

$\mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Enc}(\mathsf{sk}, m)$	$\mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Dec}(\mathsf{sk}, c)$
$N \leftarrow \{0,1\}^{2\lambda}$ // Nonce	**parse** $(N, c', t) = c$
$c' = \mathsf{RO}_M(\mathsf{sk}, N) \oplus m$	$m = \mathsf{RO}_M(\mathsf{sk}, N) \oplus c'$
$t = \mathsf{RO}_I(\mathsf{sk}, N, c', m)$	$t' = \mathsf{RO}_I(\mathsf{sk}, N, c', m)$
return (N, c', t)	**if** $t = t'$ **return** m
	else return $\bot$

$\mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Sim}()$	$\mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Expln}(\mathsf{sk}, c, m)$
$N \leftarrow \{0,1\}^{2\lambda}$	**parse** $(N, c', t) = c$
$c' \xleftarrow{r} \{0,1\}^{\mathsf{poly}(\lambda)}$	**program** $\mathsf{RO}_M(esk) := c' \oplus m$
$t \leftarrow \{0,1\}^\lambda$	**program** $\mathsf{RO}_I(\mathsf{sk}, N, c', m) := t$
return (N, c', t)	

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

Proof (Proof sketch). By standard arguments, $\mathsf{SKE}_{\mathsf{NCE}}$ is actually a authenticated encryption scheme, so in particular IND-CCA secure. Similar to $\mathsf{PKE}_{\mathsf{NCE}}$,

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-committment of $\mathsf{SKE}_{\mathsf{NCE}}$ 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* $(\mathsf{TPKE.Gen}, \mathsf{TPKE.Enc}, \mathsf{TPKE.Dec}, \mathsf{TPKE.TDec}, \mathsf{TPKE.ShareVer}, \mathsf{TPKE.Combine}, \mathsf{TPKE.Sk2Pk})$ *of* PPT *algorithms such that*

- $\mathsf{TPKE.Gen}(1^\lambda)$ *outputs a public key* pk*, secret key* sk *and sets* $\{\mathsf{vk}_i\}_{i\in[n]}$ *of verification keys and* $\{\mathsf{sk}_i\}_{i\in[n]}$ *of secret key shares.*
- $\mathsf{TPKE.Enc}(\mathsf{pk}, m)$ *takes a public key* pk*, a message* m *in the message space* $\mathcal{M}$*, and outputs a ciphertext* ct*.*
- $\mathsf{TPKE.Dec}(\mathsf{sk}, ct)$ *takes a secret key* sk*, a ciphertext* ct *and outputs a message* $m \in \mathcal{M}$*.*
- $\mathsf{TPKE.TDec}(\mathsf{sk}_i, ct)$ *takes a secret key share* sk_i*, a ciphertext* ct*, and outputs a decryption share* ct_i*.*
- $\mathsf{TPKE.ShareVer}(ct, \mathsf{vk}_i, ct_i)$ *takes a ciphertext* ct*, a verification key* vk_i *and a decryption share* ct_i*, and outputs either* 1 *or* 0*. If the output is* 1*,* ct_i *is called a* valid *decryption share.*
- $\mathsf{TPKE.Combine}(ct, T)$ *takes a ciphertext* ct *and a set of valid decryption shares* T *such that* $|T| = t + 1$*, and outputs a message* $m \in \mathcal{M}$*.*
- $\mathsf{TPKE.Sk2Pk}(\mathsf{sk})$ *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 $(\mathsf{pk}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Gen}(1^\lambda)$*:*

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

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

A TPKE scheme is called simulatable *if addinionaly an efficient simulation algorithm* $\mathsf{TPKE.SimTDec}(\mathsf{pk}, ct, m, \{ct_k\}_{k\in\mathcal{B}:|\mathcal{B}|\leq t})$ *exists that takes a public key, ciphertext, target message and up to* t *decryption shares of corrupted parties as input and outputs decryption shares* $\{ct_i\}_{i\in[n]\setminus\mathcal{B}}$ *for the honest parties that cause*

$\mathsf{TPKE.Combine}$ *to output the desired message* m *and for which* $\mathsf{TPKE.ShareVer}$ *outputs* 1.

A TPKE scheme is called re-randomizable *if it also has a* PPT *algorithm* $\mathsf{TPKE.Rand}$ *such that* $\forall(\mathsf{pk}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Gen}(1^\lambda)$*:*

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

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

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* $\mathcal{A}$ *and all efficiently resamplable distributions* $\mathcal{D}$ *we have that*

$$\mathsf{Adv}^{\mathsf{RIND-SO}}_{\mathsf{PKE}}(\mathcal{A}, \lambda) := \left| \Pr \mathsf{Exp}^{\mathsf{RIND-SO}}_{\mathsf{PKE}}(\mathcal{A}, \lambda) - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda)$$

$\mathsf{Exp}^{\mathsf{RIND-SO}}_{\mathsf{PKE}}(\mathcal{A}, \lambda)$

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

Corruption Oracle $\mathcal{C}(i)$

1 : **if** $i \notin [n]$**return** $\perp$
2 : $\mathcal{I} := \mathcal{I} \cup \{i\}$
3 : **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 Σ.

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

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

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

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

A.6 Commitment Scheme

We use a correct, (computationally) hiding, (statistically) binding, extractable and equivocable commitment scheme COM.

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

- $\mathsf{COM.Setup}$ *is a PPT algorithm, which takes* 1^λ *as input and outputs public parameters* crs_{com}.
- $\mathsf{COM.Com}$ *is a PPT algorithm, which takes as input parameters* crs_{com} *and a message* $m \in \mathcal{M}$ *and outputs a commitment* com_m *to* m *and some decommitment value* decom_m.
- $\mathsf{COM.Open}$ *is a deterministic polynomial-time algorithm, which takes as input parameters* crs_{com}*, a commitment* com_m*, a decommitment value* decom_m *and a message* m*. It returns either* 0 *or* 1.

COM *is* correct *if for all* λ, $crs_{com} \leftarrow \mathsf{COM.Setup}(1^\lambda)$, $m \in \mathcal{M}$, *and* $(\mathsf{com}_m, \mathsf{decom}_m) \leftarrow \mathsf{COM.Com}(crs_{com}, m)$ *it holds that*

$$1 = \mathsf{COM.Open}(crs_{com}, \mathsf{com}_m, \mathsf{decom}_m, m).$$

An extractable *commitment scheme has the following additional algorithms:*

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

An equivocable *commitment scheme has the following additional algorithms:*

- $(crs^{eq}_{com}, td^{eq}_{com}) \leftarrow \mathsf{COM.Setup}^{Equiv}(1^\lambda)$
- $(\widetilde{\mathsf{com}}, eq) \leftarrow \mathsf{COM.SimCom}(td^{eq}_{com})$
- $\mathsf{decom}_m \leftarrow \mathsf{COM.Equiv}(td^{eq}_{com}, m, eq)$

Definition 9 (Equivocal Commitment scheme). *A commitment scheme is* equivocal *if there exist PPT algorithms* $\mathsf{COM.Setup}^{Equiv}$, $\mathsf{COM.SimCom}$ *and* $\mathsf{COM.Equiv}$ *such that for all PPT adversaries* $\mathcal{A}$ *we have that the advantage defined by*

$$\left| \begin{array}{c} \Pr\left[\, 1 \leftarrow \mathcal{A}(crs_{com}) \mid crs_{com} \leftarrow \mathsf{COM.Setup}(1^\lambda) \,\right] \\ - \Pr\left[\, 1 \leftarrow \mathcal{A}(crs^{eq}_{com}) \mid (crs^{eq}_{com}, td^{eq}_{com}) \leftarrow \mathsf{COM.Setup}^{Equiv}(1^\lambda) \,\right] \end{array} \right|$$

is negligible in λ *and we have that the advantage defined by*

$$\left| \begin{array}{c} \Pr\left[\begin{array}{r} 1 \leftarrow \mathcal{A}\,(crs^{eq}_{com}, td^{eq}_{com}, \\ m, c, d) \end{array} \middle| \begin{array}{c} (crs^{eq}_{com}, td^{eq}_{com}) \leftarrow \mathsf{COM.Setup}^{Equiv}(1^\lambda), \\ m \leftarrow \mathcal{M}, \\ (\mathsf{com}_m, \mathsf{decom}_m) \leftarrow \mathsf{COM.Com}(crs^{eq}_{com}, m) \end{array} \right] \\ - \Pr\left[\begin{array}{r} 1 \leftarrow \mathcal{A}\,(crs^{eq}_{com}, td^{eq}_{com}, \\ m, \widetilde{\mathsf{com}}, \mathsf{decom}_m) \end{array} \middle| \begin{array}{c} (crs^{eq}_{com}, td^{eq}_{com}) \leftarrow \mathsf{COM.Setup}^{Equiv}(1^\lambda), \\ (\widetilde{\mathsf{com}}, eq) \leftarrow \mathsf{COM.SimCom}(td^{eq}_{com}), \\ m \leftarrow \mathcal{M}, \\ \mathsf{decom}_m \leftarrow \mathsf{COM.Equiv}(td^{eq}_{com}, m, eq) \end{array} \right] \end{array} \right|$$

is zero.

Definition 10 (Extractable Commitment Scheme). *A commitment scheme is* extractable *if there exist PPT algorithms* $\mathsf{COM.Setup}^{Ext}$ *and* $\mathsf{COM.Extract}$ *such that for all PPT adversaries* $\mathcal{A}$ *we have that the advantage defined by*

$$\left| \begin{array}{c} \Pr\left[\, 1 \leftarrow \mathcal{A}(crs_{com}) \mid crs_{com} \leftarrow \mathsf{COM.Setup}(1^\lambda) \,\right] \\ - \Pr\left[\, 1 \leftarrow \mathcal{A}(crs^{ext}_{com}) \mid (crs^{ext}_{com}, td^{ext}_{com}) \leftarrow \mathsf{COM.Setup}^{Ext}(1^\lambda) \,\right] \end{array} \right|$$

is negligible in λ *and we have that the advantage defined by*

$$\Pr\left[\begin{array}{c} \mathsf{COM.Extract}(td^{ext}_{com}, \mathsf{com}_m) \\ \neq \\ m \end{array} \middle| \begin{array}{c} (crs^{ext}_{com}, td^{ext}_{com}) \leftarrow \mathsf{COM.Setup}^{Ext}(1^\lambda), \\ c \leftarrow \mathcal{A}(crs^{ext}_{com}), \\ \exists! m \in \mathcal{M}, r : \\ c \leftarrow \mathsf{COM.Com}(crs^{ext}_{com}, m; r) \end{array} \right]$$

is zero.

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 NIZK *for NP-relation* $\mathcal{R}$ *is a tuple* $(\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$ *of PPT algorithms, where*

- $\mathsf{NIZK.Setup}(1^\lambda)$ *outputs* (crs, td)*, where* crs *is the CRS and* td *is a trapdoor.*
- $\mathsf{NIZK.Prove}(crs, stmt, wit)$ *generates a proof* π *given* $(stmt, wit) \in \mathcal{R}$.

– $\mathsf{NIZK.Verify}(crs, stmt, \pi)$ *verifies a proof* π *for statement* $stmt$ *and outputs* 0 *or* 1*.*

It is a NIZKPoK *if additionally there are algorithms* NIZK.Sim *and* NIZK.Ext *with*

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

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

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

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

We say that 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 *crs* (and proving consistency as part of the relation)).

A.8 Introduction to the UC Framework

The Universal Composability (UC) framework [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 $\mathcal{F}$ – 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 Π that computes the same function.

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

B Full System Model

In the following, we provide our full system model, apart from the auditable surveillance functionality $\mathcal{F}_{\text{AS}}$ which was already presented in Section 3. We start with some preliminary remarks in Appendix B.1. Then we present the protocol Π_{AS} that realizes $\mathcal{F}_{\text{AS}}$ in Appendix B.2. We continue by presenting the auditable decryption functionality $\mathcal{F}_{\text{AD}}$ in Appendix B.3 and the protocol Π_{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 $\mathcal{F}_{\text{AS}}$, $\mathcal{F}_{\text{AD}}$ and $\mathcal{F}_{\text{ASTE}}$:

- 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 *sid*s and *ssid*s 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 $(\textsc{Output}, \textsc{TaskName}, \mathrm{P})$ to the adversary and wait for message $(\textsc{Allow}, \textsc{TaskName}, \mathrm{P})$ from the adversary. Then send (*value*) to P.

[23] The functionality can do that because it knows the party identifiers (*pid*s) of all parties in the system

Global Functionality $\mathcal{G}_{\text{CLOCK}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $P = (pid, sid)$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} \coloneqq \emptyset$ and $F \coloneqq \emptyset$.

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

Synchronization:

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

Party Management:

- Upon receiving $(\text{REGISTER}, sid_C)$ from some party P_i (or from $\mathcal{A}$ on behalf of a corrupted P_i), set $\mathcal{P} = \mathcal{P} \cup \{P_i\}$. Return $(\text{REGISTER}, sid_C, P_i)$ to the caller.
- Upon receiving $(\text{DE-REGISTER}, sid_C)$ from some party $P_i \in \mathcal{P}$, the functionality updates $\mathcal{P} \coloneqq \mathcal{P} \setminus \{P_i\}$ and returns $(\text{DE-REGISTER}, sid_C, P_i)$ to P_i.
- Upon receiving $(\text{IS-REGISTERED}, sid_C)$ from some party P_i, return $(\text{REGISTER}, sid_C, b)$ to the caller, where the bit b is 1 if and only if $P_i \in \mathcal{P}$.
- Upon receiving $(\text{GET-REGISTERED}, sid_C)$ from $\mathcal{A}$, the functionality returns the response $(\text{GET-REGISTERED}, sid_C, \mathcal{P})$ to $\mathcal{A}$.
- Upon receiving $(\text{REGISTER}, sid_C)$ from a functionality $\mathcal{F}$ (with session-id sid), update $F \coloneqq F \cup \{(\mathcal{F}, sid)\}$.
- Upon receiving $(\text{DE-REGISTER}, sid_C)$ from a functionality $\mathcal{F}$ (with session-id sid), update $F \coloneqq F \setminus \{(\mathcal{F}, sid)\}$.
- Upon receiving $(\text{GET-REGISTERED-F}, sid_C)$ from $\mathcal{A}$, return $(\text{GET-REGISTERED-F}, sid_C, F)$ to $\mathcal{A}$

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

Fig. 7: The global clock functionality from [5]

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

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

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

Fig. 8: Random oracle functionality from [12]

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

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

1. When activated for the first time on input $(\textsc{value}, sid)$, 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 $\mathcal{F}_{\mathrm{BB}}$

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

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

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

Retrieve by UID: Upon receiving $(\textsc{Retrieve}, uid)$ from party P_j (including the adversary), generate public delayed output $(\textsc{Retrieve}, P_i, uid, v)$ to P_j where (P_i, v) is such that $(P_i, uid, v) \in \mathsf{L}_{\mathsf{BB}}$ or $(\perp, \perp)$ if no entry is recorded in L_{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 Π_{AS}

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

$wit := (secret, sec_2, \mathsf{decom}_{\mathrm{sec}}, \mathsf{decom}_{sec_2}, r)$
$stmt := (sec_1, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{sec_2}, \mathsf{pk}, ct, crs_{\mathrm{com}})$

$\mathcal{R}_{ss}^{AS} = \{(stmt, wit) \mid$

- $secret = sec_1 \oplus sec_2$
- $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}, secret) = 1$
- $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_2}, \mathsf{decom}_{sec_2}, sec_2) = 1$
- $ct = \mathsf{TPKE.Enc}(\mathsf{pk}, sec_2; r)$

$\}$

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

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

- Relation $\mathcal{R}_{ss}^{AS}$ (see Fig. 11) is used in Store Secret by a user to prove to the system operator that they honestly encrypted the partial secret sec_2
- Relation $\mathcal{R}_{zk}^{AS}$ (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 Π_{AS} that UC-realizes the auditable surveillance functionality $\mathcal{F}_{\mathrm{AS}}$.

Details of the relation $\mathcal{R}_{zk}^{AS}$

$\mathit{wit} := (\mathit{uid}, \mathit{secret}, \mathit{wit}', \sigma_{\mathrm{ss}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \mathsf{com}_{\mathrm{sec}},$
$\mathsf{decom}_{\mathrm{sec}}, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}})$
$\mathit{stmt} := (\mathit{vper}, \mathit{stmt}', \mathsf{vk}_{\mathrm{SO}}, \mathit{crs}_{\mathrm{com}})$

$\mathcal{R}_{zk}^{AS} = \{(\mathit{stmt}, \mathit{wit}) \mid$

- $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{SO}}, (\mathsf{com}_{\mathrm{uid}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{\mathrm{vper}}), \sigma_{\mathrm{ss}}) = 1$
- $\mathsf{COM.Open}(\mathit{crs}_{\mathrm{com}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \mathit{uid}) = 1$
- $\mathsf{COM.Open}(\mathit{crs}_{\mathrm{com}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}, \mathit{secret}) = 1$
- $\mathsf{COM.Open}(\mathit{crs}_{\mathrm{com}}, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}}, \mathit{vper}) = 1$
- $\mathit{wit}_R = (\mathit{uid}, \mathit{secret}, \mathit{wit}')$
- $\mathit{stmt}_R = (\mathit{vper}, \mathit{stmt}')$
- $\mathcal{R}(\mathit{stmt}_R, \mathit{wit}_R) = 1$

$\}$

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

Π_{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 $W^{\mathrm{pub}} \leftarrow f_t(W)$.
- 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
- $\mathcal{S}$ — Space from which the secrets are drawn
- Signature scheme $\Sigma = (\Sigma.\mathsf{Gen}, \Sigma.\mathsf{Sign}, \Sigma.\mathsf{Vfy})$, where all algorithms are PPT and $\Sigma.\mathsf{Vfy}$ is deterministic.
- Commitment scheme $\mathsf{COM} = (\mathsf{COM.Setup}, \mathsf{COM.Com}, \mathsf{COM.Open})$, where all algorithms are PPT and $\mathsf{COM.Open}$ is deterministic.
- Threshold Public Key Encryption scheme $\mathsf{TPKE} = (\mathsf{TPKE.Gen}, \mathsf{TPKE.Enc}, \mathsf{TPKE.Dec}, \mathsf{TPKE.TDec}, \mathsf{TPKE.ShareVer}, \mathsf{TPKE.Combine}, \mathsf{TPKE.Rand}, \mathsf{TPKE.Sk2Pk})$, where all are PPT and $\mathsf{TPKE.Dec}$, $\mathsf{TPKE.TDec}$, $\mathsf{TPKE.ShareVer}$, $\mathsf{TPKE.Combine}$ and $\mathsf{TPKE.Sk2Pk}$ are deterministic.
- Public Key Encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$, where all are PPT and $\mathsf{PKE.Dec}$ is deterministic.
- Non-interactive Zero-Knowledge scheme $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$, where all are PPT and $\mathsf{NIZK.Verify}$ is deterministic.
- $\mathcal{R}$ — an NP relation. (The code threats $\mathcal{R}$ as a binary function.)
- Relations $\mathcal{R}_{ss}^{AS}$ (see Fig. 11), and $\mathcal{R}_{zk}^{AS}$ (see Fig. 12)

- System pids: pid_{SO}, pid_{J}, pid_{AU}

States of the Parties:

- Each user stores:
  - CRS: $crs = (crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}})$
  - SO public key $\mathsf{vk}_{\mathrm{SO}}$ (to verify signatures from SO)
  - Public threshold encryption key pk
  - Own signing keypair: $(\mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}})$
  - Registration data: $(uid, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}})$
  - List of own secrets. Entries are of the form $(secret, \mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}, vper, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}}, \sigma_{\mathrm{ss}})$
- LE stores:
  - List of warrants. Entries are of the form $(W, b, \widetilde{W}, \sigma_{\widetilde{W}})$
  - List of ciphertexts requested for decryption. Entries are of the form $(uid_i, vper_i, ct_i, sec_{1,i})$
- J stores:
  - Signature keypair $(\mathsf{vk}_{\mathrm{J}}, \mathsf{sk}_{\mathrm{J}})$ (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 $(\widetilde{W}, \sigma_{\widetilde{W}})$
- SO stores:
  - CRS: $crs = (crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}})$
  - Public verification key of the judge: vk_{J}
  - Public threshold encryption key pk
  - Signature keypair $(\mathsf{vk}_{\mathrm{SO}}, \mathsf{sk}_{\mathrm{SO}})$. Messages are either of the form $\mathsf{com}_{\mathrm{uid}}$ or of the form $(\mathsf{com}_{\mathrm{uid}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{\mathrm{vper}})$
  - List of registered users. Entries are of the form $(uid, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}}, \mathsf{vk}_{\mathrm{U}})$
  - List of stored partial secrets. Entries are of the form $(uid, vper, ct, \sigma_{\mathrm{U}}, sec_1, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}, \sigma_{sec_1})$

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

- $crs_{\mathrm{zk}}^{AS} \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$
- $crs_{\mathrm{com}} \leftarrow \mathsf{COM.Setup}(1^\lambda)$¸
- Return $\left(crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}}\right)$

System Init:

- Input SO: $(\textsc{Init}, \mathrm{SO})$
- Input J: $(\textsc{Init}, \mathrm{J})$
- Input AU: $(\textsc{Init}, \mathrm{AU})$
- 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 $(\mathsf{vk}_{\mathrm{SO}}, \mathsf{sk}_{\mathrm{SO}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$ and store it
  2. J: Create signing keypair $(\mathsf{vk}_{\mathrm{J}}, \mathsf{sk}_{\mathrm{J}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$ and store it
  3. Initialize $\mathcal{F}_{\mathrm{AD}}$:
     $\mathrm{SO} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Init}, \mathrm{SO}, \mathsf{vk}_{\mathrm{SO}})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{SO}$: $(\textsc{InitFinished})$
     $\mathrm{J} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Init}, \mathrm{J}, \mathsf{vk}_{\mathrm{J}})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{J}$: $(\textsc{InitFinished})$
     $\mathrm{AU} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Init}, \mathrm{AU})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{AU}$: $(\textsc{InitFinished})$
  4. SO: Get CRS and keys
     $\mathrm{SO} \rightarrow \mathcal{F}_{\mathrm{CRS}}$: $(\textsc{value})$
     $\mathcal{F}_{\mathrm{CRS}} \rightarrow \mathrm{SO}$: $\left(crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}}\right)$
     $\mathrm{SO} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetPK})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{SO}$: $(\textsc{GotPK}, \mathsf{pk})$
     $\mathrm{SO} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetSKeys})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{SO}$: $(\textsc{GotSKeys}, \mathsf{vk}_{\mathrm{SO}}, \mathsf{vk}_{\mathrm{J}})$
  5. SO: Store $crs = (crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}})$, pk and vk_{J}
  6. J: Get CRS
     $\mathrm{J} \rightarrow \mathcal{F}_{\mathrm{CRS}}$: $(\textsc{value})$
     $\mathcal{F}_{\mathrm{CRS}} \rightarrow \mathrm{J}$: $\left(crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}}\right)$
  7. J: Store $crs = (crs_{\mathrm{com}})$

– Output SO: $(\textsc{InitFinished})$
– Output J: $(\textsc{InitFinished})$
– Output AU: $(\textsc{InitFinished})$

Party Init:
– Input some Party P: $(\textsc{PInit})$
– Behavior P:
  1. Get CRS and keys
     $\mathrm{P} \rightarrow \mathcal{F}_{\mathrm{CRS}}$: $(\textsc{value})$
     $\mathcal{F}_{\mathrm{CRS}} \rightarrow \mathrm{P}$: $\left(crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}}\right)$
     $\mathrm{P} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetSKeys})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{P}$: $(\textsc{GotSKeys}, \mathsf{vk}_{\mathrm{SO}}, \mathsf{vk}_{\mathrm{J}})$
     $\mathrm{P} \rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetPK})$
     $\mathcal{F}_{\mathrm{AD}} \rightarrow \mathrm{P}$: $(\textsc{GotPK}, \mathsf{pk})$
  2. Store $crs = (crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}})$ and $(\mathsf{pk}, \mathsf{vk}_{\mathrm{J}}, \mathsf{vk}_{\mathrm{SO}})$

– Output to P: $(\textsc{PInitFinished})$

User Registration:
– Input U: $(\textsc{Register}, uid)$

– Input SO: $\big(\textsc{Register}, \mathit{uid}'\big)$
– Behavior:
1. U: If no keypair is stored yet, generate keypair $(\mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}}) \leftarrow \Sigma.\mathsf{Gen}(1^{\lambda})$ and store it
2. U: Send (uid) to SO
3. SO: If $\mathit{uid} \neq \mathit{uid}'$, abort (wrong inputs)
4. SO: If there is already an entry $(\mathit{uid}, \cdot, \cdot, \cdot)$ in the list of registered users, abort (User already successfully registered.)
5. SO: Send $(\textsc{uid_ok})$ to U
6. $\mathrm{U} \to \mathcal{F}_{\mathrm{BB}}$: $(\textsc{Register}, \mathit{uid}, \mathsf{vk}_{\mathrm{U}})$
7. U: Check if registration was successful:
   $\mathrm{U} \to \mathcal{F}_{\mathrm{BB}}$: $(\textsc{Retrieve}, \mathit{uid})$
   $\mathcal{F}_{\mathrm{BB}} \to \mathrm{U}$: $(\textsc{Retrieve}, {\mathit{pid}_{\mathrm{U}}}^{*}, \mathit{uid}, \mathsf{vk}^{*}_{\mathrm{U}})$
8. U: If ${\mathit{pid}_{\mathrm{U}}}^{*} \neq \mathit{pid}_{\mathrm{U}}$ or $\mathsf{vk}^{*}_{\mathrm{U}} \neq \mathsf{vk}_{\mathrm{U}}$, abort (UID already taken by someone else)
9. U: Send $(\textsc{Ok})$ to SO
10. $\mathrm{SO} \to \mathcal{F}_{\mathrm{BB}}$: $(\textsc{Retrieve}, \mathit{uid})$
    $\mathcal{F}_{\mathrm{BB}} \to \mathrm{SO}$: $(\textsc{Retrieve}, {\mathit{pid}_{\mathrm{U}}}^{*}, \mathit{uid}, \mathsf{vk}^{*}_{\mathrm{U}})$
11. SO: If ${\mathit{pid}_{\mathrm{U}}}^{*} \neq \mathit{pid}_{\mathrm{U}}$, abort (UID already taken by another user)
12. SO: Set $\mathsf{vk}_{\mathrm{U}} \coloneqq \mathsf{vk}^{*}_{\mathrm{U}}$.
13. SO: Commit-and-Sign uid:
    $(\mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}) = \mathsf{COM.Com}(\mathit{crs}_{\mathrm{com}}, \mathit{uid})$ and $\sigma_{\mathrm{reg}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{SO}}, \mathsf{com}_{\mathrm{uid}})$
14. SO: Send $(\sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}})$ to U
15. U: Get public key of SO:
    $\mathrm{U} \to \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetSKeys})$
    $\mathcal{F}_{\mathrm{AD}} \to \mathrm{U}$: $(\textsc{GotSKeys}, \mathsf{vk}_{\mathrm{SO}}, \mathsf{vk}_{\mathrm{J}})$
16. U: Store $\mathsf{vk}_{\mathrm{SO}}$
17. U: Get CRS:
    $\mathrm{U} \to \mathcal{F}_{\mathrm{CRS}}$: $(\textsc{value})$
    $\mathcal{F}_{\mathrm{CRS}} \to \mathrm{U}$: $\big(\mathit{crs}^{AS}_{\mathrm{zk}}, \mathit{crs}_{\mathrm{com}}\big)$
18. U: Store $\mathit{crs} = (\mathit{crs}^{AS}_{\mathrm{zk}}, \mathit{crs}_{\mathrm{com}})$
19. U: Get public key of TPKE
    $\mathrm{U} \to \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetPK})$
    $\mathcal{F}_{\mathrm{AD}} \to \mathrm{U}$: $(\textsc{GotPK}, \mathsf{pk})$
20. U: Store pk
21. U: Verify commitment and signature: If $\mathsf{COM.Open}(\mathit{crs}_{\mathrm{com}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \mathit{uid}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{SO}}, \sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}) = 0$, abort.
22. U: Store $(\mathit{uid}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}})$
23. SO: Store $(\mathit{uid}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}}, \mathsf{vk}_{\mathrm{U}})$

– Output U: $(\textsc{Registered})$
– Output SO: $(\textsc{Registered})$

Store Secret:
– Input U: $(\textsc{StoreSecret}, \mathit{uid}, \mathit{vper})$

- Input SO: (StoreSecret, $vper'$)
- Behavior:
  1. U: Draw part of *secret*: $sec_2 \xleftarrow{r} \mathcal{S}$
  2. U: Commit on sec_2: $(\mathsf{com}_{sec_2}, \mathsf{decom}_{sec_2}) \leftarrow \mathsf{COM.Com}(crs_{\mathrm{com}}, sec_2)$
  3. U: Commit on *vper*: $(\mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}}) \leftarrow \mathsf{COM.Com}(crs_{\mathrm{com}}, vper)$
  4. U: Send $(uid, \sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \mathsf{com}_{sec_2}, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}})$ to SO
  5. SO: Verify $\mathsf{com}_{\mathrm{vper}}$: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}}, vper') = 0$, abort.
  6. SO: If $(uid, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}}, \cdot)$ is not in the list of registered Users, abort.
  7. SO: If there already exists an entry $(uid, vper, \cdot, \cdot)$ in the list of stored secrets, abort (user already registered a secret for the current validity period).
  8. SO: Draw part of *secret*: $sec_1 \xleftarrow{r} \mathcal{S}$
  9. SO: Send (sec_1) to U
  10. U: Compute *secret* as $secret \coloneqq sec_1 \oplus sec_2$ ($\oplus$ can be XOR or $+$, but it must hold that $secret \in \mathcal{S}$).
  11. U: Commit on *secret*: $(\mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}) \leftarrow \mathsf{COM.Com}(crs_{\mathrm{com}}, secret)$
  12. U: Commit and sign sec_1: $(\mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}) \leftarrow \mathsf{COM.Com}(crs_{\mathrm{com}}, sec_1)$ and $\sigma_{sec_1} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{U}}, \mathsf{com}_{sec_1})$
  13. U: Encrypt secret: $ct \leftarrow \mathsf{TPKE.Enc}(\mathsf{pk}, sec_2; r)$ for fresh randomness r
  14. U: Sign ciphertext: $\sigma_{\mathrm{U}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{U}}, (ct, uid, vper))$
  15. U: Assemble ZK-Witness $wit_{\mathrm{ss}} := (secret, sec_2, \mathsf{decom}_{\mathrm{sec}}, \mathsf{decom}_{sec_2}, r)$
  16. U: Assemble ZK-Statement: $stmt_{\mathrm{ss}} := (sec_1, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{sec_2}, \mathsf{pk}, ct, crs_{\mathrm{com}})$
  17. U: Compute Proof: $\pi_{\mathrm{ss}} \leftarrow \mathsf{NIZK.Prove}(crs_{\mathrm{zk}}^{AS}, stmt_{\mathrm{ss}}, wit_{\mathrm{ss}}, \mathcal{R}_{ss}^{AS})$
  18. U: Send $(ct, \mathsf{com}_{\mathrm{sec}}, \pi_{\mathrm{ss}}, \sigma_{\mathrm{U}}, \sigma_{sec_1}, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1})$ to SO
  19. SO: Assemble ZK-Statement: $stmt_{\mathrm{ss}} := (sec_1, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{sec_2}, \mathsf{pk}, ct, crs_{\mathrm{com}})$
  20. SO: Verify Proof: If $\mathsf{NIZK.Verify}(crs_{\mathrm{zk}}^{AS}, stmt_{\mathrm{ss}}, \pi_{\mathrm{ss}}, \mathcal{R}_{ss}^{AS}) = 0$, abort.
  21. SO: For *uid*, retrieve entry $(uid, \cdot, \cdot, \cdot, \mathsf{vk}_{\mathrm{U}})$ from list of registered users.
  22. SO: Verify Commitment: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}, sec_1) = 0$, abort.
  23. SO: Verify Signatures: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U}}, (ct, uid, vper), \sigma_{\mathrm{U}}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U}}, \mathsf{com}_{sec_1}, \sigma_{sec_1}) = 0$, abort.
  24. SO: $\sigma_{\mathrm{ss}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{SO}}, (\mathsf{com}_{\mathrm{uid}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{vper}))$
  25. SO: Send (σ_{ss}) to U
  26. U: Verify signature: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{SO}}, (\mathsf{com}_{\mathrm{uid}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{com}_{\mathrm{vper}}), \sigma_{\mathrm{ss}}) = 0$, abort.
  27. U: Store $(secret, \mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}, vper, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}}, \sigma_{\mathrm{ss}})$
  28. SO: Store $(uid, vper, ct, \sigma_{\mathrm{U}}, sec_1, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}, \sigma_{sec_1})$
- Output U: (SecretStored, *secret*)
- Output SO: (SecretStored, *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 $0 \leftarrow f_p(W)$, abort the protocol (Warrant not allowed by policy function).
- Output J: (RequestWarrant, W)
- Input J: (b)
- Behavior:
  1. J: If $b = 0$, set $\sigma_W = \perp$. Else, set $\sigma_W = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, W)$
  2. J: Store (W, b) in list of already requested preliminary warrants
  3. J: Send (b, σ_W) to LE
  4. LE: Store $(W, b, \perp, \perp)$ in list of warrants
  5. LE: If $b = 0$, skip the next steps and directly go to output
  6. LE $\rightarrow \mathcal{F}_{\mathrm{AD}}$: (GetSKeys)
     $\mathcal{F}_{\mathrm{AD}} \rightarrow$ LE: (GotSKeys, $\mathsf{vk}_{\mathrm{SO}}, \mathsf{vk}_{\mathrm{J}}$)
  7. LE: Verify Signature: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{J}}, W, \sigma_W) = 0$, abort.
  8. LE: Send (W, σ_W) to SO
  9. SO: Verify Signature: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{J}}, W, \sigma_W) = 0$, abort.
  10. SO: Parse preliminary warrant: $(W_1, \ldots, W_v) \leftarrow W$ and $(uid_i, vper_i, meta_i) \leftarrow W_i$
  11. SO: For each $(uid_i, vper_i, \cdot)$ entry in W, retrieve the matching $(uid_i, vper_i, ct_i, \sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}})$ entry from internal storage. If none exists, abort (User has not Stored a Secret)
  12. SO: Build enhanced warrant $\widetilde{W}$ by adding ct_i to each W_i
  13. SO: Send $(\widetilde{W}, \{\sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}\}_{i\in\{1,\ldots,v\}})$ to LE
  14. LE: Parse enhanced warrant: $(\widetilde{W_1}, \ldots, \widetilde{W_v}) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W_i}$
  15. LE: Build preliminary warrant W' out of $\widetilde{W}$ by deleting ct_i from each $\widetilde{W_i}$
  16. LE: If $W' \neq W$, abort. (SO sent wrong $\widetilde{W}$)
  17. LE: For i from 1 to v:
      - LE $\rightarrow \mathcal{F}_{\mathrm{BB}}$: (Retrieve, uid_i)
        $\mathcal{F}_{\mathrm{BB}} \rightarrow$ LE: $\big($Retrieve, $uid_i, \mathsf{vk}_{\mathrm{U,i}}\big)$
      - Check signatures: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, (ct_i, uid_i, vper_i), \sigma_{\mathrm{U,i}}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, \mathsf{com}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}) = 0$, abort
      - Check commitment: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, sec_{1,i}) = 0$, abort.
  18. LE: Send $(\widetilde{W}, \{\sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}\}_{i\in\{1,\ldots,v\}})$ to J
  19. J: Build preliminary warrant W'' out of $\widetilde{W}$ by deleting ct_i from each $\widetilde{W_i}$
  20. J: If there is no entry $(W'', 1)$ in the list of already processed preliminary warrants, abort.

