Nik Unger and Ian Goldberg

Improved Strongly Deniable Authenticated Key Exchanges for Secure Messaging

Abstract: A deniable authenticated key exchange (DAKE) protocol establishes a secure channel without producing cryptographic evidence of communication. A DAKE offers *strong deniability* if transcripts provide no evidence even if long-term key material is compromised (*offline deniability*) and no outsider can obtain evidence even when interactively colluding with an insider (*online deniability*). Unfortunately, existing strongly deniable DAKEs have not been adopted by secure messaging tools due to security and deployability weaknesses.

In this work, we propose three new strongly deniable key exchange protocols—DAKEZ, ZDH, and XZDH—that are designed to be used in modern secure messaging applications while eliminating the weaknesses of previous approaches. DAKEZ offers strong deniability in synchronous network environments, while ZDH and XZDH can be used to construct asynchronous secure messaging systems with offline and partial online deniability. DAKEZ and XZDH provide forward secrecy against active adversaries, and all three protocols can provide forward secrecy against future quantum adversaries while remaining classically secure if attacks against quantum-resistant cryptosystems are found.

We seek to reduce barriers to adoption by describing our protocols from a practitioner's perspective, including complete algebraic specifications, cryptographic primitive recommendations, and prototype implementations. We evaluate concrete instantiations of our DAKEs and show that they are the most efficient strongly deniable schemes; with all of our classical security guarantees, our exchanges require only 1 ms of CPU time on a typical desktop computer and at most 464 bytes of data transmission. Our constructions are nearly as efficient as key exchanges with weaker deniability, such as the ones used by the popular OTR and Signal protocols.

Keywords: Key exchange, deniability, secure messaging

DOI 10.1515/popets-2018-0003
Received 2017-05-31; revised 2017-09-15; accepted 2017-09-16.

Nik Unger: School of Computer Science, University of Waterloo, njunger@uwaterloo.ca
Ian Goldberg: School of Computer Science, University of Waterloo, iang@cs.uwaterloo.ca

1 Introduction

In recent decades, our society has become heavily dependent on electronic communication. The Internet has become the platform upon which our most critical discourse is conducted. Protecting the security and privacy of this communication has never been more important. At the same time, revelations of widespread surveillance of the Internet and interference in security technologies has led to increased public interest in the security and privacy of their communications. Most Americans want to maintain control over their information online, while few have confidence that they can do so [74].

In response to these developments, we have seen an explosion of new secure messaging protocols, components, and applications in recent years [94]. While this proliferation has led to many options, each providing a selection of security and privacy protections, industry adoption by WhatsApp [80], Google Allo [81], and Facebook Messenger [82] have led to a de facto standardization on the Signal protocol [79] as a means to secure communications. One of the most important privacy properties of secure messaging protocols, as first identified in the Off-The-Record Messaging (OTR) protocol [19] and later reinforced by Signal, is deniability (sometimes also called repudiation). A deniable secure messaging protocol allows users to plausibly deny exchanging messages using the protocol in the sense that the protocol produces no convincing cryptographic evidence of an exchange. This property gained renewed interest recently after DKIM signatures were used to confirm the authenticity of emails leaked from the Clinton presidential campaign in the United States [72].

Unfortunately, popular secure messaging protocols like OTR and Signal do not provide *strong deniability*. A protocol is *strongly deniable* if transcripts provide no evidence even if long-term key material is compromised (*offline deniability*) and no outsider can obtain evidence even if an insider interactively colludes with them (*online deniability*). The limited deniability of current secure messaging tools creates severe privacy weaknesses. A protocol lacking unrestricted offline deniability permits production of irrefutable transcripts that could be generated only by one of a few potential entities. A protocol without online deniability allows a participant to

generate irrefutable cryptographic proof of a conversation with the aid of an interactive third party [42].

For illustrative purposes, Appendix A describes how a malicious Signal or OTR participant can use a third-party service to reserve the capability, on a per-session basis, to produce non-repudiable conversation transcripts. Appendix A also describes another attack in which an authority coerces an honest OTR or Signal user into participating in a malicious protocol that produces proof of message authorship in real time without compromising the user's long-term secret key. In general, this attack works by forcing the user to interactively authenticate ephemeral decryption keys privately generated by the authority.

