NOTRY: Deniable messaging with retroactive avowal

Faxing Wang
University of Melbourne
wang39@student.unimelb.edu.au

Riad Wahby
Carnegie Mellon University
riad@cmu.edu

Shaanan Cohney
University of Melbourne
coheyns@unimelb.edu.au

Joseph Bonneau
a16z Crypto Research & New York University
jbonneau@gmail.com

ABSTRACT

Modern secure messaging protocols typically aim to provide deniability. Achieving this requires that convincing cryptographic transcripts can be forged without the involvement of genuine users. In this work, we observe that parties may wish to revoke deniability and avow a conversation after it has taken place. We propose a new protocol called Not-on-the-Record-Yet (NOTRY) which enables users to prove a prior conversation transcript is genuine. As a key building block we propose avowable designated verifier proofs which may be of independent interest. Our implementation incurs roughly 8× communication and computation overhead over the standard Signal protocol during regular operation. We find it is nonetheless deployable in a realistic setting as key exchanges (the source of the overhead) still complete in just over 1ms on a modern computer. The avowal protocol induces only constant computation and communication performance for the communicating parties and scales linearly in the number of messages avowed for the verifier—in the tens of milliseconds per avowal.

KEYWORDS

proof of non-knowledge, deniable messaging, AKE, avowal

1 INTRODUCTION

Imagine Mallory wrongfully accuses Alice and Bob of a heinous crime. Alice and Bob have an alibi attested in their chat history, but alas! They communicated using the Off-the-Record protocol (OTR), which is cryptographically deniable. They are therefore unable to satisfactorily prove their innocence to the public!

In this paper, we propose a new protocol called Not-On-The-Record-Yet (NOTRY), which preserves the benefits of deniable protocols while enabling Alice and Bob to jointly avow the contents of a conversation should it later become necessary. NOTRY aims to provide the best of both worlds between deniable protocols while enabling Alice and Bob to jointly avow the transcript is correct (and hence that any other claims are forgeries including those generated prior to t1).

Beyond the exonerating evidence example provided above, there are many possible scenarios in which two communicating parties might later wish to avow a conversation that they initially intended to remain off the record. A transcript may contain evidence useful to a proceeding in which the interests of the two parties are aligned (e.g., co-inventors who wish to prove first-to-invent in patent proceedings, an alibi for a crime, or a journalist and source proving the authenticity of a conversation after events have elapsed).

1.1 Proofs of non-knowledge

The essential technical idea underlying our work is a trick to prove non-knowledge of a discrete log relationship. Given a value X and two bases g, h chosen such that the discrete log relationship between them is unknown, proving knowledge of the discrete logarithm of X to the base g serves as a proof of non-knowledge of the discrete logarithm of X to the base h. Symmetrically, proving knowledge of the discrete logarithm of X to the base h proves non-knowledge of X to the base g. This works because if the discrete logarithm of X were known to both bases, this would enable computing of the discrete log of h to the base g, therefore the discrete log of X may be known to at most one of the bases g, h. The two bases can be chosen using a nothing up my sleeve technique such as hashing a constant value in two different ways to produce two pseudorandom group elements.

A similar idea has been used in protocols for oblivious transfer (OT)\cite{OT1,OT2,OT3,OT4}. For example, the sender in a classic 1-out-of-2 OT chooses two bases g, h such that the receiver doesn’t know the discrete log relation between them. The receiver then generates two public keys K and K’ = gK. The sender is convinced that the receiver can’t know the private key (discrete log) of both keys (to the base g) without learning a discrete log relation between g and h, and therefore can only decrypt one of two messages.

This works effectively as a proof (to the sender) of non-knowledge of one of the two private keys. In this work, we generalize this
idea and provide an explicit description (where previously it was implicit). We extend the idea to proofs-of-non-knowledge in a publicly verifiable setting (whereas in OT the proofs are effectively designated-verifier). We also apply this idea in a new context, avowal of (formerly) deniable proofs. Consider classic designated verifier proofs that prove either knowledge of some witness \( w \) for a statement \( S \) or knowledge of a secret key \( s_k \). We can transform this proof by proving the following:

\[
\text{(knowledge of } w \text{ AND the discrete log of } X \text{ to the base } g) \text{ OR} \\
\text{(knowledge of } s_k \text{ AND the discrete log of } X \text{ to the base } h).
\]

If nothing is known about \( X \), then this serves as a designated verifier proof, as it may have been satisfied either by knowing the genuine witness \( w \) or the verifier’s key \( s_k \). However, if a separate proof establishes that the discrete log of \( X \) is known to either base, this establishes which half of the disjunction was satisfied and reveals if the original designated verifier proof is real or a forgery.

We use an extension of this technique during NOTRY’s authenticated key exchange, in which Alice and Bob mutually create a value \( X = AB \) such that its discrete log to the base \( g \) is secret-shared between the two parties (e.g. \( AB = g^{\ast\ast} \) for values \( y, \delta \) chosen by Alice and Bob respectively). If desired, the two parties can later collaborate to prove knowledge of this discrete logarithm and revoke deniability of the key exchange.

### 1.2 Contributions

With NOTRY we present the first Deniable Authenticated Key Exchange (DAKE) satisfying the security properties described by Unger [66] while additionally supporting retroactive avowal. Our technical contributions are as follows:

- We first develop a new primitive: avowable designated verifier proofs. As with standard designated verifier proofs, they can be forged by a party with a specified public key (the designated verifier) and hence are not convincing to other parties. However, with avowable designated verifier proofs the prover later convince a verifier that a previously generated proof is genuine and not a forgery. We formally define the notion of an avowable designated verifier proof as well as providing a concrete instantiation.

- We construct a modified DAKE that incorporates our avowable designated verifier proof construction, leading to a secure messaging protocol with retroactive avowal. Our proposal (NOTRY) is the first DAKEA (DAKE with avowal) protocol. We assume that both parties store avowal secrets and verifiers store a genuine log of ciphertext transcripts to be avowed. Without authentic ciphertext it is trivial for parties to retroactively create a fake conversation and avow it. Designing an efficient and secure logging protocol is out of our scope.

- We prove the security of our construction using the Universal-Composability framework [16] under a special random oracle model. The proof is conducted under the standard decisional Diffie-Hellman hardness assumption (noting that our protocol is therefore not quantum secure). We show that NOTRY guarantees a strong form of deniability, online deniability [33, 68], and also demonstrate that NOTRY offers standard AKE security properties of AKE.

- We implement NOTRY and evaluate its performance. In comparison to Signal [55, 59], clients incur an 8x times communication overhead and an 8x times computational overhead per key exchange—a manageable performance hit in the face of likely future improvements. When parties wish to avow a message, both they and the judge pay further computation and communication costs to run the avowal protocol.

- We further show that our avowal protocol scales effectively to an entire transcript. Parties are free to avow a stream of messages in a conversation without incurring communication overhead above that of avowing a single message. They only need to operate an extra scalar addition for each additional message avowal. The computation and communication performance overhead of the designated verifier grows linearly with the number of messages to be avowed for the verifier, with the slope of 0.006 for computation and receiving extra 32 bytes each time for avowing one more message.

The remainder of the paper proceeds as follows: We commence in Section 2 with a review of work relevant to our design space, which includes message franking schemes and work on retroactivity in cryptography. In Section 3 we introduce all the primitives needed to build our scheme. This proceeds with reintroducing designated verifier proofs (DVs) for the unfamiliar reader, before proceeding on our novel construction of avowable designated verifier proofs (ADVs)—allowing a prover to render a proof verifiable at some point after they first generate it. In Section 3.4 we extend ADVs by introducing mutually avowable DVs, whereby it takes two cooperating provers to avow a message transcript to a third-party verifier. We are then in Section 3.4 able to introduce our overall deniable messaging scheme with retroactive avowal (NOTRY), deferring its security proofs until Section 5. We conclude with implementing and evaluating the protocol in Section 6 followed by a few closing remarks.

### 2 RELATED WORK

Our work draws from several similar notions to that of cryptographic avowal—disavowal (its mirror image) and message franking (a one-sided complement to avowal). Here we highlight works that relate across two axes: similar cryptographic notions, and deniable messaging protocols on which we build.

#### 2.1 Avowal, Disavowal, and Franking

Chaum[24] defined cryptographic disavowal in his work on zero-knowledge signatures wherein a signer could convince others under zero-knowledge that a signature does not correspond to the signer’s public key and purported message. Cryptographic avowal is also closely related to message franking [41, 65]. Message franking schemes allow a single party in a conversation to prove the authenticity of part of a conversation, typically only to one specific third party (the judge). Such schemes are useful for abuse management in encrypted communication systems, as they can provide a method for a message recipient to report content to a platform
hosting the otherwise encrypted messaging service. Our scheme differs in two ways: 1) both parties to a conversation must agree to avow a conversation and 2) they can choose any third-party as a judge. This vitriifies its usefulness as an abuse reporting mechanism but leads to the other use cases we described in Section 1.

2.2 Retroactivity in Signature Schemes

There are several other primitives of a similar flavor—providing the ability for parties to change the epistemic status of signatures ex post facto, which we here review.

Punctual and Sealfon introduced the notion of claimable ring signatures [37]. Traditional ring signatures provide a form of deniability by allowing a signer to choose an arbitrary set of public keys and prove that a message was signed by some key from that set (without revealing which key). Claimable ring signatures allow the signer to later claim the signature by proving that their specific key was the one used to sign, similar to our notion of revoking deniability after the fact. The corollary notion of unclaimable ring signatures also exists in which it is not possible to provably claim who signed.

Much like ring signature schemes, some threshold signature schemes offer accountability in that the signature reveals which key shares were used to sign. Others may offer privacy in that signatures are indistinguishable regardless of which shares were used. Boneh and Komlo [11] proposed threshold signature schemes with private accountability by introducing a separate accountability key that can be used to compute (and prove) which shares were used to sign, similar to our notion of avowal.

Chaum and van Antwerpen introduced undeniable signatures [25]. Undeniable signatures cannot be verified without online interaction with the original signer (though the verification protocol does not provide transferable proof). Effectively, the signer can “disavow” a past signature by refusing to verify it, meaning signatures are effectively undeniable until avowed. This is somewhat similar to the goal of proofs with retroactive avowal, although (among other differences) we are working in a two-party setting and support public avowal.

2.3 Secure Messaging and Deniability

Modern secure messaging protocols, such as Signal, possess a variety of cryptographic properties, many derived from the extensions on a key-exchange protocol (Diffie-Hellman [32]). The most basic required additional property is authentication, achieved in later authenticated key exchange (AKE) protocols [7, 31, 46, 51]. Modern protocols also possess more sophisticated properties such as composability [6, 8, 14, 20, 29, 52], security under distinct adversarial powers [20, 27, 50], forward secrecy [27, 46], key confirmation [29, 38], and repudiability/deniability [34].

Deniability allows users to convince a judge that they never had such a conversation. It works by providing an easy mechanism for a third-party to generate a forgery that a judge would be unable to distinguish from a legitimate transcript.

Deniability is of particular interest because it directly contrasts with the otherwise desired property of non-repudiability—like past work on franking, our contribution suggests that in the context of secure messaging, both properties may be desirable depending on local context.

Early work explored potential deniable protocols [12, 13, 47, 48] but lacked a corresponding formalism. Di Raimondo et al. [30] formally introduced the notion of deniable authenticated key exchange protocol (DAKE). Deniability now spans an independent line of research [34, 35, 44, 71] and is a standard feature for secure messaging applications, such as Signal and OTR. Considerable work has gone into improving the performance characteristics of DAKEs [30, 33, 43, 55, 63, 66, 70]. Finally, we point to two works that we particularly draw on for our extensions: Walfish [69] who devised a DAKE protocol which provides deniability, forward secrecy, adaptive secure, and without a trusted third party, and Unger [66] who presented protocols enhanced against insider threats and compatible with group messaging.

3 BUILDING BLOCKS

3.1 Notations

Let q be a prime and G be a cyclic group of order q. We use g and h to denote two independently sampled generators of G (that is, the discrete log relationship of g and h is unknown). Suppose Alice holds a pair of identity keys, the corresponding public key pk and private key sk are denoted as kA = (pkA, skA). For i-th round key exchange protocol, we denote yi and δi to be secrets independently chosen by each party and Ai and Bi to be the corresponding public exponents to the base h. We denote the secret for avowal as α and β respectively.

We write a uniform random number r sampled from set U as r ← U. We use X ← Y to denote X being set to Y and 𝜖c to denote computational indistinguishability. Our security parameter overall is denoted as λ. We use || to denote message concatenation, as in m1||m2. Finally, σ denotes a signature and K denotes an established symmetric key.

3.2 Designated Verifier Proofs

To develop our construction we start with Jakobsson, Sako, and Impagliazzo’s idea of designated verifier (DV) proofs. They used a disjunctive statement to provide deniability [42].1 To illustrate this notion we introduce two parties, Alice (the prover) who possesses a statement S, and a Judge, Judy (the verifier).