21. J: Check if $(\widetilde{W}, \cdot)$ 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 $\rightarrow \mathcal{F}_{\mathrm{BB}}$: $(\textsc{Retrieve}, uid_i)$
      $\mathcal{F}_{\mathrm{BB}} \rightarrow$ J: $\big(\textsc{Retrieve}, uid_i, \mathsf{vk}_{\mathrm{U,i}}\big)$
    - Check signatures: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, (ct_i, uid_i, vper_i), \sigma_{\mathrm{U,i}}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, \mathsf{com}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}) = 0$, abort
    - Check commitment: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, sec_{1,i}) = 0$, abort.
24. J: Sign enhanced warrant: $\sigma_{\widetilde{W}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, \widetilde{W})$
25. J: Store $(\widetilde{W}, \sigma_{\widetilde{W}})$ in list of stored enhanced warrants
26. J: Send $\big(\sigma_{\widetilde{W}}\big)$ 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 $\rightarrow \mathcal{F}_{\mathrm{AD}}$: $\Big(\textsc{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\Big)$
    $\mathcal{F}_{\mathrm{AD}} \rightarrow$ LE: $(\textsc{Request})$
29. LE: Store each $(uid_i, vper_i, ct_i, sec_{1,i})$ pair for $i \in \{1, \ldots, v\}$

– Output LE: $(\textsc{RequestWarrant}, b)$

Get Secrets:

– Input LE: $(\textsc{GetSecrets}, W)$
– 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 $(uid_i, vper_i, ct_i, \sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_{1,i}}, \mathsf{decom}_{sec_{1,i}}, \sigma_{sec_1,\mathrm{i}})$ from the list of stored partial secrets
4. LE $\rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Retrieve})$
   $\mathcal{F}_{\mathrm{AD}} \rightarrow$ LE: $\Big(\textsc{Retrieve}, \mathsf{L}^{\mathsf{Ready}}_{\mathsf{Requests}}\Big)$
5. For each $(uid_i, vper_i, ct_i, sec_{1,i})$:
   (a) Search for an entry $(ct_j, sec_{2,j})$ in $\mathsf{L}^{\mathsf{Ready}}_{\mathsf{Requests}}$ 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: $(\textsc{GotSecrets}, (secret_1, \ldots, secret_v))$

Get Statistics:

– Input some Party P: $(\textsc{GetStatistics})$
– Behavior P:
1. P $\rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{GetStatistics})$
   $\mathcal{F}_{\mathrm{AD}} \rightarrow$ P: $(\textsc{GotStatistics}, \mathsf{L}_{\mathsf{Stats}})$

– Output to P: $(\textsc{GotStatistics}, \mathsf{L}_{\mathsf{Stats}})$

Audit:
– Input AU: $(\textsc{AuditRequest})$
– Behavior AU:
1. If the party invoking this task has *not* the pid pid_{AU}, then abort.
2. $\mathrm{AU} \to \mathcal{F}_{\mathrm{AD}}$: $(\textsc{AuditRequest})$
   $\mathcal{F}_{\mathrm{AD}} \to \mathrm{AU}$: $(\textsc{AuditAnswer}, \mathsf{L}_{\mathsf{Warrants}})$
3. Build L_{AU} out of $\mathsf{L}_{\mathsf{Warrants}}$ by deleting ct_i from each W_i

– Output AU: $(\textsc{AuditAnswer}, \mathsf{L}_{\mathsf{AU}})$

Prove:
– Input U: $(\textsc{Prove}, stmt_R, wit_R)$
– Behavior U:
1. Parse $(uid, secret, wit') := wit_R$
2. Parse $(vper, stmt') := stmt_R$
3. If $\mathcal{R}(stmt_R, wit_R) \neq 1$, abort
4. For uid, if there is no registration data $(uid, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \cdot)$ stored, abort (User not registered)
5. For $(secret, vper)$, if there is no entry $(secret, \mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}, vper, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}}, \sigma_{\mathrm{ss}})$ in the internal list of encrypted secrets, abort (User has not encrypted a secret for the period)
6. Assemble witness: $wit := (uid, secret, wit', \sigma_{\mathrm{ss}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \mathsf{com}_{\mathrm{sec}}, \mathsf{decom}_{\mathrm{sec}}, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}})$
7. Assemble statement: $stmt := (vper, stmt', \mathsf{vk}_{\mathrm{SO}}, crs_{\mathrm{com}})$
8. Compute proof: $\pi \leftarrow \mathsf{NIZK.Prove}(crs^{AS}_{\mathrm{zk}}, stmt, wit, \mathcal{R}^{AS}_{zk})$

– Output U: $(\textsc{Proof}, stmt_R, wit_R, \pi)$

Verify:
– Input some party P: $(\textsc{Verify}, stmt_R, \pi)$
– Behavior P:
1. If the values $crs^{AS}_{\mathrm{zk}}, crs_{\mathrm{com}}, \mathsf{vk}_{\mathrm{SO}}$ are not stored internally, abort.
2. Parse $(vper, stmt') := stmt_R$
3. Assemble statement: $stmt := (vper, stmt', \mathsf{vk}_{\mathrm{SO}}, crs_{\mathrm{com}})$
4. Verify proof: $b \leftarrow \mathsf{NIZK.Verify}(crs^{AS}_{\mathrm{zk}}, stmt, \pi, \mathcal{R}^{AS}_{zk})$

– Output to P: $(\textsc{Verification}, stmt_R, \pi, b)$

B.3 The Functionality $\mathcal{F}_{\mathbf{AD}}$

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

$\mathcal{F}_{\mathrm{AD}}$

System Parameters:
– t_{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 $\Sigma = (\Sigma.\mathsf{Gen}, \Sigma.\mathsf{Sign}, \Sigma.\mathsf{Vfy})$, where all are PPT and $\Sigma.\mathsf{Vfy}$ is deterministic.
- Threshold Public Key Encryption scheme $\mathsf{TPKE} = (\mathsf{TPKE.Gen}, \mathsf{TPKE.Enc}, \mathsf{TPKE.Dec}, \mathsf{TPKE.TDec}, \mathsf{TPKE.ShareVer}, \mathsf{TPKE.Combine}, \mathsf{TPKE.Rand}, \mathsf{TPKE.Sk2Pk})$, where all are PPT and $\mathsf{TPKE.Dec}$, $\mathsf{TPKE.TDec}$, $\mathsf{TPKE.ShareVer}$, $\mathsf{TPKE.Combine}$ and $\mathsf{TPKE.Sk2Pk}$ are deterministic.
- Public Key Encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$, where all are PPT and $\mathsf{PKE.Dec}$ is deterministic.
- Non-interactive Zero-Knowledge scheme $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$, where all are PPT and $\mathsf{NIZK.Verify}$ is deterministic.
- System pids: pid_{SO}, pid_{J}, pid_{AU}

The functionality has access to a global clock $\mathcal{G}_{\text{CLOCK}}$.

Functionality State:
- t_{last}: Time of last committee handover.
- $(\mathsf{pk}, \mathsf{sk})$: Keypair for TPKE
- crs: Common reference string for NIZK
- $\mathsf{L}_{\mathsf{Warrants}}$: List of Warrants on the blockchain. Entries are of the form $\widetilde{W}$.
- $\mathsf{L}_{\mathsf{Requests}}$: List of Requests for decryption not yet ready for retrieval.
- $\mathsf{L}^{\mathsf{Ready}}_{\mathsf{Requests}}$: Lists of Requests for decryption ready to be retrieved.
- vk_{SO}: Public signing key of SO. Not needed in this functionality, it only exists so parties in Π_{AS} can get it.
- vk_{J}: Public signing key of J. Used to verify warrant signatures and so parties in Π_{AS} can get it.
- $\{d_{pid}\}, pid \in \{N_i\}$: Flag whether the node pid has called 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: $(\text{Init}, \text{SO}, \mathsf{vk}_{\text{SO}})$
- Behavior:
  1. Send (Register) to $\mathcal{G}_{\text{CLOCK}}$
  2. Set $cnum := 0$
  3. Generate encryption key pair: $(\mathsf{pk}, \mathsf{sk}, \{\mathsf{vk}_i\}, \{\mathsf{sk}_i\}) \leftarrow \mathsf{TPKE.Gen}(1^\lambda)$ and store $(\mathsf{pk}, \mathsf{sk})$
  4. Set $t_{\mathsf{last}} := 0$
  5. Send $(\text{Init}, \text{SO}, \mathsf{vk}_{\text{SO}}, \mathsf{pk})$ to $\mathcal{A}$
  6. Store vk_{SO}
- Output SO: (InitFinished)

– Input J: $(\textsc{Init}, \mathrm{J}, \mathsf{vk}_{\mathrm{J}})$
– Behavior:
  1. Send $(\textsc{Init}, \mathrm{J}, \mathsf{vk}_{\mathrm{J}})$ to $\mathcal{A}$
  2. Store vk_{J}
– Output J: (InitFinished)

– Input AU: $(\textsc{Init}, \mathrm{AU})$
– Behavior:
  1. Send $(\textsc{Init}, \mathrm{AU})$ to $\mathcal{A}$
– Output AU: (InitFinished)

Get Public Key:
– Input some Party P: (GetPK)
– Behavior:
  1. Send (GetPK) to the adversary
  2. Retrieve $(\mathsf{pk}, \mathsf{sk}) := (\mathsf{pk}, \mathsf{sk})$
– Output P: $(\textsc{GotPK}, \mathsf{pk})$

Get System Keys:
– Input some Party P: (GetSKeys)
– Behavior:
  1. Send (GetSKeys) to the adversary
  2. Retrieve $\mathsf{vk}_{\mathrm{SO}}$ and vk_{J}
– Output P: $(\textsc{GotSKeys}, \mathsf{vk}_{\mathrm{SO}}, \mathsf{vk}_{\mathrm{J}})$

Request Decryption:
– Input LE: $\left(\textsc{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$
– Behavior:
  1. Check Warrant Signature: $b \leftarrow \Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{J}}, \widetilde{W}, \sigma_{\widetilde{W}})$
  2. If $b = 0$, abort (Warrant not valid)
  3. Parse warrant: $\left(\widetilde{W}_1, \ldots, \widetilde{W}_v\right) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i$
  4. For each i, store ct_i in list of requests $\mathsf{L}_{\mathsf{Requests}}$
  5. Send $(\textsc{Request}, pid, f_t(W), |\widetilde{W}|, v)$ to the adversary, where pid is the id of the calling party
  6. When the adversary allows to deliver output, store $\widetilde{W}$ in List of Warrants $\mathsf{L}_{\mathsf{Warrants}}$
– 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 $\left(\textsc{Retrieve}, \mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}\right)$ to the adversary

– Output LE: $\left(\textsc{Retrieve}, \mathsf{L}^{\mathsf{Ready}}_{\mathsf{Requests}}\right)$

Get Statistics:
– Input some Party P: (GetStatistics)
– Behavior:
1. Send (GetStatistics) to the adversary
2. Initialize empty list $\mathsf{L}_{\mathsf{Stats}}$
3. Retrieve list of Warrants $\mathsf{L}_{\mathsf{Warrants}}$, entries are of the form $\widetilde{W}$
4. For each $\widetilde{W} \in \mathsf{L}_{\mathsf{Warrants}}$: Build W from $\widetilde{W}$, calculate $W^{\mathrm{pub}} \leftarrow f_t(W)$ and append W^{pub} to $\mathsf{L}_{\mathsf{Stats}}$

– Output P (immediately): (GotStatistics, $\mathsf{L}_{\mathsf{Stats}}$)

Audit:
– Input AU: (AuditRequest)
– Behavior:
1. If the party invoking this task has *not* the pid pid_{AU}, then abort.
2. Send (AuditRequest) to the adversary
3. Retrieve list of Warrants $\mathsf{L}_{\mathsf{Warrants}}$

– Output AU (immediately): (AuditAnswer, $\mathsf{L}_{\mathsf{Warrants}}$)

RoleExecute:
– Input N: (ExecuteRole)
– Behavior:
1. Send (ExecuteRole, pid_N) to the adversary
2. Set $d_N \coloneqq 1$
3. If for all honest nodes N_i $d_{N_i} = 1$, execute ClockTick

– Output N: (ExecuteRole)

Handling corruptions:
– Upon receiving a message (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 $d_{N_i} = 1$, execute ClockTick

ClockTick:
1. Reset $d_{pid} \coloneqq 0$ for all nodes
2. Send (Clock-Update, sid) to $\mathcal{G}_{\mathrm{CLOCK}}$
3. Send (Clock-Read, sid) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive (Clock-Read, sid, t_{Now})
4. If $t_{\mathsf{last}} + t_{\mathsf{com}} \leq t_{\mathrm{Now}}$ execute Handover.

Handover:
1. Set $\mathsf{L}^{\mathsf{Ready}}_{\mathsf{Requests}} \coloneqq \mathsf{L}^{\mathsf{Ready}}_{\mathsf{Requests}} \bigcup \mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}$ and clear $\mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}$
2. For each entry $ct_i in \mathsf{L}_{\mathsf{Requests}}$:
   (a) $secret_i \leftarrow \mathsf{TPKE.Dec}(\mathsf{sk}, ct_i)$
   (b) Add $(ct_i, secret_i)$ to $\mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}$
3. Clear $\mathsf{L}_{\mathsf{Requests}}$
4. Set time of last handover: $t_{\mathsf{last}} \coloneqq t_{\mathrm{Now}}$

B.4 The Protocol Π_{AD}

Details of the relation $\mathcal{R}_W^{AD}$

$wit := (\widetilde{W}, r, \sigma_{\widetilde{W}}, \{ct_i, r_i\}_{i \in [v]})$
$stmt := (\mathsf{ek}_{\text{AU}}, \mathsf{vk}_{\text{J}}, W^{\text{enc}}, W^{\text{pub}}, \{\widehat{ct}_i\})$

$\mathcal{R}_W^{AD} = \{(stmt, wit) \mid$
- $W^{\text{enc}} = \mathsf{PKE.Enc}(\mathsf{ek}_{\text{AU}}, \widetilde{W}; r)$
- $W^{\text{pub}} = f_t(W)$
- $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\text{J}}, \widetilde{W}, \sigma_{\widetilde{W}})$
- W is $\widetilde{W}$ but without the ciphertexts
- For each $i \in [v]$:
  - $\widehat{ct}_i \leftarrow \mathsf{TPKE.Rand}(ct_i; r_i)$

$\}$

Fig. 13: ZK-Relation $\mathcal{R}_W^{AD}$

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

- Relation $\mathcal{R}_W^{AD}$ (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 $\mathcal{R}_{KG1}^{AD}$ and $\mathcal{R}_{KG2}^{AD}$ (see Fig. 14) are used during initial key generation to ensure the resulting keypair is generated properly
- Relation $\mathcal{R}_{KS}^{AD}$ (see Fig. 15) is used during each committee handover to ensure correct resharing of the secret key
- Relation $\mathcal{R}_{Dec}^{AD}$ (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 $\mathcal{F}_{\text{AD}}$.

Details of the relation $\mathcal{R}_{KG1}^{AD}$

$wit \coloneqq (\mathsf{sk}_i, \{r_j\}_{j\in[n]}, F)$
$stmt \coloneqq (\mathsf{pk}_i, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]})$

$\mathcal{R}_{KG1}^{AD} = \{(stmt, wit) \mid$
– F is a degree t polynomial
– For each $j \in [n]$:
- $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, F(j); r_j)$

$\}$

Details of the relation $\mathcal{R}_{KG2}^{AD}$

$wit \coloneqq (\mathsf{sk}_i, \{r_j\}_{j\in[n]}, \mathsf{dk}_R, F)$
$stmt \coloneqq (\mathsf{pk}_i, \{ct^k\}_{k\in[t+1]}, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]}, \mathsf{ek}_R)$

$\mathcal{R}_{KG2}^{AD} = \{(stmt, wit) \mid$
– $\mathsf{PKE.Sk2Pk}(\mathsf{dk}_R) = \mathsf{ek}_R$
– $\mathsf{sk}_i = \sum_{k=1}^{t+1} \mathsf{PKE.Dec}(\mathsf{dk}_R, ct^k)$
– $\mathsf{TPKE.Sk2Pk}(sk_i) = \mathsf{pk}_i$
– F is a degree t polynomial with $F(0) = \mathsf{sk}_i$
– For each $j \in [n]$:
- $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, F(j); r_j)$

$\}$

Fig. 14: ZK-Relations $\mathcal{R}_{KG1}^{AD}$ and $\mathcal{R}_{KG2}^{AD}$

Details of the relation $\mathcal{R}^{AD}_{KS}$

$wit \coloneqq (\mathsf{dk}_R, \mathsf{sk}_i, \{r_j\}_{j\in[n]}, F, G)$
$stmt \coloneqq (\mathsf{ek}_R, \{ct^k\}_{k\in[t+1]}, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]}, \mathsf{pk}_i)$

$\mathcal{R}^{AD}_{KS} = \{(stmt, wit) \mid$
- $\mathsf{PKE.Sk2Pk}(\mathsf{dk}_R) = \mathsf{ek}_R$
- $\mathsf{TPKE.Sk2Pk}(sk_i) = \mathsf{pk}_i$
- G and F are both degree t polynomials
- $G(i) = F(0) = \mathsf{sk}_i$
- For each $k \in [t+1]$:
  - $sh_k \leftarrow \mathsf{PKE.Dec}(\mathsf{dk}_R, ct^k)$
- G is obtained by interpolating $sh_1, \ldots, sh_{t+1}$
- For each $j \in [n]$:
  - $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, G(j); r_j)$

$\}$

Fig. 15: ZK-Relation $\mathcal{R}^{AD}_{KS}$

Details of the relation $\mathcal{R}^{AD}_{Dec}$

$wit \coloneqq (\mathsf{sk}_i)$
$stmt \coloneqq (\mathsf{vk}_i, ct, ct^*)$

$\mathcal{R}^{AD}_{Dec} = \{(stmt, wit) \mid$
- $\mathsf{TPKE.Sk2Pk}(\mathsf{sk}_i) = \mathsf{vk}_i$
- $ct^* \leftarrow \mathsf{TPKE.TDec}(\mathsf{sk}_i, ct)$

$\}$

Fig. 16: ZK-Relation $\mathcal{R}^{AD}_{Dec}$

Π_{AD}

Building Blocks:
- NIZK proof system $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$

System Parameters:
- n — Number of committee members
- t_{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 $W^{\mathrm{pub}} \leftarrow f_t(W)$.
- Signature scheme $\Sigma = (\Sigma.\mathsf{Gen}, \Sigma.\mathsf{Sign}, \Sigma.\mathsf{Vfy})$, where all algorithms are PPT and $\Sigma.\mathsf{Vfy}$ is deterministic.
- Threshold Public Key Encryption scheme $\mathsf{TPKE} = (\mathsf{TPKE.Gen}, \mathsf{TPKE.Enc}, \mathsf{TPKE.Dec}, \mathsf{TPKE.TDec}, \mathsf{TPKE.ShareVer}, \mathsf{TPKE.Combine}, \mathsf{TPKE.Rand}, \mathsf{TPKE.Sk2Pk})$, where all algorithms are PPT and $\mathsf{TPKE.Dec}$, $\mathsf{TPKE.TDec}$, $\mathsf{TPKE.ShareVer}$, $\mathsf{TPKE.Combine}$ and $\mathsf{TPKE.Sk2Pk}$ are deterministic.
- Public Key Encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$, where all are PPT and $\mathsf{PKE.Dec}$ is deterministic.
- Non-interactive Zero-Knowledge scheme $\mathsf{NIZK} = (\mathsf{NIZK.Setup}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$, where all are PPT and $\mathsf{NIZK.Verify}$ is deterministic.
- ZK-Relations $\mathcal{R}_{W}^{AD}$ (see Fig. 13), $\mathcal{R}_{KG1}^{AD}, \mathcal{R}_{KG2}^{AD}$ (see Fig. 14), $\mathcal{R}_{KS}^{AD}$ (see Fig. 15) and $\mathcal{R}_{Dec}^{AD}$ (see Fig. 16).
- System pids: pid_{SO}, pid_{J}, pid_{AU}, pid_{LE}

State of the Parties:
- LE stores:
  - $(\mathsf{ek}_{\mathrm{LE}}, \mathsf{sk}_{\mathrm{LE}})$
- AU stores:
  - $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}})$

System Setup ($\mathcal{F}_{\mathrm{CRS}}$):
- $crs_{\mathrm{zk}}^{SS} \leftarrow \mathsf{NIZK.Setup}(1^{\lambda})$
- Return crs_{zk}^{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: $(\textsc{Init}, \mathrm{SO}, \mathsf{vk}_{\mathrm{SO}})$
- Behavior SO:
  1. Send $(\textsc{Post}, (\mathrm{OperatorKey}, \mathsf{vk}_{\mathrm{SO}}))$ to $\mathcal{F}_{\mathrm{BCRA}}$
- Output SO: $(\textsc{InitFinished})$

– Input J: $(\textsc{Init}, \mathrm{J}, \mathsf{vk}_{\mathrm{J}})$
– Behavior J:
  1. Send $(\textsc{Post}, (\mathrm{JudgeKey}, \mathsf{vk}_{\mathrm{J}}))$ to $\mathcal{F}_{\mathrm{BCRA}}$
– Output J: $(\textsc{InitFinished})$

– Input AU: $(\textsc{Init}, \mathrm{AU})$
– Behavior AU:
  1. Generate $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}}) \leftarrow \mathsf{PKE.Gen}(1^{\lambda})$
  2. Send $(\textsc{Post}, (\mathrm{AuditorKey}, \mathsf{ek}_{\mathrm{AU}}))$ to $\mathcal{F}_{\mathrm{BCRA}}$
  3. Store $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}})$
– Output AU: $(\textsc{InitFinished})$

Get Public Key:
– Input some Party P: $(\textsc{GetPK})$
– Behavior:
  1. Send $(\textsc{Read})$ to $\mathcal{F}_{\mathrm{BCRA}}$ and receive $(\textsc{Ordered})$
  2. For each role R_i in the first key generation committee do the following:
     (a) Find $(\texttt{roleassign}, t, (\textsc{Generate}, R_i, \mathsf{ek}_R, \mathsf{vk}_R))$ in $\textsc{Ordered}$ and retrieve vk_R
     (b) Find (R, msg, σ) with $msg \coloneqq (\textsc{KeyGen1}, \left(\{ct_j\}_{j \in [n]}, \pi_{KG}\right))$ in $\textsc{Ordered}$
     (c) Check $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, msg, \sigma) = 1$, otherwise ignore this role
     (d) Assemble ZK-Statement:
     $stmt_{KG} \coloneqq (\mathsf{pk}_i, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j \in [n]})$
     (e) Check $\mathsf{NIZK.Verify}(crs^{SS}_{\mathrm{zk}}, stmt_{KG}, \pi_{KG}, \mathcal{R}^{AD}_{KG1}) = 1$, otherwise ignore this role
     (f) Add $\{ct_j\}_{j \in [n]}$ to $\mathsf{L}^{ct}_{\mathsf{Qual}}$
  3. Sort $\mathsf{L}^{ct}_{\mathsf{Qual}}$ lexicographically
  4. For each role R_i in the second key generation committee do the following:
     (a) Find $(\texttt{roleassign}, t, (\textsc{Generate}, R_i, \mathsf{ek}_R, \mathsf{vk}_R))$ in $\textsc{Ordered}$ and retrieve $\mathsf{ek}_R, \mathsf{vk}_R$
     (b) Find (R, msg, σ) with $msg \coloneqq (\textsc{KeyGen2}, \left(\mathsf{pk}_i, \{ct_j\}_{j \in [n]}, \pi_{KG}\right))$ in $\textsc{Ordered}$
     (c) Check $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, msg, \sigma) = 1$, otherwise ignore this role
     (d) Retrieve the first $t+1$ ciphertexts $ct^1_i, \ldots, ct^{t+1}_i$ for this role from $\mathsf{L}^{ct}_{\mathsf{Qual}}$ as $\left\{ct^k\right\}_{k \in [t+1]}$
     (e) Assemble ZK-Statement:
     $stmt_{KG} \coloneqq (\mathsf{pk}_i, \left\{ct^k\right\}_{k \in [t+1]}, \{ct_i\}_{i \in [n]}, \mathsf{ek}_R)$
     (f) Check $\mathsf{NIZK.Verify}(crs^{SS}_{\mathrm{zk}}, stmt_{KG}, \pi_{KG}, \mathcal{R}^{AD}_{KG2}) = 1$, otherwise ignore this role
     (g) Add pk_i to $\mathsf{L}_{\mathsf{Qual}}$
  5. Sort $\mathsf{L}_{\mathsf{Qual}}$ lexicographically

6. Reconstruct pk via Lagrange-interpolation from the first $t+1$ entries in $\mathsf{L}_{\mathsf{Qual}}$
- Output P: (GotPK, pk)

Get System Keys:
- Input some Party P: (GetSKeys)
- Behavior P:
  1. Send (Read) to $\mathcal{F}_{\mathrm{BCRA}}$ and receive (Ordered)
  2. Find $(pid_{\mathrm{SO}}, t_1, (\mathrm{OperatorKey}, \mathsf{vk}_{\mathrm{SO}}))$ in Ordered, otherwise abort
  3. Find $(pid_{\mathrm{J}}, t_2, (\mathrm{JudgeKey}, \mathsf{vk}_{\mathrm{J}}))$ in Ordered, otherwise abort
- Output P: (GotSKeys, $\mathsf{vk}_{\mathrm{SO}}, \mathsf{vk}_{\mathrm{J}}$)

Request Decryption:
- Input LE: $\left(\text{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$
- Behavior LE:
  1. Initialize empty list $\mathsf{L}_{\mathsf{CommitteePK}}$
  2. If $\mathsf{sk}_{\mathrm{LE}} = \perp$ run $(\mathsf{ek}_{\mathrm{LE}}, \mathsf{sk}_{\mathrm{LE}}) \leftarrow \mathsf{PKE.Gen}(1^{\lambda})$
  3. Send (Read) to $\mathcal{F}_{\mathrm{BCRA}}$ and receive (Ordered)
  4. Find $(pid_{\mathrm{J}}, t_2, (\mathrm{JudgeKey}, \mathsf{vk}_{\mathrm{J}}))$ in Ordered, otherwise abort
  5. Find $(pid_{\mathrm{AU}}, t_3, (\mathrm{AuditorKey}, \mathsf{ek}_{\mathrm{AU}}))$ in Ordered, otherwise abort
  6. Build preliminary warrant W out of $\widetilde{W}$ by deleting the ciphertext from each entry.
  7. Compute transparency function of warrant: $W^{\mathrm{pub}} \leftarrow f_t(W)$
  8. Encrypt warrant to the auditor: $W^{\mathrm{enc}} \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{\mathrm{AU}}, \widetilde{W}; r)$ with randomness $r \xleftarrow{\mathrm{r}} \mathcal{R}$
  9. Parse warrant: $\left(\widetilde{W}_1, \ldots, \widetilde{W}_v\right) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i$
  10. For each $i \in [v]$, re-randomize $\widehat{ct}_i \leftarrow \mathsf{TPKE.Rand}(ct_i; r_i)$
  11. Assemble ZK-Witness: $wit_{\widetilde{W}} := (\widetilde{W}, r, \sigma_{\widetilde{W}}, \{ct_i, r_i\}_{i\in[v]})$
  12. Assemble ZK-Statement: $stmt_{\widetilde{W}} := (\mathsf{ek}_{\mathrm{AU}}, \mathsf{vk}_{\mathrm{J}}, W^{\mathrm{enc}}, W^{\mathrm{pub}}, \{\widehat{ct}_i\}_{i\in[v]})$
  13. Compute Proof: $\pi_{\widetilde{W}} \leftarrow \mathsf{NIZK.Prove}(crs^{SS}_{\mathrm{zk}}, stmt_{\widetilde{W}}, wit_{\widetilde{W}}, \mathcal{R}^{AD}_{W})$
  14. Send $\big(\text{Post}, (\text{Request}, W^{\mathrm{pub}}, W^{\mathrm{enc}}, \{\widehat{ct}_i\}_{i\in[v]}, \pi_{\widetilde{W}}, \mathsf{ek}_{\mathrm{LE}})\big)$ to $\mathcal{F}_{\mathrm{BCRA}}$
- Output LE: (Request)

Retrieve Secret:
- Input LE: (Retrieve)
- Behavior of LE:
  1. Send (Read) to $\mathcal{F}_{\mathrm{BCRA}}$ and receive (Ordered)
  2. For each entry $(pid, t, R, (\text{Request}, C), \sigma)$ in Ordered:
     (a) Find $(\texttt{roleassign}, t, R, \mathsf{ek}_R, \mathsf{vk}_R)$ in Ordered
     (b) Verify $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, (\text{Request}, C), \sigma)$, otherwise ignore this entry
     (c) Find $(pid, t, R, msg, \sigma')$ with $msg := (\text{KeyShare}, (\mathsf{vk}_i, \{ct_k\}, \pi_{KS}))$ in Ordered

(d) Verify $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, msg, \sigma')$, otherwise ignore this entry
(e) Verify π_{KS}, otherwise ignore this entry
(f) Decrypt $(ct, ct^*, \pi_{Dec}) \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathrm{LE}}, C)$
(g) Assemble ZK-Statement:
$stmt_{Dec} := (\mathsf{vk}, ct, ct^*)$
(h) Verify $\mathsf{NIZK.Verify}(crs_{\mathrm{zk}}^{SS}, stmt_{Dec}, \pi_{Dec}, \mathcal{R}_{Dec}^{AD})$, otherwise ignore this entry
(i) Add ct^* to $\mathsf{L}_{\mathsf{Qual}}^{ct}$

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

– Output LE: $\left(\textsc{Retrieve}, \mathsf{L}_{\mathsf{Requests}}^{\mathsf{Ready}}\right)$

Get Statistics:
– Input some Party P: (GetStatistics)
– Behavior P:
1. Initialize empty list $\mathsf{L}_{\mathsf{Stats}}$
2. Send (Read) to $\mathcal{F}_{\mathrm{BCRA}}$ and receive (Ordered)
3. Find $(pid_{\mathrm{J}}, t_2, (\mathrm{JudgeKey}, \mathsf{vk}_{\mathrm{J}}))$ in Ordered, otherwise abort
4. Find $(pid_{\mathrm{AU}}, t_3, (\mathrm{AuditorKey}, \mathsf{ek}_{\mathrm{AU}}))$ in Ordered, otherwise abort
5. For each entry $(pid, t, (id, W^{\mathrm{pub}}, W^{\mathrm{enc}}, \pi_{\widetilde{W}}))$:
(a) Assemble ZK-Statement: $stmt_{\widetilde{W}} := (\mathsf{ek}_{\mathrm{AU}}, \mathsf{vk}_{\mathrm{J}}, W^{\mathrm{enc}}, W^{\mathrm{pub}})$
(b) Verify Proof:
$b \leftarrow \mathsf{NIZK.Verify}(crs_{\mathrm{zk}}^{SS}, stmt_{\widetilde{W}}, \pi_{\widetilde{W}})$
(c) If $b = 1$ and $W^{\mathrm{pub}} \notin \mathsf{L}_{\mathsf{Stats}}$, store W^{pub} in $\mathsf{L}_{\mathsf{Stats}}$

– Output P: (GotStatistics, $\mathsf{L}_{\mathsf{Stats}}$)

Audit:
– Input AU: (AuditRequest)
– Behavior AU:
1. Initialize empty list $\mathsf{L}_{\mathsf{Warrants}}$
2. Retrieve stored $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}})$
3. Send (Read) to $\mathcal{F}_{\mathrm{BCRA}}$ and receive (Ordered)
4. Find $(pid_{\mathrm{J}}, t_2, (\mathrm{JudgeKey}, \mathsf{vk}_{\mathrm{J}}))$ in Ordered, otherwise abort
5. For each entry $(pid, t, (id, W^{\mathrm{pub}}, W^{\mathrm{enc}}, \pi_{\widetilde{W}}))$ in Ordered:
(a) Assemble ZK-Statement:
$stmt_{\widetilde{W}} := (\mathsf{ek}_{\mathrm{AU}}, \mathsf{vk}_{\mathrm{J}}, W^{\mathrm{enc}}, W^{\mathrm{pub}})$
(b) Verify Proof: If $\mathsf{NIZK.Verify}(crs_{\mathrm{zk}}^{SS}, stmt_{\widetilde{W}}, \pi_{\widetilde{W}}) = 0$, skip the following steps and proceed to next entry
(c) Decrypt W^{enc}: $\widetilde{W} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{\mathrm{AU}}, W^{\mathrm{enc}})$
(d) If $\widetilde{W} \notin \mathsf{L}_{\mathsf{Warrants}}$, store $\widetilde{W}$ in $\mathsf{L}_{\mathsf{Warrants}}$

– Output AU: (AuditAnswer, $\mathsf{L}_{\mathsf{Warrants}}$)

Behavior of Nodes:

– Upon being activated for the first time, send (Register) to $\mathcal{G}_{\text{CLOCK}}$.
– Upon receiving a message (Generate, $R, \mathsf{ek}_R, \mathsf{dk}_R, \mathsf{vk}_R, \mathsf{sk}_R$) from $\mathcal{F}_{\text{BCRA}}$, store $(R, \mathsf{ek}_R, \mathsf{dk}_R, \mathsf{vk}_R, \mathsf{sk}_R)$ in $\mathsf{L}_{\mathsf{Roles}}$.
– Upon receiving input (ExecuteRole):
  1. Send (Clock-Read) to $\mathcal{G}_{\text{CLOCK}}$ and receive (t_{Now}).
  2. Set $cnum := \left\lfloor \frac{t_{\text{Now}}}{t_{\mathsf{com}}} \right\rfloor$.
  3. Send (Read) to $\mathcal{F}_{\text{BCRA}}$ and receive (Ordered).
  4. Find $(pid_{\text{J}}, t, (\text{JudgeKey}, \mathsf{vk}_{\text{J}}))$ in Ordered, otherwise abort
  5. Find $(pid_{\text{AU}}, t, (\text{AuditorKey}, \mathsf{ek}_{\text{AU}}))$ in Ordered, otherwise abort
  6. For each role in $\mathsf{L}_{\mathsf{Roles}}$, 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 $\mathcal{G}_{\text{CLOCK}}$.
– Executing Role KeyGen1:
  1. Retrieve sk_R associated with this role R_i from $\mathsf{L}_{\mathsf{Roles}}$
  2. Initialize empty list $\mathsf{L}_{\mathsf{CommitteePK}}^{\text{next}}$
  3. For each role R_j in the next committee
     (a) Find $(\texttt{roleassign}, t, (\text{Generate}, R_j, \mathsf{ek}_{R_j}, \mathsf{vk}_{R_j}))$ in Ordered
     (b) Insert ek_{R_j} into $\mathsf{L}_{\mathsf{CommitteePK}}^{\text{next}}$
  4. Generate key share: $s_i \leftarrow \mathbb{Z}_p$
  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 $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, sh_j; r_j)$
  7. Assemble ZK-Witness: $wit_{KG} := (s_i, \{r_j\}_{j\in[n]}, F)$
  8. Assemble ZK-Statement: $stmt_{KG} := (\left\{ ct_j, \mathsf{ek}_{R_j} \right\}_{j\in[n]})$
  9. Compute Proof: $\pi_{KG} \leftarrow \mathsf{NIZK.Prove}(crs_{\mathsf{zk}}^{SS}, stmt_{KG}, wit_{KG}, \mathcal{R}_{KG1}^{AD})$
  10. Prepare message: $msg := (\text{KeyGen1}, \left(\{ct_j\}_{j\in[n]}, \pi_{KG}\right))$
  11. Sign message: $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_R, msg)$
  12. Remove the entry for this role from $\mathsf{L}_{\mathsf{Roles}}$
  13. Delete all state except msg, σ
  14. Send (Post, (R_i, msg, σ)) to $\mathcal{F}_{\text{BCRA}}$
– Executing Role KeyGen2:
  1. Retrieve sk_R associated with this role R_i from $\mathsf{L}_{\mathsf{Roles}}$
  2. Initialize empty lists $\mathsf{L}_{\mathsf{CommitteePK}}^{\text{next}}, \mathsf{L}_{\mathsf{CommitteePK}}^{\text{current}}, \mathsf{L}_{\mathsf{CommitteeVK}}$
  3. For each role R_j in the next committee
     (a) Find $(\texttt{roleassign}, t, (\text{Generate}, R_j, \mathsf{ek}_{R_j}, \mathsf{vk}_{R_j}))$ in Ordered

(b) Insert ek_{R_j} into $\mathsf{L}_{\mathsf{CommitteePK}}{}^{\text{next}}$
4. For each role R_j in the current committee
  (a) Find $(\texttt{roleassign}, t, (\text{Generate}, R_j, \mathsf{ek}_R, \mathsf{vk}_R))$ in Ordered
  (b) Insert ek_R into $\mathsf{L}_{\mathsf{CommitteePK}}{}^{\text{current}}$
5. For each role R_j in the previous committee:
  (a) Find $(\texttt{roleassign}, t, (\text{Generate}, R_j, \mathsf{ek}_{R_j}, \mathsf{vk}_{R_j}))$ in Ordered
  (b) Insert vk_{R_j} into $\mathsf{L}_{\mathsf{CommitteeVK}}$
6. For each entry $(pid, t, (R_j, (\text{KeyGen1}, \left(\{ct_j\}_{j\in[n]}, \pi_{KG}\right)), \sigma))$ in Ordered
  (a) Retrieve vk_{R_j} from $\mathsf{L}_{\mathsf{CommitteeVK}}$
  (b) Check if $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, (\text{KeyGen1}, \left(\{ct_j\}_{j\in[n]}, \pi_{KG}\right)), \sigma) = 1$, otherwise skip this entry.
  (c) Assemble ZK-Statement:
  $stmt_{KG} \coloneqq (\left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]})$
  (d) If $\mathsf{NIZK.Verify}(crs^{SS}_{\text{zk}}, stmt_{KG}, \pi_{KG}, \mathcal{R}^{AD}_{KG1}) \neq 1$ skip this entry
  (e) Add ct_i to $\mathsf{L}_{\mathsf{Qual}}$ (the one addressed to the currently executing role)
7. Sort $\mathsf{L}_{\mathsf{Qual}}$ lexicographically
8. Obtain the first $t+1$ entries $ct_i^1, \ldots, ct_i^{t+1}$ from $\mathsf{L}_{\mathsf{Qual}}$
9. Decrypt ct_i^k as $sh_k \leftarrow \mathsf{PKE.Dec}(\mathsf{dk}_R, ct_i^k)$
10. Set $\mathsf{sk}_i = \sum_{k=1}^{t+1} sh_k$ and $\mathsf{pk}_i = \mathsf{TPKE.Sk2Pk}(\mathsf{sk}_i)$
11. 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 $F(0) = \mathsf{sk}_i$
12. For each role R_j in the next committee:
  (a) Set $sh_j \coloneqq F(j)$
  (b) Generate ciphertext $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, sh_j; r_j)$
13. Assemble ZK-Witness: $wit_{KG} \coloneqq (\mathsf{sk}_i, \{r_j\}_{j\in[n]}, \mathsf{dk}_R, F)$
14. Assemble ZK-Statement:
  $stmt_{KG} \coloneqq (\mathsf{pk}_i, \left\{ct^k\right\}_{k\in[t+1]}, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]}, \mathsf{ek}_R)$
15. Compute Proof:
  $\pi_{KG} \leftarrow \mathsf{NIZK.Prove}(crs^{SS}_{\text{zk}}, stmt_{KG}, wit_{KG}, \mathcal{R}^{AD}_{KG2})$
16. Prepare message: $msg \coloneqq (\text{KeyGen2}, \left(\mathsf{pk}_i, \{ct_j\}_{j\in[n]}, \pi_{KG}\right))$
17. Sign message: $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_R, msg)$
18. Remove the entry for this role from $\mathsf{L}_{\mathsf{Roles}}$
19. Delete all state except msg, σ
20. Send $(\text{Post}, (R_i, msg, \sigma))$ to $\mathcal{F}_{\text{BCRA}}$

– Executing Role Handover:
1. Retrieve $\mathsf{dk}_R, \mathsf{sk}_R$ associated with this role R_i from $\mathsf{L}_{\mathsf{Roles}}$
2. Initialize empty lists $\mathsf{L}_{\mathsf{CommitteePK}}{}^{\text{next}}$, $\mathsf{L}_{\mathsf{CommitteePK}}{}^{\text{current}}$, $\mathsf{L}_{\mathsf{CommitteeVK}}$, $\mathsf{L}_{\mathsf{Shares}}$ and $\mathsf{L}_{\mathsf{Requests}}$
3. For each role R_j in the next committee
  (a) Find $(\texttt{roleassign}, t, (\text{Generate}, R_j, \mathsf{ek}_R, \mathsf{vk}_R))$ in Ordered
  (b) Insert ek_R into $\mathsf{L}_{\mathsf{CommitteePK}}{}^{\text{next}}$

4. For each role R_j in the current committee
   (a) Find $(\texttt{roleassign}, t, (\textsc{Generate}, R_j, \mathsf{ek}_R, \mathsf{vk}_R))$ in $\textsc{Ordered}$
   (b) Insert ek_R into $\mathsf{L}_{\mathsf{CommitteePK}}^{\text{current}}$
5. For each role R_j in the previous two committees:
   (a) Find $(\texttt{roleassign}, t, (\textsc{Generate}, R_j, \mathsf{ek}_{R_j}, \mathsf{vk}_{R_j}))$ in $\textsc{Ordered}$
   (b) Insert vk_{R_j} into $\mathsf{L}_{\mathsf{CommitteeVK}}$
6. For each entry $(pid, t, (R_j, msg, \sigma))$ with $msg := (\textsc{KeyShare}, (\mathsf{vk}_i, \{ct_j\}, \pi_{KG}))$ for the **previous** committee in $\textsc{Ordered}$
   (a) Retrieve vk_{R_j} from $\mathsf{L}_{\mathsf{CommitteeVK}}$
   (b) Check $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, msg, \sigma) = 1$, otherwise ignore this message
   (c) Assemble ZK-Statement:
   $stmt_{KS} := (\mathsf{vk}_i, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]})$
   (d) Check $\mathsf{NIZK.Verify}(crs_{\mathsf{zk}}^{SS}, stmt_{KS}, \pi_{KS}, \mathcal{R}_{KS}^{AD}) = 1$, otherwise ignore this message
   (e) Add $\{ct_j\}_{j\in[n]}$ to $\mathsf{L}_{\mathsf{Qual}}^{ct}$
7. Sort $\mathsf{L}_{\mathsf{Qual}}^{ct}$ lexicographically
8. For each entry $(pid, t, (R_j, msg, \sigma))$ with $msg := (\textsc{KeyShare}, (\mathsf{vk}_i, \{ct_k\}, \pi_{KS}))$ for the **current** committee in $\textsc{Ordered}$:
   (a) Retrieve vk_{R_j} from $\mathsf{L}_{\mathsf{CommitteeVK}}$
   (b) Check if $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{R_j}, (\textsc{KeyShare}, msg, \sigma) = 1$, otherwise skip this entry.
   (c) Retrieve from $\mathsf{L}_{\mathsf{Qual}}^{ct}$ from the first $t+1$ entries the ciphertext ct_j for R_j each, as $\mathsf{L}_{\mathsf{KCom}} := \{ct_j^1, \ldots, ct_j^n\}$
   (d) Assemble ZK-Statement:
   $stmt_{KS} := (\mathsf{pk}_j, \{ct_k\}_{k\in[n]}, \mathsf{L}_{\mathsf{KCom}})$
   (e) Check if $\mathsf{NIZK.Verify}(crs_{\mathsf{zk}}^{SS}, stmt_{KS}, \pi_{KS}, \mathcal{R}_{KS}^{AD}) = 1$, otherwise skip this entry.
   (f) Add ct_i (the one addressed to the currently executing role) to $\mathsf{L}_{\mathsf{Qual}}$
9. Sort $\mathsf{L}_{\mathsf{Qual}}$ lexicographically
10. Reconstruct and reshare secret key share:
   (a) Obtain the first $t+1$ entries $ct_i^1, \ldots, ct_i^{t+1}$ from $\mathsf{L}_{\mathsf{Qual}}$
   (b) Decrypt ct_i^k as $sh_k \leftarrow \mathsf{PKE.Dec}(\mathsf{dk}_R, ct_i^k)$
   (c) Lagrange-interpolate $G(x)$ from $sh_1, \ldots, sh_{t+1}$ to obtain secret key share $\mathsf{sk}_i := G(i)$ and set $\mathsf{vk}_i = \mathsf{TPKE.Sk2Pk}(\mathsf{sk}_i)$
   (d) Choose a random degree t polynomial $F(x) = a_0 + a_1 * x + a_2 * x^2 + \ldots + a_t * x^t$ with $F(0) = \mathsf{sk}_i$
   (e) For each role R_j in the next committee:
      i. Generate ciphertext $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, F(j); r_j)$
   (f) Assemble ZK-Witness:
   $wit_{KS} := (\mathsf{dk}_R, \mathsf{sk}_i, \{r_j\}_{j\in[n]}, F, G)$
   (g) Assemble ZK-Statement:
   $stmt_{KS} := (\mathsf{ek}_R, \{ct_i^k\}_{k\in[t+1]}, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]}, \mathsf{vk}_i)$

(h) Compute Proof:
$\pi_{KS} \leftarrow \mathsf{NIZK.Prove}(crs_{\text{zk}}^{SS}, stmt_{KS}, wit_{KS}, \mathcal{R}_{KS}^{AD})$

(i) Add $msg \coloneqq (\text{KeyShare}, \Big(\mathsf{vk}_i, \{ct_j\}_{j\in[n]}, \pi_{KS}\Big))$ to $\mathsf{L_M}$

11. For each entry $(pid, t, (\text{Request}, W^{\text{pub}}, W^{\text{enc}}, \{\widehat{ct}_i\}_{i\in[v]}, \pi_{\widetilde{W}}, \mathsf{ek}_{\text{LE}}))$ in Ordered
    (a) Assemble ZK-Statement:
    $stmt_{\widetilde{W}} \coloneqq (\mathsf{ek}_{\text{AU}}, \mathsf{vk}_{\text{J}}, W^{\text{enc}}, W^{\text{pub}}, \{\widehat{ct}_i\}_{i\in[v]})$
    (b) If $\mathsf{NIZK.Verify}(crs_{\text{zk}}^{SS}, stmt_{\widetilde{W}}, \pi_{\widetilde{W}}, \mathcal{R}_{W}^{AD}) \neq 1$, abort
    (c) For each $i \in [v]$:
        i. Store $(\mathsf{ek}_{\text{LE}}, \widehat{ct}_i)$ in $\mathsf{L_{Requests}}$
12. For each entry $(\mathsf{ek}_{\text{LE}}, ct)$ in $\mathsf{L_{Requests}}$,
    (a) Partially decrypt ct to $ct^* \leftarrow \mathsf{TPKE.TDec}(\mathsf{sk}_i, ct)$
    (b) Assemble ZK-Witness:
    $wit_{Dec} \coloneqq (\mathsf{sk}_i)$
    (c) Assemble ZK-Statement:
    $stmt_{Dec} \coloneqq (\mathsf{vk}_i, ct, ct^*)$
    (d) Compute Proof:
    $\pi_{Dec} \leftarrow \mathsf{NIZK.Prove}(crs_{\text{zk}}^{SS}, stmt_{Dec}, wit_{Dec}, \mathcal{R}_{Dec}^{AD})$
    (e) Encrypt answer: $msg \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{\text{LE}}, (ct, ct^*, \pi_{Dec}))$
    (f) Add $(\text{Request}, msg)$ to $\mathsf{L_M}$
13. For each entry (msg) in $\mathsf{L_M}$
    (a) Generate signature $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_R, msg)$
    (b) Update $msg \coloneqq (R_i, msg, \sigma)$
14. Remove the entry for this role from $\mathsf{L_{Roles}}$
15. Delete all state except for $\mathsf{L_M}$
16. For each entry (msg) in $\mathsf{L_M}$
    (a) Send (Post, msg) to $\mathcal{F}_{\text{BCRA}}$

C Auditably Sender-Traceable Encryption (ASTE)

In this section, we give the full description of the Auditably Sender-Traceable Encryption Transfer ideal functionality $\mathcal{F}_{\text{ASTE}}$ and the protocol Π_{ASTE}.

C.1 Overview

The Auditably Sender-Traceable Message Transfer (ASTE) functionality $\mathcal{F}_{\text{ASTE}}$ is a toy example with a two-fold purpose. Firstly, it demonstrates the applicability of $\mathcal{F}_{\text{AS}}$. 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 $\mathcal{F}_{\text{AS}}$.

The $\mathcal{F}_{\text{ASTE}}$ 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 $\mathcal{F}_{\text{ASTE}}$ 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, $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathcal{F}_{\mathrm{AS}}$, statistics about the surveillance are published and an audit trail is accessible for the auditor.

On a more technical level, $\mathcal{F}_{\mathrm{ASTE}}$ is very similar to $\mathcal{F}_{\mathrm{AS}}$. 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 $(\mathit{uid}, \mathit{vper})$, thus deanonymizing the subject of surveillance U during period vper. Finally, users can use the encryption scheme via **encrypt message**s and **decrypt ciphertext**.

The protocol Π_{ASTE} reflects the close relation of $\mathcal{F}_{\mathrm{ASTE}}$ to $\mathcal{F}_{\mathrm{AS}}$: Indeed, except for the tasks **encrypt message**, **decrypt message** and **execute data**, Π_{ASTE} can delegate most of the work to $\mathcal{F}_{\mathrm{AS}}$, 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, Π_{ASTE} relies on a special "encryption scheme" $\mathsf{PKE}_{\mathsf{AS}}$ where:

- $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Gen}$ generates a long-term public key pk which is also used as the user identity uid.
- $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Enc}$ 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 $\mathcal{F}_{\mathrm{ASTE}}$, 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 $\mathcal{F}_{\mathrm{AS}}$. All ciphertexts carry a consistency proof which is checked by an honest recipient. In particular, if $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}(crs, \mathsf{sk}, c, vper) \neq \bot$, then the consistency proof holds. Moreover, if the consistency proof holds, then $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(crs, usk, c, vper)$ returns uid which correctly deanonymizes the user (with secret key sk). Importantly, the consistency proof also demonstrates possession of the (long-term) secret key sk to pk, thus it is impossible to send messages in the name of another (honest) user.

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

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

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.

$\mathcal{F}_{\mathrm{ASTE}}$

System Parameters:

– f_t, f_p: Transparency and policy function as in $\mathcal{F}_{\mathrm{AS}}$.
– $\mathsf{poly}(\lambda)_{uid}$: length of uid.
– $\mathsf{leak}(m, uid, vper)$: Leakage to adversary when a message is sent, usually $\mathsf{leak}(m, uid, vper) = (|m|, vper)$.
– System pids: pid_{SO}, pid_{J}, pid_{AU}, pid_{LE}

Functionality State:

– L_{I}: Identical to $\mathcal{F}_{\mathrm{AS}}$. (List of initialized parties (initially empty). Contains (pid) entries.)
– L_{U}: Identical to $\mathcal{F}_{\mathrm{AS}}$. (List of registered users (initially empty). Contains (pid, uid) pairs.)
– $\mathsf{L}_{\mathsf{uvper}}$: List of tuples $(uid, vper)$ to keep track of which users have updated in which periods. (Initially empty.)
– $\mathsf{L}_{\mathsf{msgs}}$: List of all messages sent by honest users of $\mathcal{F}_{\mathrm{ASTE}}$. Contains tuples of the form $(c, m, uid, uvper)$.
– L_{W}: Identical to $\mathcal{F}_{\mathrm{AS}}$.
– $\mathsf{L}_{\mathsf{W\text{-}cached}}$: List of $(uid_i, vper_i)$ tuples tracking access rights of LE. (Initially empty.)
– $ureg_i \in \{0, 1\}$ is 1 if user U_i received output of registration. Else 0.
– $uvper_i$: (Most recent) validity period for which user U_i has registered (all initialized to $\bot$).

Init:

- This task is identical to **Init** in $\mathcal{F}_{\text{AS}}$, except that $(\textsc{Init}, P)$ is leaked to the adversary when party P inputs $(\textsc{Init}, P)$:
- $\mathcal{F}_{\text{ASTE}}$ sends $(\textsc{Register}, sid)$ to $\mathcal{G}_{\text{CLOCK}}$.

Party Init:

- This task is identical to **Party Init** in $\mathcal{F}_{\text{AS}}$.

User Registration:

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

Next Period:

- Input SO: $(\textsc{NextPeriod})$
- Behavior: $\mathcal{F}_{\text{ASTE}}$ sends $(\textsc{Clock-Update}, sid_C)$ to $\mathcal{G}_{\text{CLOCK}}$.

Update:

- Input U_i: $(\textsc{Update})$
- Input SO: $\emptyset$
- Behavior:
  - $\mathcal{F}_{\text{ASTE}}$ queries $(\textsc{Clock-Read}, sid_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and receives response $(\textsc{Clock-Read}, sid_C, vper)$.
  - If $uvper_i = vper$, immediately output $(\textsc{UpdateDone}, vper)$ to U_i. Else continue.
  - Send $(\textsc{Update}, \text{U}_i, vper)$ to the adversary and wait for $(\textsc{Ok})$ from the adversary.
  - $\mathcal{F}_{\text{ASTE}}$ queries $(\textsc{Clock-Read}, sid_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and receives response $(\textsc{Clock-Read}, sid_C, vper')$.
  - If $vper \neq vper'$ abort this task.
  - Generate delayed output $(\textsc{UpdateDone}, vper)$ to SO.
  - Generate delayed output $(\textsc{UpdateDone}, vper)$ to U_i, and, when delivered, set $uvper_i := vper$ and add $(\text{U}_i, uvper_i)$ to $\mathsf{L}_{\mathsf{uvper}}$.

Encrypt message:

- Input U_i: $(\textsc{Encrypt}, \text{U}_j, m)$
- Behavior:
  1. Copy $uvper_i$ and use this copy in the rest of this task. (Protect against concurrent calls to **update**.)
  2. If $uvper_i = \perp$, abort.

3. If U_j is corrupted, let $m' = m$, else let $m' = \bot$.
4. If $(uid_i, uvper_i) \in \mathsf{L}_{\text{W-cached}}$, let $uid' = uid$, else let $uid' = \bot$.
5. Send $(\textsc{Encrypt}, \mathrm{U}_j, \mathsf{leak}(m, uid_i, uvper_i), m', uid')$ to the adversary.
6. Once $(\textsc{Ciphertext}, \mathrm{U}_j, c)$ is received from the adversary, add $(c, m, uid_i, uvper_i)$ to L_{msgs}.
7. Generate immediate output $(\textsc{Ciphertext}, c)$ to U_i.

Decrypt Ciphertext:

– Input U_j: $(\textsc{Decrypt}, c)$
– Behavior:
  1. If $(c, \cdot, \cdot, \cdot) \notin \mathsf{L}_{\text{msgs}}$ then
     - Send urgent request $(\textsc{SimInjectMsg}, c)$ to the adversary and wait for immediate response $(\textsc{InjectMsg}, c, m, vper)$.
     - Run **inject message** for input $(\textsc{InjectMsg}, c, m, vper)$.
  2. Find $(c, m, uid, uvper)$ in L_{msgs} and let $m = \bot$ if no entry was found.
  3. $\mathcal{F}_{\text{ASTE}}$ sends $(\textsc{Clock-Read}, sid_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and receives response $(\textsc{Clock-Read}, sid_C, vper)$.
  4. If $uvper \neq vper$ let $m = \bot$.
  5. Generate immediate output $(\textsc{Plaintext}, m)$ to U_j.

Inject Message:

– Note: This is an auxiliary "task". It is only available as a subroutine within other tasks.
– Input: $(\textsc{InjectMsg}, c, m, uid, vper)$
  1. If $(m, uid, vper) = (\bot, \bot, \bot)$, skip the rest.
  2. If $\exists \mathrm{U}$: $(\mathrm{U}, uid) \in \mathsf{L}_{\text{U}}$ and U is not corrupted, abort.
  3. If SO is honest and $(uid, vper) \notin \mathsf{L}_{\text{uvper}}$, abort.
  4. Store $(c, m, uid, vper)$ in L_{msgs}.

Request Warrant:

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

Prepare Access:

– Input LE: $(\textsc{AccessPrep},\ W)$
– Behavior:
  1. Send $(\textsc{AccessLeak})$ 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 $(uid', vper') \in \mathsf{L}_{\text{uvper}}$ store $(uid', vper') \in \mathsf{L}_{\text{W-cached}}$.
  4. Generate delayed output $(\textsc{AccessPrepDone})$ for LE.

Execute Access:

– Input LE: $(\textsc{AccessExec}, c, (uid, vper))$
– Behavior:
  1. If $(uid, vper) \notin \mathsf{L}_{\text{W-cached}}$, return $\bot$.

2. If $(c, \cdot, \cdot, \cdot) \notin \mathsf{L}_{\mathsf{msgs}}$ then:
   (a) Send urgent request $(\textsc{SimAccessInject}, c)$ to the adversary and wait for immediate response $(\textsc{InjectMsg}, c, m, \mathit{uid}, \mathit{vper})$.
   (b) Run **inject message** for input $(\textsc{InjectMsg}, c, m, \mathit{uid}, \mathit{vper})$.
3. If there exist $m', \mathit{uid}', \mathit{vper}'$ such that $(c, m', \mathit{uid}', \mathit{vper}') \in \mathsf{L}_{\mathsf{msgs}}$ and $\mathit{uid} = \mathit{uid}'$, then set $\mathit{uid}' = \mathit{uid}$, else $\mathit{uid}' = \perp$.
4. Output $(\textsc{AccessExecDone}, \mathit{uid}')$ immediately.

Get Statistics:
– This task is identical to **Get Statistics** in $\mathcal{F}_{\mathrm{AS}}$.

Audit:
– This task is identical to **Audit** in $\mathcal{F}_{\mathrm{AS}}$.

Remark 4 (Handling of leakage). We handle leakages in **encrypt message** and **prepare access** in two different ways: In **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 **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 $(\mathit{uid}_k, \mathit{uvper}_k) \in \mathsf{L}_{\mathsf{W\text{-}cached}}$ for honest users U_k.

Indeed, in **prepare access**, we do not add an explicit leak of all affected ciphertexts, and let simulator use the trial-decryption strategy.

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

1. The access task is split into **prepare access** and **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, **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 $\mathit{\Pi}_{\mathrm{ASTE}}$, honestly generated ciphertexts of corrupted parties which are never received by honest parties can still be decrypted be LE. To model this, $\mathcal{F}_{\mathrm{ASTE}}$ leaks a non-honest ciphertext (in $(\textsc{SimAccessInject}, c')$) in **execute access**, and then runs **inject message** to inject a suitable tuple into $\mathsf{L}_{\mathsf{msgs}}$, 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, **inject message** does not require a recipient.[25] This modelling artefact leaks some information of the supposedly offline decryption queries to the ideal adversary, but any $c \in \mathsf{L}_{\mathsf{msgs}}$ (in particular, ciphertexts en- or decrypted by honest users) is accessed "silently".

3. Intuitively, a warrant $W = \{(uid_i, vper_i, meta_i)\}_i$ empowers law enforcement to extract information from *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 **decrypt ciphertext** or **inject message** "accepts", i.e. which ends up in $\mathsf{L}_{\mathsf{msgs}}$ 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 restriction w.r.t. registration, and consequently thwart deanonymization. This is inherited from $\mathcal{F}_{\text{AS}}$, because the NIZK provided by $\mathcal{F}_{\text{AS}}$ 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 contractual obligation by such actions, and is therefore strongly incentivized to not misbehave.

C.3 The Protocol Π_{ASTE}

In this section, we define the protocol for realizing $\mathcal{F}_{\text{ASTE}}$. The high-level idea is the following:

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

Passthrough tasks Except perhaps for some smaller adaptation, **Init**, **Party Init**, **RequestWarrant**, **Get Statistics** and **Audit** basically pass their inputs through to the $\mathcal{F}_{\text{AS}}$ hybrid functionality.

Next Period SO simply advances the clock. (SO registered with $\mathcal{G}_{\text{CLOCK}}$ during **Init**.)

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

[25] 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 $\mathcal{F}_{\text{ASTE}}$ (and Π_{ASTE}) does not enforce it.

to check whether the registration succeeded.[26] ($\mathcal{F}_{\text{BB}}$ 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 $vper$, 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 **store secret** from $\mathcal{F}_{\text{AS}}$. The resulting secret is the user's escrow secret usk_i and remembers the period in $uvper_i := vper$.

Encrypt Message This is merely an application of $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Enc}$.

Decrypt Ciphertext The receiver first retrieves the current time $vper$, and then decrypts the ciphertext via $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(crs, \mathsf{sk}_j, c, vper)$. 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 **get secrets** in $\mathcal{F}_{\text{AS}}$. After receiving (GotSecrets, ...), LE stores these escrow secrets (in $\mathsf{L}_{\mathsf{W\text{-}cached}}$) for later use.

Execute Access LE uses a stored escrow secret $(uid, vper, usk)$ from $\mathsf{L}_{\mathsf{W\text{-}cached}}$ to check via $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(usk, c, vper)$ whether a ciphertext c decrypts to uid when trying $(usk, vper)$. If so, the sender uid was deanonymized, otherwise it returns $\perp$.

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

Π_{ASTE}
The protocol Π_{ASTE} uses hybrid functionalities $\mathcal{F}_{\text{CRS}}$, $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{AS}}$, $\mathcal{F}_{\text{RO}}$, and global functionality $\mathcal{G}_{\text{CLOCK}}$. **System Parameters** – f_t, f_p: Transparency and policy function as in $\mathcal{F}_{\text{AS}}$. – System pids: pid_{SO}, pid_{J}, pid_{AU}, pid_{LE} **Hybrid functionalities** – $\mathcal{F}_{\text{CRS}}$: The distribution of $\mathcal{F}_{\text{CRS}}$ is $crs_{\text{ASTE}} = crs_{\text{com}}$, where $crs_{\text{com}} \leftarrow \mathsf{COM.Setup}(1^\lambda)$. – $\mathcal{F}_{\text{AS}}$: The secret key space $\mathcal{S}$ of $\mathcal{F}_{\text{AS}}$ is defined as the user secret key space $\mathsf{SKE}_{\mathsf{NCE}}.\mathcal{K}$, where $\mathsf{SKE}_{\mathsf{NCE}}$ is the scheme used in $\mathsf{PKE}_{\mathsf{AS}}$. The relation $\mathcal{R}$ is set to $\mathcal{R}_{\mathsf{PKEAS}}$ or some NP-complete relation. Other system parameters of $\mathcal{F}_{\text{AS}}$ are set as in Π_{AS} above. – $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{RO}}$, $\mathcal{G}_{\text{CLOCK}}$ have no parameterization. **States of the Parties:** – User, SO and LE: Cache crs from $\mathcal{F}_{\text{CRS}}$.

[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 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: $(\mathsf{pk}_i, \mathsf{sk}_i)$, where $uid_i = \mathsf{pk}_i$ is the user identify.
  - User secret (with period): $(usk_i, uvper_i)$, 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: $uid_j = \mathsf{pk}_j$ (once retrieved from $\mathcal{F}_{\mathrm{BB}}$).
- LE: Cache $\mathsf{L}_{\mathsf{W\text{-}cached}}$ with entries $(uid_i, uvper_i, usk_i)$.
- SO, J and AU only need to interact with $\mathcal{F}_{\mathrm{AS}}$.

Init:
- Input SO: $(\textsc{Init}, \mathrm{SO})$
- Input J: $(\textsc{Init}, \mathrm{J})$
- Input AU: $(\textsc{Init}, \mathrm{AU})$
- Note: For all parties:
  - This task can only be invoked once.
  - Ignore all other messages until this task has finished.
- Behavior
  1. SO: Send $(\textsc{Register})$ to $\mathcal{G}_{\textsc{Clock}}$.
  2. SO: Send $(\textsc{value})$ to $\mathcal{F}_{\mathrm{CRS}}$ and cache the result as crs.
  3. SO: Send $(\textsc{Init}, \mathrm{SO})$ to $\mathcal{F}_{\mathrm{AS}}$.
  4. $P \in \{\mathrm{J}, \mathrm{AU}\}$: Send $(\textsc{Init}, P)$ to $\mathcal{F}_{\mathrm{AS}}$.
  5. SO, J, AU: Upon receiving $(\textsc{InitFinished})$, output $(\textsc{InitFinished})$.

Party Init:
- This is merely a proxy to **Party Init** from $\mathcal{F}_{\mathrm{AS}}$.

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

11. SO: Receive and propagate output (REGISTERED, *uid*) (or propagate an abort) from $\mathcal{F}_{\text{AS}}$.
12. U_i: Receive output (REGISTERED) (or propagate an abort) from $\mathcal{F}_{\text{AS}}$.
13. U_i: Set $ureg_i := 1$ 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 $uvper = vper$, U_i output (UPDATEDONE, *vper*). Else continue.
  3. U_i: Send (UPDATE, uid_i, *vper*) to SO.
  4. U_i: Send (STORESECRET, uid_i, *vper*) to $\mathcal{F}_{\text{AS}}$.
  5. SO: Upon receiving (UPDATE, *uid*, *vper*) from U_i:
  6. SO: Query (CLOCK-READ) to $\mathcal{G}_{\text{CLOCK}}$ and receive (CLOCK-READ, $vper'$). If $vper' \neq vper$, abort.
  7. SO: Send (STORESECRET, *vper*) to $\mathcal{F}_{\text{AS}}$.
  8. U_i: Upon receiving (SECRETSTORED, usk_i'), update $(usk_i, uvper_i) := (usk_i', vper)$.
  9. U_i: Generate output (UPDATEDONE, *vper*).
  10. 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, $\text{U}_j, uid_j, uid_j'$). If $uid_j \neq uid_j'$ or $uid = \perp$, abort. Else cache $\mathsf{pk}_j = uid_j$.
  4. $c \leftarrow \mathsf{PKE_{AS}.Enc}(crs, \mathsf{pk}_j, m, (\mathsf{sk}_i, usk_i, uid_i, uvper_i))$
  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 $m = \mathsf{PKE_{AS}.Dec_{RE}}(crs, \mathsf{sk}_j, c, vper)$ and output (PLAINTEXT, m).

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

Prepare Access:
- Input LE: (ACCESSPREP, W)

– Behavior:
  1. Parse $W = (W_1, \dots, W_v)$ and $W_i = (uid_i, vper_i, meta_i)$.
  2. Send (GetSecrets, W) to $\mathcal{F}_{\text{AS}}$ and receive response (GotSecrets, $(usk_1, \dots, usk_v)$) (or an abort happens). For $i = 1, \dots, v$, if $usk_i \neq \bot$ add $(uid_i, vper_i, usk_i)$ to $\mathsf{L}_{\text{W-cached}}$.
  3. Output (AccessPrepDone)

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

Get Statistics:
– This is merely a proxy to **Get Statistics** from $\mathcal{F}_{\text{AS}}$.

Audit:
– This is merely a proxy to **Audit** from $\mathcal{F}_{\text{AS}}$.

In Appendix G we show the following theorem.

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

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

In this section, we define the subroutines which implement the encryption scheme used in Π_{ASTE}, called $\mathsf{PKE}_{\mathsf{AS}}$. To simplify presentation, we use a notation where $\mathsf{PKE}_{\mathsf{AS}}$ 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 Π_{ASTE}. Namely, as Π_{ASTE} is in the $\{\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{AS}}, \mathcal{G}_{\text{CLOCK}}\}$-hybrid model, $\mathcal{F}_{\text{RO}}$ will be used to implement the random oracle(s), and $\mathcal{F}_{\text{AS}}$ provides the "anonymously identity-bound" NIZK prove and verify interface. For simplicity, we simply write $\mathsf{RO}(x)$ for a random oracle query (instead of "send (Hash, x) to $\mathcal{F}_{\text{RO}}$, receive (HashConfirm, m, x)"), and likewise, we write $\mathsf{NIZK}_{\mathsf{AS}}$ for the "anonymously identity-bound" NIZK, i.e. run $\pi \leftarrow \mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Prove}((usk, uid, uvper), stmt, wit)$ for a prove request, and $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Verify}(vper, stmt, \pi)$ for a verification request. See Remark 7 for the full specification of the shorthand notation.