The primary component that provides deniability in secure messaging protocols is a deniable authenticated key exchange (DAKE) [37]. In this work, we present three new DAKEs—DAKEZ, ZDH, and XZDH—that are designed to patch this weakness in modern secure messaging environments while overcoming barriers to adoption. Our schemes can act as drop-in replacements for the DAKEs in protocols like OTR and Signal in order to efficiently provide strong deniability without sacrificing any existing security properties. Our definitions also explicitly include the option to add quantum resistance. Our other contributions in this work include:

1. concrete instantiations of our protocols and their constituent primitives to simplify development, including explicitly defined zero-knowledge proofs;
2. definition and construction of *dual-receiver encryption with associated data*, a natural extension of dual-receiver encryption, and its security properties;
3. a concrete performance comparison between efficient implementations of our DAKEs and existing key exchanges used by secure messaging applications; and
4. a discussion of the relationship between online deniability and key compromise impersonation attacks—a rarely mentioned and important topic—and techniques for messaging tools to mitigate these attacks.

The remainder of this paper is structured as follows: Section 2 surveys related work on deniability and key exchanges; Section 3 defines the security properties and features of our DAKEs, and outlines our approach for achieving these properties; Section 4 establishes notation and introduces constructions for cryptographic primitives; Section 5 defines DAKEZ; Section 6 defines ZDH; Section 7 defines XZDH; Section 8 covers practical considerations for deploying our DAKEs; Section 9 describes our implementation; Section 10 evaluates the time and space efficiency of our protocols and existing key exchanges; Section 11 discusses key compromise impersonation attacks in the context of strongly deniable key exchanges; and Section 12 concludes.

2 Related Work

An authenticated key exchange (AKE) is a protocol that allows two parties—an initiator and a responder—to securely derive a shared secret and authenticate each other. Bellare and Rogaway first formalized the definition of AKEs in 1993 [10]. Shortly afterward, several AKEs claimed to offer deniability informally [20, 66, 67]. Each of these DAKEs lacks some aspect of strong deniability. In a mostly independent line of research, deniability was widely studied in the context of authentication [43, 44, 62, 101].

With the release of the Off-the-Record Messaging protocol in 2004, deniability was recognized as a desirable feature for secure messaging [19]. Since then, a variety of DAKEs have been published [37, 42, 61, 75, 87, 93, 97, 99]. Walfish [95] was the first to introduce a DAKE, Φ_{dre}, that simultaneously provides strong deniability, (weak) forward secrecy [14], security against active attackers, and operation without trusted authorities. This work was later reiterated in a publication by Dodis et al. [42]. In 2015, we introduced two DAKEs designed for secure messaging—RSDAKE and Spawn—with comparable security proofs [93]. Notably, Spawn was the first DAKE with (partial) online deniability that can be used in non-interactive applications.

The most popular secure messaging protocols in practice, OTR [3] and Signal [79], use DAKEs with weaker deniability. OTR's variant of the SIGMA DAKE offers no online deniability, and requires fragments of legitimate exchanges to forge transcripts offline. Signal originally used 3DH [78], an implicit DAKE with unrestricted offline deniability (i.e., anyone can forge transcripts using only public keys), but lacking online deniability. Signal recently switched to a DAKE known as "Extended Triple Diffie-Hellman" (X3DH) [75] that improves forward secrecy but regresses to the deniability properties of OTR's DAKE. Consequently, real-world deployments of "deniable" secure messaging protocols still lack strong deniability.

While RSDAKE and Spawn represent the state of the art in terms of deniability for secure messaging protocols, they lack some key properties that prevent their adoption by the designers of OTR and Signal:

1. the protocols are too slow to use in real-world secure messaging settings like smartphone applications;

2. the protocols cannot provide forward secrecy against active adversaries in non-interactive settings;
3. the protocols do not provide forward secrecy against future quantum adversaries; and
4. the protocols make users susceptible to key compromise impersonation (KCI) attacks;
5. the protocols are not contributory—shared secrets are unilaterally determined by one party, which permits sophisticated attacks against key secrecy.

Our key exchanges match the deniability of RSDAKE and Spawn while overcoming these limitations.

3 Design Overview

3.1 General Approach

Our three new protocols, DAKEZ ("DAKE with Zero-knowledge"), ZDH ("Zero-knowledge Diffie-Hellman"), and XZDH ("Extended ZDH"), are meant to be used in common secure messaging scenarios. DAKEZ is designed for use in interactive settings such as instant messaging applications, while ZDH and XZDH are designed to be used in non-interactive settings such as text messaging. ZDH is the most efficient, but XZDH provides stronger forward secrecy than ZDH. Our DAKEs share many design similarities with the RSDAKE and Spawn protocols that we introduced in UG15 [93], but our new protocols are 4000–5000 times more efficient in practice. We achieve this significant efficiency advantage by constructing and using cryptographic primitives that rely on the random oracle model (ROM) for security proofs.