Given a statement S to be proven to a verifier in zero knowledge, Jakobsson et al. proposed issuing a proof of the modified statement:

\[ S' = \{ \text{Either } S \text{ is true or I know Judy's private key} \} \]  

We will abbreviate this construction here:

\[ S' = S \lor \text{Know}(sk_j) \]  

A proof of the compound statement S’ is only convincing to Judy: If Judy is confident that nobody else knows her private key sk_j (and that she did not compute the proof), then she knows the second clause is false and therefore S is true.

For anyone other than Judy, it is unclear which side of the disjunction is true—it is possible that either S is true or that Judy herself created a valid proof of S’ by satisfying the second clause.

---

1Another approach to constructing signatures with the designated-verifier property is chameleon signatures [49], which uses a standard hash-and-sign construction but with a chameleon hash function whose trapdoor is known by the intended verifier, providing deniability if the verifier attempts to transfer the signature to another party.
Therefore, the prover has deniability—the prover can always claim to an outsider that Judy generated the proof whether or not \( S \) is true, and the outsider will be unable to tell.

### 3.3 Avowable designated verifier proofs

We build on DV proofs to introduce what we believe is a novel tool: **avowable designated verifier proofs** (ADV proofs). We later use this to develop NOTRY, however ADV proofs may be of independent interest for other applications.

Given a statement \( S \), an ADV proof \( \pi \) is a proof of a modified statement \( \bar{S} \) with two key properties:

1. at time \( t_0 \) no party, not even the judge, can ascertain the truth of \( \bar{S} \) from \( \bar{S} \) alone and
2. at some future time \( t_1 \), the generator of the proof can complete a protocol such that the judge can validate that \( \bar{S} \) attests to the truth of \( S \).

Note that it is not enough to merely provide an unconditional (non-designated-verifier) or direct proof that \( S \) is true. Providing an unconditional proof would not preclude that previous proofs of \( S \) were created as forgeries.

A **one-time ADV scheme**: A strawman approach would be to prove that nobody knows Judy’s private key. Assume that Judy’s public key is a group element of the form \( pk_f = g^a \) and Judy’s private key is \( sk_f = \tilde{x} \). If Judy chooses her public key pseudorandomly as \( pk_f = H(y) \) for some pre-image \( y \), then under the discrete-log assumption revealing \( y \) proves that nobody knows the corresponding private key \( sk_f \) and therefore any proofs issued to \( pk_f \) were genuine. However, this scheme means that Judy must use a new key for every proof, and the key must be chosen by the prover in the case of genuine proofs.

A **candidate scheme**: Instead, we allow the prover to create a new proof-specific statement \( P \) which can later be easily proven to be true or false, and notify the auxiliary statement \( P \) to our ADV proof scheme works by creating a proof of the following statement:

\[
\bar{S} = (S \land P) \lor (\text{Know}(sk_f) \land \neg P) \tag{3.3}
\]

Revealing the veracity (or falsehood) of \( P \) then makes it clear which half of the disjunction was satisfied and hence whether such a proof was created by an honest prover or by Judy. Note that in either case, the proof creator must choose \( P \) and remain a witness to either prove \( P \) (in the case of the honest prover) or prove \( \neg P \) (in case of the forger). The construction symmetrically enables both avowing an honestly created proof and avowing a forgery.

Based on the candidate scheme, we provide a formal definition of the security properties of an ADV in Section 3.4 and proofs in Section 5.4.1 (deferring them for overall clarity of exposition).

#### 3.3.1 ADV Functions

An ADV proof scheme consists of the following five functions:

- **Setup(\( \lambda \))** generates public parameters which can be used concurrently and repeatedly for multiple ADV proofs. Separately, we assume a PKI to establish public/private key pairs for all parties and a designated judge.
- **Gen(\( \bar{S}, W \))** outputs an ADV proof \( \bar{\pi} \) of the disjunction \( \bar{S} \) with the witness \( W \in \{(w, wp), (sk_f, wp)\} \), where \( w \) for the statement \( S, sk_f \) for the knowledge of judge’s private key, and \( wp \) for the statement \( P \) whereas \( wp \) for \( \neg P \).

**Avow(\( P, wp \))** generates the avowal proof \( \pi' \) by (dis)proving an auxiliary statement \( P \) for telling which clause in \( \bar{S} \) was proved. The witness \( wp \) can either be \( wp \) to avow \( S \) was satisfied via \( S \) or \( wp \) to avow \( S \) was satisfied via the knowledge of \( sk_f \).

**Verify(\( \pi, \bar{S} \))** asserts that an ADV proof \( \bar{\pi} \) is a valid proof for an accompanying ADV statement \( \bar{S} \).

**Judge(\( \pi', P \))** verifies the avowal proof \( \pi' \) and asserts the validity of the avowal claim.

#### 3.3.2 Constructing ADV Proofs

**Discrete-log ADV construction**: For \( \Sigma \)-protocols, a compelling choice for \( P \) is knowledge of the discrete log \( x \) of a value \( y = g^x \) (in an appropriate group). There is a classic Schnorr \( \Sigma \)-protocol for proving knowledge of this discrete-log relationship [64] that one can combine with \( \Sigma \)-OR proofs [26] to create an efficient proof of \( \bar{S} \). The Schnorr protocol does not permit us to directly prove \( \neg P \) in a cyclic group: given the construction of \( y \) (raising the base \( g \) to the power of \( x \)) it is clear that group element \( y \) has a unique discrete log to the base \( g \). Instead, we utilize another generator \( h \) sampled independently, ensuring that nobody knows the discrete log relationship between \( g \) and \( h \).\(^2\) Proving knowledge of the discrete log of \( y \) to the base \( h \) suffices to show that the discrete log of \( y \) to the base \( g \) is unknown. If the discrete log of \( y \) was known to both bases \( g \) and \( h \), this would enable easily computing the discrete log of \( g \) to the base. This yields an efficient candidate for \( P \) when constructing \( \Sigma \)-protocols: proof of knowledge of an element \( y \) to either the base \( g \) (equivalent to proving \( P \)) or to the base \( h \) (equivalent to proving \( \neg P \)).

Alternatively, if \( g \) and \( h \) are known to generate subgroups of a group \( G \) which are disjoint modulo the identity element, and the subgroup membership problem is assumed to be hard, then proving knowledge of the discrete log of \( y \) to the base \( g \) is an unconditionally sound proof that the discrete log of \( y \) to the base \( h \) is unknown (since it will not exist). However, the discrete log assumption is still required for privacy.

**Hash-based ADV construction**: An alternative, simple construction that might offer efficiency benefits in some circuit-based proof systems utilized preimages of a collision-resistant hash function. Specifically, \( P \) might state that given a value \( b \), there exists a preimage \( b = H(a) \) under hash function \( H \) such that \( a \) has odd parity. The corresponding \( \neg P \) simply states that \( a \) has even parity. This statement can be proven true or false by revealing \( a \) (or if necessary this could be proved in zero-knowledge). Note that if two values \( a \) and \( a' \) are known such that \( b = H(a) = H(a') \) (possibly with differing parity), this would be a hash collision. Thus, only one of \( P, \neg P \) will be provable assuming the hash function remains collision-resistant.

### 3.4 Mutual ADVs

Finally, the full NOTRY protocol requires an extension to ADV proofs which enables mutual avowal, in which both Alice and Bob’s

\(^2\)Given a hash function \( H \) which outputs pseudorandom generators of a group \( \mathbb{G}_1 \), \( g \) and \( h \) can be chosen as \( g = H(0), h = H(1) \), for example.
cooperation is needed to avow the proof. We term such a proof a mutual-ADV proof or MADV.

The security goal of MADV avowal proof is to convince the Judge that Alice and Bob together do have the witness \( w \) to an ADV statement \( \tilde{S} \) while revealing no more information to the Judge. This means that MADV proof avowal is a zero-knowledge proof of knowledge (ZKPoK). Since this paper centers on avowal proofs, security definitions and corresponding proofs of ADV proofs are deferred to appendix A.

**Definition:** Formally, a secure MADV avowal proof \( \pi' \) against the MADV proof \( \pi \) and statement \( \tilde{S} \) consists of a triple of algorithms (Setup, Avow, Judge) with the following properties:

1. **Completeness:** Any prover who generated an MADV proof \( \pi \) can construct the corresponding MADV proof avowal \( \pi' \) such that:
   \[
   \Pr \left[ 1 \leftarrow \text{Judge}(\pi, \tilde{S}) \mid \pi' \leftarrow \text{Avow}(P, W_P) \right] = 1 \tag{3.4}
   \]
   Completeness indicates that Judge always accepts the MADV avowal proof \( \pi' \) from an honest prover.

2. **Knowledge Soundness:** There exists a PPT extractor \( E \) which, given two transcripts of avowal proof generation \( T_1 \) and \( T_2 \), can extract a valid witness:
   \[
   \Pr \left[ 1 \leftarrow \text{Judge}(\pi', P) \mid W'_P \leftarrow E(1^\lambda, T_1, T_2) \pi' \leftarrow \text{Avow}(P, W'_P) \right] = 1 \tag{3.5}
   \]
   where \( W'_P \) is the extracted witness. Special soundness implies that any party who generates a valid ADV proof \( \pi \) must gain the knowledge of \( W_P \) to \( P \).

3. **Avowal Soundness:** For any PPT adversary \( A \), given a MADV proof \( \pi \), it is unable to produce a valid avowal proof:
   \[
   \Pr \left[ 1 \leftarrow \text{Judge}(\tilde{\pi}', P) \mid \tilde{\pi}' \leftarrow A(1^\lambda, \pi) \right] \leq \text{negl}(\lambda) \tag{3.6}
   \]
   where \( \text{negl} \) is a negligible function.

4. **Honest Verifier Zero-Knowledge:** There exists a PPT simulator \( S \) that generates an MADV avowal proof \( \tilde{\pi}'_S \) without knowledge of the witness \( w \) such that \( 1 \leftarrow \text{Judge}(\tilde{\pi}'_S) \). \( \tilde{\pi}'_S \) holds that:
   \[
   \tilde{\pi}'_S \approx_c \tilde{\pi}' \tag{3.7}
   \]

After defining a secure MADV avowal, we proceed to provide an MADV construction. Based on our discrete-log ADV construction, our construction of MADV begins with Alice and Bob generating a value \( AB = h^{\gamma + \delta} \). The value \( \gamma + \delta \) is a shared secret between Alice and Bob; let \( \gamma \) and \( \delta \) be individual secrets belonging to Alice and Bob respectively. Generating the shared secret is easy: Alice generates \( \gamma \) randomly and sends \( A = h^\gamma \) to Bob, who generates \( \delta \) randomly and sends \( B = h^\delta \) back to Alice. Both sides can compute \( AB = A \cdot B = h^\gamma \cdot h^\delta = h^{\gamma + \delta} \).

Recalling that proving non-knowledge of a discrete log to base \( g \) is achieved by demonstrating knowledge of the discrete log of the same value to another base \( h \) where \( g \) and \( h \) are two independently sampled generators of \( G \). Let \( h \) be the generator used to produce the shared secret. Thus, Alice and Bob can use the mutually created \( AB \) as the value in our ADV construction, making it an MADV as avowal will require both Alice and Bob’s participation to prove knowledge of the discrete log of \( AB \) to the base.

Note that if either party is malicious, they can permanently prevent future avowal. For example, Alice can choose \( A = g^\delta \) for a random \( \gamma \), which will prevent ever avowing that the discrete log of \( AB \) to the base \( h \) is known. This simple attack could be prevented if both sides include a zero-knowledge proof of \( A, B \) respectively to the base \( h \) (possibly a designated verifier proof for the other party), but this doesn’t prevent either party from deleting their avowal share (\( a \) or \( b \)) immediately after the handshake completes.

To see why this is convincing, take Alice’s statement (Equation 3.8). For an honest Bob, who has sampled \( B = h^\delta \), he can be sure Alice could not prove \( \text{dlog}_h A \) and hence Alice must know \( \text{dlog}_h A \). This means that \( \text{dlog}_h AB \) is unknown, and hence Alice must know both \( \text{dlog}_g p_A \) (authenticating her) and \( \text{dlog}_h A \) (meaning her contribution to the avowal secret is well-formed). A symmetric argument applies to Alice’s reasoning about Bob’s proof.

Observe that this construction is still deniable—a forger can first randomly generate \( A = h^\gamma \), then \( AB = g^\delta \) for a random \( \chi \) and derive \( B = AB/A \). This forger can then satisfy both Equation 3.8 and Equation 3.9 by proving \( \text{dlog}_g A \) on the left side (using knowledge of \( \gamma \) and \( \text{dlog}_h AB \) on the right side (using knowledge of \( \chi \)). Note that this forger does not need to know either secret key.