Remark 7 (Shorthand notation in $\mathsf{PKE}_{\mathsf{AS}}$*).* We write $\mathsf{RO}(m)$ for the protocol:

1. Send $(\textsc{Hash}, m)$ to $\mathcal{F}_{\mathrm{RO}}$.
2. Immediately receive $(\textsc{HashConfirm}, m, h)$ and output h.

For $\mathsf{NIZK}_{\mathsf{AS}}$, statement resp. witness have the form $stmt_R = (vper, stmt)$ resp. $wit_R = (uid, usk, wit)$. We add an explicit witness relation $\mathcal{R}$ 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 $\mathcal{R}$ (if it does not coincide with the relation used in $\mathcal{F}_{\mathrm{AS}}$). We write $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Prove}((usk, uid, vper), stmt_R, wit, \mathcal{R})$ for the protocol:

1. Let $stmt_R = (vper, stmt)$ and $wit_R = (uid, usk, wit)$. (If necessary, apply (NP-)reduction.)
2. Send $(\textsc{Prove}, stmt_R, wit_R)$ to $\mathcal{F}_{\mathrm{AS}}$.
3. Immediately receive $(\textsc{Proof}, stmt_R, wit_R, \pi)$ and output π.

We write $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Verify}(stmt_R, \pi, \mathcal{R})$ for the protocol:

1. Send $(\textsc{Verify}, stmt_R)$ to $\mathcal{F}_{\mathrm{AS}}$. (If necessary, apply (NP-)reduction.)
2. Immediately receive $(\textsc{Verification}, stmt_R, \pi, b)$ and output b.

We also write $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Sim}(stmt_R, \mathcal{R})$ for the proof simulation subroutine in the $\mathcal{F}_{\mathrm{AS}}$-hybrid model. This means:

1. The simulator sends $(\textsc{Prove}, stmt_R)$ to the (dummy) adversary (i.e. the environment).
2. The simulator immediately receives $(\textsc{Proof}, \pi)$ and the subroutine outputs π.

Note that the (dummy) adversary (i.e. the environment) must immediately answer the proof-string request $(\textsc{Prove}, stmt_R)$, this request contains no information on wit_R, and the immediate response $(\textsc{Proof}, \pi)$ must be an accepting proof. Thus, exactly corresponds to simulation of proofs. Moreover, from the adversary's perspective, the ideal NIZK in the (hybrid) $\mathcal{F}_{\mathrm{AS}}$-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 $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Ext}(stmt_R, \pi)$ for the witness to $(stmt_R, \pi)$ (assuming $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Verify}(stmt_R, \pi) = 1$).

Construction 1. The construction of $\mathsf{PKE}_{\mathsf{AS}}$ is given as pseudocode for the algorithms $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Gen}$, $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Enc}$, $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}$, $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}$, $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Sim}$, $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExplIn}_{\mathsf{RE}}$, $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExplIn}_{\mathsf{LE}}$. We let $\ell(\lambda) = 2\lambda$. The relation $\mathcal{R}_{\mathsf{PKEAS}}$ for the NIZK proofs is stated once in $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Enc}$. The message space is $\{0,1\}^{\mathsf{poly}_{\mathsf{msg}}(\lambda)}$.[27] The space $\mathsf{SKE}_{\mathsf{NCE}}.\mathcal{K}$ is $\{0,1\}^{\mathsf{poly}(\lambda)}$ for some poly. The space $\mathcal{UID}$ is $\{0,1\}^{\mathsf{poly}(\lambda)}$ for some poly; in our case, $\mathcal{UID}$ is the same as public keys to $\mathsf{PKE}_{\mathsf{NCE}}$ (i.e. public keys to

[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.

$\mathsf{PKE_{AS}}$), which is w.l.o.g. $\{0,1\}^{\mathsf{poly}(\lambda)}$ 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 $\mathsf{PKE_{AS}}$ 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 $\mathsf{PKE_{AS}}$ for clarity. The ciphertext size can, presumably, be reduced by a (small) constant factor without compromising security. Also for simplicity, $\mathsf{PKE_{AS}}$ includes the recipients public key in a ciphertext, as this allows to define and use $\mathsf{PKE_{AS}.Vfy}$ independent of the receiver's key.

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

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

$\mathsf{PKE_{AS}.Gen}(1^\lambda)$

1 : // For simplicity, sk equals the random coins.

2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE_{NCE}.Gen}(1^\lambda; \mathsf{sk})$

$\mathsf{PKE_{AS}.Enc}(ck, \mathsf{pk}_R, m, (\mathsf{sk}_S, usk, uid, vper))$

1 : // Prepare cut-and-choose encryption for LE
2 : **for** $i = 1, \ldots, \ell(\lambda)$
3 : $\quad m_{i,0} + m_{i,1} = m$ // Additive secret shares
4 : $\quad usk_{i,0} + usk_{i,1} = usk$
5 : $\quad uid_{i,0} + uid_{i,1} = uid$
6 : $\quad$ **for** $b \in \{0, 1\}$
7 : $\quad\quad ct^{RE}_{i,b} = \mathsf{PKE_{NCE}.Enc}(\mathsf{pk}_R, m_{i,b}; r^{RE}_{i,b})$
8 : $\quad\quad ct^{LE}_{i,b} = \mathsf{SKE_{NCE}.Enc}(usk_{i,b}, uid_{i,b}; r^{LE}_{i,b})$
9 : $\quad\quad w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, r^{RE}_{i,b}, r^{LE}_{i,b})$
10 : $\quad\quad com_{i,b} = \mathsf{COM.Com}(ck, w_{i,b}; r^{com}_{i,b})$
11 : $\quad\quad d_{i,b} = (w_{i,b}, r^{com}_{i,b})$
12 : // Consistency proof for commitments and ctxts.
13 : $stmt = (vper, \mathsf{pk}_R, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b})$
14 : $wit = (usk, uid, (d_{i,b})_{i,b})$
15 : $\pi_{\mathsf{con}} = \mathsf{NIZK_{AS}.Prove}((usk, uid, vper),$
16 : $\quad stmt, wit, \mathcal{R}_{\mathsf{PKEAS}})$ for relation
17 : $\quad \mathcal{R}_{\mathsf{PKEAS}} = \{(stmt, wit) \mid$
18 : $\quad (uid, \mathsf{sk}_S) = \mathsf{PKE_{NCE}.Gen}(1^\lambda; \mathsf{sk}_S)$
19 : $\quad com_{i,b} = \mathsf{COM.Com}(ck, (m_{i,b}, usk_{i,b}, uid_{i,b},$
20 : $\quad\quad r^{RE}_{i,b}, r^{LE}_{i,b}); r^{com}_{i,b})$ for all i, b
21 : $\quad \forall i \colon m = m_{i,0} + m_{i,1}$
22 : $\quad \forall i \colon usk = usk_{i,0} + usk_{i,1}$
23 : $\quad \forall i \colon uid = uid_{i,0} + uid_{i,1}$
24 : $\quad \}$
25 : // Cut-and-choose: Query challenge γ
26 : $\gamma = \mathsf{RO}(stmt, \pi_{\mathsf{con}}) \in \{0, 1\}^\ell$
27 : $\pi_{\mathsf{cut}} = (d_{i,\gamma_i})_i$
28 : $\pi = (\pi_{\mathsf{cut}}, \pi_{\mathsf{con}})$
29 : **return** $(\mathsf{pk}_R, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b}, \pi)$.

$\mathsf{PKE_{AS}.Vfy}(ck, c, vper)$

1 : **parse** $c = (\mathsf{pk}, ((com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b}),$
2 : $\quad ((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$
3 : $stmt = (vper, \mathsf{pk}_R, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b})$
4 : $\gamma = \mathsf{RO}(stmt, \pi_{\mathsf{con}}) \in \{0, 1\}^\ell$
5 : **if** $\mathsf{NIZK_{AS}.Verify}(stmt, \pi_{\mathsf{con}}, \mathcal{R}_{\mathsf{PKEAS}}) = 0$ **then**
6 : $\quad$ **return** 0
7 : // Check cut-and-choose proof
8 : **for** $i = 1, \ldots, \ell(\lambda)$
9 : $\quad b = \gamma_i$
10 : $\quad$ **parse** $d_{i,b} = (w_{i,b}, r^{com}_{i,b})$
11 : $\quad$ **parse** $w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, r^{RE}_{i,b}, r^{LE}_{i,b})$
12 : $\quad$ **check** $com_{i,b} == \mathsf{COM.Com}(ck, w_{i,b}; r^{com}_{i,b})$
13 : $\quad$ **check** $ct^{RE}_{i,b} == \mathsf{PKE_{NCE}.Enc}(\mathsf{pk}, m_{i,b}; r^{RE}_{i,b})$
14 : $\quad$ **check** $ct^{LE}_{i,b} == \mathsf{SKE_{NCE}.Enc}(usk_{i,b}, uid_{i,b}; r^{LE}_{i,b})$
15 : $\quad$ **return** 0 **if** any **check** fails
16 : **return** 1

$\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}(ck, \mathsf{sk}, c, vper)$

1 : **if** $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Vfy}(ck, c, vper) = 0$ **then return** $\perp$
2 : **parse** $c = (\mathsf{pk}, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b}, ((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$
3 : **if** pk does not belong to sk **then return** $\perp$
4 : $stmt = (vper, \mathsf{pk}_R, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b})$
5 : // Reconstruct m from secret-shares
6 : **for** $i = 1, \ldots, \ell(\lambda)$
7 : **for** $b \in \{0,1\}$
8 : $m_{i,b} = \mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Dec}(\mathsf{sk}, ct_{i,b}^{RE})$
9 : $m_i = m_{i,0} + m_{i,1}$ // $m_i \coloneqq \perp$ if one share is $\perp$
10 : **if** m is absolute majority of m_i **then**
11 : **return** m
12 : **else return** $\perp$

$\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(ck, usk, c, vper)$

1 : **if** $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Vfy}(ck, c, vper) = 0$ **then return** $\perp$
2 : **parse** $c = (\mathsf{pk}, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b}, ((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$
3 : $stmt = (vper, \mathsf{pk}_R, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b})$
4 : $\gamma = \mathsf{RO}(stmt, \pi_{\mathsf{con}}) \in \{0,1\}^{\ell}$
5 : // $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Vfy}(ck, c, vper) \neq \perp$ ensures $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Verify}(stmt, \pi_{\mathsf{con}}, \mathcal{R}_{\mathsf{PKEAS}}) = 1$.
6 : // Extract possible uid from cut-and-choose proof
7 : **for** $i = 1, \ldots, \ell(\lambda)$
8 : $b = \gamma_i$
9 : **parse** $d_{i,b} = (w_{i,b}, r_{i,b}^{com})$
10 : **parse** $w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, r_{i,b}^{RE}, r_{i,b}^{LE})$
11 : $usk_{i,1-b} = usk_{i,b} + usk$
12 : $uid_{i,1-b} = \mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Dec}(usk_{i,1-b}, ct_{i,1-b}^{LE})$
13 : $uid_i = uid_{i,0} + uid_{i,1}$ // $\perp$ if one is $\perp$
14 : **if** uid is absolute majority of uid_i **then**
15 : **return** uid
16 : **else return** $\perp$

$\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Sim}(ck, \mathsf{pk}, vper)$

1 : // Prepare cut-and-choose encryption

2 : **for** $i = 1, \ldots, \ell(\lambda)$

3 : $m_{i,0}, m_{i,1} \xleftarrow{r} \{0,1\}^{\mathsf{poly}_{\mathsf{msg}}}$ // Simulate random shares

4 : $usk_{i,0}, usk_{i,1} \xleftarrow{r} \mathsf{SKE}_{\mathsf{NCE}}.\mathcal{K}$

5 : $uid_{i,0}, uid_{i,1} \xleftarrow{r} \mathcal{UID}$

6 : **for** $b \in \{0,1\}$ $d_{i,b} = (w_{i,b}, r_{i,b}^{com})$

7 : $w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, r_{i,b}^{RE}, r_{i,b}^{LE})$

8 : $com_{i,b} = \mathsf{COM.Com}(ck, w_{i,b}; r_{i,b}^{com})$

9 : $com_{i,b} = \mathsf{COM.Com}(ck, (usk_{i,b}, uid_{i,b}); r_{i,b}^{com})$

10 : $ct_{i,b}^{RE} = \mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Sim}(1^{\lambda}, \mathsf{pk})$ // Simulate encryptions

11 : $ct_{i,b}^{LE} = \mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Sim}(1^{\lambda})$

12 : // Non-interactively compute challenge γ

13 : $stmt = (vper, \mathsf{pk}_R, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b})$

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

15 : // Simulate the consistency proof.

16 : $\pi_{\mathsf{con}} = \mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Sim}(stmt, \mathcal{R}_{\mathsf{PKEAS}})$

17 : $\pi_{\mathsf{cut}} = (d_{i,\gamma_i})_i$

18 : **return** $(\mathsf{pk}, ((com_{i,b}, ct_{i,b})_{i,b}, ct_R, (\pi_{\mathsf{cut}}, \pi_{\mathsf{con}})).$

$\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Expln}_{\mathsf{RE}}(ck, \mathsf{sk}, c, m, vper)$

1 : **parse** $c = (\mathsf{pk}, ((com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b}), ((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$

2 : $\gamma = \mathsf{RO}(vper, \mathsf{pk}, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b})$

3 : // Explain all unopened $ct_{i,b}^{RE}$

4 : **for** $i = 1, \ldots, \ell(\lambda)$

5 : $b = 1 - \gamma_i$

6 : $\mathsf{PKE}_{\mathsf{NCE}}.\mathsf{Expln}(\mathsf{sk}, ct_{i,b}^{RE}, m - m_{i,b})$

$\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Expln}_{\mathsf{LE}}(ck, usk, c, uid, vper)$

1 : **parse** $c = (\mathsf{pk}, ((com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b}), ((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$

2 : $\gamma = \mathsf{RO}(vper, \mathsf{pk}, (com_{i,b}, ct_{i,b}^{RE}, ct_{i,b}^{LE})_{i,b})$

3 : // Explain all unopened $ct_{i,b}^{LE}$

4 : **for** $i = 1, \ldots, \ell(\lambda)$

5 : $b = 1 - \gamma_i$

6 : $\mathsf{SKE}_{\mathsf{NCE}}.\mathsf{Expln}(usk - usk_{i,b}, ct_{i,b}^{LE}, uid - uid_{1,b})$

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 $\mathsf{PKE}_{\mathsf{AS}}$, 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 $\mathcal{F}_{\mathrm{ASTE}}$ *and* $\mathcal{F}_{\mathrm{AS}}$ Since we do not consider adaptive corruption, there is no immediate relation between $\mathcal{F}_{\mathrm{ASTE}}$ and NINCE. However, there is an informal connection which suggests that implementing $\mathcal{F}_{\mathrm{ASTE}}$ based on $\mathcal{F}_{\mathrm{AS}}$ requires similar assumptions and techniques as NINCE. Namely, if $\mathcal{F}_{\mathrm{AS}}$ is used to store escrow keys, and non-interactive encryption and decryption algorithms are used, as is done in $\mathit{\Pi}_{\mathrm{ASTE}}$, then the request of the escrow keys behaves very similar to adaptive corruption of the key. In particular, this suggests that realizing $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathsf{PKE}_{\mathsf{AS}}$

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

Clearly, we require $2\ell(\lambda)$ encryptions with $\mathsf{PKE}_{\mathsf{NCE}}$ and $\mathsf{SKE}_{\mathsf{NCE}}$, where $\ell(\lambda) = 2\lambda$ is the minimal choice for λ bits of security.[28] With space-efficient public key encryption schemes based on DLOG, the size of public key encryptions is two

[28] If k of the cut-and-choose ciphertexts are "bad", e.g., don't share the message m extracted from π_{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 $\mathsf{PKE}_{\mathsf{AS}}$ to fail/be inconsistent with π_{con}. For λ 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^{λ} tries are required to get a constant success chance.

group elements of size 2λ each. For secret key encryptions, the size is (at least) plaintext length, which is at least 2λ 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 8λ group exponentiations are required.

For the cut-and-choose proof, there are many options and trade-offs. Our definition of $\mathsf{PKE}_{\mathsf{AS}}$ 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 π_{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 $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Enc}$):

- At least 1536kiB are required per ciphertext.
- For succinct (and efficient) zero-knowledge proofs and commitments, the total size should be around 1536kiB to 2MiB, 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 Σ-protocols and ElGamal commitments:
  - At least additionally 8λ group exponentiations are required for computing commitments and the same amount for the proofs.[29] This leads to at least 24λ 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 4.5MiB for $\lambda = 128$.

Overall, $\mathsf{PKE}_{\mathsf{AS}}$ 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 $\mathsf{PKE}_{\mathsf{AS}}$ 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 Σ-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) 2λ 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 $\mathcal{F}_{\mathrm{AS}}$ nor in $\mathcal{F}_{\mathrm{AD}}$): 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: Π_{AS} UC-realizes $\mathcal{F}_{\text{AS}}$

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

Theorem 1. *Π_{AS} UC-realizes $\mathcal{F}_{\text{AS}}$ in the $\{\mathcal{F}_{\text{AD}}, \mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BB}}, \mathcal{G}_{\text{CLOCK}}\}$-hybrid model under the assumptions that* COM *is a (computationally) hiding, (statistically) binding and (dual-mode) extractable and equivocable commitment scheme, Σ 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 $\mathcal{S}$ for those corruptions and then proceed to prove the security. The simulator uses the following writing conventions:

- If $\mathcal{F}_{\text{AS}}$ 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 $\mathcal{F}_{\text{CRS}}$ and $\mathcal{F}_{\text{BB}}$ are simulated honestly. We assume in the following that whenever calls to those functionalities are made, $\mathcal{S}$ simulates them honestly.
- Hybrid functionality $\mathcal{F}_{\text{AD}}$ is mostly simulated honestly, except the tasks Request Decryption, Get Statistics and Handover (for details see the paragraph "simulation of $\mathcal{F}_{\text{AD}}$")

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

Simulator for System Security

State of the Simulator

- Setup
  - TPKE keypair: $(\mathsf{pk}, \mathsf{sk})$
  - SO Signing keypair: $(\mathsf{vk}_{\mathrm{SO}}, \mathsf{sk}_{\mathrm{SO}})$
  - J Signing keypair: $(\mathsf{vk}_{\mathrm{J}}, \mathsf{sk}_{\mathrm{J}})$
  - Common reference strings $(crs_{\mathrm{zk}}^{AS}, crs_{\mathrm{com}})$
  - Trapdoor $td_{\mathrm{com}}^{\mathrm{ext}}$ for extracting commitments
  - Trapdorr td_{NIZK} for simulating and extracting NIZKs
- $\mathsf{L}_{\mathsf{Registered}}^{\mathsf{sys-sec}}$ with $(uid, \sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}})$ entries for *corrupted* users. Filled in the task User Registration. This is the same list that SO keeps in Π_{AS}.
- $\mathsf{L}_{\mathsf{HonestU}}^{\mathsf{sys-sec}}$ with $(pid, uid, \mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}})$ entries for *honest* users. Filled in the task User Registration.
- $\mathsf{L}_{\mathsf{StoreSecret}}^{\mathsf{sys-sec}}$ with $(uid, vper, secret, sec_1, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}, \sigma_{sec_1}, sec_2, ct, \sigma_{\mathrm{U}})$ entries for *corrupted* users. Filled during StoreSecret.

- If LE is corrupted:
  - The list $\mathsf{L}_{\mathsf{Warrants}}^{\mathsf{sys-sec}}$ with $(W, 1, \widetilde{W}, \sigma_{\widetilde{W}})$ entries. Stored in Request Warrant (like J in Π_{AS})
  - The list $\mathsf{L}_{\mathsf{HonestSecrets}}^{\mathsf{sys-sec}}$ stores $(uid_i, vper_i, sec_{1,i})$ entries for *honest* users. Filled during Request Warrant and used in $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption
  - The list $\mathsf{L}_{\mathsf{ReqNotReady}}^{\mathsf{sys-sec}}$ temporarily stores $(ct_i, sec_{2,i})$ entries for *honest* users. Filled during the simulation of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption and cleared during the simulation of $\mathcal{F}_{\mathrm{AD}}$.Handover.
- If LE is honest:
  - The lists $\mathsf{L}_{\mathsf{Warrants}}$, $\mathsf{L}_{\mathsf{Requests}}$, $\mathsf{L}_{\mathsf{Requests}}^{\mathsf{Ready}}$ and $\mathsf{L}_{\mathsf{Stats}}$ in $\mathcal{F}_{\mathrm{AD}}$ are not used. Instead, manage a separate list for statistics: $\mathsf{L}_{\mathsf{Stats}}^{\mathsf{sys-sec}}$ with $f_t(W)$ entries. This list is filled during RequestWarrant.
- List for simulation of $\mathcal{F}_{\mathrm{BB}}$: L_{BB} (initially empty).

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

- $(crs_{\mathrm{zk}}^{AS}, td_{\mathrm{NIZK}}) \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$
- $(crs_{\mathrm{com}}, td_{\mathrm{com}}^{\mathrm{ext}}) \leftarrow \mathsf{COM.Setup}^{\mathrm{Ext}}(1^\lambda)$
- Store all generated values

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

- Create empty list L_{BB}.

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

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

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

- Choose the same system parameters for $\mathcal{F}_{\text{AS}}$ as Π_{AS}

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

$\mathcal{S}$ answers all calls honestly.

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

$\mathcal{S}$ answers all calls honestly.

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

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

- **Request Decryption**
  1. Retrieve $\left(\text{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$ as input from LE
  2. Check Warrant Signature: $b \leftarrow \Sigma.\mathsf{Vfy}(\mathsf{vk}_{\text{J}}, \widetilde{W}, \sigma_{\widetilde{W}})$
  3. If $b = 0$, abort (Warrant not valid)
  4. Find the entry $(W, 1, \widetilde{W}, \sigma_{\widetilde{W}}) \in \mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Warrants}}$ containing $\widetilde{W}$. If none exists, abort (warrant not granted)
  5. Call $\mathcal{F}_{\text{AS}}$ 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: $(\widetilde{W}_1, \ldots, \widetilde{W}_v) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i$

7. For each i from 1 to v:
   (a) Check if there is an entry $(\cdot, uid_i, \cdot, \cdot) \in \mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestU}}$. If yes (case: user is honest) go to step 7b, if not (case: user is corrupted) go to step 7f.
   (b) For uid_i, retrieve $(uid_i, vper_i, sec_{1,i})$ from $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestSecrets}}$
   (c) Compute $sec_{2,i}$ such that $sec_{1,i} \oplus sec_{2,i} = secret_i$
   (d) Add $(ct_i, sec_{2,i})$ to $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{ReqNotReady}}$
   (e) Then go to step 8
   (f) Append ct_i to $\mathsf{L}_{\mathsf{Requests}}$ of $\mathcal{F}_{\mathrm{AD}}$
   (g) Then go to step 8
8. Send $(\textsc{Request}, pid_{\mathrm{LE}}, f_t(W), |\widetilde{W}|, v)$ to $\mathcal{A}$
9. When $\mathcal{A}$ allows to deliver output, store $\widetilde{W}$ in list of warrants $\mathsf{L}_{\mathsf{Warrants}}$ of $\mathcal{F}_{\mathrm{AD}}$
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 $(\textsc{GotStatistics}, \mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Stats}})$

– **Handover**

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

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

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

Party Init (honest P)

1. Upon receiving the leak (PInit) from $\mathcal{F}_{\mathrm{AS}}$
   (a) Send (GetSKeys) and (GetPK) to the adversary
   (b) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to all parties

Party Init (corrupted P)

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

User Registration (honest U, honest SO)

1. Upon receiving the leak $(\textsc{Register}, pid, uid)$ from $\mathcal{F}_{\mathrm{AS}}$
   (a) Send (Ok) to $\mathcal{F}_{\mathrm{AS}}$

2. When $\mathcal{F}_{\mathrm{AS}}$ wants to deliver output¸
   (a) Generate keypair for the user: $(\mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
   (b) Store $(pid, uid, \mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}})$ in list of honest users $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestU}}$
   (c) Honestly simulate REGISTER and RETRIEVE calls to $\mathcal{F}_{\mathrm{BB}}$ including leaks to the adversary
   (d) Send (GETSKEYS) and (GETPK) to the adversary
   (e) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output

User Registration (corrupted U, honest SO)

1. Upon receiving the message (uid) from U with pid $pid_{\mathrm{U}}{}'$
   (a) If there exists an entry $(uid, \cdot, \cdot, \cdot)$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Registered}}$, let $\mathcal{S}$ abort in the name of SO (User already successfully registered)
   (b) Call $\mathcal{F}_{\mathrm{AS}}$ in the name of U with $(\text{REGISTER}, uid)$ under the pid $pid_{\mathrm{U}}{}'$.
      i. After receiving $\big(\text{REGISTER}, pid_{\mathrm{U}}{}', uid\big)$ from $\mathcal{F}_{\mathrm{AS}}$, send (OK) to $\mathcal{F}_{\mathrm{AS}}$
      ii. If $\mathcal{F}_{\mathrm{AS}}$ aborts, $\mathcal{S}$ aborts as well.
2. Upon receiving output (REGISTERED) from $\mathcal{F}_{\mathrm{AS}}$ to U
   (a) Note that receiving this output implies that $uid = uid'$, where uid' is the secret SO input.
   (b) Report message (UID_OK) from SO to U
3. Upon receiving (OK) from U to SO
   (a) Honestly simulate call $(\text{RETRIEVE}, uid)$ to $\mathcal{F}_{\mathrm{BB}}$ including leaks to the adversary
   (b) Receive answer $(\text{RETRIEVE}, pid_{\mathrm{U}}{}^*, uid, \mathsf{vk}^*_{\mathrm{U}})$ from $\mathcal{F}_{\mathrm{BB}}$. If $pid_{\mathrm{U}}{}^* \neq pid_{\mathrm{U}}{}'$, abort (UID already taken by another user or not registered).
   (c) $(\mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}) = \mathsf{COM.Com}(crs_{\mathrm{com}}, uid)$
   (d) $\sigma_{\mathrm{reg}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{SO}}, \mathsf{com}_{\mathrm{uid}})$
   (e) Store $(uid, \sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}})$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Registered}}$
   (f) Report message $(\sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}})$ from SO to U
   (g) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to SO

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

Store Secret (corrupted U, honest SO)

1. Upon receiving $(uid, \sigma_{\mathrm{reg}}, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \mathsf{com}_{sec_2}, \mathsf{com}_{\mathrm{vper}}, \mathsf{decom}_{\mathrm{vper}})$ from U to SO:
   (a) Extract Commitment $\mathsf{com}_{\mathrm{vper}}$ to get $vper$: $vper \leftarrow \mathsf{COM.Extract}(td^{\mathrm{ext}}_{\mathrm{com}}, \mathsf{com}_{\mathrm{vper}})$
   (b) Call $\mathcal{F}_{\mathrm{AS}}$ with input $(\text{STORESECRET}, uid, vper)$ in the name of U
   (c) If $\mathcal{F}_{\mathrm{AS}}$ aborts, let $\mathcal{S}$ abort as well (in the name of SO)

2. Upon receiving ($\textsc{SecretStored}$, *secret*) as output from $\mathcal{F}_{\text{AS}}$ to U
   (a) Extract Commitment com_{sec_2} to get sec_2: $sec_2 \leftarrow \mathsf{COM.Extract}(td^{\text{ext}}_{\text{com}}, \mathsf{com}_{sec_2})$
   (b) Calculate sec_1 such that $secret = sec_1 \oplus sec_2 \in \mathcal{S}$
   (c) Report message (sec_1) from SO to U
3. Upon receiving $(ct, \mathsf{com}_{\text{sec}}, \pi_{\text{ss}}, \sigma_{\text{U}}, \sigma_{sec_1}, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1})$ from U to SO:
   (a) Compute signature $\sigma_{\text{ss}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\text{SO}}, (\mathsf{com}_{\text{uid}}, \mathsf{com}_{\text{sec}}, \mathsf{com}_{vper}))$
   (b) Report message (σ_{ss}) from SO to U
   (c) Store $(uid, vper, secret, sec_1, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}, \sigma_{sec_1}, sec_2, ct, \sigma_{\text{U}})$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{StoreSecret}}$
   (d) Allow $\mathcal{F}_{\text{AS}}$ 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 $\mathcal{F}_{\text{AS}}$ in the name of LE with input ($\textsc{RequestWarrant}$, W)
   (b) Allow $\mathcal{F}_{\text{AS}}$ to deliver output to J
2. *Case 1:* If then $\mathcal{F}_{\text{AS}}$ delivers the output $(\textsc{RequestWarrant}, 0)$ to LE:
   (a) Report message $(0, \perp)$ from J to LE
   Note that this concluded the simulation for this task in this case
3. *Case 2:* If then $\mathcal{F}_{\text{AS}}$ leaks $\left(\textsc{RequestWarrant}, f_t(W), |\widetilde{W}|, v\right)$:
   (a) Set $b := 1$
   (b) Set $\sigma_W = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\text{J}}, W)$
   (c) Store $(W, b, \perp, \perp)$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Warrants}}$
   (d) Report message (b, σ_W) from J to LE
4. Upon receiving the message (W, σ'_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 $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{C}}$ and $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{MsgReqW}}$
   (d) For each i from 1 to v:
      i. Check if there is an entry $(\cdot, uid_i, \cdot, \cdot) \in \mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestU}}$. If yes (case: user is honest) go to step 4(d)ii, if not (case: user is corrupted) go to step 4(d)x
      ii. Fake ciphertext $ct_i \leftarrow \mathsf{TPKE.Enc}(\mathsf{pk}, 0)$ and append ct_i to $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{C}}$
      iii. For uid_i, retrieve entry $(pid_i, uid_i, \mathsf{vk}_{\text{U}}, \mathsf{sk}_{\text{U}}) \in \mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestU}}$.
      iv. Create signature $\sigma_{\text{U,i}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\text{U}}, (ct_i, uid_i, vper_i))$
      v. Draw random $sec_{1,i} \xleftarrow{r} \mathcal{S}$ and commit to it: $(\mathsf{com}_{sec_{1,i}}, \mathsf{decom}_{sec_{1,i}}) \leftarrow \mathsf{COM.Com}(crs_{\text{com}}, sec_{1,i})$
      vi. Create signature $\sigma_{sec_1,\text{i}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\text{U}}, \mathsf{com}_{sec_{1,i}})$
      vii. Append $(\sigma_{\text{U,i}}, sec_{1,i}, \mathsf{com}_{sec_{1,i}}, \mathsf{decom}_{sec_{1,i}}, \sigma_{sec_1,\text{i}})$ to $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{MsgReqW}}$
      viii. Append $(uid_i, vper_i, sec_{1,i})$ to $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestSecrets}}$
      ix. Then go to step 4e

x. For uid_i retrieve the entry $(uid_i, vper_i, secret_i, sec_{1,i}, \mathsf{com}_{sec_{1,i}}, \mathsf{decom}_{sec_{1,i}}, \sigma_{sec_1,\mathrm{i}}, sec_{2,i}, ct_i, \sigma_{\mathrm{U,i}})$ from $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{StoreSecret}}$. If none exists, abort in the name of SO (User has not stored a secret)

xi. Append ct_i to $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{C}}$ and $(\sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_{1,i}}, \mathsf{decom}_{sec_{1,i}}, \sigma_{sec_1,\mathrm{i}})$ to $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{MsgReqW}}$

xii. Then go to step 4e

(e) Build enhanced warrant $\widetilde{W}$ by adding ct_i (from $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{C}}$) to each W_i

(f) Report message $(\widetilde{W}, \mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{MsgReqW}})$ from SO to LE

5. Upon receiving the message $(\widetilde{W}, \{\sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_{1,i}}, \mathsf{decom}_{sec_{1,i}}, \sigma_{sec_1,\mathrm{i}}\}_{i\in\{1,\ldots,v\}})$ from LE to J
   (a) Build W out of $\widetilde{W}$ by deleting ct_i from each $\widetilde{W_i}$
   (b) If there is no entry $(W, 1, \perp, \perp)$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Warrants}}$, abort in the name of J
   (c) Parse enhanced warrant: $(\widetilde{W_1}, \ldots, \widetilde{W_v}) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W_i}$
   (d) For i from 1 to v:
      – For uid_i, retrieve entry $(pid_{\mathrm{U,i}} uid_i, \mathsf{vk}_{\mathrm{U,i}}) \in \mathsf{L}_{\mathsf{BB}}$
      – Check signatures: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, (ct_i, uid, vper_i), \sigma_{\mathrm{U,i}}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, \mathsf{com}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}) = 0$, abort in the name of J
      – Check commitment: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, sec_{1,i}) = 0$, abort in the name of J
   (e) Sign enhanced warrant: $\sigma_{\widetilde{W}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, \widetilde{W})$
   (f) Change the entry $(W, 1, \perp, \perp)$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Warrants}}$ to $(W, 1, \widetilde{W}, \sigma_{\widetilde{W}})$
   (g) Report message $\left(\sigma_{\widetilde{W}}\right)$ from J to LE
6. When the adversary allows $\mathcal{F}_{\mathrm{AD}}$ to deliver output in $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption
   (a) Finish simulation of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption
   (b) Send (Ok) to $\mathcal{F}_{\mathrm{AS}}$
   (c) Receive output (WarrantAnswer, b) from $\mathcal{F}_{\mathrm{AS}}$

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

1. Upon being notified that $\mathcal{F}_{\mathrm{AS}}$ wants to deliver output to J, allow the output
2. Upon receiving the leak $\left(\text{RequestWarrant}, f_t(W), |\widetilde{W}|, v\right)$ from $\mathcal{F}_{\mathrm{AS}}$:
   (a) Send (Request, $pid_{\mathrm{LE}}, f_t(W), |\widetilde{W}|, v$) to $\mathcal{A}$
3. When $\mathcal{A}$ allows to deliver output in $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption
   (a) Store $f_t(W)$ in $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Stats}}$ for managing statistics in $\mathcal{F}_{\mathrm{AD}}$
   (b) Send (Ok) to $\mathcal{F}_{\mathrm{AS}}$
   (c) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to LE

Get Secrets (corrupted LE)
This is handled by simulation of $\mathcal{F}_{\text{AD}}$.