Conversely, avowal works by the parties jointly proving knowledge of \( \text{dlog}_h AB \), using the shared secret value \( \gamma + \delta \). This rules out that the simulator was used, as the simulator (having chosen \( AB \) with a known discrete log to the base \( g \)), cannot prove knowledge of \( \text{dlog}_h AB \).

There are two possible flavors of avowal: public and designated verifier avowal. Depending on different applications, parties can choose to avow to a designated Judge for partial deniability (avowed transcripts are still deniable for anyone other than the Judge) or publicly avow and completely give up deniability. Constructions are similar for both types of avowal, using either an unconditional proof-of-knowledge of \( \text{dlog}_h AB \) for public avowal or its corresponding designated verifier version for designated-verifier avowal.

### 3.5 Signatures of Knowledge

Mutual ADVs alone are not sufficient to build an avowable key exchange protocol, as they are vulnerable to meddler-in-the-middle (MITM) attacks. A standard solution to address MITM attacks in key exchange protocol is to ask parties to include signatures on their messages. However, classic digital signatures will undermine deniability as they provide non-repudiable evidence that a specific private key was used to sign.

To build a deniable authenticated key exchange protocol with retroactive avowal (DAKEA) defending against MITM attack, we instead use *signatures of knowledge* [23]. Unlike classical digital signature schemes which require a specific secret key, computing a signature of knowledge (SoK) requires knowing a witness \( w \) for some statement \( x \in L \) for an NP language \( L \). Specifically, NOTRY
requires parties to compute a signature of knowledge using a witness for the MADV proof statement, like eq. (3.8), on a message which commits to all the public information and the prefix of the transcript.

Definition 1. A signature of knowledge scheme for message space \(M\) and an NP language \(L\) decided by a Turning Machine \(M_L\) is defined by three polynomial-time algorithms (Setup, Sign, Verify):

\[
\text{Setup}(\lambda) \rightarrow pp: \text{produces a set of public parameters } pp.
\]

\[
\text{Sign}(m, w, x, sk) \rightarrow \sigma: \text{generates a SoK } \sigma \text{ on message } m \text{ by } (pp, L, x, w) \text{ with } sk \text{ if } M_L \text{ decides } x \in L \text{ such that } M_L(x, w) = 1, \text{ otherwise outputting } \bot.
\]

\[
\text{Verify}(m, x, \sigma, pk) \rightarrow \text{Accept/Reject}: \text{verifies the signature } \sigma \text{ on message } m \text{ for statement } x \text{ with } pk.
\]

A signature of knowledge has three following security properties:

Correctness: If a signature \(\sigma\) is produced using a valid witness, then it should be accepted by a verifier with overwhelming probability.

Simulatability: By checking a signature, a verifier learns nothing beyond that the message is signed properly and the statement is true. This means that the signature of knowledge reveals nothing about the witness which is utilized to generate the signature.

Extraction: guarantees that a party who can create a valid signature must “know” a witness \(w\) for the statement \(x\), in the sense that an extractor exists which can output a witness given access to the signer. In other words, a signature of knowledge is also a proof of knowledge issued by a signer which indicates that she knows the witness to the target statement.

We use this primitive by setting the message \(m\) to be signed as the corresponding key exchange protocol messages and letting the statement \(F\) be the ADV disjunction \(S\). Therefore, by using the discrete-log ADV construction in Section 3.3.2, a concrete SoK can be constructed. Each ADV prover generates a Schnorr Signature [64] with its secret key and sets the message \(m\) to be an instantiation of the ADV statement \(S\).

4 DESIGN OF NOTRY

Our NOTRY protocol consists of two sub-protocols: NOTRY-Kex and NOTRY-Avow. Like other (D)AKEs, we design our NOTRY-Kex around the Diffie-Hellman protocol with signatures for authentication. We also include a ratcheting scheme in NOTRY-Kex-Ratchet to enable future secrecy in the context of ongoing messaging—typical among secure messaging protocols [12].

4.1 NOTRY-Kex

Based on the above constructions and primitives we propose our NOTRY DAKEA key exchange protocol NOTRY-Kex. Before starting NOTRY-Kex, we assume that the authority properly sets up public parameters and identity tokens. Initially, Alice and Bob register their key pairs with a server (similar to Signal’s coordinating server or a CA). NOTRY-Kex depicted in Figure 1 proceeds as follows:

First, Alice starts the protocol by randomly selecting an ephemeral secret \(\gamma \in \mathbb{Z}_q\). She sends Bob the public exponent \(A \leftarrow h^\gamma\) as with DH protocol.

\[
\begin{align*}
\text{Alice}(pk_A, pk_B, sk_A) & \quad \text{Bob}(pk_A, pk_B, sk_B) \\
\gamma & \leftarrow \mathbb{Z}_q \\
A & \leftarrow h^\gamma \\
\delta & \leftarrow \mathbb{Z}_q \\
B & \leftarrow h^\delta \\
\sigma_B & \leftarrow \text{SoK-Sign}((A, B), \delta, S_B, sk_B) \\
S_B & \leftarrow \text{Sign}((A, B), S_B, pk_B) \\
\sigma_A & \leftarrow \text{SoK-Sign}((A, B), S_A, \gamma, sk_A) \\
\end{align*}
\]

Figure 1: NOTRY-Kex. Alice and Bob follow the key exchange protocol and store the masked secrets \(\alpha, \beta\) for future avowal.

Second, Bob chooses his ephemeral secret \(\delta \in \mathbb{Z}_q\) and computes \(B \leftarrow h^\delta\). He generates the ADV proof \(\pi\) to the statement \(S_A\) defined in eq. (3.8) with the knowledge of ephemeral secret \(\delta\) and his secret key \(sk_B\). Based on \(\pi\) he generates a signature of knowledge, \(\sigma_B\) on message \(m \leftarrow (A, B)\). Bob sends \(\sigma_B\) and \(B\) back to Alice.

Third, Alice verifies the \(\sigma_B\) from Bob with his public key \(pk_B\) and \(B\). She aborts if \(\sigma_B\) is invalid. Alice computes her signature of knowledge \(\sigma_A\) on the statement \(S_B\) in eq. (3.9), with knowledge of her private key \(sk_A\) and \(\gamma\) on message \(m \leftarrow (A, B, \sigma_B)\). In parallel, Alice is able to derive the DH key \(K \leftarrow B^\gamma\). Alice can now generate the session key \(K_{sess}\) which is generated by a KDF that takes as input: the two SoKs \(\sigma_A, \sigma_B\), the DH key \(K\), and DH transcripts \(A, B\). Finally, Alice masks her ephemeral secret with the digest \(\rho\) of the session key concatenated with the message “avow” to produce her avowal witness \(\alpha\). (Alice now erases \(\gamma\) and \(\rho\)). To finish NOTRY-Kex, Alice sends her SoK, \(\sigma_A\), to Bob.

At last, Bob verifies \(\sigma_A\) from Alice and aborts if the verification fails. Next, Bob computes his DH key \(K \leftarrow A^\delta\). Bob generates an identical session key to Alice, \(K_{sess}\), following the same process. Bob gets his avowal witness \(\beta\) in such a way that \(\alpha + \beta\) equals the addition of two ephemeral secrets \(\gamma + \delta\). Finally, Bob securely deletes temporary secrets \(\delta, \rho\).

Note that the SoK statement in NOTRY-Kex is actually consistent with Equation 3.8. Take Alice as an example—she demonstrably does not know \(\text{dlog}_h\). Therefore, she is unable to prove both clauses...
involving B but to prove knowledge of her own ephemeral secret \( y \leftarrow dlog_y A \) and her private key \( sk_A \leftarrow dlog_y pk_A \).

**Forging key exchange transcripts.** Using the simulator algorithm NOTRY-Kex-Sim, defined in Figure 3, anyone, including Alice and Bob, is able to generate indistinguishable NOTRY-Kex transcripts, which are \( A, B, \sigma_A, \sigma_B \). The core idea of constructing the simulator is to generate \( AB \leftarrow h^\gamma, q^\delta \) where \( \gamma, \delta \) is sampled from \( Z_q \). The simulator generates \( B \leftarrow AB / A \). Therefore, the simulator is able to produce a valid ADV-based SoK by proving clauses \( dlog_y AB \wedge dlog_y A \).

**NOTRY-Kex-Ratchet.** In most cases, a conversation between Alice and Bob consists of multiple messages. Therefore, a stream of keys is needed to secure a complete conversation consisting of a stream of messages between the two parties. To ensure future secrecy across an ongoing conversation we introduce a ratcheted [59] version of our protocol, NOTRY-Kex-Ratchet. By sending a new SoK each time Alice and Bob update a key, we can extend NOTRY-Kex. As shown in Figure 2, NOTRY-Kex-Ratchet works as follows:

- For their first session key \( K_{0,0} \), parties follow NOTRY-Kex as expected.
- For a message key \( K_{i,j}(i > 0 \text{ or } j > 0) \) indexed with \( i, j \) as \( i \)-th contribution from Alice and \( j \)-th contribution from Bob, first Alice samples a random ephemeral secret \( y_i \leftarrow Z_q \) and setups the DH key \( K_{i,j} \leftarrow B_i^j \). The message key \( K_{\text{sec},i,j} \) is the output of the KDF with \((A_i||B_j||\sigma_A||\sigma_B||K_{i,j})\) as input.

We instantiated the statement \( S^{i,j}_A \) (and \( S^{i,j}_B \) accordingly) with \( A_i, B_j, \) and \( pk_A, pk_B \), where all of them are group elements.

#### 4.2 NOTRY-Avow

Up to this point, our protocol allows Alice and Bob to establish a secure channel with NOTRY-Kex protocol and to deny a transcript using NOTRY-Kex-Sim. We now introduce the retroactive avowal routine within NOTRY protocol to support retroactive avowal. Recall that to avow, Alice and Bob aim to ‘revoke’ deniability. Deniability arises from the first AND clause in Equation 3.8 and 3.9, which proves knowledge of \( dlog_y B \) or \( dlog_y A \) and \( dlog_y (AB) \). The avowal procedure works by proving that the proof statement was generated through the second clause (involving the private key), i.e. by giving evidence that the first clause is false.

In a genuine conversation, Alice and Bob are incapable of proving knowledge of \( dlog_y AB \) but can prove knowledge of \( dlog_y (AB) \) — noting that this relies on the absence of a known relationship between \( g \) and \( h \). The *Judge* can therefore be persuaded that the first clause is false by receiving \( dlog_y (AB) \). Note that Equation 3.8 and 3.9 both contain \( dlog_y (AB) \), proving non-knowledge of \( dlog_y AB \) indicating both statements are proven via their second clauses. That shows the two SoK signatures \( \sigma_A, \sigma_B \) were generated with knowledge of \( sk_A, sk_B \).

Recall that two styles of avowal are possible, here we choose to give a construction for the designated verifier avowal where public avowal can be trivially derived by excluding the designated clause. Essentially, the designated verifier avowal proves non-knowledge of the discrete log to the designated verifier *Judge*. Therefore, the

<table>
<thead>
<tr>
<th>Alice({pk_A, pk_B, sk_A} )</th>
<th>Bob({pk_A, pk_B, sk_B} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_0 \leftarrow Z_q )</td>
<td>( A_0 \leftarrow h^{B_0} )</td>
</tr>
<tr>
<td>( A_0 )</td>
<td>( \delta_0 \leftarrow Z_q )</td>
</tr>
<tr>
<td>( B_0 \leftarrow h^{\delta_0} )</td>
<td>( \sigma_{B_0} \leftarrow \text{SoK-Sign}\left( (A_0, B_0), \delta_0, S_{A,0}^0, sk_B \right) )</td>
</tr>
</tbody>
</table>
| \( \sigma_{B_0} \) & Abort if \(~\neg\text{SoK-Verify}\left( (A_0, B_0), S_{A,0}^0, \sigma_{B_0}, pk_B \right) \)

| \( K_{10} \leftarrow B_{10}^{1\theta} \) | \( K_{10} \leftarrow A_{10}^{\delta_0} \) |
| \( k_{\text{sec}10} \leftarrow \text{KDF}\left( A_1 || B_0 || \sigma_A || \sigma_B || K_{10} \right) \) | \( k_{\text{sec}10} \leftarrow \text{KDF}\left( A_1 || B_0 || \sigma_A || \sigma_B || K_{10} \right) \) |
| \( \text{Alice now encrypts with hash-ratcheted } k_{\text{sec}10} \) | \( \text{Bob now encrypts with hash-ratcheted } k_{\text{sec}10} \) |

| \( y_1 \leftarrow Z_q \) | \( A_1 \leftarrow h^{B_1} \) |
| \( \sigma_{A_1} \leftarrow \text{SoK-Sign}\left( (A_1, B_0), y_1, S_A^{1,0}, sk_A \right) \) | \( \sigma_{B_1} \leftarrow \text{SoK-Sign}\left( (A_1, B_1), \delta_1, S_B^{1,1}, sk_B \right) \) |

| \( \sigma_{B_1} \) & Abort if \(~\neg\text{SoK-Verify}\left( (A_1, B_1), S_A^{1,1}, \sigma_{B_1}, pk_B \right) \)

| \( K_{11} \leftarrow B_{11}^{1\theta} \) | \( K_{11} \leftarrow A_{11}^{\delta_1} \) |
| \( k_{\text{sec}11} \leftarrow \text{KDF}\left( A_1 || B_1 || \sigma_A || \sigma_B || K_{11} \right) \) | \( \text{Bob now encrypts with hash-ratcheted } k_{\text{sec}11} \) |

| \( y_2 \leftarrow Z_q \) | \( A_2 \leftarrow h^{B_2} \) |
| \( \sigma_{A_2} \leftarrow \text{SoK-Sign}\left( (A_2, B_1), y_2, S_A^{2,1}, sk_A \right) \) | \( \sigma_{A_2} \leftarrow \text{SoK-Sign}\left( (A_2, B_1), y_2, S_A^{2,1}, sk_A \right) \) |

| \( \sigma_{A_2} \) & Abort if \(~\neg\text{SoK-Verify}\left( (A_2, B_1), S_A^{2,1}, \sigma_{A_2}, pk_A \right) \)

| \( K_{21} \leftarrow B_{21}^{1\theta} \) | \( K_{21} \leftarrow A_{21}^{\delta_2} \) |
| \( k_{\text{sec}21} \leftarrow \text{KDF}\left( A_2 || B_1 || \sigma_A || \sigma_B || K_{21} \right) \) | \( \text{Alice now encrypts with hash-ratcheted } k_{\text{sec}21} \) |

| \( k_{\text{sec}21} \) & Abort if \(~\neg\text{SoK-Verify}\left( (A_2, B_1), S_A^{2,1}, \sigma_{A_2}, pk_A \right) \)

\( K_{xy} \) are used in Figure 1. For every ratcheted resulting prekey \( K_{xy} \), we have \( dlog_y K_{xy} = dlog_y B_y \) or \( dlog_y K_{xy} = dlog_y A_x \).
Σ with \( P \) statement avowal proof is an aggregation of the two proofs. Note that the final avowal statement to be proven is:

\[
\{ \text{dlog}_{g}(AB) \} \lor \{ \text{Know}(sk) \} \quad (4.1)
\]

As the \( \text{dlog}_{g}AB \) is separately held by Alice (α) and Bob (β), we, therefore, devise a two-party Σ proof generation by asking the two to generate their corresponding proof-of-knowledge of their secrets with Σ-OR Schnorr and exchange their proofs. The ultimate ADV avowal proof is an aggregation of the two proofs. Note that the statement \( P (\neg P) \) in eq. (3.3) is instantiated with \( \text{dlog}_{g}AB \) (\( \text{dlog}_{g}AB \)).

As shown in Figure 4, NOTRY-Avow works as follows:

1. NOTRY-Avow begins by both parties executing NOTRY-Kex to get a session \( k_{rel} \) to communicate securely for the course of the avowal protocol. Note that a simpler deniable key exchange could also be used here to produce a session key for the avowal protocol, for simplicity, we assume we reuse NOTRY-Kex.

2. Either Alice or Bob can request an avowal. Their counterparty approves this request by joining NOTRY-Avow. For convenience, we start NOTRY-Avow with Alice as the initiator. Alice samples random \( c_{A}, z_{A}, r_{A} \) for the ADV proof avowel, and samples another random value \( s_{A} \) to prevent message replay. Alice computes \( E_{A} = g^{c_{A} h^{z_{A}} \ell^{A}} \) and \( R_{A} \) and sends them to Bob. Note that \( E_{A} \) is necessary to prohibit Alice from illegitimately generating a valid ADV proof that passes verification. We mitigate Alice’s advantage (wherein she receives secrets first) by asking her to commit to all the secrets. Note that \( E_{A} \) is a natural adaptation of a Pedersen Commitment [58], where \( s_{A} \) is the random value to complete a Pedersen Commitment.

3. Bob receives messages from Alice and randomly generates \( c_{B}, z_{B}, r_{B} \) as his proof of the ADV proof avowel. Bob proceeds with getting \( R_{B} \) and sending it with encrypted \( c_{B}, z_{B} \).

\[
\text{Alice}(pk_{A}, pk_{B}, pk_{A} \cdot pk_{j}, sk_{A}, \alpha) \quad \text{Bob}(pk_{A}, pk_{B}, pk_{j}, sk_{B}, \beta)
\]

Execute NOTRY-Kex to produce \( k_{rel} \), discarding the resulting release secrets.

\[
c_{A} \leftarrow \{0, 1\}^\lambda \\
z_{A}, s_{A}, r_{A} \leftarrow \mathbb{Z}_{q} \\
E_{A} \leftarrow g^{c_{A} h^{z_{A}} \ell^{A}} \\
R_{A} \leftarrow h^{r_{A}}
\]

\[
E_{A}, R_{A} \\
c_{B} \leftarrow \{0, 1\}^\lambda \\
z_{B}, r_{B} \leftarrow \mathbb{Z}_{q} \\
R_{B} \leftarrow h^{r_{B}}
\]

\[
R_{B}, \text{Enc}(k_{rel}, (c_{B}, z_{B})) \quad \text{Enc}(k_{rel}, (c_{A}, z_{A}, s_{A}))
\]

Abort if \( E_{A} \neq g^{c_{A} h^{z_{A}} \ell^{A}} \)

\[
c_{j} \leftarrow c_{A} \oplus c_{B} \\
z_{j} \leftarrow z_{A} + z_{B} \\
R_{j} \leftarrow g^{z_{j}} / pk_{j}^{c_{j}} \\
R_{AB} \leftarrow R_{A}R_{B} \\
c \leftarrow H(R_{AB} || R_{j} || \cdots) \\
c_{AB} \leftarrow c + c_{j}
\]

\[
z_{\alpha} \leftarrow c_{AB} \alpha + r_{A} \\
z_{\beta} \leftarrow c_{AB} \beta + r_{B} \\
\text{Output } z_{\alpha} \\
z_{\alpha} \leftarrow z_{A} + z_{B} \\
\text{Output } z_{\beta} \\
z_{AB} \leftarrow c_{AB} z_{A} + c_{AB} z_{B}
\]

Figure 3: NOTRY-Kex-Sim. Anyone can run the simulator to generate an indistinguishable transcript to deny.

Figure 4: NOTRY-Avow

4. Alice first decrypts the ciphertext to get Bob’s partial proof and sends her partial proof \( c_{A}, z_{A} \) with fresh secret \( s_{A} \) secretly to Bob.

5. After decrypting the ciphertext from Alice, Bob first checks message freshness by computing \( E_{A} = g^{c_{A} h^{z_{A}} \ell^{A}} \). Observe that after three-more message exchanges both Alice and Bob will have acquired all of \( c_{A}, z_{A}, z_{B}, \) and \( R_{B} \). Both Alice and Bob are then capable of generating the next stage of the avowel proof by setting \( c_{j} = c_{A} @ c_{B} \) and \( z_{j} = z_{A} + z_{B} \) and \( R_{j} \) and \( R_{AB} \). After that, the simulated proof of \( sk_{j} \) is \( R_{j} - g^{z_{j} / pk_{j}^{c_{j}}} \). The Fiat-Shamir-based random challenge \( c \) of the disjunction proof is generated by hashing the message \( R_{AB} || R_{j} \). The final challenge \( c_{AB} \) required to prove \( \text{dlog}_{g}AB \) (which is jointly generated Alice and Bob) is \( c_{AB} \leftarrow c \oplus c_{j} \).

6. Respectively, Alice and Bob generate the last piece of the avowel proof \( z_{w} \) to get \( z_{AB} \) with their partial witness \( w \in [\alpha, \beta] \). They complete NOTRY-Avow by exchanging \( z_{w} \) to get \( z_{AB} \) and output the avowel proof \( \pi \leftarrow (c_{AB}, z_{AB}, c_{j}, z_{j}) \).

The designated Judge verifies \( \pi \) by checking whether it proves the knowledge \( \text{dlog}_{g}AB \) or not. As shown in Figure 5, after parsing
We will analyze three algorithms of NOTRY. A secure AKE protocol ensures that they established the key with Section 5.4 overviews our UC-secure proof of overviews proof techniques that will be used for our proof, Sec-
a UC-secure be a ZKPoK protocol. Second, for the interactive avowal generation, the framework.

\[ \text{NOTRY-Kex-Ratchet} \]
\[ \text{indistinguishable from the genuine transcript} \]
\[ \text{one running the simulator could output a transcript} \]
\[ \text{change protocol’s transcript,} \]
\[ \text{their intended partner and the key is a fresh secret. The key ex-

\[ \text{399} \]
\[ \text{NOTRY-Avow} \]
\[ \text{taneously avow multiple messages, claiming ownership of an ex-

\[ \text{NOTRY-Judge} \]
\[ \text{Judge's verification procedure:} \]
\[ \text{applying (by not having any corresponding transcript to be avowed).} \]
\[ \text{avowal even if they haven’t exchanged any keys. Of course, this}
\[ \text{attempt will either fail (generate an invalid proof) or be unconvinc-

\[ \text{NOTRY-Avow} \]
\[ \text{are not necessarily sequential. Parties can start}
\[ \text{avow even if they haven’t exchanged any keys. Of course, this}
\[ \text{attempt will either fail (generate an invalid proof) or be unconvinc-

\[ \text{NOTRY-Kex-Sim} \]
\[ \text{proofs can be simplified by constraining shared functionalities to}
\[ \text{a single common functionality} \]
\[ \text{the GUC framework more naturally allows us to express the deniability property. GUC-based}
\[ \text{proves it under the GUC model.} \]

\[ \text{NOTRY-Kex and NOTRY-Avow are not necessarily sequential. Parties can start}
\[ \text{avow even if they haven’t exchanged any keys. Of course, this}
\[ \text{attempt will either fail (generate an invalid proof) or be unconvinc-

\[ \text{NOTRY-Kex} \]
\[ \text{is guaranteed to maintain security in an environment where different}
\[ \text{protocol instances run concurrently, even if the protocol is}
\[ \text{composed with arbitrary protocols.} \]

\[ \text{EUC is an extended generalized universal composability (GUC) framework} [3, 17]. The GUC framework grants every entity in the model access to all the functionalities across composited protocol sessions. Therefore, it is regarded as a more convincing and accurate model of real-world settings. In addition, the GUC framework more naturally allows us to express the deniability property. GUC-based proofs can be simplified by constraining shared functionalities to a single common functionality \( \mathcal{G} \), in the EUC model. Canetti et al. [17] proved that security under the EUC model is equivalent to proving it under the GUC model.

The key observation of our proof strategy is that NOTRY-Kex and NOTRY-Avow are not necessarily sequential. Parties can start avowal even if they haven’t exchanged any keys. Of course, this attempt will either fail (generate an invalid proof) or be unconvincing (by not having any corresponding transcript to be avowed). Therefore, after defining ideal functionalities for NOTRY-Kex and NOTRY-Avow separately, proving NOTRY is secure requires proving NOTRY-Kex and NOTRY-Avow are UC-secure respectively. This special flavor of UC is called multi-protocol UC[15].

5.2 NOTRY Security Goals

NOTRY has the following security properties. We note that the first four are standard security notions for secure messaging and key exchange, whereas the fifth (mutually-agreed avowal) is novel to our setting:

(1) Universally composable AKE [22]: NOTRY-Kex-Ratchet realizes the ideal AKE functionality \( F_{\text{kek}} \) in the Ideal World in the UC framework [16]. UC-secure NOTRY-Kex-Ratchet enables the protocol to acquire arbitrary composability beyond the standard notions of an AKE protocol, which are mutual authentication, key privacy and freshness [7, 22].

(2) Deniability [30, 33, 39]: Transcripts in NOTRY-Kex can be efficiently forged with access only to the public parameters. Obviously, this deniability property demolishes the dependability of transcripts as proof of genuine key exchange protocol executions. A stronger notion of deniability, online deniability [33] guarantees that even a corrupted protocol participant (sometimes called an informant) cannot convince others of the authenticity of a transcript. NOTRY provides online deniability since NOTRY-Kex-Sim works without using Alice or Bob’s long-term secret key.

(3) Future secrecy and post-compromise secrecy [6]: The exposure of long-term secret key(s) cannot undermine the confidentiality of past or future session key(s) respectively (following recovery by the protocol). We achieve future secrecy through ratcheting with NOTRY-Kex-Ratchet, in which both parties update their contribution to \( k_{\text{sess}} \) generation. The
5.3 UC Security functionalities

In this section we outline our UC security functionalities, deferring full specifications to Appendix B. Our proof involves three ideal functionalities:

- \( \mathcal{F}_{\text{Avow}} \). For clarity and simplicity, we separate \( \mathcal{F}_{\text{Avow}} \) into two major functions, an ideal functionality for avowal proof generation and another one for avowal proof verification.
- \( \mathcal{G}^{\phi,n,\mathcal{G},q,h,g}_{\text{krk}} \), which models miscellaneous key registration with knowledge (krk) PKI and random oracles (ro). It’s parameterized by a set of protocols interacting with \( P \), a number of random oracles \( n \), a group \( \mathcal{G} \) of order \( q \) with two independently sampled generators \( h, g \). This is closely based on the global shared functionality defined in [68].