Get Secrets (honest LE)

1. Upon receiving the leak (GetSecrets) from $\mathcal{F}_{\text{AS}}$
   (a) Leak (Retrieve) to $\mathcal{A}$
   (b) Allow $\mathcal{F}_{\text{AS}}$ to deliver output

Get Statistics (corrupted P)
This is handled by simulation of $\mathcal{F}_{\text{AD}}$.

Get Statistics (honest P)

1. Upon receiving the leak (GetStatistics) from $\mathcal{F}_{\text{AS}}$
   (a) Send (GetStatistics) to $\mathcal{A}$ and wait for okay from $\mathcal{A}$
   (b) Allow $\mathcal{F}_{\text{AS}}$ to deliver output

Audit (honest AU)

1. Upon receiving the leak (AuditRequest) from $\mathcal{F}_{\text{AS}}$
   (a) Send (AuditRequest) to $\mathcal{A}$ and wait for okay from $\mathcal{A}$
   (b) Allow $\mathcal{F}_{\text{AS}}$ 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 $\mathcal{F}_{\text{AS}}$
   (a) Simulate proof: $\pi \leftarrow \mathsf{NIZK.Sim}(td_{\text{NIZK}}, stmt_R, \mathcal{R}_{zk}^{AS})$
   (b) Send (Proof, π) to $\mathcal{F}_{\text{AS}}$

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 $\mathcal{F}_{\text{AS}}$
   (a) Extract Witness: $wit_R \leftarrow \mathsf{NIZK.Ext}(td_{\text{NIZK}}, stmt_R, \pi, \mathcal{R}_{zk}^{AS})$
   (b) Send (witness, wit_R) to $\mathcal{F}_{\text{AS}}$

We now prove the following theorem:

Theorem 5 (System Security). *Π_{AS} UC-realizes $\mathcal{F}_{\text{AS}}$ in the $\{\mathcal{F}_{\text{AD}}, \mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BB}}, \mathcal{G}_{\text{CLOCK}}\}$-hybrid model under the assumptions that*

- COM *is a (computationally) hiding, (statistically) binding and extractable commitment scheme*
- Σ *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 $\mathcal{A}$ 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 $\mathcal{F}_{\mathrm{AS}}^{i+1}$ behaves the same as $\mathcal{F}_{\mathrm{AS}}^{i}$ except for the stated changes and $\mathcal{S}^{i+1}$ behaves the same as $\mathcal{S}^{i}$ except for the stated changes.

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

$$H_0 := \mathsf{Exp}^{\Pi_{\mathrm{AS}}}_{\mathcal{F}_{\mathrm{AD}},\mathcal{F}_{\mathrm{CRS}},\mathcal{F}_{\mathrm{BB}},\mathcal{G}_{\mathrm{CLOCK}},\mathcal{S}^0,\mathcal{Z}}$$

with $\mathcal{S}^0$ being the dummy adversary. This means that all parties execute the real protocol.

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

Proof Sketch: Game 0 $\approx$ Game 1.
$\mathcal{S}^1$ executes the same code as the real parties on the same inputs (that have been forwarded by $\mathcal{F}_{\mathrm{AS}}^1$). Thus, Game 0 and Game 1 are identical from the environment's view.

Game 2. In this game, $\mathcal{S}^2$ generates the common reference string for the commitment scheme with an extraction trapdoor $td_{\mathrm{com}}^{\mathrm{ext}}$ as $(crs_{\mathrm{com}}, td_{\mathrm{com}}^{\mathrm{ext}}) \leftarrow \mathsf{COM.Setup}^{\mathrm{Ext}}(1^\lambda)$. $\mathcal{S}^2$ also generates the common reference string for the non-interactive proof system with a trapdoor for simulating and extracting proofs, i.e., $(crs_{\mathrm{zk}}^{AS}, td_{\mathrm{NIZK}}) \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$. Thus, $\mathcal{S}^2$ simulates $\mathcal{F}_{\mathrm{CRS}}$ the same way $\mathcal{S}$ does.

Proof Sketch: Game 1 $\approx$ Game 2.
Indistinguishability follows from the extractability of COM and the simulation-extractability of NIZK.

Game 3. During setup of $\mathcal{F}_{\mathrm{AD}}$, $\mathcal{S}^3$ is asked for the keypairs for SO, J and AU. $\mathcal{S}^3$ generates the keypairs honestly and stores them.

Proof Sketch: Game 2 $\approx$ Game 3.
$\mathcal{S}^3$ runs the same key generation algorithm as the real protocol does for honest parties, therefore $\mathcal{Z}$'s view remains unchanged.

Game 4. $\mathcal{S}^4$ keeps track of additional information. For each honest U, it stores $(uid, vper, sec_1)$ in the list $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestSecrets}}$ at the end of the Store Secret task. For each corrupted U, it stores $(uid, vper, \perp, sec_1, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}, \sigma_{sec_1}, sec_2,$

ct, σ_{U}) in the list $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{StoreSecret}}$ at the end of the Store Secret task. Note that normally instead of $\perp$ 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.
$\mathcal{S}^4$ only keeps track of more information (that the honest parties acquire during the real protocol), nothing to distinguish for $\mathcal{Z}$.

Game 5. In this game, $\mathcal{F}^5_{\mathrm{AS}}$ additionally handles the tasks System Init and Party Init the same way $\mathcal{F}_{\mathrm{AS}}$ does, and $\mathcal{S}^5$ handles these task the same way $\mathcal{S}$ 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: $\mathcal{Z}$'s view remains identical, since $\mathcal{S}^5$ only needs to leak (GetSKeys) and (GetPK) to $\mathcal{A}$ to emulate the call to $\mathcal{F}_{\mathrm{AD}}$.
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, $\mathcal{F}^6_{\mathrm{AS}}$ additionally handles the Audit task the same way $\mathcal{F}_{\mathrm{AS}}$ does, and $\mathcal{S}^6$ handles this task the same way $\mathcal{S}$ does. Note that since the auditor AU is assumed to be honest, simulation only requires a simulated (AuditRequest) to $\mathcal{F}_{\mathrm{BCRA}}$.

Proof Sketch: Game 5 $\approx$ Game 6.
Since AU is honest, these games are identical.

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

Proof Sketch: Game 6 $\approx$ Game 7.
For honest U: $\mathcal{Z}$'s view remains identical, since $\mathcal{S}^7$ only needs to leak (GetSKeys) and (GetPK) to $\mathcal{A}$ to emulate the calls to $\mathcal{F}_{\mathrm{AD}}$ and to honestly emulate the calls to $\mathcal{F}_{\mathrm{BB}}$. Note that additionally some information about honest users (pid, uid and keys) are stored in a designated list $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{HonestU}}$.
For corrupted U: Since $\mathcal{S}$ essentially simulates SO in the User Registration task honestly, the hybrids are indistinguishable for $\mathcal{Z}$. Note that $\mathcal{S}$ keeps track of the list of registered users like SO does in Π_{AS}.

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

Proof Sketch: Game 7 $\approx$ Game 8.
$\mathcal{F}^8_{\mathrm{AS}}$ sends $\left(\textsc{RequestWarrant}, f_t(W), |\widetilde{W}|, v\right)$ to $\mathcal{S}^8$. $\mathcal{S}^8$ then simulates the execution of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption by sending ($\textsc{Request}, pid_{\mathrm{LE}}, f_t(W), |\widetilde{W}|$, v) to $\mathcal{A}$, 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 $\mathsf{L}_{\mathsf{Warrants}}$ (since we do not actually know the warrant). This is fixed by directly storing $f_t(W)$ in a separate list $\mathsf{L}^{\mathsf{sys-sec}}_{\mathsf{Stats}}$ during Request Warrant and using this list in $\mathcal{F}_{\mathrm{AD}}$.GetStatistics to return the same statistics as the real game does. The task $\mathcal{F}_{\mathrm{AD}}$.GetStatistics now behaves like $\mathcal{S}$ 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, $\mathcal{F}_{\mathrm{AS}}^{9}$ additionally handles the Request Warrant task the same way $\mathcal{F}_{\mathrm{AS}}$ does for corrupted LE, and $\mathcal{S}^{9}$ handles this task the same way $\mathcal{S}$ does for corrupted LE. Additionally, $\mathcal{S}^{9}$ simulates $\mathcal{F}_{\mathrm{AD}}$ as $\mathcal{S}$ does for the Request Decryption and Handover tasks in $\mathcal{F}_{\mathrm{AD}}$. Note that if LE is honest, this game changes nothing. Also note that now $\mathcal{S}^{9}$ simulates $\mathcal{F}_{\mathrm{AD}}$ as $\mathcal{S}$ does (for every task and every corruption scenario).

Proof Sketch: Game 8 $\approx$ Game 9.
$\mathcal{S}^{9}$ receives the necessary information from LE to call $\mathcal{F}_{\mathrm{AS}}^{9}$ with the correct input. After receiving output from $\mathcal{F}_{\mathrm{AS}}^{9}$, $\mathcal{S}^{9}$ 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 $\mathsf{L}_{\mathsf{StoreSecret}}^{\mathsf{sys-sec}}$ that was filled during Store Secret (for corrupted users). For honest users, the strategy is a bit different, since the list $\mathsf{L}_{\mathsf{StoreSecret}}^{\mathsf{sys-sec}}$ has no entries for honest users. For honest users, $\mathcal{S}^{9}$ 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 Σ is EUF-CMA-secure and the commitment scheme COM is binding, $\mathcal{S}^{9}$ aborts in the same cases as SO/J would do in Π_{AS}. Note that $\mathcal{S}^{9}$ stores the granted warrants in $\mathsf{L}_{\mathsf{Warrants}}^{\mathsf{sys-sec}}$ during the Request Warrant task.
Now, the simulation of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption needs to be handled as well. Since Σ is EUF-CMA-secure, $\mathcal{S}^{9}$ aborts in the same cases as the real $\mathcal{F}_{\mathrm{AD}}$ would. $\mathcal{S}^{9}$ has access to the list $\mathsf{L}_{\mathsf{Warrants}}^{\mathsf{sys-sec}}$ that is filled during Request Warrant (which needs to be called before this task). Using $\mathsf{L}_{\mathsf{Warrants}}^{\mathsf{sys-sec}}$, $\mathcal{S}^{9}$ can call $\mathcal{F}_{\mathrm{AS}}$ 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 $\mathsf{L}_{\mathsf{Requests}}$ of $\mathcal{F}_{\mathrm{AD}}$ like an honest $\mathcal{F}_{\mathrm{AD}}$. The remainder of the secret retrieval (for corrupted users) is then handled by the honest simulation of $\mathcal{F}_{\mathrm{AD}}$. For honest users, we retrieve the entry $(uid, vper, sec_1)$ from $\mathsf{L}_{\mathsf{HonestSecrets}}^{\mathsf{sys-sec}}$ and use the output of $\mathcal{F}_{\mathrm{AS}}^{9}$ to calculate the secret sec_2 (in the clear) that needs to be retrieved. We then store this secret sec_2 directly to a special list $\mathsf{L}_{\mathsf{ReqNotReady}}^{\mathsf{sys-sec}}$. Then the $\mathcal{F}_{\mathrm{AD}}$.Handover task has to be adapted as well. It now additionally moves all entries from $\mathsf{L}_{\mathsf{ReqNotReady}}^{\mathsf{sys-sec}}$ to $\mathsf{L}_{\mathsf{Requests}}^{\mathsf{Pending}}$. 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, $\mathcal{F}_{\mathrm{AS}}^{10}$ additionally handles the tasks Get Secrets and Get Statistics the same way $\mathcal{F}_{\mathrm{AS}}$ does, and $\mathcal{S}^{10}$ handles this task the same way $\mathcal{S}$ does.

Proof Sketch: Game 9 $\approx$ Game 10.
Get Secrets: For honest LE, $\mathcal{Z}$'s view remains identical, since $\mathcal{S}^{10}$ only needs to leak (RETRIEVE) to $\mathcal{A}$ to emulate the $\mathcal{F}_{\mathrm{AD}}$.RetrieveSecret task. For corrupted

LE, only the simulation of $\mathcal{F}_{\text{AD}}$.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, $\mathcal{S}^{10}$ only needs to send (GetStatistics) to $\mathcal{A}$. For a corrupted party, this game is identical to Game 9 since this task is already handled by the simulation of $\mathcal{F}_{\text{AD}}$.

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

Proof Sketch: Game 10 ≈ Game 11.
For corrupted U: $\mathcal{S}^{11}$ can obtain the required inputs to call $\mathcal{F}_{\text{AS}}^{11}$ from the message $(uid, \sigma_{\text{reg}}, \mathsf{com}_{\text{uid}}, \mathsf{decom}_{\text{uid}}, \mathsf{com}_{sec_2}, \mathsf{com}_{\text{vper}}, \mathsf{decom}_{\text{vper}})$ from U by extracting $vper$ from $\mathsf{com}_{\text{vper}}$. Since COM is extractable, this is possible with overwhelming probability and allows $\mathcal{S}^{11}$ to call $\mathcal{F}_{\text{AS}}^{11}$ with the correct input. Using the output (SecretStored, *secret*) of $\mathcal{F}_{\text{AS}}^{11}$ and by extracting sec_2 from com_{sec_2} (which is again possible with overwhelming probability since COM is extractable), $\mathcal{S}^{11}$ 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, $\mathcal{F}_{\text{AS}}^{12}$ additionally handles the Prove and Verify tasks the same way $\mathcal{F}_{\text{AS}}$ does, and $\mathcal{S}^{12}$ handles this task the same way $\mathcal{S}$ does.

Proof Sketch: Game 11 ≈ Game 12.
Since NIZK is a simulation extractable zero-knowledge proof system, $\mathcal{Z}$ can not distinguish between the outputs it gets from $\mathcal{F}_{\text{AS}}^{12}$ and the outputs it gets from $\mathcal{F}_{\text{AS}}$.

Note that $\mathcal{S}^{12}$ equals the final simulator $\mathcal{S}$. 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. □

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 $\mathcal{S}$ for those corruptions and then proceed to prove the security. The simulator uses the following writing conventions:

- If $\mathcal{F}_{\text{AS}}$ 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 $\mathcal{F}_{\text{CRS}}$, $\mathcal{F}_{\text{BB}}$ are simulated honestly. We assume in the following that whenever calls to those functionalities are made, $\mathcal{S}$ simulates them honestly.

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

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

Simulator for User Security

State of the Simulator

– Setup:
  - TPKE keypair: $(\mathsf{pk}, \mathsf{sk})$
  - Verification key of SO: $\mathsf{vk}_{\mathrm{SO}}$
  - J Signing keypair: $(\mathsf{vk}_{\mathrm{J}}, \mathsf{sk}_{\mathrm{J}})$
  - Common reference strings $(crs^{AS}_{\mathrm{zk}}, crs^{\mathrm{eq}}_{\mathrm{com}})$
  - Trapdoor td_{NIZK} for simulating and extracting proofs
  - Trapdoor $td^{\mathrm{eq}}_{\mathrm{com}}$ for equivocating commitments
– Stored in User Registration:
  - $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Registered}}$ with $(pid_{\mathrm{U}}, uid, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}}, \mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}})$ entries. Note that only honest users are in that list.
– Stored in Request Warrant:
  - (W, b) — list of already requested preliminary warrants
  - $(\widetilde{W}, \sigma_{\widetilde{W}})$ — list of stored enhanced warrants
– Stored for simulation of $\mathcal{F}_{\mathrm{AD}}$:
  - $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Secrets}}$ with $(uid, vper, sec_1, ct)$ entries. Entries are stored in AS.StoreSecret (for honest U) and used in $\mathcal{F}_{\mathrm{AD}}$.RequestSecret.
  - Only for corrupted LE: The list $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{ReqNotReady}}$ temporarily stores $(ct_i, sec_{2,i})$ entries (for honest U). Filled during the simulation of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption and cleared during the simulation of $\mathcal{F}_{\mathrm{AD}}$.Handover.
  - Everything else that is stored inside $\mathcal{F}_{\mathrm{AD}}$
– List for simulation of $\mathcal{F}_{\mathrm{BB}}$:
  - L_{BB} (initially empty). Entries are of the form $(pid_{\mathrm{U}}, uid, \mathsf{vk}_{\mathrm{U}})$

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

– $(crs^{AS}_{\mathrm{zk}}, td_{\mathrm{NIZK}}) \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$
– $(crs^{\mathrm{eq}}_{\mathrm{com}}, td^{\mathrm{eq}}_{\mathrm{com}}) \leftarrow \mathsf{COM.Setup}^{\mathrm{Equiv}}(1^\lambda)$
– Store all values

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

– Create empty list L_{BB}.

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

– See simulation of Init-Task

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

– Choose the same system parameters for $\mathcal{F}_{\text{AS}}$ as Π_{AS}

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

$\mathcal{S}$ answers all calls honestly.

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

$\mathcal{S}$ answers all calls honestly.

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

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

– **Request Decryption**
  1. Receive $\left(\textsc{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$ as input from LE
  2. Check Warrant Signature: $b \leftarrow \Sigma.\mathsf{Vfy}(\mathsf{vk}_{\text{J}}, \widetilde{W}, \sigma_{\widetilde{W}})$
  3. If $b = 0$, abort (Warrant not valid)
  4. Build preliminary warrant W out of $\widetilde{W}$ by deleting ct_j from each $\widetilde{W}_j$
  5. Call $\mathcal{F}_{\text{AS}}$ in the name of LE with input $(\textsc{GetSecrets}, W)$
  6. If $\mathcal{F}_{\text{AS}}$ aborts, $\mathcal{S}$ lets $\mathcal{F}_{\text{AD}}$ abort as well (Warrant either not requested or not granted)
  7. Receive output $(\textsc{GotSecrets}, (secret_1, \ldots, secret_v))$ from $\mathcal{F}_{\text{AS}}$ to LE
  8. Parse warrant: $\left(\widetilde{W}_1, \ldots, \widetilde{W}_v\right) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i$
  9. For i from 1 to v:
     (a) Check if there is an entry $(uid_i, vper_i, \cdot, ct_i)$ in $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Secrets}}$. If yes (case: user is honest) go to step 9b, if not (case: user is corrupted) go to step 9f.

(b) Retrieve the entry $(uid_i, vper_i, \cdot, ct_i)$ from $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Secrets}}$.
(c) Compute $sec_{2,i}$ such that $sec_{1,i} \oplus sec_{2,i} = secret_i$
(d) Add $(ct_i, sec_{2,i})$ to $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{ReqNotReady}}$
(e) Then go to step 10
(f) Append ct_i to $\mathsf{L}_{\mathsf{Requests}}$ of $\mathcal{F}_{\mathrm{AD}}$
(g) Then go to step 10

10. Send $(\textsc{Request}, pid_{\mathrm{LE}}, f_t(W), |\widetilde{W}|, v)$ to $\mathcal{A}$
11. When $\mathcal{A}$ allows to deliver output, store $\widetilde{W}$ in list of warrants $\mathsf{L}_{\mathsf{Warrants}}$ of $\mathcal{F}_{\mathrm{AD}}$
12. Give output $(\textsc{Request})$ to LE

– **Handover**
Simulate honestly, but additionally execute the following steps at the end:
- For each item $(ct_i, sec_{2,i})$ in $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{ReqNotReady}}$, append $(ct_i, sec_{2,i})$ to $\mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}$ of $\mathcal{F}_{\mathrm{AD}}$
- Clear $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{ReqNotReady}}$

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

1. Upon receiving $(\textsc{Init}, \mathrm{SO}, \mathsf{vk}_{\mathrm{SO}})$ from SO to $\mathcal{F}_{\mathrm{AD}}$
(a) Create signing keypair for J: $(\mathsf{vk}_{\mathrm{J}}, \mathsf{sk}_{\mathrm{J}}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$ and store it
(b) Store $\mathsf{vk}_{\mathrm{SO}}$
(c) Honestly simulate the Init-Task of $\mathcal{F}_{\mathrm{AD}}$ with the following inputs:
J $\rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Init}, \mathrm{J}, \mathsf{vk}_{\mathrm{J}})$
AU $\rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Init}, \mathrm{AU})$
SO $\rightarrow \mathcal{F}_{\mathrm{AD}}$: $(\textsc{Init}, \mathrm{SO}, \mathsf{vk}_{\mathrm{SO}})$
(d) During simulation of the Init-Task, store all generated variables and lists and leak all necessary information to the adversary.
(e) Call $\mathcal{F}_{\mathrm{AS}}$ in the name of SO with input $(\textsc{Init}, \mathrm{SO})$
2. Upon receiving output $(\textsc{InitFinished})$ from $\mathcal{F}_{\mathrm{AS}}$ to SO
(a) Report message $(\textsc{InitFinished})$ from $\mathcal{F}_{\mathrm{AD}}$ to SO
(b) Send $(\textsc{GetSKeys})$ and $(\textsc{GetPK})$ to the adversary
(c) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to all honest parties

Party Init (honest P)

1. Upon receiving the leak $(\textsc{PInit})$ from $\mathcal{F}_{\mathrm{AS}}$
(a) Send $(\textsc{GetSKeys})$ and $(\textsc{GetPK})$ to the adversary
(b) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to all parties

Party Init (corrupted P)
This is handled by simulation of $\mathcal{F}_{\text{AD}}$.

User Registration (honest U, corrupted SO):

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

User Registration (corrupted U, corrupted SO):
This is handled by simulation of $\mathcal{F}_{\text{AD}}$.

Store Secret (honest U, corrupted SO):

1. Upon receiving the leak (StoreSecret, $vper$) from $\mathcal{F}_{\text{AS}}$
   (a) Call $\mathcal{F}_{\text{AS}}$ in the name of SO with input (StoreSecret, $vper$)
   (b) If $\mathcal{F}_{\text{AS}}$ 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 (SecretStored, uid) from $\mathcal{F}_{\text{AS}}$ to SO, but delay output to U
   (d) For uid, retrieve entry $(pid_{\text{U}}, uid, \mathsf{com}_{\text{uid}}, \mathsf{decom}_{\text{uid}}, \sigma_{\text{reg}}, \mathsf{vk}_{\text{U}}, \mathsf{sk}_{\text{U}})$ from $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Registered}}$
   (e) Create fake commitment on sec_2: $(\widetilde{\mathsf{com}_{sec_2}}, eq_2) \leftarrow \mathsf{COM.SimCom}(td^{\text{eq}}_{\text{com}})$
   (f) Create real commitment on $vper$: $(\mathsf{com}_{\text{vper}}, \mathsf{decom}_{\text{vper}}) \leftarrow \mathsf{COM.Com}(crs^{\text{eq}}_{\text{com}}, vper)$
   (g) Report message $(uid, \sigma_{\text{reg}}, \mathsf{com}_{\text{uid}}, \mathsf{decom}_{\text{uid}}, \widetilde{\mathsf{com}_{sec_2}}, \mathsf{com}_{\text{vper}}, \mathsf{decom}_{\text{vper}})$ from U to SO

2. Upon receiving the message (sec_1) from SO to U
   (a) Create fake commitment on *secret*: $(\widetilde{\mathsf{com}}_{\mathrm{sec}}, eq_s) \leftarrow \mathsf{COM.SimCom}(td^{\mathrm{eq}}_{\mathrm{com}})$
   (b) Fake ciphertext $ct \leftarrow \mathsf{TPKE.Enc}(\mathsf{pk}, 0)$
   (c) Honestly commit and sign sec_1: $(\mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1}) \leftarrow \mathsf{COM.Com}(crs_{\mathrm{com}}, sec_1)$ and $\sigma_{sec_1} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{U}}, \mathsf{com}_{sec_1})$
   (d) Create signature $\sigma_{\mathrm{U}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{U}}, (ct, uid, vper))$
   (e) $stmt_{\mathrm{ss}} := (sec_1, \widetilde{\mathsf{com}}_{\mathrm{sec}}, \widetilde{\mathsf{com}}_{sec_2}, \mathsf{pk}, ct, crs_{\mathrm{com}})$
   (f) Fake proof: $\widetilde{\pi_{\mathrm{ss}}} \leftarrow \mathsf{NIZK.Sim}(td_{\mathrm{NIZK}}, stmt_{\mathrm{ss}}, \mathcal{R}^{AS}_{ss})$
   (g) Report message $(ct, \mathsf{com}_{\mathrm{sec}}, \pi_{\mathrm{ss}}, \sigma_{\mathrm{U}}, \sigma_{sec_1}, \mathsf{com}_{sec_1}, \mathsf{decom}_{sec_1})$ from U to SO
3. Upon receiving the message (σ_{ss}) from SO to U
   (a) Verify signature: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{SO}}, (\mathsf{com}_{\mathrm{uid}}, \widetilde{\mathsf{com}}_{\mathrm{sec}}, \mathsf{com}_{\mathrm{vper}}), \sigma_{\mathrm{ss}}) = 0$, let $\mathcal{S}$ abort in the name of U
   (b) Store $(uid, vper, sec_1, ct)$ in $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Secrets}}$
   (c) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to U

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

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

1. Upon receiving the message (W) from LE to J
   (a) Call $\mathcal{F}_{\mathrm{AS}}$ in the name of LE with input (RequestWarrant, W)
   (b) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to J
2. Upon receiving the message (RequestWarrant, W, b) from $\mathcal{F}_{\mathrm{AS}}$:
   (a) If $b = 0$, set $\sigma_W = \perp$. Else, set $\sigma_W = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, W)$
   (b) Store (W, b) in list of already requested preliminary warrants
   (c) Report message (b, σ_W) from J to LE
3. Upon receiving the message $(\widetilde{W}, \{\sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}\}_{i \in \{1,\ldots,v\}})$ from LE to J
   (a) Build preliminary warrant W'' out of $\widetilde{W}$ by deleting ct_i from each $\widetilde{W_i}$
   (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 $(\widetilde{W}, \cdot)$ is already in list of stored enhanced warrants. If yes, abort in the name of J (Warrant already processed).
   (d) Parse enhanced warrant: $(\widetilde{W}_1, \ldots, \widetilde{W}_v) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i$
   (e) For i from 1 to v:
      – For uid_i, honestly simulate a call to $\mathcal{F}_{\mathrm{BB}}$ in the name of J with (Retrieve, uid_i) and get answer (Retrieve, $pid_{\mathrm{U,i}}, uid_i, \mathsf{vk}_{\mathrm{U,i}}$)
      – Check signatures: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, (ct_i, uid_i, vper_i), \sigma_{\mathrm{U,i}}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, \mathsf{com}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}) = 0$, abort in the name of J
      – Check commitment: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, sec_{1,i}) = 0$, abort in the name of J

(f) Sign enhanced warrant: $\sigma_{\widetilde{W}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, \widetilde{W})$
(g) Store $(\widetilde{W}, \sigma_{\widetilde{W}})$ in list of stored enhanced warrants
(h) Report message $(\sigma_{\widetilde{W}})$ from J to LE

4. When the adversary allows $\mathcal{F}_{\mathrm{AD}}$ to deliver output in $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption
   (a) Finish simulation of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption (like described above)
   (b) Send (Ok) to $\mathcal{F}_{\mathrm{AS}}$
   (c) Receive output (WarrantAnswer, b) from $\mathcal{F}_{\mathrm{AS}}$

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

1. Upon $\mathcal{F}_{\mathrm{AS}}$ asking to deliver output to J, allow the output
2. Upon receiving the leak (RequestWarrant, W, b) from $\mathcal{F}_{\mathrm{AS}}$
   (a) If $b = 0$, set $\sigma_W = \perp$. Else, set $\sigma_W = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, W)$
   (b) Store (W, b) 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, σ_W) from LE to SO
3. Upon receiving the message $(\widetilde{W}, \{\sigma_{\mathrm{U,i}}, sec_{1,i}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}\}_{i\in\{1,\ldots,v\}})$ from SO to LE
   (a) Parse enhanced warrant: $(\widetilde{W}_1, \ldots, \widetilde{W}_v) \leftarrow \widetilde{W}$ and $(uid_i, vper_i, meta_i, ct_i) \leftarrow \widetilde{W}_i$
   (b) Build preliminary warrant W' out of $\widetilde{W}$ by deleting ct_i from each $\widetilde{W}_i$
   (c) If $W' \neq W$, abort in the name of LE. (SO sent wrong $\widetilde{W}$)
   (d) For i from 1 to v:
      i. For uid_i, honestly simulate a call to $\mathcal{F}_{\mathrm{BB}}$ in the name of LE with (Retrieve, uid_i) and get answer (Retrieve, $pid_{\mathrm{U,i}}, uid_i, \mathsf{vk}_{\mathrm{U,i}}$)
      ii. Check signatures: If $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, (ct_i, uid_i, vper_i), \sigma_{\mathrm{U,i}}) = 0$ or $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{\mathrm{U,i}}, \mathsf{com}_{sec_1,\mathrm{i}}, \sigma_{sec_1,\mathrm{i}}) = 0$, abort in the name of LE
      iii. Check commitment: If $\mathsf{COM.Open}(crs_{\mathrm{com}}, \mathsf{com}_{sec_1,\mathrm{i}}, \mathsf{decom}_{sec_1,\mathrm{i}}, sec_{1,i}) = 0$, abort in the name of LE
      iv. For uid_i, honestly simulate a call to $\mathcal{F}_{\mathrm{BB}}$ in the name of J with (Retrieve, uid_i)
   (e) Sign enhanced warrant: $\sigma_{\widetilde{W}} = \Sigma.\mathsf{Sign}(\mathsf{sk}_{\mathrm{J}}, \widetilde{W})$
   (f) Store $(\widetilde{W}, \sigma_{\widetilde{W}})$ in list of stored enhanced warrants
   (g) Simulate call to $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption by sending (Request, $pid_{\mathrm{LE}}, f_t(W), |\widetilde{W}|, v$) to $\mathcal{A}$
4. When the adversary allows $\mathcal{F}_{\mathrm{AD}}$ to deliver output in $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption
   (a) Store $\widetilde{W}$ in list of warrants $\mathsf{L}_{\mathsf{Warrants}}$ of $\mathcal{F}_{\mathrm{AD}}$
   (b) Send (Ok) to $\mathcal{F}_{\mathrm{AS}}$
   (c) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output to LE

Get Secrets (honest LE):

1. Upon receiving the leak (GetSecrets) from $\mathcal{F}_{\mathrm{AS}}$
   (a) Leak (Retrieve) to $\mathcal{A}$
   (b) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output

Get Secrets (corrupted LE):
This is handled by simulation of $\mathcal{F}_{\mathrm{AD}}$.

Get Statistics (honest P):

1. Upon receiving the leak (GetStatistics) from $\mathcal{F}_{\mathrm{AS}}$
   (a) Leak (GetStatistics) to $\mathcal{A}$ and wait for okay from $\mathcal{A}$
   (b) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output

Get Statistics (corrupted P):
This is handled by simulation of $\mathcal{F}_{\mathrm{AD}}$.

Audit (honest AU):

1. Upon receiving the leak (AuditRequest) from $\mathcal{F}_{\mathrm{AS}}$
   (a) Leak (AuditRequest) to $\mathcal{A}$ and wait for okay from $\mathcal{A}$
   (b) Allow $\mathcal{F}_{\mathrm{AS}}$ to deliver output

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

Prove (honest U)

1. Upon receiving the message $(\textsc{Prove}, \mathit{stmt}_R)$ from $\mathcal{F}_{\mathrm{AS}}$
   (a) Simulate proof: $\pi \leftarrow \mathsf{NIZK.Sim}(td_{\mathrm{NIZK}}, \mathit{stmt}_R, \mathcal{R}_{zk}^{AS})$
   (b) Send $(\textsc{Proof}, \pi)$ to $\mathcal{F}_{\mathrm{AS}}$

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

Verify (honest P)

1. Upon receiving the message $(\textsc{Verify}, \mathit{stmt}_R, \pi)$ from $\mathcal{F}_{\mathrm{AS}}$
   (a) Extract Witness: $\mathit{wit}_R \leftarrow \mathsf{NIZK.Ext}(td_{\mathrm{NIZK}}, \mathit{stmt}_R, \pi, \mathcal{R}_{zk}^{AS})$
   (b) Send $(\textsc{witness}, \mathit{wit}_R)$ to $\mathcal{F}_{\mathrm{AS}}$

We now prove the following theorem:

Theorem 6 (User Security). *Π_{AS} UC-realizes $\mathcal{F}_{\mathrm{AS}}$ in the $\{\mathcal{F}_{\mathrm{AD}}, \mathcal{F}_{\mathrm{CRS}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{G}_{\mathrm{CLOCK}}\}$-hybrid model under the assumptions that*

- COM *is a (computationally) hiding, (statistically) binding and equivocal commitment scheme*
- *Σ 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 $\mathcal{A}$ 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 $\mathcal{F}_{\mathrm{AS}}^{i+1}$ behaves the same as $\mathcal{F}_{\mathrm{AS}}^{i}$ except for the stated changes and $\mathcal{S}^{i+1}$ behaves the same as $\mathcal{S}^{i}$ except for the stated changes.

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

$$H_0 := \mathsf{Exp}^{\Pi_{\mathrm{AS}}}_{\mathcal{F}_{\mathrm{AD}},\mathcal{F}_{\mathrm{CRS}},\mathcal{F}_{\mathrm{BB}},\mathcal{G}_{\mathrm{CLOCK}},\mathcal{S}^0,\mathcal{Z}}$$

with $\mathcal{S}^0$ being the dummy adversary. This means that all parties execute the real protocol.

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

Proof Sketch: Game 0 $\approx$ Game 1.
$\mathcal{S}^1$ executes the same code as the real parties on the same inputs (that have been forwarded by $\mathcal{F}_{\mathrm{AS}}^{1}$). Thus, Game 0 and Game 1 are identical from the environments view.

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

Proof Sketch: Game 1 $\approx$ Game 2.
Indistinguishability follows from the equivocality of COM and the simulation-extractability of NIZK.

Game 3. During setup of $\mathcal{F}_{\mathrm{AD}}$, $\mathcal{S}^3$ is asked for the keypairs for J and AU. $\mathcal{S}^3$ generates the keypairs honestly and stores them.

Proof Sketch: Game 2 $\approx$ Game 3.
$\mathcal{S}^3$ runs the same key generation algorithm as the real protocol does for honest parties, therefore $\mathcal{Z}$'s view remains unchanged.

Game 4. $\mathcal{S}^4$ keeps track of additional information. For J it stores a list of (W, b) elements and a list of $(\widetilde{W}, \sigma_{\widetilde{W}})$ elements in the Request Warrant task. For each honest U, it stores $(pid_{\mathrm{U}}, uid, \mathsf{com}_{\mathrm{uid}}, \mathsf{decom}_{\mathrm{uid}}, \sigma_{\mathrm{reg}}, \mathsf{vk}_{\mathrm{U}}, \mathsf{sk}_{\mathrm{U}})$ in the

list $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Registered}}$ at the end of the User Registration task and $(uid, vper, sec_1, ct)$ in the list $\mathsf{L}^{\mathsf{user-sec}}_{\mathsf{Secrets}}$ at the end of the Store Secret task.