Ideal Functionality \( \mathcal{F}_{\text{Avow}} \). To prove \( \mathcal{F}_{\text{Avow}} \) is UC-secure within the EUC-framework, we compile an ideal functionality \( \mathcal{F}_{\text{Avow}} \) to emulate \( \mathcal{F}_{\text{Avow}}\)-Ratchet which captures all the security properties and features of this protocol. In addition to \( \mathcal{F}_{\text{Avow}}, \) shown in Algorithm 1, in the EUC-framework, there is an external environment \( Z \), intended to distinguish between a simulator \( \mathcal{S} \) in the Ideal World attacking the \( \mathcal{F}_{\text{Avow}} \) and a Real World adversary \( A \) attacking the real protocol. We denote dummy parties in the Ideal World as \( \mathcal{P} \), the corresponding real parties in the Real World as \( P \), and parties simulated by \( \mathcal{S} \) for \( A \) as \( \mathcal{P}^{\mathcal{S}} \).

Interactions between different entities in the context of EUC-framework are summarized as follows. First, \( Z \) is allowed to communicate with \( \mathcal{P} \) (or \( \mathcal{P}^{\mathcal{S}} \)) and to write the inputs to every party \( P \) (or \( \mathcal{P}^{\mathcal{S}} \)) in the Ideal World or the Real World and read their outputs. Likewise, \( A \) controls all the communication between every party \( P \) (or \( \mathcal{P}^{\mathcal{S}} \)) whereas \( \mathcal{S} \) is permitted to interact with \( \mathcal{F}_{\text{Avow}} \) under prescribed rules. Third, \( \mathcal{S} \) and \( A \) are allowed to corrupt parties while \( Z \) will be immediately notified of this corruption. Finally, the ideal functionality \( \mathcal{F}_{\text{Avow}, \mathcal{S}, A, \mathcal{P}, \mathcal{P}^{\mathcal{S}}} \) are able to interact with global shared functionality \( \mathcal{G}^{\phi,n,\mathcal{G},q,h,g}_{\text{krk}} \).

We model ratcheting by additionally introducing a round id \( rid \). NOTRY-Avow-Ratchet is guaranteed to execute in rounds even in the general concurrent operation since NOTRY-Avow checks the consistency of \( rid \) before setting the shared key.

Ideal Functionality \( \mathcal{F}_{\text{Avow}} \). For clarity and simplicity, taking advantage of the composability theorem, we defined a separate modular ideal functionality of avow \( \mathcal{F}_{\text{Avow}} \). \( \mathcal{F}_{\text{Avow}} \) consists of two major functions: \( \text{Avow} \) and \( \text{Judge} \). Note that, since \( \text{Judge} \) can be anyone to whom parties intend to avow, we therefore, consider \( \text{Judge} \) as part of \( \mathcal{F}_{\text{Avow}} \). So the primary task to initialize \( \mathcal{F}_{\text{Avow}} \) is to register a Judge and then publish her public key. This models that parties know the identity of the \( \text{Judge} \) in the Real World. Naturally, as \( \mathcal{F}_{\text{Avow}} \) plays as \( \text{Judge} \), it runs NOTRY-Avow-Sim\(\text{Judge} \) to produce transcripts for the simulator \( \mathcal{S} \) in the Ideal World.

Ideal Functionality \( \mathcal{G}^{\phi,n,\mathcal{G},q,h,g}_{\text{krk}} \), depicted in Algorithm 4, is defined to model publicly accessible features/functions like PKI or CA. The Share functionality models three primary cross-session states. First is PKI since we assume that each party learns the other’s long-term public key, as well as the key of the designated verifier judge \( pk_A \). Second, a random oracle is needed for KDF functions, signatures-of-knowledge, avowal, and secret sharing in NOTRY-Kex. Essentially, \( \mathcal{G}^{\phi,n,\mathcal{G},q,h,g}_{\text{krk}} \) combines the idea of \( \mathcal{G}_{\text{krk}} \) key registration of knowledge, from Dodis et al. [33] to distribute public keys and reveal private keys to corresponding corrupted party. In addition, we also integrated the shared random oracle \( \mathcal{G}_{\text{ro}} \) defined in Walfish [69] into \( \mathcal{G}^{\phi,n,\mathcal{G},q,h,g}_{\text{krk}} \) for our proof working on EUC-framework.
The non-information oracle functionality is defined to ensure our ideal \( F_{\text{kex}} \) is reifiable. Since \( F_{\text{kex}} \) outputs a random key in the Ideal World, even classic DH key exchange does not securely realize \( F_{\text{kex}} \) under an adaptive attacker. \( Z \) can easily distinguish when a corrupted party’s secret state is leaked to \( Z \) because there is no way for \( S \) to generate the counterpart’s transcript to match the random output key and leaked secret. To handle adaptive adversarial corruptions, Canetti and Krawczyk [22] proposed a special non-information oracle. \( N \) as part of \( F_{\text{kex}} \). \( N \) emulates a round of key exchange and exposes the secret state of the uncorrupted party to \( S \) once corruption happens. This helps \( S \) properly simulate a consistent key exchange under an adaptive adversary.

To capture online-deniable, Dodis et al. [33] observed that by relaxing the notion of fully adaptive adversaries to semi-adaptive adversaries we can achieve an online-deniable authentication given the PKI model. Semi-adaptive adversaries are restricted to corrupting a party only at the beginning of a protocol or at the end of it, but they can arbitrarily abort the protocol. After aborting, \( S \) gets the corresponding un-simulatable information from an incriminate procedure. This notion of deniability is known as key exchange with incrimination abort (KEIA), which assures deniability once the protocol is terminated by outputting a shared key. Note that if a protocol is executed without abort, it also implies perfect forward secrecy. This is because the session key is chosen randomly and it is independent of any information, including the protocol transcript.

5.4 UC-security proof sketch

Finally, we state our main security theorems and a proof sketch, deferring full details to Appendix C and Appendix D. We show a key lemma, that NOTRY-Avow is a ZKPoK scheme, in Section 5.4.1. Upon all the functionalities discussed above, we define UC-secure NOTRY-Kex and UC-secure NOTRY-Avow as following.

**Theorem 2.** NOTRY-Kex-Ratchet EUC-realizes \( F_{\text{kex}} \) under the erasure \( G^\text{pr,co,q,h,δ} \)-hybrid model with adaptive security, given the DDH assumption and access to a non-information oracle NonInfo\_NOTRY.

**Theorem 3.** NOTRY-Avow EUC-realizes \( F_{\text{avow}} \) under the erasure \( G^\text{pr,co,q,h,δ} \)-hybrid model with adaptive security given the DDH assumption.

**Proof sketch** To show that NOTRY-Kex-Ratchet actually EUC-realizes the \( F_{\text{kex}} \) is to show that for any PPT \( A \) attacking NOTRY in the Real World, there exists a corresponding PPT \( S \) in the Ideal World attacking \( F_{\text{kex}} \) such that the environment \( Z \) under the hybrid \( G^\text{pr,co,q,h,δ} \) model is unable to distinguish whether it’s in the Real World or Ideal World (Theorem 2, proof in Appendix C). The security proof (Appendix B) of Theorem 3 follows the same paradigm described above.

5.4.1 NOTRY-Avow is a secure MADV avowal.

We now present our proof that our NOTRY-Avow is a ZKPoK scheme, a key component of proving Theorem 3.

**Lemma 1.** NOTRY-Avow is a secure avowal of MADV proofs under Decisional Diffie–Hellman (DDH) in the random oracle model.

**Remark 1.** For simplicity, here we discuss single-message avowal rather than multi-message avowal. But our argument still holds in the latter case.

We divide proof of Lemma 1 into the following three lemmas:

**Lemma 2.** NOTRY-Avow MADV avowal proof is complete.

**Proof.** We argue the correctness of MADV avowal by examining the verification process. Since \( R_j \) is the simulator’s output, \( Judge \) can simply recompute it again to get the identical value. For \( R_{\text{AB}} \), first observe that \( h^{x_{\text{AB}}} \) can be expanded to \( h^{x_{\text{AB}}(1+y)+r_{\text{AB}}} \) and \( AB^{x_{\text{AB}}} \) equals to \( h^{(1+y)x_{\text{AB}}} \). After dividing \( h^{x_{\text{AB}}} \) with \( AB^{x_{\text{AB}}} \), \( Judge \) gets \( h^{x_{\text{AB}}} \), which is identical to \( R_{\text{A}} \times R_{\text{B}} \), for \( R_{\text{A}} \leftarrow h^\text{h_{\text{A}}} \), \( R_{\text{B}} \leftarrow h^\text{h_{\text{B}}} \). The random oracle will output an equivalent value as \( Judge \) and proves request with the same values. Therefore, verification will always pass for correctly generated proofs.

**Lemma 3.** There exists a PPT extractor \( E \) for NOTRY-Avow MADV avowal can extract a valid witness while \( A \) is unable to get the witness from a genuine ADV proof \( \pi \).

**Proof.** We construct a knowledge extractor to demonstrate how to get secrets \( \alpha, \beta \) with the forking lemma[60]. Note that \( E \) works the same way to extract both secrets. Suppose that \( E \) can rewind a verifier in the NOTRY-Avow to have the challenge \( c \) in two proofs. The extractor \( E \) will run NOTRY-Avow twice to extract \( \alpha \). In the first round, upon observing that Alice outputs her partial proof \( z_{\alpha} \), the \( E \) records the partial proof \( z_{\alpha} \) and reboots Bob to compel him to output the same \( c \). In this case, Alice will generate two proofs for the same challenge \( c \). With two partial proofs \( z_{\alpha}, z_{\beta} \) received from Alice, to get secret \( \alpha \), \( E \) first collects two common challenge values running NOTRY-Avow \( c_{\alpha}, c_{\beta} \). Now \( E \) can derive \( \alpha \) by dividing \( z_{\alpha} - z_{\beta} \) with \( c_{\alpha} - c_{\beta} \). Observe that \( z_{\alpha} - z_{\beta} = (c_{\alpha}α + r_{\alpha}) - (c_{\beta}α + r_{\beta}) \) equals to \( c_{AB}α - c_{AB}α \). So \( α \leftarrow (z_{\alpha} - z_{\beta})/(c_{\alpha} - c_{\beta}) \).

Avowal soundness can be trivially satisfied for NOTRY-Avow. Even though the witness of avowal proof serves as a partial witness to the ADV proof, \( A \) learns nothing about the ADV witness from a zero-knowledge ADV proof \( \pi \).

**Remark 2.** To illustrate the idea, we follow the classical way of constructing the extractor \( E \) with the rewinding technique. However, since rewinding is impermissible in the UC model, straight-line compilers, which are extensively studied in the literature[37, 40, 45, 54, 62], can be directly applied to NOTRY-Avow to transform this special \( \Sigma \)-OR proof to a UC-secure one.

**Remark 3.** A global random oracle is not enough to enable a straight-line compiler in the UC model[18]. To extract the witness, a knowledge extractor \( E \) without rewinding gets a special power: oberserving all requests to the random oracle and their responses. We, therefore, adopt restricted observable global random oracles[19]. Also, note that since we apply the Fiat–Shamir transform to get a non-interactive protocol, this special flavor of random oracle is also needed for special-soundness for both \( \pi \) and \( \pi' \).

**Lemma 4.** NOTRY-Avow MADV avowal is zero-knowledge under DDH/RO and a semantic-secure encryption scheme.

**Proof.** Since it is a designated verifier proof, \( Judge \) can easily simulate a correct transcript. Note that this doesn’t damage generalization because anyone can serve as a \( Judge \). Therefore, we can construct a simulator NOTRY-Avow-Sim\_\text{Judge} (abbreviate it to
We designed our protocol for use in real-world systems and accordingly developed a proof-of-concept implementation, which we evaluate here. An anonymized review-ready copy of our code is available at https://github.com/xxsqwe/notry.

Our NOTRY implementation is based on the official Rust implementation of Signal[53] and is designed to achieve an equivalent security level (128 bits). In particular, we use the same Curve 25519 [9] as Signal. Our implementation extends the Signal implementation to support both single- and multiple-message transcript avowal. All experiments were performed on an Intel 12th generation core i7-12700K pinned to 3.6GHz with 32GB RAM.