Proof Sketch: Game 3 $\approx$ Game 4.
$\mathcal{S}^4$ only keeps track of more information (that the honest parties acquire during the real protocol), nothing to distinguish for $\mathcal{Z}$.

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

Proof Sketch: Game 4 $\approx$ Game 5.
$\mathcal{S}^5$ simulates the System Init task honestly, therefore $\mathcal{Z}$s view remains unchanged.
Party Init for honest Parties: $\mathcal{Z}$s view remains identical, since $\mathcal{S}^5$ only needs to leak (GetSKeys) and (GetPK) to $\mathcal{A}$ to emulate the call to $\mathcal{F}_{\mathrm{AD}}$.
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, $\mathcal{F}^6_{\mathrm{AS}}$ additionally handles the Audit task the same way $\mathcal{F}_{\mathrm{AS}}$ does, and $\mathcal{S}^6$ handles this task the same way $\mathcal{S}$ does. Note that since the auditor AU is assumed to be honest, simulation only requires a simulated (AuditRequest) to $\mathcal{F}_{\mathrm{BCRA}}$.

Proof Sketch: Game 5 $\approx$ Game 6.
Since AU is honest, these games are identical.

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

Proof Sketch: Game 6 $\approx$ Game 7.
For honest U, $\mathcal{S}^7$ learns the *uid* the environment has chosen for the user and can thus invoke $\mathcal{F}_{\mathrm{AS}}$ with the correct input. Apart from that, $\mathcal{S}^7$ plays the role of the user honestly. If the signature scheme Σ is EUF-CMA-secure and the commitment scheme COM is binding, $\mathcal{S}^7$ aborts in the same cases as U would do in Π_{AS}.
For corrupted U, this game is identical to Game 6 since this task is already handled by the simulation of $\mathcal{F}_{\mathrm{AD}}$.

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

Proof Sketch: Game 7 $\approx$ Game 8.
$\mathcal{S}^8$ essentially simulates the real protocol for honest parties (only the call to $\mathcal{F}_{\mathrm{AD}}$ is simulated instead of executed honestly), therefore $\mathcal{Z}$ can not distinguish those games. The simulated call to $\mathcal{F}_{\mathrm{AD}}$ is indistinguishable since $\mathcal{S}^8$ leaks the correct information to the adversary. Note that if the signature scheme Σ is EUF-CMA-secure and the commitment scheme COM is binding, $\mathcal{S}^8$ aborts in the same cases as LE or J would do in Π_{AS}.

Game 9. If LE is corrupted, $\mathcal{F}^9_{\mathrm{AS}}$ additionally handles the Request Warrant task the same way $\mathcal{F}_{\mathrm{AS}}$ does for corrupted LE, and $\mathcal{S}^9$ handles this task the same way $\mathcal{S}$ does for corrupted LE. Additionally, $\mathcal{S}^9$ simulates $\mathcal{F}_{\mathrm{AD}}$ as $\mathcal{S}$ does for the Request Decryption and Handover tasks in $\mathcal{F}_{\mathrm{AD}}$. Note that if LE is honest, this

game changes nothing. Also note that now $\mathcal{S}^9$ simulates $\mathcal{F}_{\mathrm{AD}}$ as $\mathcal{S}$ does (for every task and every corruption scenario).

Proof Sketch: Game 8 ≈ Game 9.
In Request Warrant, $\mathcal{S}^9$ receives the necessary information from LE to call $\mathcal{F}_{\mathrm{AS}}^9$ with the correct input. After receiving output from $\mathcal{F}_{\mathrm{AS}}^9$, $\mathcal{S}^9$ essentially executes the real protocol for J and SO. Now, the simulation of $\mathcal{F}_{\mathrm{AD}}$.RequestDecryption needs to be handled as well. Since Σ is EUF-CMA-secure, $\mathcal{S}^9$ aborts in the same cases as the real $\mathcal{F}_{\mathrm{AD}}$ would. $\mathcal{S}^9$ can call $\mathcal{F}_{\mathrm{AS}}^9$ 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 $\mathsf{L}_{\mathsf{Requests}}$ of $\mathcal{F}_{\mathrm{AD}}$ like an honest $\mathcal{F}_{\mathrm{AD}}$. The remainder of the secret retrieval (for corrupted users) is then handled by the honest simulation of $\mathcal{F}_{\mathrm{AD}}$. For honest users, we retrieve the entry $(uid, vper, sec_1, ct)$ from $\mathsf{L}_{\mathsf{Secrets}}^{\mathsf{user-sec}}$ and use the output of $\mathcal{F}_{\mathrm{AS}}^9$ to calculate the secret sec_2 (in the clear) that needs to be retrieved. We then store this secret directly to a special list $\mathsf{L}_{\mathsf{ReqNotReady}}^{\mathsf{user-sec}}$. Then, the $\mathcal{F}_{\mathrm{AD}}$.Handover task has to be adapted as well. It now additionally moves all entries from $\mathsf{L}_{\mathsf{ReqNotReady}}^{\mathsf{user-sec}}$ to $\mathsf{L}_{\mathsf{Requests}}^{\mathsf{Pending}}$. 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, $\mathcal{F}_{\mathrm{AS}}^{10}$ additionally handles the tasks Get Secrets and Get Statistics the same way $\mathcal{F}_{\mathrm{AS}}$ does, and $\mathcal{S}^{10}$ handles these tasks the same way $\mathcal{S}$ does.

Proof Sketch: Game 9 ≈ 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 $\mathcal{F}_{\mathrm{AD}}$. For honest parties, $\mathcal{S}^{10}$ only needs to leak the same messages to the adversary as $\mathcal{F}_{\mathrm{AD}}$ would do. All the information $\mathcal{S}^{10}$ needs for that are provided by leaks from $\mathcal{F}_{\mathrm{AS}}$.

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

Proof Sketch: Game 10 ≈ Game 11.
For corrupted U: nothing to do in this case.
For honest U: With the message (StoreSecret, $vper$) $\mathcal{S}^{11}$ is able to invoke $\mathcal{F}_{\mathrm{AS}}^{11}$ with the correct inputs for SO. Instead of computing real commitments on sec_2 and $secret$, $\mathcal{S}^{11}$ computes equivocal commitments on them. Since COM is hiding and equivocable, $\mathcal{Z}$ can not differentiate between the real and the simulated commitments. Also, $\mathcal{S}^{11}$ simulates the proof π_{ss} instead of computing a real proof for honest users. Due to the zero-knowledge property of NIZK, $\mathcal{Z}$ can not detect this. $\mathcal{S}^{11}$ also creates a fake ciphertext (by encrypting 0) which is not detected by $\mathcal{Z}$ since TPKE is IND-CPA-secure and we do not use this ciphertext during retrieval anymore. Apart from these changes, $\mathcal{S}^{11}$ simulates the role of the user honestly.

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

Proof Sketch: Game 11 ≈ Game 12.
For corrupted parties invoking this task, there is nothing to do for $\mathcal{S}^{12}$, since both tasks are *local tasks*. So let us assume the calling parties are honest in the following. Since NIZK is a simulation extractable zero-knowledge proof system, $\mathcal{Z}$ can not distinguish between the outputs it gets from $\mathcal{F}_{\mathrm{AS}}^{12}$ and the outputs it gets from $\mathcal{F}_{\mathrm{AS}}$. Note that in the $\mathcal{F}_{\mathrm{AS}}$.Verify task, it may be the case that no corresponding entry exists in $\mathsf{L}_{\mathsf{Secrets}}$. But since SO is corrupted, the additional check in $\mathcal{F}_{\mathrm{AS}}$.Verify that involves $\mathsf{L}_{\mathsf{Secrets}}$ is skipped and the correct output is given.

Note that $\mathcal{S}^{12}$ equals the final simulator $\mathcal{S}$. 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. □

F Security Proof: Π_{AD} UC/YOSO-realizes $\mathcal{F}_{\mathrm{AD}}$

Here we prove that the protocol Π_{AD} from Appendix B.4 UC-realizes the functionality $\mathcal{F}_{\mathrm{AD}}$ from Appendix B.3 in the YOSO model. In particular, we prove the following theorem:

Theorem 2. *If* NIZK *is a straight-line simulation-extractable non-interactive zero-knowledge proof system, Σ 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 $\mathcal{F}_{\mathrm{BCRA}}$ is a RIND-SO secure public key encryption scheme, and* TPKE *is an IND-CPA secure, randomizable and binding (t, n)-threshold PKE, then Π_{AD} UC-realizes $\mathcal{F}_{\mathrm{AD}}$ in the $\{\mathcal{F}_{\mathrm{BCRA}}, \mathcal{F}_{\mathrm{CRS}}, \mathcal{G}_{\mathrm{CLOCK}}\}$-hybrid model with respect to adversaries $\mathcal{A}$ 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 Π_{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 $\mathcal{S}$ and then proceed to prove indistinguishability between the real and ideal worlds. Again, we use the convention that when $\mathcal{F}_{\mathrm{AD}}$ 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 $\mathcal{S}$ is defined as follows.

Simulator $\mathcal{S}$
State of the Simulator

- Setup
  - SO verification key $\mathsf{vk}_{\mathrm{SO}}$
  - J verification key vk_{J}
  - AU encryption keypair $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}})$
  - Functionality public key pk
  - Common reference string crs_{zk}^{SS}
  - Trapdoor td for extracting and simulating zero-knowledge proofs
- Lists to keep track of simulation
  - L_{C} to store (C, nym) pairs (where $C \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}, nym)$)
  - $\mathsf{L}_{\mathsf{Secrets}}$ to store shares of retrieved secrets
  - ${\mathsf{L}_{\mathsf{CommitteePK}}}^{i}$: List of public keys of the committee with number i. Entries are of the form (ek_j).
  - ${\mathsf{L}_{\mathsf{Committee}}}^{i}$: 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 $\mathcal{F}_{\mathrm{BCRA}}$:
  - Set Posted, initialized to $\emptyset$
  - Sequence Ordered, initialized to $()$

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

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

Simulation of $\mathcal{F}_{\mathrm{BCRA}}$

- Upon receiving (Post), (Order) or (Read) from corrupted P simulate $\mathcal{F}_{\mathrm{BCRA}}$ honestly
- Upon receiving (NextCommittee, R, n, S) with $|S| < \epsilon n$ 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 committte. If ${\mathsf{L}_{\mathsf{Committee}}}^{j} \neq \emptyset$, ignore this message.
  2. Otherwise, send (Clock-Read) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive t_{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 ${\mathsf{L}_{\mathsf{Committee}}}^{j}$
       (c) Set $m = $ (NextCommittee, $R, \mathsf{ek}_i, \mathsf{vk}_i$), add (`roleassign`, t_{Now}, m) to Posted and send m to the adversary on behalf of $\mathcal{F}_{\mathrm{BCRA}}$

- For $i \in (|S|+1,\dots,n)$:
  (a) Sample $\left(\mathsf{ek}_{R_i},\mathsf{dk}_{R_i}\right) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
  (b) Sample $\left(\mathsf{vk}_{R_i},\mathsf{sk}_{R_i}\right) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  (c) Add $(R_i,(\mathsf{ek}_{R_i},\mathsf{dk}_{R_i},\mathsf{vk}_{R_i},\mathsf{sk}_{R_i}))$ to ${\mathsf{L}_{\mathsf{Committee}}}^j$
  (d) Set $m = \left(\textsc{NextCommittee}, R, \mathsf{ek}_{R_i}, \mathsf{vk}_{R_i}\right)$, add (`roleassign`, t_{Now}, m) to Posted and send m to the adversary on behalf of $\mathcal{F}_{\text{BCRA}}$

Init (honest SO)

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

Init (corrupted SO)

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

Init (honest J)

– Upon receiving $(\textsc{Init}, \text{J}, \mathsf{vk}_{\text{J}})$ from $\mathcal{F}_{\text{AD}}$
  1. Send (Clock-Read) to $\mathcal{G}_{\text{CLOCK}}$ and receive t_{Now}
  2. Set $m \coloneqq (\text{JudgeKey}, \mathsf{vk}_{\text{J}})$
  3. Add $(pid_{\text{J}}, t_{\text{Now}}, m)$ to Posted and send $(\textsc{Post}, pid_{\text{J}}, m)$ 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 $\mathcal{F}_{\mathrm{AD}}$
  1. Send (CLOCK-READ) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive t_{Now}
  2. Generate $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$ and store $(\mathsf{ek}_{\mathrm{AU}}, \mathsf{sk}_{\mathrm{AU}})$
  3. Set $m := (\mathrm{AuditorKey}, \mathsf{ek}_{\mathrm{AU}})$
  4. Add $(pid_{\mathrm{AU}}, t_{\mathrm{Now}}, m)$ to POSTED and send $(\mathrm{POST}, pid_{\mathrm{AU}}, m)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathrm{BCRA}}$
  5. Allow $\mathcal{F}_{\mathrm{AD}}$ to deliver output

Get Public Key (honest P)

- Upon receiving (GETPK) from $\mathcal{F}_{\mathrm{AD}}$
  1. Send (READ) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathrm{BCRA}}$
  2. Allow $\mathcal{F}_{\mathrm{AD}}$ to deliver output

Get Public Key (corrupted P)

- This is handled by simulation of key generation roles and $\mathcal{F}_{\mathrm{BCRA}}$

Get System Keys (honest P)

- Upon receiving (GETSKEYS) from $\mathcal{F}_{\mathrm{AD}}$
  1. Send (READ) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathrm{BCRA}}$
  2. Allow $\mathcal{F}_{\mathrm{AD}}$ to deliver output

Get System Keys (corrupted P)

- This is handled by simulation of $\mathcal{F}_{\mathrm{BCRA}}$

Request Decryption (honest LE)

- Upon receiving $(\mathrm{REQUEST}, pid, W^{\mathrm{pub}}, len, v)$ from $\mathcal{F}_{\mathrm{AD}}$
  1. Send (CLOCK-READ) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive t_{Now}
  2. If $\mathsf{sk}_{\mathrm{LE}} = \perp$ set $(\mathsf{ek}_{\mathrm{LE}}, \mathsf{sk}_{\mathrm{LE}}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
  3. Send (READ) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathrm{BCRA}}$
  4. Generate $W^{\mathrm{enc}} \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{\mathrm{AU}}, 0^{len})$
  5. For $i = 1, \ldots, v$ do:
     (a) $ct_i \leftarrow \mathsf{TPKE.Enc}(\mathsf{pk}, 0)$
  6. $stmt_{\widetilde{W}} := (\mathsf{ek}_{\mathrm{AU}}, \mathsf{vk}_{\mathrm{J}}, W^{\mathrm{enc}}, W^{\mathrm{pub}}, \{ct_i\}_{i\in[v]})$
  7. $\pi_{\widetilde{W}} \leftarrow \mathsf{NIZK.Sim}(crs^{SS}_{\mathrm{zk}}, stmt_{\widetilde{W}}, td)$
  8. Set $msg := (\mathrm{REQUEST}, W^{\mathrm{pub}}, W^{\mathrm{enc}}, \{ct_i\}_{i\in[v]}, \pi_{\widetilde{W}}, \mathsf{ek}_{\mathrm{LE}})$, add $(pid, t_{\mathrm{Now}}, msg)$ to POSTED and send $(\mathrm{POST}, pid, msg)$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathrm{BCRA}}$
  9. Allow $\mathcal{F}_{\mathrm{AD}}$ to deliver output

Request Decryption (corrupted LE)

- Upon receiving $\left(\text{POST}, (\text{REQUEST}, W^{\text{pub}}, W^{\text{enc}}, \{ct_i\}_{i\in[v]}, \pi_{\widetilde{W}}, \mathsf{ek}_{\text{LE}})\right)$ from LE to $\mathcal{F}_{\text{BCRA}}$
  1. $stmt_{\widetilde{W}} := (\mathsf{ek}_{\text{AU}}, \mathsf{vk}_{\text{J}}, W^{\text{enc}}, W^{\text{pub}}, \{ct_i\}_{i\in[v]})$
  2. If $\mathsf{NIZK.Verify}(crs^{SS}_{\text{zk}}, stmt, \pi_{\widetilde{W}}) \neq 1$ ignore this message
  3. Otherwise, extract $wit_{\widetilde{W}} := (\widetilde{W}, r, \sigma_{\widetilde{W}}, \{ct_i, r_i\}_{i\in[v]}) \leftarrow \mathsf{NIZK.Ext}(crs, \pi_{\widetilde{W}}, td)$
  4. Send $\left(\text{REQUEST}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$ to $\mathcal{F}_{\text{AD}}$ on behalf of LE

Retrieve Secret (honest LE)

- Upon receiving (RETRIEVE) 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

Retrieve Secret (corrupted LE)

- This is handled during simulation of handover and $\mathcal{F}_{\text{BCRA}}$

Get Statistics (honest P)

- Upon receiving (GETSTATISTICS) 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 Statistics (corrupted P)

- This is handled by simulation of $\mathcal{F}_{\text{BCRA}}$

Auditor Audit (honest AU)

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

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

- All roles that are already corrupted are assigned to $\mathcal{B}$
- $t - |\mathcal{B}|$ random roles that are not corrupted are assigned to $\mathcal{SH}$
- The remaining roles are assigned to $\mathcal{H}$

It holds that $\mathcal{SH} \bigcap \mathcal{B} = \emptyset$ and $\mathcal{H} \bigcap (\mathcal{SH} \bigcup \mathcal{B}) = \emptyset$ and $\mathcal{H} \bigcup \mathcal{SH} \bigcup \mathcal{B}$ contains all roles of that committee. Whenever the adversary corrupts a party that is assigned a role in $\mathcal{H}$ and has not executed yet, the simulator picks a random role $R_i \in \mathcal{SH}$ instead, sets $\mathcal{SH} := \mathcal{SH} \setminus \{R_i\}$, $\mathcal{B} := \mathcal{B} \bigcup \{R_i\}$ 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 in $\mathcal{SH}$, the simulator sets $\mathcal{SH} := \mathcal{SH} \setminus \{R\}$, $\mathcal{B} := \mathcal{B} \bigcup \{R\}$ and returns the state of that role.

- Upon receiving (ExecuteRole, pid_N) from $\mathcal{F}_{\mathrm{AD}}$
  1. Send (Read) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathrm{BCRA}}$
  2. Send (Clock-Read) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive t_{Now}
  3. If $t_{\mathsf{last}} + t_{\mathsf{com}} \leq t_{\mathrm{Now}}$ execute **PrepareExecution**
  4. If this node is scheduled to execute a role R_i this round, execute the respective task **KeyGen1**, **KeyGen2** or **Handover** below.
- **PrepareExecution**
  1. Set $t_{\mathsf{last}} = t_{\mathrm{Now}}$
  2. Initialize empty lists $\mathsf{L}_{\mathsf{CommitteePK}}$ and $\mathsf{L}_{\mathsf{CommitteeVK}}$
  3. For each role R_j in the next committee
     (a) Find $(\texttt{roleassign}, t, (\mathrm{Generate}, R_j, \mathsf{ek}_R, \mathsf{vk}_R))$ in Ordered
     (b) Insert ek_R into $\mathsf{L}^{\mathsf{next}}_{\mathsf{CommitteePK}}$
  4. For each role R_j in the current committee
     (a) Find $(\texttt{roleassign}, t, (\mathrm{Generate}, R_j, \mathsf{ek}_R, \mathsf{vk}_R))$ in Ordered
     (b) Insert ek_R into $\mathsf{L}_{\mathsf{CommitteePK}}$
  5. For each role R_j in the previous two committees
     (a) Find $(\texttt{roleassign}, t, (\mathrm{Generate}, R_j, \mathsf{ek}_R, \mathsf{vk}_R))$ in Ordered
     (b) Insert vk_R into $\mathsf{L}_{\mathsf{CommitteeVK}}$
  6. Assign the roles of the current committee to $\mathcal{H}, \mathcal{SH}, \mathcal{B}$ as described above
  7. If KeyGen1 has been executed, but KeyGen2 not, do the following:
     (a) For each entry $(pid, t, (R_j, (\mathrm{KeyGen1}, \Big(\{ct_j\}_{j\in[n]}, \pi_{KG}\Big)), \sigma))$ in Ordered
        i. Retrieve vk_{R_j} from $\mathsf{L}_{\mathsf{CommitteeVK}}$
        ii. Check if $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, (\mathrm{KeyGen1}, \Big(\{ct_j\}_{j\in[n]}, \pi_{KG}\Big)), \sigma) = 1$, otherwise skip this entry.
        iii. Assemble ZK-Statement: $stmt_{KG} := (\Big\{ct_j, \mathsf{ek}_{R_j}\Big\}_{j\in[n]})$
        iv. If $\mathsf{NIZK.Verify}(crs^{SS}_{\mathsf{zk}}, stmt_{KG}, \pi_{KG}, \mathcal{R}^{AD}_{KG1}) \neq 1$ skip this entry
        v. Add ct_i to $\mathsf{L}_{\mathsf{Qual}}$ (the one addressed to the currently executing role)

(b) Sort $\mathsf{L}_{\mathsf{Qual}}$ lexicographically

(c) For each of the first $t+1$ entries in $\mathsf{L}_{\mathsf{Qual}}$ 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 $R_j \in \mathcal{SH} \bigcup \mathcal{B}$ compute $\mathsf{sk}_j := \sum_{k=1}^{t+1} G_k(j)$ and $\mathsf{pk}_j := \mathsf{TPKE.Sk2Pk}(\mathsf{sk}_j)$

iii. For each role $R_j \in \mathcal{H}$ set $\mathsf{pk}_j := \mathsf{pk}^{\lambda_{j,0}} \cdot \prod_{k \in \mathcal{SH} \bigcup \mathcal{B}} \mathsf{pk}_k^{\lambda_{j,k}}$, where $\lambda_{i,j}$ are appropriate Lagrange coefficients.

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

(a) For each entry $(pid, t, (R_j, msg, \sigma))$ with $msg := (\text{KeyShare}, (\mathsf{vk}_i, \{ct_j\}, \pi_{KG}))$ for the **previous** committee in Ordered

i. Retrieve vk_{R_j} from $\mathsf{L}_{\mathsf{CommitteeVK}}$

ii. Check $\Sigma.\mathsf{Vfy}(\mathsf{vk}_R, msg, \sigma) = 1$, otherwise ignore this message

iii. Assemble ZK-Statement:
$stmt_{KS} := (\mathsf{vk}_i, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j \in [n]})$

iv. Check $\mathsf{NIZK.Verify}(crs_{\mathrm{zk}}^{SS}, stmt_{KS}, \pi_{KS}, \mathcal{R}_{KS}^{AD}) = 1$, otherwise ignore this message

v. Add $\{ct_j\}_{j \in [n]}$ to $\mathsf{L}_{\mathsf{Qual}}^{ct}$

(b) Sort $\mathsf{L}_{\mathsf{Qual}}^{ct}$ lexicographically

(c) For each entry $(pid, t, (R_j, msg, \sigma))$ with $msg := (\text{KeyShare}, (\mathsf{vk}_i, \{ct_k\}, \pi_{KS}))$ for the **current** committee in Ordered:

i. Retrieve vk_{R_j} from $\mathsf{L}_{\mathsf{CommitteeVK}}$

ii. Check if $\Sigma.\mathsf{Vfy}(\mathsf{vk}_{R_j}, (\text{KeyShare}, msg, \sigma) = 1$, otherwise skip this entry.

iii. Retrieve from $\mathsf{L}_{\mathsf{Qual}}^{ct}$ from the first $t+1$ entries the ciphertext ct_j for R_j each, as $\mathsf{L}_{\mathsf{KCom}} := \{ct_j^1, \ldots, ct_j^n\}$

iv. Assemble ZK-Statement:
$stmt_{KS} := (\mathsf{pk}_j, \{ct_k\}_{k \in [n]}, \mathsf{L}_{\mathsf{KCom}})$

v. Check if $\mathsf{NIZK.Verify}(crs_{\mathrm{zk}}^{SS}, stmt_{KS}, \pi_{KS}, \mathcal{R}_{KS}^{AD}) = 1$, otherwise skip this entry.

vi. Add ct_i (the one addressed to the currently executing role) to $\mathsf{L}_{\mathsf{Qual}}$

(d) Sort $\mathsf{L}_{\mathsf{Qual}}$ lexicographically

(e) For each of the first $t+1$ entries in $\mathsf{L}_{\mathsf{Qual}}$ 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 $R_j \in \mathcal{SH} \bigcup \mathcal{B}$ compute $\mathsf{sk}_j := \sum_{k=1}^{t+1} G_k(j)$ and $\mathsf{vk}_j := \mathsf{TPKE.Sk2Pk}(\mathsf{sk}_j)$

iii. For each role $R_j \in \mathcal{H}$ set $\mathsf{vk}_j := \mathsf{pk}^{\lambda_{j,0}} \cdot \prod_{k \in \mathcal{SH} \bigcup \mathcal{B}} \mathsf{pk}_k^{\lambda_{j,k}}$, where $\lambda_{i,j}$ are appropriate Lagrange coefficients.

(f) For each entry $(pid, t, (\textsc{Request}, W^{\text{pub}}, W^{\text{enc}}, \{\widehat{ct}_i\}_{i\in[v]}, \pi_{\widetilde{W}}, \mathsf{ek}_{\text{LE}}))$ in $\textsc{Ordered}$
   i. Assemble ZK-Statement:
      $stmt_{\widetilde{W}} := (\mathsf{ek}_{\text{AU}}, \mathsf{vk}_{\text{J}}, W^{\text{enc}}, W^{\text{pub}}, \{\widehat{ct}_i\}_{i\in[v]})$
   ii. If $\mathsf{NIZK.Verify}(crs^{SS}_{\text{zk}}, stmt_{\widetilde{W}}, \pi_{\widetilde{W}}, \mathcal{R}^{AD}_{W}) \neq 1$, ignore this message
   iii. For each $i \in [v]$:
      A. Store $(\mathsf{ek}_{\text{LE}}, ct_i, \perp)$ in $\mathsf{L}_{\mathsf{Requests}}$

(g) If LE is corrupted:
   i. Send $(\textsc{Retrieve})$ to $\mathcal{F}_{\text{AD}}$ and receive $\left(\textsc{Retrieve}, \mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}\right)$
   ii. For each entry $(\mathsf{ek}_{\text{LE}}, ct_i, \perp)$ in $\mathsf{L}_{\mathsf{Requests}}$:
      A. Find $(ct_i, secret_i)$ in $\mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}$
      B. For each role R_j in $\mathcal{SH} \bigcup \mathcal{B}$, compute $\widehat{ct}^{j}_{i} \leftarrow \mathsf{TPKE.TDec}(\mathsf{sk}_j, ct_i)$, where sk_j was computed earlier
      C. Run $\left\{\widehat{ct}^{k}_{i}\right\}_{k\in\mathcal{H}} \leftarrow \mathsf{TPKE.SimTDec}(\mathsf{pk}, ct_i, secret_i, \left\{\widehat{ct}^{j}_{i}\right\}_{i\in(\mathcal{SH}\bigcup\mathcal{B})})$
      D. Update the entry in $\mathsf{L}_{\mathsf{Requests}}$ to $(\mathsf{ek}_{\text{LE}}, ct_i, \left\{\widehat{ct}^{k}_{i}\right\}_{k\in\mathcal{H}})$

– **KeyGen1**
1. Retrieve the respective $\mathsf{dk}_R, \mathsf{sk}_R$
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 $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, sh_j; r_j)$
5. Assemble ZK-Witness: $wit_{KG} := (s_i, \{r_j\}_{j\in[n]}, F)$
6. Assemble ZK-Statement:
   $stmt_{KG} := (\left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]})$
7. Compute Proof:
   $\pi_{KG} \leftarrow \mathsf{NIZK.Prove}(crs^{SS}_{\text{zk}}, stmt_{KG}, wit_{KG}, \mathcal{R}^{AD}_{KG1})$
8. Prepare message: $msg := (\textsc{KeyGen1}, \left(\{ct_j\}_{j\in[n]}, \pi_{KG}\right))$
9. Sign message: $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_R, msg)$
10. Add $(pid, t_{\text{Now}}, (R_i, msg, \sigma))$ to $\textsc{Posted}$ and send $(\textsc{Post}, pid, (R_i, msg, \sigma))$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$

– **KeyGen2**
1. If $R_i \in \mathcal{SH}$ execute this role honestly
2. Otherwise $(R_i \in \mathcal{H})$ 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 $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, sh_j; r_j)$

(c) Retrieve pk_i computed during preparation

(d) Assemble ZK-Statement:
$stmt_{KG} \coloneqq (\mathsf{pk}_i, \{ct^k\}_{k\in[t+1]}, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]}, \mathsf{ek}_R)$

(e) Simulate Proof:
$\pi_{KG} \leftarrow \mathsf{NIZK.Sim}(crs^{SS}_{\text{zk}}, stmt_{KG}, td, \mathcal{R}^{AD}_{KG2})$

(f) Prepare message: $msg \coloneqq (\textsc{KeyGen2}, \left(\mathsf{pk}_i, \{ct_j\}_{j\in[n]}, \pi_{KG}\right))$

(g) Sign message: $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_R, msg)$

(h) Add $(pid, t_{\text{Now}}, (R_i, msg, \sigma))$ to Posted and send (Post, $pid, (R_i, msg, \sigma)$) to $\mathcal{A}$ on behalf of $\mathcal{F}_{\text{BCRA}}$

– **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 $\mathsf{L}^{ct}_{\mathsf{Qual}}$ as $\left\{ct^k_i\right\}_{k\in[t+1]}$ 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 $ct_j \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{R_j}, F(j); r_j)$

iv. Assemble ZK-Statement:
$stmt_{KS} \coloneqq (\mathsf{ek}_R, \left\{ct^k_i\right\}_{k\in[t+1]}, \left\{ct_j, \mathsf{ek}_{R_j}\right\}_{j\in[n]}, \mathsf{vk}_i)$

v. Simulate Proof:
$\pi_{KS} \leftarrow \mathsf{NIZK.Sim}(crs^{SS}_{\text{zk}}, stmt_{KS}, td, \mathcal{R}^{AD}_{KS})$

vi. Add $msg \coloneqq (\textsc{KeyShare}, \left(\mathsf{vk}_i, \{ct_j\}_{j\in[n]}, \pi_{KS}\right))$ to L_M

(b) For each entry $(\mathsf{ek}_{\text{LE}}, ct, \mathsf{L}_{\mathsf{Decryptions}})$ in $\mathsf{L}_{\mathsf{Requests}}$,

- If LE is honest:

  i. Partially decrypt ct to $ct^* \leftarrow \mathsf{TPKE.TDec}(\mathsf{sk}_i, ct)$

  ii. Assemble ZK-Statement:
  $stmt_{Dec} \coloneqq (\mathsf{vk}_i, ct, ct^*)$

  iii. Simulate Proof:
  $\pi_{Dec} \leftarrow \mathsf{NIZK.Sim}(crs^{SS}_{\text{zk}}, stmt_{Dec}, td, \mathcal{R}^{AD}_{Dec})$

  iv. Generate response ciphertext: $msg \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{\text{LE}}, (0^{|ct|}, 0^{|ct^*|}, 0^{|\pi_{Dec}|}))$

  v. Add (Request, msg) to L_M

- If LE is corrupted:

  i. Retrieve ct^* for this role from $\mathsf{L}_{\mathsf{Decryptions}}$

  ii. Assemble ZK-Statement:
  $stmt_{Dec} \coloneqq (\mathsf{vk}_i, ct, ct^*)$

  iii. Simulate Proof:
  $\pi_{Dec} \leftarrow \mathsf{NIZK.Sim}(crs^{SS}_{\text{zk}}, stmt_{Dec}, td, \mathcal{R}^{AD}_{Dec})$

  iv. Encrypt answer: $msg \leftarrow \mathsf{PKE.Enc}(\mathsf{ek}_{\text{LE}}, (ct, ct^*, \pi_{Dec}))$

  v. Add (Request, msg) to L_M

(c) For each entry (msg) in $\mathsf{L_M}$
    i. Generate signature $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_R, msg)$
    ii. Update $msg \coloneqq (R_i, msg, \sigma)$
(d) Remove the entry for this role from $\mathsf{L_{Roles}}$
(e) Delete all state except for $\mathsf{L_M}$
(f) For each entry (msg) in $\mathsf{L_M}$
    i. Send $(\textsc{Post}, msg)$ to $\mathcal{F}_{\text{BCRA}}$

RoleExecute (corrupted N)

– This is handled by simulation of $\mathcal{F}_{\text{BCRA}}$

To prove Theorem 2, we proceed in a series of games, where $\mathcal{F}_{\text{AD}}^{i+1}$ behaves the same as $\mathcal{F}_{\text{AD}}^{i}$ except for the stated changes and $\mathcal{S}^{i+1}$ behaves the same as $\mathcal{S}^{i}$ except for the stated changes.

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

$$H_0 \coloneqq \mathsf{Exp}^{\Pi_{\text{AD}}}_{\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{BCRA}}, \mathcal{G}_{\text{CLOCK}}, \mathcal{S}^0, \mathcal{Z}}$$

with $\mathcal{S}^0$ being the dummy adversary. This means that all parties execute the real protocol.

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

Proof Sketch: Game 0 ≈ Game 1.
$\mathcal{S}^1$ executes the same code as the real parties on the same inputs (that have been forwarded by $\mathcal{F}_{\text{AD}}^1$). Thus, Game 0 and Game 1 are identical from the environments view.

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

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

Game 3. In this game, $\mathcal{F}_{\text{AD}}^3$ now handles $(\textsc{Init}, \ldots)$ messages the same way $\mathcal{F}_{\text{AD}}$ does, except that during SO init it also sends the generated secret key sk to $\mathcal{S}^3$, and $\mathcal{S}^3$ handles this task the same way $\mathcal{S}$ does, except that it also stores the received secret key sk. Note that $\mathcal{S}^3$ still chooses the outputs for (GetPK) and can thus still return the public encryption key pk' generated during the protocol.

Proof Sketch: Game 2 ≈ Game 3.
If SO is honest, $\mathcal{S}^3$ receives the message $(\textsc{Init}, \text{SO}, \mathsf{vk}_{\text{SO}}, \mathsf{pk})$ from $\mathcal{F}_{\text{AD}}^3$ and can thus perfectly simulate the real protocol for this task. If SO is corrupted, $\mathcal{S}^3$ can wait for a message $(\textsc{Post}, \text{OperatorKey}, \mathsf{vk}_{\text{SO}})$ from SO to $\mathcal{F}_{\text{BCRA}}$ and then

send ($\textsc{Init}, \mathrm{SO}, \mathsf{vk}_{\mathrm{SO}}$) to $\mathcal{F}^3_{\mathrm{AD}}$ on behalf of SO. Since both J and AU are assumed to be honest, $\mathcal{S}^3$ receives the respective messages from $\mathcal{F}^3_{\mathrm{AD}}$ and can perfectly simulate the real protocol for the respective task.

Game 4. In this game, $\mathcal{F}^4_{\mathrm{AD}}$ now handles ($\textsc{GetSKeys}$) the same way $\mathcal{F}_{\mathrm{AD}}$ does, and $\mathcal{S}^4$ handles this task the same way $\mathcal{S}$ does.

Proof Sketch: Game 3 $\approx$ Game 4.
For corrupted parties, this is perfectly simulated by executing $\mathcal{F}_{\mathrm{BCRA}}$. For honest parties, $\mathcal{F}^4_{\mathrm{AD}}$ outputs the keys if they are available via $\mathcal{F}_{\mathrm{BCRA}}$. Thus, these games are statistically indistinguishable.

Game 5. In this game, after simulation of $\mathcal{F}_{\mathrm{BCRA}}$ resulted in all secret keys for the next committee being delivered, $\mathcal{S}^5$ fills the three sets $\mathcal{H}, \mathcal{B}, \mathcal{SH}$ as follows:

- All roles that are already corrupted are assigned to $\mathcal{B}$
- $t - |\mathcal{B}|$ random roles that are not corrupted are assigned to $\mathcal{SH}$
- The remaining roles are assigned to $\mathcal{H}$

It holds that $\mathcal{SH} \bigcap \mathcal{B} = \emptyset$ and $\mathcal{H} \bigcap (\mathcal{SH} \bigcup \mathcal{B}) = \emptyset$ and $\mathcal{H} \bigcup \mathcal{SH} \bigcup \mathcal{B}$ 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 $\mathcal{H}$ and has not executed yet, the simulator picks a random role $R_i \in \mathcal{SH}$ instead, sets $\mathcal{SH} := \mathcal{SH} \setminus \{R_i\}$, $\mathcal{B} := \mathcal{B} \bigcup \{R_i\}$ 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 in $\mathcal{SH}$, the simulator sets $\mathcal{SH} := \mathcal{SH} \setminus \{R\}$, $\mathcal{B} := \mathcal{B} \bigcup \{R\}$ 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, $\mathcal{S}^7$ performs the steps listed under **PrepareExecution** in $\mathcal{S}$, except that in step 8g) instead of querying $\mathcal{F}_{\mathrm{AD}}$ 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 $\mathcal{H}$, $\mathcal{S}^9$ uses $\widehat{ct}$ computed during **PrepareExecution** instead of executing $\mathsf{TPKE.TDec}$. (This corresponds to step 2b), LE corrupted in $\mathcal{S}$)

Proof Sketch: Game 8 ≈ Game 9.
If LE is honest, these two games are the same. Otherwise, $\widehat{ct}$ has been computed to ensure that $\mathsf{TPKE.Combine}$ 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 $\mathcal{H}$, $\mathcal{S}^{10}$ encrypts (the appropriate amount of) zeroes instead. (This corresponds to step 2b), LE honest in $\mathcal{S}$)

Proof Sketch: Game 9 ≈ 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, $\mathcal{S}^{11}$ simulates the distributed key generation protocol to output pk (received from $\mathcal{F}_{\text{AD}}^{11}$ during SO Init) as resulting public encryption key. This is achieved by following the steps listed under **KeyGen2** and **Handover** in $\mathcal{S}$. Additionally, sk (received from $\mathcal{F}_{\text{AD}}^{11}$ during SO Init) is used during **PrepareExecution** to decrypt ciphertexts. $\mathcal{F}_{\text{AD}}^{11}$ now handles (ExecuteRole) messages the same way $\mathcal{F}_{\text{AD}}$ does.

Proof Sketch: Game 10 ≈ 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, $\mathcal{F}_{\text{AD}}^{12}$ now handles (GetPK) the same way $\mathcal{F}_{\text{AD}}$ does, and $\mathcal{S}^{12}$ handles this task the same way $\mathcal{S}$ does.

Proof Sketch: Game 11 ≈ Game 12.
Since the previous games already ensured that the same public key pk is used in both $\mathcal{F}_{\text{AD}}^{12}$ and the protocol simulation, these two games are perfectly indistinguishable.

Game 13. In this game, when receiving $\left(\text{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$ messages, $\mathcal{F}_{\text{AD}}^{13}$ checks if the signature is valid. If the signature is valid, it forwards $\left(\text{Request}, \widetilde{W}\right)$ to $\mathcal{S}^{13}$ (but not the signature $\sigma_{\widetilde{W}}$!), otherwise it ignores the message. $\mathcal{S}^{13}$ follows the protocol honestly except for the NIZK proof, where it uses the simulation trapdoor to generate $\pi_{\widetilde{W}} \leftarrow \mathsf{NIZK.Sim}(crs_{\text{zk}}^{SS}, stmt_{\widetilde{W}}, td, \mathcal{R}_{W}^{AD})$ without knowing the witness.

Proof Sketch: Game 12 ≈ 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 (Post, (Request, W^{pub}, W^{enc}, $\{ct_i\}_{i\in[v]}$, $\pi_{\widetilde{W}}, \mathsf{ek}_{\text{LE}}$)) from LE to $\mathcal{F}_{\text{BCRA}}$, if the zero-knowledge proof verifies, $\mathcal{S}^{14}$ uses the trapdoor td to extract a witness and aborts the simulation of this fails.

Proof Sketch: Game 13 ≈ 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 ($\textsc{Post}$, ($\textsc{Request}$, W^{pub}, W^{enc}, $\{ct_i\}_{i\in[v]}$, $\pi_{\widetilde{W}}, \mathsf{ek}_{\text{LE}}$)) from LE to $\mathcal{F}_{\text{BCRA}}$, $\mathcal{S}^{15}$ sends $\left(\textsc{Request}, \widetilde{W}, \sigma_{\widetilde{W}}\right)$ to $\mathcal{F}_{\text{AD}}^{15}$ in the name of LE. $\mathcal{F}_{\text{AD}}^{15}$ now fills the lists $\mathsf{L}_{\mathsf{Warrants}}, \mathsf{L}_{\mathsf{Requests}}$ the same way $\mathcal{F}_{\text{AD}}$ does, and as a consequence also fills the lists $\mathsf{L}_{\mathsf{Requests}}^{\mathsf{Pending}}$ and $\mathsf{L}_{\mathsf{Requests}}^{\mathsf{Ready}}$ during Handover.

Proof Sketch: Game 14 ≈ Game 15.
This only affects the internal view of $\mathcal{F}_{\text{AD}}^{15}$, but no output depends on that view (yet). Thus, these games are perfectly indistinguishable.

Game 16. In this game, $\mathcal{F}_{\text{AD}}^{16}$ handles ($\textsc{Retrieve}$) messages the same way as $\mathcal{F}_{\text{AD}}$, and $\mathcal{S}^{16}$ handles this task the same way as $\mathcal{S}$.

Proof Sketch: Game 15 ≈ Game 16.
If LE is corrupted, this changes nothing. If LE is honest, the list $\mathsf{L}_{\mathsf{Requests}}^{\mathsf{Ready}}$ has been correctly filled in Game 15. Since by assumption, at least $t+1$ roles in each committee are not corrupted, every ciphertext that $\mathcal{F}_{\text{AD}}$ 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 $\mathcal{F}_{\text{AD}}^{17}$ now stores $\widetilde{W}$ in $\mathsf{L}_{\mathsf{Warrants}}$ when $\mathcal{S}^{17}$ instructs it to deliver output while handling ($\textsc{Request}$) messages.

Proof Sketch: Game 16 ≈ Game 17.
This change is only syntactical.

Game 18. In this game, $\mathcal{F}_{\text{AD}}^{18}$ additionally handles ($\textsc{AuditRequest}$) messages the same way $\mathcal{F}_{\text{AD}}$ does, and $\mathcal{S}^{18}$ handles this task the same way $\mathcal{S}$ does. Note that since the auditor AU is assumed to be honest, simulation only requires a simulated ($\textsc{Read}$) to $\mathcal{F}_{\text{BCRA}}$.

Proof Sketch: Game 17 ≈ Game 18.
If LE is honest, these games are identical, since $\mathcal{S}^{17}$ generated entries in $\textsc{Ordered}$ with valid proofs for exactly the warrants that $\mathcal{F}_{\text{AD}}^{18}$ has in $\mathsf{L}_{\mathsf{Warrants}}$, 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 $\textsc{Ordered}$ for which the zero-knowledge proof is valid. For exactly those warrants, $\mathcal{S}^{18}$ sends ($\textsc{Request}, \ldots$) messages to $\mathcal{F}_{\text{AD}}^{18}$, resulting in them being in $\mathsf{L}_{\mathsf{Warrants}}$. 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 $\mathcal{S}^{19}$ computes $W^{\text{enc}} \leftarrow \mathsf{PKE}(\mathsf{ek}_{\text{AU}}, 0^{|\widetilde{W}|})$ instead of encrypting $\widetilde{W}$.

Proof Sketch: Game 18 ≈ 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, $\mathcal{F}_{\text{AD}}^{20}$ no longer sends the secret decryption key sk to $\mathcal{S}^{20}$ during Init. Instead of using sk to decrypt ciphertexts during **PrepareExecute** handling, $\mathcal{S}^{20}$ now queries $\mathcal{F}_{\text{AD}}^{20}$ for the decryption as described in $\mathcal{S}$.

Proof Sketch: Game 19 ≈ 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 $(\mathsf{ek}_{\mathrm{LE}}, ct_i, \perp)$ in $\mathsf{L}_{\mathsf{Requests}}$ for which no matching entry $(ct_i, secret_i)$ exists in $\mathsf{L}^{\mathsf{Pending}}_{\mathsf{Requests}}$ returned by $\mathcal{F}^{20}_{\mathrm{AD}}$. This can not be the case: Only ciphertexts ct_i corresponding to valid zero-knowledge proofs are inserted into $\mathsf{L}_{\mathsf{Requests}}$. But for each valid zero-knowledge proof, the extracted warrant (and thus all listed ciphertexts) was send as input to $\mathcal{F}^{20}_{\mathrm{AD}}$, and thus a valid decryption is returned. Therefore, these games are indistinguishable.

Game 21. In this game, $\mathcal{F}^{21}_{\mathrm{AD}}$ handles (Request) messages the same way as $\mathcal{F}_{\mathrm{AD}}$, and $\mathcal{S}^{21}$ handles this task the same way as $\mathcal{S}$.

Proof Sketch: Game 20 ≈ Game 21.
For corrupted LE, this changes nothing. For honest LE, $\mathcal{S}^{21}$ no longer receives $\widetilde{W}$ but only $f_t(W), |\widetilde{W}|, v$, i.e. the output of the transparency function, the length of the warrant and the number of listed ciphertexts. $\widetilde{W}$ 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 $\mathcal{F}^{21}_{\mathrm{AD}}$, 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 $\mathcal{S}^{21}$ since Game 10, except for parties in $\mathcal{SH}$, for which it is ensured that there are at most t parties in $\mathcal{SH} \bigcup \mathcal{B}$. Thus, indistinguishability follows from IND-CPA security of TPKE.

Game 22. In this game, $\mathcal{F}^{22}_{\mathrm{AD}}$ additionally handles (GetStatistics) messages the same way $\mathcal{F}_{\mathrm{AD}}$ does, and $\mathcal{S}^{22}$ handles this task the same way $\mathcal{S}$ does.

Proof Sketch: Game 21 ≈ 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 $\mathcal{F}^{18}_{\mathrm{AD}}$

Note that Game 22 is the same as the ideal world. Since we showed that Game i is indistinguishable from Game $i+1$ for $i \in \{0, \dots, 21\}$, it follows that Game 0 (the real protocol) is indistinguishable from Game 22 (ideal execution) and thus Theorem 2 follows. □

G Security Proof: Π_{ASTE} UC-realizes $\mathcal{F}_{\mathrm{ASTE}}$

In this section, we prove that the protocol Π_{ASTE} from Appendix C.3 UC-realizes the functionality $\mathcal{F}_{\mathrm{ASTE}}$ from Appendix C.2.

G.1 Simulator

In this section, we describe the inner workings of the simulator for Π_{ASTE}. Apart from message sending and law enforcement access, most other tasks are trivial to simulate. They are basically taken care of by $\mathcal{F}_{\mathrm{AS}}$, due to Π_{ASTE} working

the $\mathcal{F}_{\mathrm{AS}}$-hybrid model, and the simulator merely implements the bookkeeping of $\mathcal{F}_{\mathrm{AS}}$. 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 $\mathcal{F}_{\mathrm{AS}}$ together with $\mathsf{PKE_{AS}}$ 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 $\mathsf{PKE_{AS}}$. We implicitly use that the simulator can program $\mathcal{F}_{\mathrm{RO}}$ and simulate and extract NIZK proofs of $\mathcal{F}_{\mathrm{AS}}$.

Notation We use the notation introduced for $\mathsf{PKE_{AS}}$, see Remark 7. Like $\mathcal{F}_{\mathrm{ASTE}}$ (and Π_{ASTE}), $\mathcal{S}$ internally uses lists, e.g. $\mathsf{L_{uvper}}$, $\mathsf{L_{msgs}}$, L_W, $\mathsf{L_{W\text{-}cached}}$, and variables $ureg_i$, usk_i, uid_i, and so on. To distinguish the variables of $\mathcal{F}_{\mathrm{ASTE}}$ and $\mathcal{S}$, if not clear from the context, we write $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{uvper}}$ for $\mathcal{F}_{\mathrm{ASTE}}$. Variables without extra specification (e.g. $\mathsf{L_{uvper}}$) belong to the simulator.

High level simulation strategy The simulator keeps its "shadow copies" or lists and variables of $\mathcal{F}_{\mathrm{ASTE}}$ in perfect synchronization with those of $\mathcal{F}_{\mathrm{ASTE}}$, to the extent possible. Clearly, $\mathsf{L_{msgs}}$ cannot equal $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$, since $\mathcal{S}$ cannot know all messages or *uid*s in $\mathsf{L_{msgs}}$; 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 *uid*s to ciphertexts become known.

Most simulation steps are quite straightforward, thanks to $\mathcal{F}_{\mathrm{AS}}$ which provides a powerful and easy-to-use basis. The steps where simulation requires most care (besides ensuring synchronization of variables with $\mathcal{F}_{\mathrm{ASTE}}$) are the tasks involving $\mathsf{PKE_{AS}}$, 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 $\mathsf{PKE_{AS}.Sim}$, i.e. these ciphertexts have no message m or *uid* associated with them (yet), and thus in $\mathsf{L_{msgs}}$ the respective components are set to $\perp$ (whereas in $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$, m and *uid* are filled in). In case m or *uid* is leaked to $\mathcal{S}$ (because the receiver is corrupted or U_i is under surveillance), $\mathcal{S}$ runs $\mathsf{PKE_{AS}.ExpIn_{RE}}$ or $\mathsf{PKE_{AS}.ExpIn_{LE}}$ to ensure consistency under decryption, and also updates the entry in $\mathsf{L_{msgs}}$. When an honest user decrypts a ciphertext c which is part of $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$, $\mathcal{S}$ learns nothing (and does nothing). If $(c, \cdot, \cdot, \cdot) \notin \mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$, then $\mathcal{F}_{\mathrm{ASTE}}$ requests $(\textsc{SimInjectMsg}, c)$ of $\mathcal{S}$. To respond, $\mathcal{S}$ extract the NIZK to learn (m, uid_i) from c.

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 *uid*s are explained (via $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExplIn}_{\mathsf{LE}}$). The simulation of **execute access** for (honest) LE need only answer **inject message** queries for ciphertexts not in $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$. Again, the contents of the ciphertext are obtained by extracting the NIZK consistency proof.

The detailed description of $\mathcal{S}$ 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 $\mathcal{F}_{\mathrm{AS}}$ 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 $\mathsf{NIZK}_{\mathsf{AS}}$, programming $\mathcal{F}_{\mathrm{RO}}$) and exploiting their extractability (i.e. $\mathcal{S}$ learns every query to a hybrid functionality). We are explicit about any deviations from honest emulation.

Trivially simulatable tasks In all these tasks, $\mathcal{S}$ does the necessary bookkeeping so all lists remain synchronized with $\mathcal{F}_{\mathrm{ASTE}}$. Otherwise, simulation is trivial.

Party Init The simulation is trivial, since this task is identical to **party init** in $\mathcal{F}_{\mathrm{AS}}$ and the protocol merely proxies to $\mathcal{F}_{\mathrm{AS}}$.

Request Warrant The simulation is trivial, since this task is identical to **request warrant** in $\mathcal{F}_{\mathrm{AS}}$ and the protocol merely proxies to $\mathcal{F}_{\mathrm{AS}}$.

Get statistics The simulation is trivial, since this task is identical to **get statistics** in $\mathcal{F}_{\mathrm{AS}}$ and the protocol merely proxies to $\mathcal{F}_{\mathrm{AS}}$.

Audit The simulation is trivial, since this task is identical to **audit** in $\mathcal{F}_{\mathrm{AS}}$ and the protocol merely proxies to $\mathcal{F}_{\mathrm{AS}}$.

Next Period For honest SO, the simulator does nothing. In fact, $\mathcal{F}_{\mathrm{ASTE}}$ (and $\mathcal{G}_{\mathrm{CLOCK}}$) 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 $\mathcal{F}_{\mathrm{ASTE}}$ whenever SO intends to send (Clock-Update) to $\mathcal{G}_{\mathrm{CLOCK}}$ and SO is currently registered with $\mathcal{G}_{\mathrm{CLOCK}}$. (Note that the registration state can be checked by sending (Is-Registered, sid_C) to $\mathcal{G}_{\mathrm{CLOCK}}$.)

Init Upon receiving $(\textsc{Init}, P)$ for $P \in \{\mathrm{J}, \mathrm{AU}\}$ from $\mathcal{F}_{\mathrm{ASTE}}$, $\mathcal{S}$ executes the honest protocol for P (i.e. sends $(\textsc{Init})$ to $\mathcal{F}_{\mathrm{AS}}$ and forwards the output, once it is delivered). Since J and AU are assumed uncorruptable, this completes their simulation.

For honest SO: Upon receiving $(\textsc{Init}, \mathrm{SO})$, the simulator executes the honest protocol for SO.

For corrupted SO: When $\mathcal{S}$ receives $(\textsc{Init}, \mathrm{SO})$ in place of $\mathcal{F}_{\mathrm{AS}}$ from SO, it sends $(\textsc{Init}, \mathrm{SO})$ to $\mathcal{F}_{\mathrm{ASTE}}$ on behalf of SO. When the simulated $\mathcal{F}_{\mathrm{AS}}$ would deliver output $(\textsc{InitFinished})$ to AU (resp. J), $\mathcal{S}$ allows the output of $(\textsc{InitFinished})$ for $\mathcal{F}_{\mathrm{ASTE}}$ to AU (resp. J).

User Registration We consider the corruption cases separately. Note that an honest user first writes to $\mathcal{F}_{\mathrm{BB}}$ and then checks if the write to $\mathcal{F}_{\mathrm{BB}}$ was successful. This ensures that it only continues the protocol if $\mathcal{F}_{\mathrm{BB}}$ contains the correct entry.Also note that the leaks to the dummy adversary (as part of simulating $\mathcal{F}_{\mathrm{AS}}$) 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 $(\textsc{Register}, \mathrm{U}_i)$ from $\mathcal{F}_{\mathrm{ASTE}}$, the simulator essentially honestly executes the protocol on behalf of both parties, with following additional steps:

1. Let $(\mathsf{pk}_i, \mathsf{sk}_i)$ be the keypair generated for U_i. When $\mathcal{S}$ on behalf of $\mathcal{F}_{\mathrm{BB}}$ emulates a response $(\textsc{Retrieve}, \mathrm{U}_i, \mathit{uid}_i, \mathit{uid}_i)$ to U_i, i.e. once U_i knows it has secured its unique uid_i on $\mathcal{F}_{\mathrm{BB}}$, $\mathcal{S}$ sends $(\textsc{Register}, \mathrm{U}_i, \mathit{uid}_i)$ to $\mathcal{F}_{\mathrm{ASTE}}$ (i.e. registers U_i with that uid_i, as uid_i is now locked).
2. *Sim* also emulates $\mathcal{F}_{\mathrm{AS}}$ honestly. (The necessary inputs and leakage are trivially known to $\mathcal{S}$.)
3. When a simulated party (i.e. U_i or SO) generates output in the protocol, $\mathcal{S}$ allows $\mathcal{F}_{\mathrm{ASTE}}$ to deliver this party's output. When $\mathcal{S}$ allows $\mathcal{F}_{\mathrm{ASTE}}$ to deliver output to U_i, it sets $\mathit{ureg}_i := 1$.

Case 2: Honest U_i and corrupted SO Upon receiving $(\textsc{Register}, \mathrm{U}_i)$ from $\mathcal{F}_{\mathrm{ASTE}}$, 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 $\mathcal{S}$ (in place of $\mathcal{F}_{\mathrm{AS}}$) receives $(\textsc{Register}, \mathit{uid})$ from SO, it sends $(\textsc{Register})$ to $\mathcal{F}_{\mathrm{ASTE}}$ on behalf of SO.

Case 3: Corrupted U_i and honest SO Upon receiving $(\textsc{Ready})$ from U_i, the simulator essentially honestly executes the protocol on behalf of SO, with following additional step: After simulating a response $(\textsc{Retrieve}, \mathrm{U}_i, \mathit{uid}, \mathit{uid}')$ of $\mathcal{F}_{\mathrm{BB}}$ for the query made by SO in this protocol run, if $\mathit{uid}' = \mathit{uid}$, $\mathcal{S}$ sends $(\textsc{Register}, \mathrm{U}_i, \mathit{uid})$ to $\mathcal{F}_{\mathrm{ASTE}}$. (At this point, uid_i for U_i is locked.)

[30] Since SO is corrupted (and of its output has no effect in $\mathcal{F}_{\mathrm{ASTE}}$), we can immediately allow it or delay it forever without affecting the simulation.

All other steps are simulated as in the actual protocol. Once $\mathcal{S}$ (in place of $\mathcal{F}_{\text{AS}}$) delivers output (REGISTERED) to U_i, set $ureg_i := 1$. As in step 3 above, once SO would generate (accepting) output, $\mathcal{S}$ allows $\mathcal{F}_{\text{ASTE}}$ 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 $uvper_i$ 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, $\mathcal{S}$ also keeps track of (a simulated) $uvper_i$ for corrupted users.

Note that for honest users, **update** sends $(\text{UPDATE}, \text{U}_i, vper)$ to $\mathcal{S}$ and uid_i is known (through $\mathcal{F}_{\text{BB}}$), thus, $\mathcal{S}$ can simulate the leaks to the environment in **store secret** of $\mathcal{F}_{\text{AS}}$ trivially.

Case 1: Honest U and honest SO Upon receiving leakage $(\text{UPDATE}, \text{U}_i, vper)$, the simulator acts as follows:

- $\mathcal{S}$ acts for U_i resp. SO exactly as in the protocol.
- When U_i (resp. SO) send $(\text{STORESECRET}, uid_i, uvper_i)$ (resp. $(\text{STORESECRET}, uvper_i)$) to $\mathcal{F}_{\text{AS}}$ (whichever sends first), send (OK) to $\mathcal{F}_{\text{ASTE}}$.
- $\mathcal{S}$ runs the remaining protocol and allows $\mathcal{F}_{\text{ASTE}}$ to deliver output once a simulated party generates output. Note that, when delivering output for U_i, $\mathcal{S}$ also sets $uvper_i := vper$, where $vper$ is the agreed upon period.
- As noted above, the leakage $(\text{STORESECRET}, vper)$ from $\mathcal{F}_{\text{AS}}$ to the dummy adversary is trivial to honestly simulate.

Case 2: Honest U and corrupted SO Upon receiving leakage $(\text{UPDATE}, \text{U}_i, vper)$, the simulator honestly executes the protocol on behalf of U_i.[31] Note again that, when $\mathcal{S}$ would generate protocol output for U_i, it allows $\mathcal{F}_{\text{ASTE}}$ to deliver output and sets $uvper_i := vper$, where $vper$ is the agreed upon period. (Observe that $\mathcal{S}$ implements $\mathcal{F}_{\text{AS}}$, and thus learns the inputs of SO to $\mathcal{F}_{\text{AS}}$ (if any). Thus simulating possible aborts is straightforward.)

Case 3: Corrupted U and honest SO Upon receiving $(\text{UPDATE}, uid, vper)$ by U_i on behalf of SO, the simulator honestly executes the protocol in the name of SO. Suppose SO calls (the simulated) $\mathcal{F}_{\text{AS}}$ with $(\text{STORESECRET}, vper)$. Once output $(\text{SECRETSTORED}, \cdot)$ is delivered to U_i by $\mathcal{S}$ in the name of $\mathcal{F}_{\text{AS}}$. Then set $uvper_i := vper$.

Case 4: Corrupted U and corrupted SO There is nothing to do for the simulator.

[31] Note that the (dummy) adversary now receives $(\text{STORESECRET}, uvper_i)$ from the simulated $\mathcal{F}_{\text{AS}}$. But as noted, $\mathcal{S}$ can trivially simulate the leakage.

Encrypt Message For encryption, we only need to consider honest users. When an honest user U_i invokes $\mathcal{F}_{\mathrm{ASTE}}$ with $(\textsc{Encrypt}, \mathrm{U}_j, m)$, then $\mathcal{S}$ receives leakage $(\textsc{Encrypt}, \mathrm{U}_j, (|m|, uvper), m', uid')$ from $\mathcal{F}_{\mathrm{ASTE}}$. (Recall that $\mathsf{leak}(m, uid, uvper) = (|m|, uvper)$ in Theorem 3, and that $m' = \bot$ if the receiver U_j is honest and $m' = m$ otherwise.) Then $\mathcal{S}$ 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 $(\textsc{Retrieve}, \ldots)$ of $\mathcal{F}_{\mathrm{BB}}$.)
2. Simulate the ciphertext c by running $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Sim}(crs, \mathsf{pk}_j, uvper_i)$.
3. If $m' \neq \bot$, then U_j is corrupted and $\mathcal{S}$ immediately runs $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExplIn}_{\mathsf{RE}}(crs, \mathsf{pk}_j, c, m, uvper_i)$ on the simulated ciphertext. (In this case $m' = m$, i.e. $\mathcal{S}$ learns m.)
4. If $uid' \neq \bot$, then $(uid_i, uvper_i) \in \mathsf{L}_{\mathsf{W\text{-}cached}}$ and $\mathcal{S}$ runs $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExplIn}_{\mathsf{LE}}(crs, usk_i, c, uid_i, uvper_i)$ immediately on the simulated ciphertext. (In this case, $uid' = uid_i$, i.e. $\mathcal{S}$ learns U_i)
5. Store $(c, m', uid', uvper_i)$ in $\mathsf{L}_{\mathsf{msgs}}$.
6. Send $(\textsc{Ciphertext}, \mathrm{U}_j, c)$ to $\mathcal{F}_{\mathrm{ASTE}}$.

Extract-and-inject ciphertext c In the tasks **decrypt ciphertext**, **prepare access** and **execute access**, the ideal functionality $\mathcal{F}_{\mathrm{ASTE}}$ may send an urgent query $(\textsc{SimInjectMsg}, c)$ or $(\textsc{SimAccessInject}, c)$ to the adversary to make sense of some ciphertext which is not (yet) contained in $\mathsf{L}_{\mathsf{msgs}}$. The simulator handles the query as follows:

1. If the NIZK proof π of consistency of c is not accepting, then $\mathcal{S}$ sends $(\textsc{InjectMsg}, c, \bot, \bot, \bot)$ to $\mathcal{F}_{\mathrm{ASTE}}$ (and skips the rest).
2. $\mathcal{S}$ extracts the NIZK proof π_{con} (from c) to obtain (m, uid). Since the NIZK of $\mathcal{F}_{\mathrm{ASTE}}$ is ideal, extraction always succeeds.
3. Add $(c, m, uid, uvper)$ to $\mathsf{L}_{\mathsf{msgs}}$.
4. Send $(\textsc{InjectMsg}, c, m, uid, uvper)$ to $\mathcal{F}_{\mathrm{ASTE}}$.

Decrypt Ciphertext For decryption, we only need to consider honest users. Moreover, if $(c, \cdot, \cdot, \cdot) \in \mathsf{L}_{\mathsf{msgs}}$, then $\mathcal{S}$ learns nothing about the call, as it is local and immediately finishes. (The simulation — doing nothing — is perfect.) Thus, $\mathcal{S}$ only handles the case where $(c, \cdot, \cdot, \cdot) \notin \mathsf{L}_{\mathsf{msgs}}$. Namely, when $\mathcal{S}$ receives $(\textsc{SimInjectMsg}, c)$, then $\mathcal{S}$ acts as follows:

1. Let $m' = \mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}(crs, \mathsf{sk}_j, c, vper)$.
2. If $m' = \bot$, send $(\textsc{InjectMsg}, \bot, \bot, \bot)$ to $\mathcal{F}_{\mathrm{ASTE}}$.
3. Else, run **extract-and-inject ciphertext** c (as defined above).

For future reference, note that whenever $\mathcal{S}$ adds $(c, m, uid_i, uvper_i)$ to $\mathsf{L}_{\mathsf{msgs}}$, it is ensured that $\mathcal{F}_{\mathrm{ASTE}}$ adds it to $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$ as well. If SO is corrupted, this is trivial. If SO is honest, $(uid, uvper) \in \mathsf{L}_{\mathsf{uvper}}$ is checked during the **inject message** task in $\mathcal{F}_{\mathrm{ASTE}}$, and this check will succeed, since for $(uid, vper) \notin \mathsf{L}_{\mathsf{uvper}}$ it is impossible to generate a (fake) NIZK proof (and thus ciphertext c associated with fake $(uid, vper)$).

Prepare Access We consider simulation of honest and corrupt LE separately. The main difference is, that honest LE only fills $\mathsf{L}_{\mathsf{W\text{-}cached}}$ at this step, so the simulator has not much to do. On the other, hand, a corrupted LE learns $(uid_i, vper_i, usk_i)$ tuples covered by the warrant. The simulator must now use $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}$ to fix the *uid*s in the simulated ciphertexts.

Honest LE Upon receiving (AccessLeak) from $\mathcal{F}_{\mathrm{ASTE}}$, $\mathcal{S}$ simulates Π_{ASTE} for LE by simulating $\mathcal{F}_{\mathrm{AS}}$ on input (GetSecrets, W) (for unknown $W = \{W_i\}_{i=1}^{v}$ with $W_i = (uid_i, vper_i, meta_i)$). Note that **get secrets** (only) leaks (GetSecrets) to the adversary, and then allows the adversary to delay the output. If $\mathcal{F}_{\mathrm{ASTE}}$ aborts, $\mathcal{S}$ instructs $\mathcal{F}_{\mathrm{AS}}$ to abort. Once the adversary allows to deliver output, $\mathcal{S}$ allows $\mathcal{F}_{\mathrm{ASTE}}$ to deliver output.

Corrupted LE Upon receiving (GetSecrets, W) from LE (in place of $\mathcal{F}_{\mathrm{AS}}$), $\mathcal{S}$ proceeds as follows: Let $W = \{(uid_i, vper_i, meta_i)\}_{i=1}^{v}$. Send (AccessPrep, W) to $\mathcal{F}_{\mathrm{ASTE}}$ on behalf of LE. If $\mathcal{F}_{\mathrm{ASTE}}$ aborts, $\mathcal{S}$ instructs $\mathcal{F}_{\mathrm{AS}}$ to abort. Else, receive (AccessLeak). Before sending (GotSecrets, ...) to LE, do the following:

Allow $\mathcal{F}_{\mathrm{ASTE}}$ the output of (AccessPrepDone) (to simulated LE, i.e. $\mathcal{S}$). This ensures that **execute access** now decrypts all ciphertexts affected by W. Now, $\mathcal{S}$ can run a trial-decryption process and explain all affected ciphertexts. For each $(uid_j, vper_j, \cdot) \in W$:

- If $(uid_j, vper_j) \in \mathsf{L}_{\mathsf{uvper}}$, add $(uid_j, vper_j)$ to $\mathsf{L}_{\mathsf{W\text{-}cached}}$. Else, skip to next $(uid, vper)$ pair.
- For each ciphertext with undefined *uid*, i.e. for all $(c, m, \perp, vper) \in \mathsf{L}_{\mathsf{msgs}}$, $\mathcal{S}$ sends (AccessExec, $c, (uid_i, vper_i)$) to $\mathcal{F}_{\mathrm{ASTE}}$, and immediately[32] receives (AccessExecDone, uid) from $\mathcal{F}_{\mathrm{ASTE}}$. If $uid \neq \perp$, then
  - If there is no *honest* U_i with $(\mathrm{U}_i, uid) \in \mathsf{L}_{\mathsf{U}}$ and $ureg_i = 1$ then abort the entire simulation with output `inconsistent`. Else we have $uid_i = uid$.
  - Explain c via $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}(crs, usk_i, c, uid_i, vper_i)$.
  - Update $(c, m, \perp, vper)$ to $(c, m, uid, vper)$ in $\mathsf{L}_{\mathsf{msgs}}$.

After this, $\mathcal{S}$ continues to simulate $\mathcal{F}_{\mathrm{AS}}$ honestly. Note that at this point, any simulated ciphertext which would be accessible with warrant W has its *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 (SimAccessInject, c). These are handled as follows:

1. Let $uid = \mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(crs, \mathsf{sk}_j, c, vper)$.
2. If $uid = \perp$, send (InjectMsg, $\perp, \perp, \perp$) to $\mathcal{F}_{\mathrm{ASTE}}$.
3. Else, run **extract-and-inject ciphertext** c (as defined above).

[32] There will never be a (SimAccessInject, c) request, since c was simulated, thus $(c, m, uid, vper) \in \mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$,

G.2 Simulator overview

Simulator $\mathcal{S}$

The simulator has access to the global functionality $\mathcal{G}_{\text{CLOCK}}$. It implements the hybrid functionalities $\mathcal{F}_{\text{CRS}}$, $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{AS}}$, $\mathcal{F}_{\text{RO}}$.

Simulation of $\mathcal{F}_{\text{CRS}}$ $\mathcal{S}$ simulates $\mathcal{F}_{\text{CRS}}$ honestly.

Simulation of $\mathcal{F}_{\text{BB}}$ $\mathcal{S}$ simulates $\mathcal{F}_{\text{BB}}$ honestly, but it observes (adversarial) messages.