Our goal in evaluation is to determine the performance penalty incurred by implementing NOTRY over a non-avowable secure messaging protocol. We, therefore, compare it against Signal, which represents the best-in-class for secure messaging with deniability. Additional overhead is paid for a UC-secure Signal [10]. Therefore, we implement NOTRY without a straight-line compiler for a fair comparison.

### 6 REAL WORLD EVALUATION

#### 6.1 Results

First, we profile the computation and communication overhead in the key exchange stage. This gives an indication of the overhead that NOTRY would exhibit in regular use.

As session keys in Signal are generated by the X3DH protocol, which employs double ratcheting with its accompanying performance costs, we implement a similarly ratcheted NOTRY key exchange. We collected our performance results from 10,000 experimental evaluations on our NOTRY-Kex implementation and Signal benchmarks, summarized in Table 1. While NOTRY incurs an approximately 4x communication and 8x computational overhead as compared with the Signal protocol, the absolute time taken per key exchange (~1ms) is still negligible—demonstrating that the protocol may be sufficiently performant for eventual use in production systems.

We also evaluate the cost of performing avowal, both for the communicating parties and for the Judge. Our protocol is designed to allow any server to play the role of a Judge and the cost of verification is therefore not a bottleneck for the protocol.

We measured the cost of running the avowal with 1, 10, 100, and 1000 transcripts respectively, and tabulate our results in Figure 7. We find that the communication and performance costs to the communicating parties are roughly constant, and scales linearly for the Judge. While it is unsurprising that the computational burden falls to the Judge overall performance costs are both small and linear in the number of transcripts to avow, and therefore pose a lesser deployment concern.

| Table 1: NOTRY and Signal performance evaluation for key exchange |
|--------------------------|--------------------------|--------------------------|
|                          | Signal                   | NOTRY                    | Ratio  |
| Key Generation [ms]      | 0.011±0.001              | 0.044±0.002              | 4×      |
| SoK [Bytes]              | -                        | 128                      | N/A     |
| SoK Verify [ms]          | -                        | 0.230±0.01               | N/A     |
| Key Exchange [ms]        | 0.151±0.03               | 1.21±0.05               | 8×      |
| Rounds                   | 2                        | 2                        | 1×      |
| Public key [Bytes]       | 32                       | 32                       | 1×      |
| Prekey [Bytes]           | 128                      | -                        | N/A     |
| SoK [Bytes]              | -                        | 256                      | N/A     |
| Key Exchange [B/key]     | 80                       | 335                      | 4×      |

#### 7 DISCUSSION AND CONCLUSIONS

We identified an interesting potential weakness with existing deniable messaging applications: sometimes both participants in a conversation initially meant to be deniable may genuinely wish to remove deniability and avow the conversation at a later time. To overcome this limitation, we developed a novel notion, ADV proofs, and applied a practical ADV proof construction to construct NOTRY. To the best of our knowledge, this is the first DAKEA protocol that retains all the critical security properties of DAKE protocols. Our evaluation shows that NOTRY protocol is also a reasonably efficient DAKEA protocol in practice.
We thank the anonymous reviewers of CCS 23 and PETS 23 for valuable suggestions. We also thank our PETs Shepherd for helpful feedbacks. Faxing Wang is supported by Melbourne Research Scholarship. Riad Wahby and Joseph Bonneau are supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

ACKNOWLEDGMENTS
We thank the anonymous reviewers of CCS 23 and PETS 23 for valuable suggestions. We also thank our PETs Shepherd for helpful feedbacks. Faxing Wang is supported by Melbourne Research Scholarship. Riad Wahby and Joseph Bonneau are supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

REFERENCES
A MADV PROOFS $\tilde{\pi}$

A.1 Secure MADV proofs $\tilde{\pi}$

**Definition 4.** A secure MADV proof $\tilde{\pi}$ consists of a triple of algorithms $\{\text{Setup}, \text{Avow}, \text{Judge}\}$ with the following properties:

1. **Completeness:** Any prover who generated an MADV proof $\tilde{\pi}$ with knowledge of $W$ satisfies:

   $$\Pr \left[ 1 \leftarrow \text{Verify}(\tilde{\pi}, \tilde{S}) \mid \tilde{\pi} \leftarrow \text{Gen}(\tilde{S}, \tilde{W}) \right] = 1$$

   Completeness indicates that Judge always accepts the MADV avowal proof $\tilde{\pi}'$ from an honest prover.

2. **Special Knowledge Soundness:** There exists a PPT extractor $E$ which, given two transcripts of avowal proof $T_1$ and $T_2$, can extract a valid witness:

   $$\Pr \left[ 1 \leftarrow \text{Verify}(\tilde{\pi}, P) \mid \tilde{W}' \leftarrow E(1^\lambda, T_1, T_2) \right] = 1 - \text{negl}(\lambda)$$

   where $W'_P$ is the extracted witness. Special soundness implies that any party who generates a valid ADV proof $\tilde{\pi}$ must gain the knowledge of the witness $W_P$ to the statement $P$.

3. **Honest Verifier Zero-Knowledge:** There exists a PPT simulator $S$ that generates an MADV proof $\tilde{\pi}_S$ without knowledge of the witness $w$ such that $1 \leftarrow \text{Judge}(\tilde{\pi}_S, \tilde{S})$. $\tilde{\pi}_S$ holds that:

   $$\tilde{\pi}_S \approx_e \tilde{\pi}$$

A.2 Construction of MADV proofs

Here, we detail the proof generation and verification protocol. Note that MADV proofs are designed under the context of *Signature of Knowledge* in NOTRY-Kex, so we instantiate the Signature of Knowledge with the Schnorr identification scheme and the Schnorr signature scheme.

**Classical Schnorr Identification Scheme:** [64] for example, targeting on proving knowledge of $y \leftarrow \text{dlog}_A g$, given global parameters $\Theta, g, a, h$. Alice first samples $r \overset{\$}{\leftarrow} \mathbb{Z}_q$ and computes $a = g^r \mod P$.

Next, she applies the Fiat-Shamir transform to get a non-interactive proof, setting $c = H(a||A||h)$. Last, she sends the proof $(a, z = r + cy)$ to Bob. Bob verifies the proof by first reconstructing the random challenge $c' = H(a||A||h)$, then checking if $h^c \overset{?}{=} a \cdot A^{c'}$.

**Remark 4.** Challenge $c$ will be generated in another way when applying a straight-line compiler to bootstrap the protocol to a UC-secure one. Briefly, given a hash function $H_{\lambda}$, Bob only accepts a proof iff the $H_{\lambda}(a||c||z||A||h)$ starts with $x$ bits long leading zeros and passes verification. In this case, given a commitment $a$, Alice is supposed to try multiple potential challenges until she gets a valid challenge $c$. Also, since $H_{\lambda}$ outputs $x$ long digest, where $x \cdot y = \lambda$, so to achieve the $\lambda$-bit level of security, Alice has to generate $y$ proof transcripts $\{(a_i, z_i)\}_{i \in [y]}$ and ensure that the corresponding hash of every proof $H_{\lambda}(a||c||z||A||h)$ has $x$ leading zeros, where $a = (a_1, \cdots, a_i)$. Alice wraps $(a, c, z)$ up as the Schnorr proof under the straight-line compiler.

**Schnorr AND Proof:** proving an AND clause via Schnorr protocol is to generate two proofs for the two clauses respectively. For example, to prove $\text{dlog}_A g \wedge \text{dlog}_A p_k A$, Alice generates the Schnorr proof to $\text{dlog}_A g$ as we discussed before. Second, in the same way,

---


Alice generates the Schnorr proof \((\tilde{a}, \tilde{z})\) to \(\text{dlog}_A pk_A\). Alice sends two proofs \(((a, z), (\tilde{a}, \tilde{z}))\) to Bob who verifies two proofs individually as the basic scheme.

**Schnorr OR Proof:** [28] for the OR clause, i.e. \(\text{dlog}_A A \lor \text{dlog}_B B\), since Alice has non-knowledge of \(\text{dlog}_B B\), she will simulate a proof for \(\text{dlog}_A A\). To get a simulated proof, Alice first samples \(c_B, z_B \leftarrow \mathbb{Z}_q\), then she finalizes simulation with \(a_B \leftarrow g^{z_B}/B^{c_B}\). For the proof to \(\text{dlog}_A A\), Alice generates \(r_A, a_A\) as the basic Schnorr scheme. Second, she gets the challenge value \(c \leftarrow H(a_A || b_B | A || B | h)\), computes her really challenge \(c_A \leftarrow c \oplus c_A Y\) as the basic scheme. Alice sends two proofs \((r_A, z_A, r_B, z_B)\) to Bob. The verifier Bob first reconstructs \(a'_A \leftarrow g^{z_A}/A^{c_A}, a'_B \leftarrow g^{z_B}/B^{c_B}\). Bob accepted a proof if \(c_A @ c_B = H(a'_A || a'_B || A || B || h)\), otherwise rejects it.

### A.3 MADV proofs in NOTRY are ZKPoK

**Lemma 5.** The instantiation of the MADV proof in NOTRY-Kex is complete, special sound, and zero-knowledge under DDH/RO assumption.

**Remark 5.** Every security property of an SoK scheme is captured by its corresponding property of MADV proofs defined in Appendix A. Correctness is defined via Completeness, Simulatability is defined by Zero-knowledge, and Extraction is defined via Special Knowledge Soundness.

**Proof.** Completeness: The MADV proof in NOTRY-Kex consists of four individual Schnorr proofs with respect to four clauses of the \(\overline{S}_A\) in Equation (3.8). Specifically, Alice generates the MADV proof (also signature of knowledge) \(\overline{\pi}\) with \(y \leftarrow \text{dlog}_A A\) and her private key \(sk_A\) where

\[
\overline{\pi} = \{(a_A, c_A, z_A), (a_B, c_B, z_B), (a_j, c_j, z_j), (a_{AB}, c_{AB}, z_{AB})\} \quad \text{(A.4)}
\]

\(\overline{\pi}_A\) and \(\overline{\pi}_{AB}\) are simulated proofs. It’s trivial to tell that all four proofs will pass verification as each proof is a Schnorr proof.

**Special soundness:** Two accepting proofs with the same first commitment are enough to retrieve the knowledge under the Schnorr protocol. With rewriting, the knowledge extractor \(E\) works similarly as the \(E\) in NOTRY-Aow. The only distinction in Schnorr-OR proofs is that \(E\) is supposed to output an additional bit to indicate which clause was proved. Taking \(\text{dlog}_A A \lor \text{dlog}_B B\) as an example, \(E\) rewrites Alice to the point where she asks the random oracle to generate two accepting proofs: \((a_A, c_A, z_A), (a_B, c_B, z_B)\) and \((a_A, c'_A, z'_A), (a_B, c'_B, z'_B)\). If \(c_A \neq c'_A\), \(E\) outputs 0 to indicate that \(\text{dlog}_A A\) is proved and yields \(y\) from \(\overline{\pi}_A\), \(\overline{\pi}_A\) the same way as the \(E\) in Lemma 3.

On the other hand, consider applying the straight-line compiler to MADV awoval proofs in Remark 4, \(E\) will get to search all the queries that Alice made to the random oracle. Since rewriting is not allowed in the UC model, observable random oracle, defined in Algorithm 4, exposes an additional interface observe-RO to \(E\) for extracting the witness. After receiving \(\{\overline{\pi}_1, \overline{\pi}_2\}\) from Alice, \(E\) first parses \((a_i, c_i, z_i)\varepsilon_{[y]} \leftarrow \overline{\pi}\) and sets \(a = (a_i)\varepsilon_{[c_i]}\). Second, \(E\) sends observe-RO(Alice) to the random oracle and gets back all the requests Alice made. Finally, \(E\) finds two requests \(i, j\) that share the a but \(c_j \neq c_i\) and \(z_j \neq z_i\) and pass Schnorr-OR verification. By the forking lemma[60], \(E\) extracts the witness \(y\) from \((z_j - z_i)/(c_j - c_i)\).

**Honest verifier zero-knowledge:** we give a simulator \(S\) that simulates indistinguishable proofs from real proof \(\overline{\pi}\). For \(\overline{\pi}\), note that the Schnorr proof is just a concatenation of two Schnorr-OR proofs. Therefore, we briefly describe the simulator for Schnorr-OR proof. Given \(A, \delta \leftarrow \text{dlog}_B B\), Simulator \(S\) first samples \(c_{AB}, z_{AB}, r_{BS} \leftarrow \mathbb{Z}_q, a_{BS} \leftarrow h^{r_{BS}}/A^{c_{AB}}\). Now \(S\) derives \(c_S \leftarrow H(a_{BS} || b_B || A || B || h), s_{BS} \leftarrow c_S @ c_{AB}, \) finally generates \(z_{BS} \leftarrow r_{BS} + e_{BS} \delta\). It’s trivial to tell that simulated proof \(\overline{\pi}_{BS}\) shall pass the verification. Note that the special power the \(S\) gets is knowledge of \(\text{dlog}_B B\) which is unknown to Alice.