Simulation of $\mathcal{F}_{\text{RO}}$ $\mathcal{S}$ simulates $\mathcal{F}_{\text{RO}}$ honestly, but it also programs $\mathcal{F}_{\text{RO}}$ and observes (adversarial) queries. Programmability and extractability of $\mathcal{F}_{\text{RO}}$ are used as part of the NINCE encryption subroutines, namely $\mathsf{SKE}_{\mathsf{NCE}}$ in $\mathsf{PKE}_{\mathsf{AS}}$.

Simulation of $\mathcal{F}_{\text{AS}}$ $\mathcal{S}$ simulates $\mathcal{F}_{\text{AS}}$ honestly, but observes (adversarial) messages.

States of the Parties:

- User, SO and LE: Cache crs from $\mathcal{F}_{\text{CRS}}$.
- User U_i:
  - User long-term key: $(\mathsf{pk}_i, \mathsf{sk}_i)$, where $uid_i = \mathsf{pk}_i$ is the user identify.
  - User secret (with period): $(usk_i, uvper_i)$, 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: $uid_j = \mathsf{pk}_j$ (once retrieved from $\mathcal{F}_{\text{BB}}$).
- LE: Cache $\mathsf{L}_{\mathsf{W\text{-}cached}}$ with entries $(uid_i, uvper_i, usk_i)$.
- SO, J and AU only need to interact with $\mathcal{F}_{\text{AS}}$.

Init:

- This is merely a proxy to **Init** from $\mathcal{F}_{\text{AS}}$. For concreteness, we specify this proxy simulation, but leave other proxies to the reader.

Init (honest J or AU):

1. Upon receiving $(\textsc{Init}, P)$ for $P \in \{\text{J}, \text{AU}\}$ from $\mathcal{F}_{\text{ASTE}}$, leak $(\textsc{Init}, P)$ in place of $\mathcal{F}_{\text{AS}}$ to the adversary.
2. When the adversary allows delivery of output $(\textsc{InitFinished})$ to P, allow $\mathcal{F}_{\text{ASTE}}$ delivery of $(\textsc{InitFinished})$ 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 $(\text{REGISTER}, \text{U}_i)$ from $\mathcal{F}_{\text{ASTE}}$, $\mathcal{S}$ emulates Π_{ASTE} for U_i:
  1. Send (VALUE) to $\mathcal{F}_{\text{CRS}}$ and cache the result as crs.
  2. Generate and store $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{PKE}_{\mathsf{AS}}.\mathsf{Gen}(1^\lambda)$ (unless already stored).
  3. Send $(\text{REGISTER}, uid_i, uid_i)$ to $\mathcal{F}_{\text{BB}}$.
  4. Send $(\text{RETRIEVE}, uid)$ to $\mathcal{F}_{\text{BB}}$ and abort if the (eventually delivered) response is not $(\text{RETRIEVE}, \text{U}_i, uid, uid)$.
  5. Send (READY) to SO.
  6. Send $(\text{REGISTER}, uid_i)$ to $\mathcal{F}_{\text{AS}}$.
  7. Send $(\text{RETRIEVE}, uid)$ to $\mathcal{F}_{\text{BB}}$.
  8. Upon output (REGISTERED) from $\mathcal{F}_{\text{AS}}$ to U_i, set $ureg_i := 1$. Moreover, $\mathcal{S}$ allows $\mathcal{F}_{\text{ASTE}}$ output for U_i.
– For SO, $\mathcal{S}$ emulates Π_{ASTE} as well:
  1. Upon receiving (READY) from U_i, send $(\text{RETRIEVE}, \text{U}_i)$ to $\mathcal{F}_{\text{BB}}$ and receive $(\text{RETRIEVE}, \text{U}_i, uid_i, uid'_i)$. If $uid_i \neq uid'_i$, abort.
  2. Call $\mathcal{F}_{\text{AS}}$ with input $(\text{REGISTER}, uid_i)$.
  3. Upon output $(\text{REGISTERED}, uid)$ 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 $(\text{REGISTER}, uid_i)$ 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 $(\text{REGISTERED}, uid)$ 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, $\mathcal{S}$ executes the honest simulation of SO with following change:
  1. Upon receiving (READ) from U_i, $\mathcal{S}$ sends (REGISTER) to $\mathcal{F}_{\mathrm{ASTE}}$ on behalf of U_i.
  2. $\mathcal{S}$ receives (REGISTER, U_i) from $\mathcal{F}_{\mathrm{ASTE}}$, but delays its response, until:
  3. After delivering (RETRIEVE, U_i, uid, uid') to the simulated SO on behalf of (the simulated) $\mathcal{F}_{\mathrm{BB}}$ and if $uid = uid'$, then $\mathcal{S}$ sends (REGISTER, U_i, uid') to $\mathcal{F}_{\mathrm{ASTE}}$. Moreover, $\mathcal{S}$ 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 $\mathcal{S}$.

Next Period:

- There is nothing to do for $\mathcal{S}$.

Update (Honest U_i and honest SO):

- Upon receiving (UPDATE, U_i, $vper$) from $\mathcal{F}_{\mathrm{ASTE}}$, $\mathcal{S}$ emulates Π_{ASTE} for U_i:
  1. Query (CLOCK-READ) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive (CLOCK-READ, $vper$).
  2. If $uvper = vper$, U_i output (UPDATEDONE, $vper$). Else continue.
  3. Send (UPDATE, uid_i, $vper$) to SO.
  4. Send (STORESECRET, uid_i, $vper$) to (the simulated) $\mathcal{F}_{\mathrm{AS}}$.
  5. Upon receiving (SECRETSTORED, usk'_i), and update $(usk_i, uvper_i) := (usk'_i, vper)$. Moreover, $\mathcal{S}$ allows output of (UPDATEDONE, $vper$) by SO to $\mathcal{F}_{\mathrm{ASTE}}$.
- For SO, $\mathcal{S}$ emulates Π_{ASTE} as well:
  1. Upon receiving (UPDATE, uid, $vper$) from U_i:
  2. Query (CLOCK-READ) to $\mathcal{G}_{\mathrm{CLOCK}}$ and receive (CLOCK-READ, $vper'$). If $vper' \neq vper$, abort.
  3. Send (STORESECRET, $vper$) to $\mathcal{F}_{\mathrm{AS}}$.
  4. Upon receiving (SECRETSTORED, uid_i), the simulator $\mathcal{S}$ allows output of (UPDATEDONE, $vper$) to SO by $\mathcal{F}_{\mathrm{ASTE}}$.
- $\mathcal{S}$ observes the behaviour of the $\mathcal{F}_{\mathrm{AS}}$ calls (in this session) and acts as follows:
  - When U_i (respectively SO) send (STORESECRET, uid_i, $uvper_i$) (respectively (STORESECRET, $uvper_i$)) to $\mathcal{F}_{\mathrm{AS}}$ (whichever sends first), send (OK) to $\mathcal{F}_{\mathrm{ASTE}}$.

Update (Honest U_i and corrupt SO):

– Simulation for honest U_i is unchanged.
– For corrupt SO, no additional special handling required. (Note that $\mathcal{F}_{\mathrm{ASTE}}$ still receives (OK) from the simulator $\mathcal{S}$ once (simulated) U_i or SO send (STORESECRET, $uvper_i$) for this session to $\mathcal{F}_{\mathrm{AS}}$.)

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 $\mathcal{F}_{\mathrm{ASTE}}$ still receives (OK) from the simulator $\mathcal{S}$ once U_i or (simulated) SO send (STORESECRET, $uid_i, uvper_i$) or (STORESECRET, $uvper_i$) for this session to $\mathcal{F}_{\mathrm{AS}}$.)

Update (Corrupt U_i and honest SO):

– There is nothing to do for $\mathcal{S}$.

Encrypt Message (Honest U_i):

– Upon receiving (CIPHERTEXT, $\mathrm{U}_j, \mathsf{leak}, m', uid'$), 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 $uvper_i = \bot$, abort.
  3. If pk_j is not cached, send (RETRIEVE, U_j) to (the simulated) $\mathcal{F}_{\mathrm{BB}}$ and later receive (RETRIEVE, $\mathrm{U}_j, uid_j, uid'_j$). If $uid_j \neq uid'_j$ or $uid = \bot$, abort. Else cache $\mathsf{pk}_j = uid_j$.
– Now, simulate the ciphertext and keep $\mathcal{S}.\mathsf{L}_{\mathsf{msgs}}$ up-to-date.
  4. Simulate the ciphertext c as $c \leftarrow \mathsf{PKE_{AS}.Sim}(crs, \mathsf{pk}_j, uvper_i)$.
  5. If $m' \neq \bot$, then U_j is corrupted and the simulator $\mathcal{S}$ immediately runs $\mathsf{PKE_{AS}.ExplIn_{RE}}(crs, \mathsf{pk}_j, c, m, uvper_i)$ on the simulated ciphertext. (In this case $m' = m$, i.e. $\mathcal{S}$ learns m.)
  6. If $uid' \neq \bot$, then $(uid_i, uvper_i) \in \mathsf{L}_{\mathsf{W\text{-}cached}}$ and $\mathcal{S}$ immediately runs $\mathsf{PKE_{AS}.ExplIn_{LE}}(crs, usk_i, c, uid_i, uvper_i)$ on the simulated ciphertext. (In this case, $uid' = uid_i$, i.e. $\mathcal{S}$ learns U_i)
  7. Store $(c, m', uid', uvper_i)$ in $\mathsf{L}_{\mathsf{msgs}}$.
  8. Send (CIPHERTEXT, U_j, c) to $\mathcal{F}_{\mathrm{ASTE}}$.
– Note that $\mathcal{S}$ programs $\mathcal{F}_{\mathrm{RO}}$ during the $\mathsf{PKE_{AS}}$ simulation and explanation steps.

Decrypt Ciphertext (Honest U_i):

– This is handled as part of **Simulator Injected Messages**.

Simulator Injected Messages:

- Upon receiving ($\textsc{SimInjectMsg}, c$) or ($\textsc{SimAccessInject}, c$) from $\mathcal{F}_{\text{ASTE}}$:
  1. If the NIZK proof π of consistency of c does not verify, then $\mathcal{S}$ sends ($\textsc{InjectMsg}, c, \perp, \perp, \perp$) to $\mathcal{F}_{\text{ASTE}}$ (and skips the rest).
  2. $\mathcal{S}$ verifies extracts the NIZK proof π_{con} (from c) to obtain (m, uid). Since the NIZK of $\mathcal{F}_{\text{ASTE}}$ is ideal, extraction always succeeds.
  3. Add $(c, m, \mathit{uid}, \mathit{uvper})$ to $\mathcal{S}.\mathsf{L}_{\mathsf{msgs}}$.
  4. Send ($\textsc{InjectMsg}, c, m, \mathit{uid}, \mathit{uvper}$) 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 ($\textsc{AccessLeak}$):
  1. Simulate sending ($\textsc{GetSecrets}, W$) to $\mathcal{F}_{\text{AS}}$, that is, send ($\textsc{GetSecrets}$) to the adversary on behalf of $\mathcal{F}_{\text{AS}}$.

Prepare Access (Corrupted LE):

- Upon receiving ($\textsc{GetSecrets}, W$) from LE in place of $\mathcal{F}_{\text{AS}}$:
  1. Send ($\textsc{AccessPrep}, W$) 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 ($\textsc{AccessLeak}$) from $\mathcal{F}_{\text{ASTE}}$.
  4. Allow $\mathcal{F}_{\text{ASTE}}$ the output of ($\textsc{AccessPrepDone}$) (to LE).
  5. Retrocatively explain all affected ciphertext using $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}$ 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 Σ' is a EUF-CMA-secure signature scheme,* COM *is a statistically binding and computationally hiding commitment scheme,* $\mathsf{PKE}_{\mathsf{NCE}}$ *is*

a strong NINCE PKE *scheme and IND-CCA secure and key-committing,* $\mathsf{SKE}_{\mathsf{NCE}}$ *is a strong NINCE* SKE *scheme and IND-CCA secure and key-committing. Moreover, suppose all schemes are perfectly correct. Let* $\mathsf{leak}(m, uid, uvper) = (|m|, uvper)$ *as the system parameter in* $\mathcal{F}_{\mathrm{ASTE}}$*, and let* f_p*,* f_t *be policy and transparency functions. Then* Π_{ASTE} *UC-realizes* $\mathcal{F}_{\mathrm{ASTE}}$ *in the* $\{\mathcal{F}_{\mathrm{CRS}}, \mathcal{F}_{\mathrm{BB}}, \mathcal{F}_{\mathrm{AS}}, \mathcal{G}_{\mathrm{CLOCK}}\}$*-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 $\mathcal{F}_{\mathrm{ASTE}}$ use. To refer to a variable of $\mathcal{F}_{\mathrm{ASTE}}$, we write, for example, $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$, 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. $ureg_i$ or $uvper_i$ for (honest or corrupt) user U_i; seldom, we write $\mathcal{F}_{\mathrm{AS}}.\mathsf{L}_{\mathsf{U}}$ to emphasize that the variable is part of the (simulated) hybrid functionality $\mathcal{F}_{\mathrm{AS}}$. We introduce also certain invariants, which are useful to see that $\mathcal{S}$ uses $\mathcal{F}_{\mathrm{ASTE}}$ correctly. Moreover, we will gradually ensure that lists and variables of $\mathcal{S}$ and $\mathcal{F}_{\mathrm{ASTE}}$ "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}, $\mathsf{L}_{\mathsf{uvper}}$, $\mathsf{L}_{\mathsf{msgs}}$, etc., but also the variables $ureg_i$, $uvper_i$ (and sk_i, usk_i, etc.) both for honest parties and corrupted parties. They are initialized as in $\mathcal{F}_{\mathrm{ASTE}}$. To gradually turn the real world into the simulation, we imagine access to the functionality $\mathcal{F}_{\mathrm{ASTE}}$ (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 $\mathcal{F}_{\mathrm{ASTE}}$ and $\mathcal{S}$.

This change is only conceptual.

Game 2 *(Simulate **Init**).* For **Init**, we replace $P \in \{\mathrm{AU}, \mathrm{J}\}$ by dummy parties which interact with $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathcal{F}_{\mathrm{ASTE}}$).

Since simulation and real protocol are effectively equivalent (namely, up to some protocol steps, they are just calls to $\mathcal{F}_{\mathrm{AS}}$ and $\mathcal{G}_{\mathrm{CLOCK}}$), these changes are only conceptual.

Game 3 *(Simulate **Party Init**).* We let honest parties call $\mathcal{F}_{\mathrm{ASTE}}$ 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 $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{I}}$ evidently always coincide.

Game 4 *(Simulate **user registration**).* We let honest parties call $\mathcal{F}_{\mathrm{ASTE}}$ instead of running the protocol for **user registration**, and run the simulator for this task. We track the variable $ureg_i$ exactly as in the simulation. Moreover, we link these with respective **user registration** task of the imagined $\mathcal{F}_{\mathrm{ASTE}}$. Concretely, the following 4 cases affect $ureg_i$:

- Case 1 (Honest U_i and honest SO): $\mathcal{S}$ 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): $\mathcal{S}$ 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): $\mathcal{S}$ runs the protocol for SO. It sets $ureg_i = 1$ after $\mathcal{F}_{\mathrm{AS}}$ outputs (Registered) to U_i.
- Case 4 (Corrupted U_i and SO): The corrupted parties act unchanged.

By definition of $\mathcal{S}$, we could in all cases simply execute $\mathcal{S}$ with $\mathcal{F}_{\mathrm{ASTE}}$ to simulate the real protocol. Indeed, $\mathcal{S}$ 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 $\mathcal{F}_{\mathrm{ASTE}}.ureg_i$ always coincide. The change is only conceptual.

Game 5 *(Simulate **update**).* We let honest parties call $\mathcal{F}_{\mathrm{ASTE}}$ instead of running the protocol for **update**, and run the simulator for this task. Now, we track $uvper_i$ and $\mathsf{L}_{\mathsf{uvper}}$. Moreover, we link these with respective **update** task of the imagined $\mathcal{F}_{\mathrm{ASTE}}$. As in Game 4, we consider 4 cases of corruption, which affect how the variables $uvper_i$ and $\mathsf{L}_{\mathsf{uvper}}$ are changed. Let $vper$ be the agreed upon $vper$ (which is the response of $\mathcal{G}_{\mathrm{CLOCK}}$ 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 $\mathsf{L}_{\mathsf{uvper}}$.
- 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 $\mathsf{L}_{\mathsf{uvper}}$.
- Case 3 (Corrupted U_i and honest SO): Run the protocol for SO. When $\mathcal{F}_{\mathrm{AS}}$ outputs (SecretStored, usk) to U_i, set $uvper_i = vper$ and add $(uid_i, uvper_i)$ to $\mathsf{L}_{\mathsf{uvper}}$.
- Case 4 (Corrupted U_i and SO): Let the corrupt parties act unchanged.

As in Game 4, by definition of $\mathcal{S}$, we could in all cases simply execute $\mathcal{S}$ with $\mathcal{F}_{\mathrm{ASTE}}$ to simulate the real protocol. Indeed, $\mathcal{S}$ 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. $\mathsf{L}_{\mathsf{uvper}}$) and $\mathcal{F}_{\mathrm{ASTE}}.uvper_i$ (resp. $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{uvper}}$) always coincide.

Game 6 *(Simulate (but immediately explain) honest ciphertexts).* In this game, we replace the computation of honest ciphertexts by simulations, using $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Sim}$, but immediately explain their contents using $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExplnRE}$ and $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Expln}_{\mathsf{LE}}$ afterwards. Thus, these ciphertexts still correctly decrypt given the secret keys. To keep track of simulated ciphertexts, the game adds c to $\mathsf{L}_{\mathsf{ctsim}}$.

This change is indistinguishable by a straightforward reduction to straight-line simulation-extractability of $\mathsf{NIZK}_{\mathsf{AS}}$, and strong NINCE of $\mathsf{PKE}_{\mathsf{AS}}$ (or rather, strong NINCE of $\mathsf{PKE}_{\mathsf{NCE}}$ and $\mathsf{SKE}_{\mathsf{NCE}}$).

Game 7 *(Consistency of ciphertexts proofs and decryptions).* In this game, we ensure that any non-simulated ciphertexts c which did not decrypt to $\perp$ by an

honest user U_j or honest LE has a consistent NIZK proof. For any $c \notin \mathsf{L}_{\mathsf{ctsim}}$ and which is part of honest party's ciphertext verification or decryption run (either $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Vfy}$, or $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}$ or $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}$) with period *vper*, we add following abort to the game: Assemble the statement $stmt_c$ for c w.r.t. validity period *vper*; let π be the proof (contained in c). Let $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Ext}(td, stmt_c, \pi) = wit$ be the extracted witness, which contains user secret *usk*, user id *uid*, and message m:

Invariant 1 (Consistency of $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}$): If $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}(crs, \mathsf{sk}_\ell, c, vper) \neq m$, abort with output `inconsistent`. (Here honest U_ℓ decrypts.)

Invariant 2 (Consistency of $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}$): If $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(crs, usk, c, vper) \neq uid$, abort with output `inconsistent`. (Here honest LE decrypts.)

Lemma 3. *Let $c = (\mathsf{pk}, \dots)$ be a ciphertext which was not generated in the name of an honest user, i.e. $c \notin \mathsf{L}_{\mathsf{ctsim}}$. Let $stmt_c$ be the assembled statement for c w.r.t. validity period vper and proof π. Then probability that $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Vfy}(crs, c, vper) = 1$, but the extracted witness $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Ext}(td, stmt_c, \pi) = wit$ contains user secret usk, user id uid or message m with $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}(crs, \mathsf{sk}, c, vper) \neq m$ or $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(crs, usk, c, vper) \neq uid$ 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. $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(crs, usk, c) \neq uid$. The claim for $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{RE}}(crs, \mathsf{sk}, c) \neq m$ follows analogously.

Let $c = (\mathsf{pk}, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b}, ((d_{i,\gamma_i})_i, \pi_{\mathsf{con}}))$, and $stmt = (vper, \mathsf{pk}, (com_{i,b}, ct^{RE}_{i,b}, ct^{LE}_{i,b})_{i,b})$. and $\gamma = \mathsf{RO}(stmt, \pi_{\mathsf{con}})$.

First, note that using $\mathsf{NIZK}_{\mathsf{AS}}.\mathsf{Ext}(stmt, \pi)$, we can extract m, uid, usk from π. This is possible, since c was not simulated (i.e. $c \notin \mathsf{L}_{\mathsf{ctsim}}$), and a failure of extraction would break straight-line simulation-extractability (which holds information-theoretically in $\mathcal{F}_{\mathrm{AS}}$). Let $w_{i,b} = (m_{i,b}, usk_{i,b}, uid_{i,b}, r^{RE}_{i,b}, r^{LE}_{i,b})$ 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 $i = 1, \dots, \ell(\lambda)$.

Now check (by recomputing) whether $\mathsf{SKE}_{\mathsf{NCE}}(usk_{i,b}, uid_{i,b}; r^{LE}_{i,b}) = ct_{i,b}$ for all $i \in \{1, \dots, \ell\}$, $b \in \{0, 1\}$. Suppose that for k choices of i, at least one share does not satisfy the check, i.e. $\mathsf{SKE}_{\mathsf{NCE}}(usk_{i,b}, uid_{i,b}; r^{LE}_{i,b}) \neq ct_{i,b}$. Then with probability at most 2^{-k}, the challenge $\gamma = \mathsf{RO}(stmt, \pi_{\mathsf{con}}) \in \{0,1\}^\ell$ 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 $k \leq \ell(\lambda)/2$ inconsistent indices is not rejected is at most $2^{-\lceil \ell(\lambda)/2 \rceil}$. Consequently, if at most $Q(\lambda)$ 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 $Q(\lambda) \cdot 2^{\lceil -\ell(\lambda)/2 \rceil} = \mathsf{negl}(\lambda)$ for polynomial Q. Finally, note that by correctness of $\mathsf{SKE_{NCE}}$, for any index i for which the ciphertexts $ct_{i,b}^{LE}$ are consistent, the decryption $ct_{i,b}^{LE}$ 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 $c \in \mathsf{L_{ctsim}}$, there is only one entry $(c, m, uid, uvper)$ in $\mathsf{L_{msgs}}$. More precisely, upon any call to $\mathsf{PKE_{AS}.Vfy}$ (e.g., as part of $\mathsf{PKE_{AS}.Dec_{LE}}$ or $\mathsf{PKE_{AS}.Dec_{RE}}$) succeeds for another $uvper' \neq uvper$ or output another m or uid than the one contained in $\mathsf{L_{msgs}}$, abort with `inconsistent`.

The uniqueness of c allows c to serve as a unique identifier, which connects *incomplete* entries $(c, *, *, uvper)$ in $\mathsf{L_{msgs}}$ with complete entries $(c, m, uid, uvper)$ in $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$. This will later be useful, to ensure that $\mathsf{L_{msgs}}$ and $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$ coincide for honest ciphertexts, up to unknown m and/or uid for simulated ciphertexts, and prevent any other entry for ciphertext c in $\mathsf{L_{msgs}}$ and $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L_{msgs}}$.

We sketch the proof: Let $c \in \mathsf{L_{ctsim}}$. 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 $c \in \mathsf{L_{ctsim}}$ and $\mathsf{PKE_{AS}.Vfy}$ accepts for two periods $uvper \neq uvper'$. 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 $\mathsf{PKE_{AS}}$ cut-and-choose. Thus, the accepting $uvper$ is unique, as claimed. Secondly, we show that m and uid are fixed for c. If c were to decrypt via $\mathsf{PKE_{AS}.Dec_{RE}}$ under more than one user's sk, or via $\mathsf{PKE_{AS}.Dec_{LE}}$ under more than one usk, then the key-committing property of $\mathsf{PKE_{NCE}}$ or $\mathsf{SKE_{NCE}}$ 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 $(m, uid, uvper)$ as claimed.

Game 9 *(Consistency checks on uid (for honest U_i)).* 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 $\mathsf{PKE_{AS}.Vfy}(crs, c, vper) = 1$ and $\mathsf{PKE_{AS}.Dec_{LE}}(crs, usk, c, vper) = uid$ and $(\mathrm{U}, uid) \in \mathsf{L_U}$ for honest U, then abort with `inconsistent`. (Recall that $c \notin \mathsf{L_{ctsim}}$.)

[33] Note that extraction is never used on ciphertexts in $\mathsf{L_{ctsim}}$. 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 $\mathsf{PKE}_{\mathsf{NCE}}$ and key-committing of $\mathsf{SKE}_{\mathsf{NCE}}$. 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 $(\mathit{uid}, \mathsf{sk}) = \mathsf{PKE}_{\mathsf{AS}}.\mathsf{Gen}(1^\lambda; \mathsf{sk})$. Moreover, since $\mathsf{SKE}_{\mathsf{NCE}}$ is key-committing, it is hard to find $\mathit{usk} \neq \mathit{usk}'$ with $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}(\mathit{crs}, \mathit{usk}', c, \mathit{vper}) \neq \bot$.

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 $(c, m, \mathit{uid}, \mathit{vper})$ 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 $(\mathit{uid}, \mathit{vper}) \notin \mathsf{L}_{\mathsf{uvper}}$, abort with `invalidperiod`. (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 $(\cdot, \mathit{uid}) \notin \mathsf{L}_{\mathsf{U}}$, abort with `invalidperiod`. (If SO is honest, a user must be registered to produce any valid ciphertexts.)

These invariants are immediately implied by the guarantees of $\mathsf{NIZK}_{\mathsf{AS}}$, or rather, the guarantees of the **Verify** task of $\mathcal{F}_{\mathrm{AS}}$. 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 $\mathsf{L}_{\mathsf{msgs}}$ (resp. $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$), are exactly as they would be by $\mathcal{S}$ (resp. $\mathcal{F}_{\mathrm{ASTE}}$). (But not everything related to $\mathsf{L}_{\mathsf{msgs}}$ is handled as in (or by) $\mathcal{F}_{\mathrm{ASTE}}$. 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 $\mathsf{L}_{\mathsf{msgs}}$ and $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$ remain synchronized.

This change is perfectly indistinguishable. Indeed, only if the ciphertext which U_j decrypts is not part of $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$, will there be any change, namely, the $(\textsc{SimInjectMsg}, c)$ 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 $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{Dec}_{\mathsf{LE}}$ 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 $(c, m, \mathit{uid}, \mathit{vper})$ fails the check that $(\mathit{uid}, \mathit{vper}) \in \mathsf{L}_{\mathsf{uvper}}$ 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 $(\textsc{SimInjectMsg}, c)$. 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 $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_W$).* In this game, we simulate the **request warrant** task in $\mathcal{F}_{\text{ASTE}}$, and switch all uses of L_W (of the hybrid functionality $\mathcal{F}_{\text{AS}}$) to uses of $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_W$. This works without any problem, since the **request warrant** task of $\mathcal{F}_{\text{ASTE}}$ is merely a proxy to the task of $\mathcal{F}_{\text{AS}}$. Thus, corrupted parties are handled as usual. For honest parties, we switch their interface to $\mathcal{F}_{\text{ASTE}}$ and plug in the simulation (exactly as in Game 2). Clearly, L_W and $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_W$ always coincide. Moreover, the change is perfectly indistinguishable.

Game 14 *(Simulate **get statistics**).* Since **get statistics** in $\mathcal{F}_{\text{ASTE}}$ is merely a proxy to $\mathcal{F}_{\text{AS}}$, simulation is trivial and perfectly indistinguishable, since we have replaced the use of L_W with $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_W$ globally (in Game 13).

Game 15 *(Simulate **audit**).* Since **audit** in $\mathcal{F}_{\text{ASTE}}$ is merely a proxy to $\mathcal{F}_{\text{AS}}$, simulation is trivial and perfectly indistinguishable, since we have replaced the use of L_W with $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_W$ 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 $\mathsf{L}_{\mathsf{uvper}}$ and $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_{\mathsf{uvper}}$ coincide by Game 5, also $\mathsf{L}_{\mathsf{W\text{-}cached}}$ and $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_{\mathsf{W\text{-}cached}}$ 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 bookkeeping between $\mathcal{F}_{\text{ASTE}}$ and the (simulated) $\mathcal{F}_{\text{AS}}$.

Corrupted LE The simulation acts whenever it receives $(\textsc{GetSecrets}, W)$ in place of $\mathcal{F}_{\text{AS}}$. Namely, the simulator sends $(\textsc{AccessPrep}, W)$ to $\mathcal{F}_{\text{ASTE}}$ on behalf of LE, and receives $(\textsc{AccessLeak})$ or $\mathcal{F}_{\text{ASTE}}$ aborts (in which case $\mathcal{S}$ causes $\mathcal{F}_{\text{AS}}$ to abort). Recall that simulation must ensure that $\mathsf{L}_{\mathsf{W\text{-}cached}}$ and $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_{\mathsf{W\text{-}cached}}$ coincide, and the above strategy ensures exactly that. Before $(\textsc{GotSecrets}, \ldots)$ would be delivered to LE, $\mathcal{S}$ does the following: $\mathcal{S}$ allows $\mathcal{F}_{\text{ASTE}}$ to deliver output. Now, the simulator runs **execute access** on all simulated ciphertexts in $\mathsf{L}_{\mathsf{msgs}}$ (i.e. the subset $\mathsf{L}_{\mathsf{ctsim}}$) with all possible $(uid_i, vper_i)$ where $(uid_i, vper_i, \cdot) \in W$. If a ciphertext c is deanonymized with $(uid, vper)$, $\mathcal{S}$ runs $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}(crs, usk, c, uid, vper)$ and updates $(c, m, uid, vper)$ in $\mathsf{L}_{\mathsf{msgs}}$.

Observe that, in the current game, the calls to $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}$ 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, $\mathsf{L}_{\mathsf{W\text{-}cached}}$ and $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_{\mathsf{W\text{-}cached}}$ 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 $\mathsf{L}_{\mathsf{msgs}}$ and $\mathcal{F}_{\text{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$ 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 $(\textsc{AccessExec}, c, (\mathit{uid}, \mathit{vper}))$, the check $(\mathit{uid}, \mathit{vper}) \in \mathsf{L}_{\mathsf{W\text{-}cached}}$ and $(\mathit{uid}, \mathit{vper}) \in \mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{W\text{-}cached}}$ are identical, since $\mathsf{L}_{\mathsf{W\text{-}cached}}$ and $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{W\text{-}cached}}$ coincide (by Game 16). The task is local, except if $c \notin \mathsf{L}_{\mathsf{msgs}}$, thus it suffices to argue that an occurrence of $(\textsc{InjectMsg}, c)$ does not result in a difference between the output of the protocol **execute access** and the output of $\mathcal{F}_{\mathrm{ASTE}}$. This is almost identical to the argument for **decrypt message** in Game 12 (for corrupted SO). Indeed, by invariants 1–4, no difference will occur (or the game would have aborted).

Remark 11. At this point, the game handles all lists, in particular $\mathsf{L}_{\mathsf{msgs}}$ and $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$, exactly as $\mathcal{S}$ (resp. $\mathcal{F}_{\mathrm{ASTE}}$). The only difference compared to a full simulation is the ciphertext handling in **encrypt message** and **prepare access**. For **encrypt message**, Game 11 still uses $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{RE}}$ and $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}$ immediately after simulating a ciphertext, and this has not been removed. For **prepare access**, Game 16, the call to $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}$ is skipped (as all ciphertext are already explained). We complete the simulation of these task in the following steps.

Game 18 *(Simulate honest ciphertext uids).* In this game, we complete the simulation of **prepare access**. Instead of running $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{LE}}$ on a honestly generated ciphertexts immediately after their simulation (in **encrypt message**), it is run only when necessary.

This change is indistinguishable. To see this, consider two cases:

- *Case 1 (Honest LE).* There is no difference in this case.
- *Case 2 (Corrupt LE).* Whenever $\mathcal{F}_{\mathrm{AS}}$ 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 $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$ directly, but since $\mathcal{S}$ has no access to $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{msgs}}$, it uses the roundabout way of repeatedly running trial calls to **execute access** until all possible cases are exhausted. Moreover, the case of $(\textsc{SimAccessInject}, c)$ is handled correctly. Furthermore, fresh honest encryptions during **encrypt message** are always explained immediately by $\mathcal{S}$ if they lie in a surveilled epoch, i.e. if $(\mathit{uid}_i, \mathit{uvper}_i) \in \mathsf{L}_{\mathsf{W\text{-}cached}}$.

The indistinguishability in Case 2 reduces to strong NINCE-security of $\mathsf{PKE}_{\mathsf{AS}}$.

The last remaining difference in ciphertext handling is addressed next.

Game 19 *(Simulate honest ciphertext m).* In this game, we complete the simulation of **encrypt message**. Instead running $\mathsf{PKE}_{\mathsf{AS}}.\mathsf{ExpIn}_{\mathsf{RE}}$ 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 $\mathsf{PKE}_{\mathsf{AS}}$.

Game 20 *(Tying up loose ends).* At this point, the game handles everything pertaining ciphertexts (in particular filling the list $\mathsf{L}_{\mathsf{msgs}}$) by simply running $\mathcal{S}$. 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 $\mathcal{S}$. 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 $\mathcal{F}_{\mathrm{ASTE}}$). Besides invariants 1–6 introduced in the games, one can also view invariants in terms of the variables kept for the simulator and for $\mathcal{F}_{\mathrm{ASTE}}$. Over the course of several games, it is ensured step by step that these "invariants" hold.

- $ureg_i$ and $\mathcal{F}_{\mathrm{ASTE}}.ureg_i$ always coincide.
- $uvper_i$ (resp. $\mathsf{L}_{\mathsf{uvper}}$) and $\mathcal{F}_{\mathrm{ASTE}}.uvper_i$ (resp. $\mathcal{F}_{\mathrm{ASTE}}.\mathsf{L}_{\mathsf{uvper}}$) always coincide.
- $ureg_i = 0 \implies \mathsf{L}_{\mathsf{uvper}}$ contains no uid_i, assuming honest SO. (Without registration, users cannot run **update**.)
- $(uid_i, vper) \notin \mathsf{L}_{\mathsf{uvper}} \implies$ no messages in $\mathsf{L}_{\mathsf{msgs}}$ from U_i in period $vper$, 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 uid of a an honest user.
- c is a unique identifier in $\mathsf{L}_{\mathsf{msgs}}$ for "honest" ciphertexts.
- If $(c, m', uid', vper) \in \mathsf{L}_{\mathsf{msgs}}$ then $\exists m, uid, vper\colon (c, m, uid, vper) \in \mathsf{L}_{\mathsf{msgs}}$ where $m' = m$ (resp. $uid' = uid$) or $m' = \bot$ (resp. $uid' = \bot$).
- 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 usk's are "released" via the (simulated) hybrid functionality $\mathcal{F}_{\mathrm{AS}}$).