We proceed with showing that the simulated proof \(\overline{\pi}_{BS}\) is indistinguishable from the MADV proof \(\overline{\pi}\). \(\overline{\pi}_{BS} \approx \overline{\pi}\). \n
**Hybrid 0.** starting with \(\overline{\pi}_S = ((a_{AS}, c_{AS}, z_{AS}), (a_{BS}, c_{BS}, z_{BS}))\).

**Hybrid 1.** consider a proof \(\overline{\pi}^1_S = ((a_A, c_A, z_A), (a_{BS}, c_{BS}, z_{BS}))\).

**Hybrid 2.** consider a proof \(\overline{\pi}^2_S = ((a_A, c_A, z_A), (a_B, c_B, z_B))\).

With \(a_{BS}\) depends on \(z_{BS}\), which is unknown to Alice.

**B FUNCTIONALITIES**

To prove our NOTRY in the UC model is to construct a simulator \(S\) that emulates all the possible transcript a Real World adversary \(A\) can produce. Note that \(S\) executes \(A\) internally. Therefore, any message exchanged between \(A\) and \(Z\) is through a communication channel by \(S\). Also, messages \(S\) received from \(Z\) are copied to \(A\). For a party \(P\) in the Ideal World, \(S\) simulates a corresponding party \(P^S\) in the Ideal World for \(A\). \(A\) is a fully adaptive adversary with arbitrary behaviors. \(S\) will compromise \(P\) accordingly after \(A\) corrupts a simulated party \(P^S\).

Algorithm 1 shows the Ideal functionality constructed for NOTRY-Kex-Ratchet. Parties join the NOTRY-Kex-Ratchet by \(Z\) sending them init or responde message. A party with init message is called initiator \(I\) and the one responde is the responder \(R\). Before joining
NOTRY-Kex-Ratchet protocol, all parties are expected to get themselves registered to the $G_{rk}\in\mathbb{G}$ with their identity key pairs, $pk$ and $sk$. To simplify simulator construction, $F_{kex}$ guarantees that the responder message from $R$ will not be handled until $S$ sees a $init$ message. $S$ controls delivery of the final shared secret key via an $ok$ request. $S$ later individually delivers the key to parties in the Ideal World. We model post-specified peers by allowing parties joining NOTRY without declaring their intended partners and $S$ specifying the remote identity when delivering the shared secret. $S$ can arbitrarily deliver the secret if the corresponding party is corrupted.

We model constructiveness by decoupling a non-information part from $S$. The initial analysis on universally composable key exchange protocol [22] observes that the ideal functionality $F_{kex}$ simply outputs a random key cannot be realized in DH family protocols. This is because an adaptive adversary can corrupt one of two participants before the final message flow to get its ephemeral states. Based on that $Z$ can distinguish the protocol in the Ideal World from the Real World. NonInfNOTRY, depicted in Algorithm 5, is defined to be part of the ideal functionality $F_{kex}$ to capture the notion of AKE under the Universal Composability Model.

The idea of NonInfNOTRY is to provide $S$ with a secret internal state to simulate a real protocol. The idea of constructing this functionality is to generate both secret components and then send their corresponding public components to the NonInfNOTRY caller $M$. $M$ is the $F_{kex}$ when it is asked to hand a key to a pair of parties. $M$ can accept or reject(by sending ok message through $F_{kex}$) the proposed keys. If $M$ agrees, NonInfNOTRY completes the key exchange process and locally outputs the final shared key. NonInfNOTRY discards them when $M$ rejects and acts as a responder to finish the key exchange by accepting one public component proposed by $M$. It’s necessary to achieve initiator resilience for $S$ still to be capable of finalizing the protocol on behalf of the remaining honest party. In addition, NonInfNOTRY also maintains an interface for $S$ to generate SoK messages since internal secret data are all coming from IncProcNOTRY-Kex.

According to the composability property, we divided our security proof of NOTRY into two parts: proof of NOTRY-Kex-Ratchet and proof of NOTRY-Aw. We argue that by proving NOTRY-Kex-Ratchet and NOTRY-Aw are UC-secure respectively we can conclude that NOTRY is UC-secure.

C PROOF OF NOTRY-KEX-RATCHET

C.1 Simulator construction

We denote three messages in a round NOTRY-Kex-Ratchet as $m_{kex}^1$, $m_{kex}^2$, $m_{kex}^3$. Other than the initial round, only $m_B$ or $m_A$ is needed. If any messages sent from $A$ in the context of the simulated environment are unrelated to NOTRY-Kex-Ratchet, they will be discarded by $S$. If messages are delayed delivery by $A$, then $S$ just waits until messages are delivered. NOTRY-Kex-Ratchet is secure against fully adaptive corruptions. $A$ is allowed to corrupt any party arbitrarily.

$S$ begins simulation by initializing the protocol via a message ($setup, G, g, g, h$) to NonInfNOTRY through $F_{kex}$. NonInfNOTRY generates messages for $S$ to simulate and sends back ($exchange, A, B$) to $S$. $S$ stores the ephemeral values $A, B$ for future use.

$S$ holds until being activated by an ($init, sid, I, \Phi$) from $F_{kex}$. After that, $S$ compiles a $m_{kex}^1$ for $I^*$ with the material $A$ received from NonInfNOTRY. $S$ broadcasts $m_{kex}^1$ through $A$ as if it was a messages sent by $I^*$.

After getting a ($responded, sid, rid, R, \Phi$) from $F_{kex}$, $S$ first checks the transmission status of $m_{kex}^1$. Since $F_{kex}$ only sends a responded message after it has already gotten an $init$ message, $m_{kex}^1$ must have been sent and delivered in the simulated environment. $S$ aborts the protocol once $m_{kex}^1$ is not in its format or invalid and marks $R$ as aborted at once. If $m_{kex}^1$ is valid then $S$ constructs the corresponding message $m_{kex}^2$ for $R$ in response to $m_{kex}^1$.

- If $m_{kex}^1$ was created by $S$ before or was from a corrupted party $I^*$, then based on the messages got from NonInfNOTRY $S$ constructs the message $m_{kex}^1$. $S$ sends a proof request with the identifier of $R$ to NonInfNOTRY. It’s NonInfNOTRY that generates the corresponding signature of knowledge $\delta_B$ by $y, x$, then $S$ setup the message $m_{kex}^1$ as $(B, \delta_B)$. Note that in the case of corruption, $S$ will send a (complete, false $A$) to NonInfNOTRY to indicate that its transcript has been rejected. $F_{kex}$ immediately sends internal states of NonInfNOTRY to $S$. $S$ sends ($ok, sid, rid, 0$) to $F_{kex}$, causing it record the output from NonInfNOTRY as the shared key.

- If the received message $I^*$ is not identical to the message $m_{kex}^1$ created by $S$ before, $I^* \neq A$, trivially we can see that $m_{kex}^1$ must have been altered by $A$ during transmission. Therefore, $m_{kex}^2$ will be constructed from IncProcNOTRY-Kex. $S$ sends ($abort, rid, sid$) to $F_{kex}$ while withold delivery of the abort message to both $I$ and $R$. Next, $S$ sends ($incrate, sid, rid, sid$) to $F_{kex}$ to invoke IncProcNOTRY-Kex. Based on $m_{kex}^1$, $S$ sends ($inc, sid, R, A^*$) to the IncProcNOTRY-Kex process and waits response ($inc, sid, B, \delta$). According to the response from NonInfNOTRY $S$ compiles $m_{kex}^2$ as $(B, \delta)$.

$S$ sends $m_{kex}^2$ through $A$, as if $R^*$ responses to $I^*$ and sends ($deliver, sid, rid, l, P$) to $F_{kex}$ to indicate that $I$ generated the shared key.

After receiving message $m_{kex}^2$ from $A$, $S$ first checks that if $I^*$ has previously broadcast the message $m_{kex}^1$. The coming message $m_{kex}^2$ will be ignored if no matching $m_{kex}^1$ is found. If a corresponding record of $m_{kex}^1$ found, $S$ parses $m_{kex}^2$ to get $(B, \sigma_B)$. $S$ will abort the protocol by sending ($abort, sid$) to $F_{kex}$ if $m_{kex}^1$ is not in an expected form, or $m_{kex}^2$ is not matching the previous message $m_{kex}^1$. Meanwhile, $S$ delivers the abort message to $I$ and withholds abort message to $R$.

Otherwise, $S$ constructs $m_{kex}^2$, which is $\sigma_A$, to emulate the message from $I$ to $R$. In addition, $S$ will set up the shared secret key. Depending on the situation of protocol emulation, $S$ constructs $m_{kex}^2$ and outputs a shared key in the following way:

- If $A$ has already corrupted $R$, then the corresponding party $R^*$ must have been corrupted by $S$ as well. $S$ simply gets $sk_B$ from $G_{rk}^{\in\mathbb{G}}$ with a retrieve secret request. Next, $\sigma_A$ is returned from NonInfNOTRY by requesting it with ($prove, l$). Finally, $I$ is supposed to output a session key with respect
Algorithm 1 Ideal Functionality of NOTRY $F_{kex}$

$F_{kex}$ proceeds with security parameter $\lambda$ in the $G_{krkro}^{0, n, q, h, g}$-hybrid model with parties $P$ and $P'$ and an ideal adversary $S$. When initializing, $F_{kex}$ calls NonInfoNOTRY $\mathcal{N}$ with fresh randomness.

1. **On** interaction with $N$:
   2. Forward messages from $S$ to $N$. If one of the two is corrupted or $R$ is "aborted" while $N$ has generated its output, reveal the state of $N$ to $S$. Session id is denoted as sid, round id is denoted as rid, $\phi$ is session state.

3. **On** (init, sid, I, aux, $\phi$) from $P$:
   4. if sid is incriminated return
   5. if I ≠ P return
   6. if sid is in null
      7. mark I as the initiator, "active"
      8. mark sid as initialized
      9. record(sid, I) as I
      10. Send(init, sid, I, $\phi$) to $S$

4. **On** (responde, sid, R, $\phi$) from $P$:
   5. if (sid is not in initialized) return
   6. if (R ≠ P) return
   7. if (the session is null) hold until being initialized
   8. if sid is initialized
      9. mark R as responder
      10. mark sid as responded
      11. init rid with a random value and mark sid as NULL
      12. record(sid, rid, R) as II
      13. Send(responde, sid, rid, R, $\phi$) to $S$

5. **On** (ok, sid, rid, k) from $S$:
   6. if (the record III of (rid, sid, K) found) return
      7. /Test the key of this round is ready or not, only allow to set
         /it once.
   8. if ((I ∧ R) are corrupted) return
   9. if both are corrupted, simulation is handled by the A.
   10. if (I ∧ R are uncorrupted)
      11. $K \leftarrow \{0, 1\}^\lambda$
      12. /If both are honest, they output a random key, this captures
         //both PCS and forward secrecy

13. Else if (R is corrupted)
   14. [K be the local output of $N$]
   15. //This captures initiator resilience, whereas the output key
      //is random or the local output of $N$ if $R$ is honest.
   16. else { $K \leftarrow k$ }
   17. record(sid, rid, K) as III
   18. **On** (deliver, sid, rid, $P$, $P'$) from $S$:
      19. if (sid is not "responded" ∨ no record III related to ($P$, $P'$) return
      20. if (a set-key have been sent)
         21. mark rid as "exchanged"
         22. Switch roles (I, R) ← (I', R')
         24. Inc round number rid ← rid +1
         25. //For multi-round key exchange
      26. **On** (abort, rid, sid) from $S$
         27. Send delayed (abort, sid, I) to I
         28. Send delayed (abort, sid, R) to R
      29. **On** (incriminate, sid, rid) from $S$
         30. if (rid is "incriminated" ∨ "exchanged") return
         31. if ((R is "aborted") ∧ (I is "active") ∧ (R is uncorrupted))
            32. Send(retrieve-secret, R) to $G_{krkro}^{0, n, q, h, g}$
            33. $sk_R \leftarrow G_{krkro}^{0, n, q, h, g} - retrieve-secret(R)$
            34. Execute IncProc(sid, I, R, $pk_I, pk_R, sk_R$)
            35. **mark** sid as incriminated

36. S sends $m^kek_A$ through $A$ as if $I^0$ sent it to $I^0$. 

When $R^0$ gets back the SoK $A'$ of $I$, $S$ examines that if $R$ has already saw message $m^kek_B$ from $I^0$ and it has sent a corresponding response $m^kek_A$. $m^kek_A$ will be ignored if either of the above requirements failed. $S$ validates $\sigma_A$ and then outputs the distributed key.

- Otherwise, SoK $\sigma_R$ is valid iff it's generated with the knowledge of the corresponding secret key $sk_R$ or discrete log of $AB$ to the base $q$. Furthermore, $\sigma_R$ is fully bounded to this instance of NOTRY-Kex-Ratchet because it depends on the exchanged message A, B. In this case, we can conclude that $S$ generated the message $m^kek_B$ and $P = R^0$. So $S$ get the

- $\sigma_A$ fails verification, which implies the $\sigma_A$ might be altered by $A$, or it does not match previous messages. In this case, $S$ sends (abort, sid) to $F_{kex}$ and delivers abort message to R at once while withholding the abort message to I.

Algorithm 3 Incriminate Procedure IncProcNOTRY-Kex

Subroutine IncProcNOTRY-Kex(sid, I, R, skR)

1. On (inc, sid, I, R, Q, g, h, A) from S:
   1. Generate δ ← Zq, B ← gδ
   2. σR ← SoK-Sign(A, B, σ, δ, skR)
   3. Send(inc, sid, B, σ) to S
   4. //As working as B as contributing his share to the exchanged key

• If σA is valid, S outputs a key for R as well. A secure valid SoK indicates that m_kex^A is sent from a corrected I or protocol executed honestly. In both cases, S has already instructed F_kex to record a session key. S completes by sending (deliver, sid, rid, R, I) to F_kex to sends indicate that R generated the shared key.

When A instructs I to broadcast m_kex^I, but S has not gotten an init message from F_kex yet, then S awaits I to be activated by Z with (init, sid, I, ⊥, ⊥) to F_kex for simulation but discards any init message from F_kex. When A instructs R to issue message m_kex^R while S has not received a responded message yet, then S awaits R to be activated by Z with (responde, sid, ⊥, R, ⊥) and discards any responded from F_kex.
Algorithm 5 Non-Information Oracle NonInfoOracle N
Subroutine NonInfoOracle
1: On setup, q, h, g, l from M:
2: if (the first setup message)
3: parse q, h, g, l as group q, prime order h, and generator
4: else discard parameters
5: generate (y, k) ← Zq, (A, AB, B) = (h^y, g^y, AB/A)
6: record("setup", q, h, g, l, y)
7: Send(exchange, A, B, q, h, g, l) to M
8: On (complete, ok, A, σA, σB) from M:
9: if (not setup yet) return
10: if (ok is True ∧ σA ∧ σB)
11: set K = KDF (A || B || σA || σB || B^y) for randomness
12: else if ((A / q) OR (A is an identity element)) return
13: else
generate new κ ← Zq, AB ← g^κ, B ← AB/A
14: generate σA and σB with (y, k)
15: set K = KDF (A || B || σA || σB || g^yk/AY)
16: locally output K
17: On (prove, P, skx) from M:
18: if (not setup) return
19: if (P ≠ (initiator, responder)) return
20: if (skx) σ ← SoK(A, B, pkB, AB, y, sky)
21: else generate σ ← SoK(A, B, pkB, AB, y, k)
22: Send(proof, σ) to M

C.2.2 Dishonest cases.

- Modified m^kex: this happens when m^kex from S is modified by A in transmission, but neither I^0 nor R^0 is corrupted until m^kex is delivered. In this case, S generates σB by IncProcNOTRYKex. IncProcNOTRYKex generates σB as the way A generates it in the Real World.
- Modified m^kex: this happens when A modifies m^kex generated by S but both of them are not corrupted when m^kex is transmitted. I^0 will immediately know the modification after it checks σA ⊆ m^kex since the verification will fail. This is also identical to the case in Real World of I.
- Modified m^kex: this happens when A modifies m^kex from S while neither I^0 nor R^0 are corrupted when m^kex is delivered. S instructs R^0 to abort and R in the Real World will abort too since σA is unable to pass the verification.

So the transcript output by S in this case will be (A^I1, B^I1, σA^I1, σA^S) where only σA^S is generated through proof simulation.

C.2.3 Corruption. This is the case when one of the parties is corrupted during times that are not covered in dishonest cases. In terms of transcript, the advantage of corruption is access to the secret key. Therefore, the corresponding SoK, say σA for I^3, is generated by taking the secret key as one of the two witnesses in the SoK statement.

When both parties are corrupted, S does nothing but copy any outputs from parties in the Ideal World to parties in the Real World. So there is no simulation at all, for Z to distinguish.

The simulated transcript of S, in this case is (A^S, B^S, σA^2, σB^2) where one of two SoKs is generated with the secret key of the corrupted party.

C.2.4 Hybrid Arguments. We start hybrid arguments with the transcript in the honest case.

Proof. (Hybrid 0) is the transcript T^kex_0 = (A^S, B^S ← (AB^y))/σA, σB), where both SoKs are simulated.

Hybrid 1: replace σB in T^kex_1 with σB to get T^kex_1. T^kex is indistinguishable from T^kex because Z is unable to distinguish a simulated valid SoK(without secret key) from a real SoK(with secret key). This is what S simulated in the Dishonest cases.

Hybrid 2: rather than replace σB in T^kex_1, replace σA with σA^2, generated with knowledge of the secret key, to get T^kex_2. The same argument in T^kex_3 can be applied to T^kex_2 and T^kex_3. It’s obvious to see that T^kex and T^kex_2 capture the output of S in the corruption case. That is, one of two SoKs is not simulated.

Hybrid 3: replace σA in T^kex_3 with σA^2, which is generated with the knowledge of the secret key, to get T^kex_4. Note that the only distinction between T^kex_3 and T^kex_4 is that σA^2 in T^kex_4 is proven by knowing the corresponding secret key whereas in T^kex_3 is simulated. Again, Z is unable to tell a simulated SoK from a real SoK to the same statement. So T^kex is indistinguishable from T^kex_4. Observe that this is exactly the case in the Real World execution.

All ’A’s and ’B’s in all the transcript(T0-3) are just DH tuples, since we assume DDH holds, all the ’A’s and ’B’s in all the transcripts should be indistinguishable from each other. That, we conclude that Z is unable to distinguish S in the Ideal World from A in the Ideal World. □

D. PROOF OF NOTRY-AVOW

D.1 Simulator construction

Proof. For convenience, we denote the three messages in NOTRY-Avow as (m_1, m_2, m_3). First, Z activates S to initialize the NOTRY-Avow protocol by sending a setup to N through F^kex and waits for an exchange response. After receiving an exchange message from N, S and A finish another round of NOTRY-Kex-Ratchet to acquire a temporary session key for future use in NOTRY-Avow. Z instructs parties to start avowal by sending Avow message to F^avow. Note here we assume the identity of Judge is common knowledge since we modeled Judge as part of the NOTRY-Avow. The arbitrary inputs of parties, [secret]s, and public components [AB]s are from Z.
F_{avow} holds only after it gets another avow message. F_{avow} proceeds only after it receives two avow messages with identical [AB]s, which captures that both parties mutually request to avow the same part of their conversation. After getting the second avow, F_{avow} matches two records that share [AB] values and tests if the avowal is successful or not. If this is a valid avowal, F_{avow} simulates session key exchanged $k_{rel}$. F_{avow} simulates transcripts of avow $m_1(E_A, R_A), m_2 = (R_B, Enc(k_{rel}, (c_A, z_A, s_A))), m_3((Enc(k_{rel}, (c_B, z_B))))$, and keep the rest of O as the ADV avowal proof $\pi$. S receives all the simulated materials from F_{avow} and stores them for later emulation. If this is a failed avowal, F_{avow} sends fail message to S. Note that S knows distributed session key in the Ideal World since it completes the emulation. In that case, S simulates the critical part of avowal $z_A$ and $z_B$ by arbitrarily sampling the $z_A$ and $z_B$ which will lead to a failed avowal.

Next, S honestly constructs $m_1$ in NOTRY-Avow with the materials it gets from F_{avow} when it observes the corresponding message sent from I to R. S sends $(E_A, R_A)$ to the simulate party in the Real World, R, through A to emulate this behavior. S constructs $m_2 = (R_B, Enc(k_{rel}, (c_B, z_B)))$ for NOTRY-Avow in response to I based on the message even if it’s different from the one it got from N and sent it to R. S just forward $m_3$ from R’ to I as the internal real-world P replies I with real $m_2$. To simulate message $m_3$ sends I to $\bar{R}$ in the Real World, S just passes message $m_3$ from R’ to I’. When S sees two parties’ final outputs in the Real World, S forward the corresponding output, which is $z_A, z_B$ to two dummy parties in the Ideal World. Note that $m_1, m_2, m_3$ can all be modified by A.

D.2 Indistinguishable realization
F_{avow} UC-realizes NOTRY-Avow iff S emulates an indistinguishable execution view in the Ideal World for any adversary A in the Real World, such that:

Lemma 7. IDEAL_{FAVOW}, S, Z \cong \text{REALNOTRY-Avow}, A, Z

To formally prove Lemma 7, we will see how the constructed S emulates a computationally indistinguishable view adaptively according to the behavior of A. For an adversary A that can modify, abort, and corrupt parties. If any of the three messages in the Real World was modified by A, S replaces the corresponding message with a random value. Note that under this circumstance with a secure encryption scheme and the DLP assumption holds, the final scenario S is unable to distinguish S from A since S works exactly as A in NOTRY-Avow.

D.2.1 Hybrid Arguments. For the one-party corruption case, the Z is unable to distinguish its communication with A in the Real World and S in the Ideal World because of the zero-knowledge properties of ADV proof of avowal. As we’ve already proved in the Lemma 4. For the honest case and two-party corruption case, we’re going to prove the transcripts between the two worlds are indistinguishable. Recall the two transcripts are $T_c = (E'_A, R'_A, R'_B, c_1', z_1', z_1', c_2', z_2', c_1', z_1')$ in the Ideal World and $T = (E_A, R_A, R_B, \bar{e}_1, \bar{c}_2, \bar{e}_2, \bar{z}_2, \bar{c}_1, \bar{z}_1)$ in the Real World.

(Hybrid 0) is the genuine transcript $T_c$.

(Hybrid 1) slightly changes $T_c$ to $T_1 = (E'_A, R_0, R_1, c_1', c_2', z_1', z_1', \bar{c}_1', \bar{z}_1')$, where $(R_0, R_1) \overset{\$}{\leftarrow} Z_q$, suppose there exists a distinguisher $D$ with advantage $\epsilon_{DDH}$ to break DDH assumption, then the advantage of $D$ to distinguish $T_c$ and $T_1$ is bounded by $2\epsilon_{DDH}$.

(Hybrid 2) $T_2$ will be a simple replacement of $E_A$ with the Peder- sen Commitment $E_A$ in $T_c$. As Pedersen Commitments are perfect hiding, D is unable to distinguish $E_A$ between $E_\alpha$.

(Hybrid 3) the two ciphertexts $c_1', c_2'$ in $T_2$ can be substituted the $\bar{c}_1, \bar{c}_2$ of $T$ to constitute $T_3$. Let’s say the advantage of $D$ to break the notion of semantic security is $\epsilon_{se}$. Therefore, we can conclude that $D$ distinguishes $T_2$ and $T_3$ with $2\epsilon_{se}$.

(Hybrid 4) $T_4$ transfers $c_1', c_2'$ in $T_2$ to the two random values $(c_1, z_1)$ in $T_4$. Since $(c_1', z_1')$ $(c_1, z_1)$ are all dependent on two random numbers contributed by I and R respectively. Naturally, under the one-time pad, the D is not capable of distinguishing the pairs from each other. That implies $T_3$ and $T_4$ are indistinguishable.

(Hybrid 5) observe that $z_1'$ is distinguishable from $c_1 + r_{AB}$ because $r_{AB}$ are randomly sampled from $Z_q$. Assuming a random oracle, it’s obvious that the output of the random oracle $c$ is indistinguishable from $r_{AB}$ as well. As $c_{AB}$ is masked by c, according to the one-time-pad, we can conclude that $c_{AB}$ is distinguishable from a random number. Therefore, $z_1'$ is distinguishable from $r_{AB} + r_0$. Same arguments applies to $z_2'$ and $(z_1', z_2')$.

With $z_1$ and $z_1'$ both indistinguishable from $r_{AB} + r_0$, it’s trivial to see $z_1(z_1')$ is indistinguishable from $z_1'(z_1')$. This directly leads to $T_5 = (E_A, R_0, c_1, z_2', z_2', c_1', z_1)$ to be inherently distinguishable from $T_4 = (E_A, R_0, c_1, z_2', z_2', c_1', z_1, z_1)$.

(Hybrid 6) $T_6$ is identical to T. follow the same arguments we used in Hybrid 1, we can get that the advantage of $D$ to distinguish $T_6$ and $T$ is less than $2\epsilon_{DDH}$. Finally, for any PPT distinguisher $D$, we have

$$\Pr[D(T) = 1] - \Pr[D(T')] \leq 4\epsilon_{DDH} + 2\epsilon_{se}$$

with all DDH, semantic security, and random oracle holds, both of $\epsilon_{DDH}, \epsilon_{se}$ are negligible. In conclusion, D only has negligible advantage to distinguish T from $T_6$, which makes T is computationally indistinguishable from $T_6$. This completes the proof. □