We construct DAKEZ by adopting RSDAKE's approach to authentication, but using several new instantiations of cryptographic primitives in the random oracle model. For comparison purposes, we also adapt Spawn to the random oracle model using our primitives. We then show how recognizing the partial online deniability of Spawn can be leveraged to dramatically improve performance in non-interactive settings, leading to our design of ZDH. Finally, we describe XZDH, a variant of ZDH that adopts X3DH's forward secrecy technique.

3.2 Proof Technique

There are two general approaches to proving the security of key exchanges: *security by indistinguishability*, where security properties are expressed in terms of indistinguishable adversarial games [10]; and *security by emulation*, where a protocol is shown to be indistinguishable from an idealized protocol with access to secure channels and a trusted third party [25]. Our protocols are designed to be proven secure using the emulation method through the generalized universal composability (GUC) framework [42], which was also used in UG15. GUC-based security proofs are generally more convincing of real-world security, more resistant to insecure composition, and able to naturally express deniability properties. On the other hand, GUC-based proofs are often overly restrictive and more complex.

While UG15 also sketched GUC-based security proofs for RSDAKE and Spawn, UG15 used the standard model with obscure hardness assumptions. Our protocols can be proven secure using random oracles and widely accepted standard assumptions. While the use of random oracles in security proofs has historically caused debate among theorists [65], it remains to be shown that instantiation of random oracles using appropriate cryptographic hash functions in real-world protocols introduces any actual security flaws.

Our DAKEs are designed to provide many security properties and features. However, proof of these properties within the GUC framework requires the definition of an *ideal functionality* and proof showing that the real key exchanges behave indistinguishably from the idealized version. Formal definitions of these ideal functionalities, explanations for how they exhibit desirable properties, and expression of the resulting security theorems, are long and complex. For ease of presentation, we formally define our ideal functionalities, security theorems, and proof sketches in the appendices, beginning in Appendix D. We refer readers interested in the technical details of the formal security properties to the appendices. The main body of this paper focuses on practical issues surrounding the deployment of DAKEs. Nonetheless, to facilitate design discussion and comparison with prior schemes, we define our high-level security objectives in the next section, and provide simplified security propositions in the main body of this paper.

3.3 Protocol Properties

Our protocols provide the following properties:

1. **Universally composable AKE** [25]: Our protocols emulate idealized authenticated key exchanges facilitated by a trusted third party (without requiring one in practice), and continue to do so even when composed within arbitrary protocols. This property includes the traditional notions of mutual authentication, key secrecy, and key freshness [10, 25].
2. **Offline deniability** [37]: Anyone can forge a DAKEZ, ZDH, or XZDH transcript between any two parties using only their long-term public keys. Con-

sequently, no transcript provides evidence of a past key exchange, because it could have been forged. This is similar to the deniability offered by 3DH [78].

3. **Online deniability** [42]: Participants in a DAKEZ exchange cannot provide proof of participation to third parties without making themselves vulnerable to KCI attacks, even if they perform arbitrary protocols with these third parties during the exchange. ZDH and XZDH provide this property for one party.
4. **Contributiveness**: Our DAKEs are *initiator-resilient* [59] in the same sense as traditional Diffie-Hellman [39] key exchanges—the initiator of the protocol cannot force the shared secret to take on a specific value. In practice, it is also computationally infeasible for the responder to select a specific shared secret, although it may attempt to select one with some desirable characteristic through a brute-force search. The initiator (and, in practice, the responder) cannot force the shared secret to be a value known to a third party before the exchange begins. Without contributiveness, an adversary can specify a secret, coerce a participant into forcing that shared secret, and then decrypt the subsequent conversation without access to the key exchange transcript [59]. A non-contributory protocol also permits a participant to force a shared secret known to an innocent third party, enabling that third party to decrypt the conversation without their consent.
5. **Forward secrecy** [14]: A classical adversary that compromises the long-term secret keys of both parties cannot retroactively compromise past session keys. DAKEZ offers *strong forward secrecy*—it protects the session key when at least one party completes the exchange. ZDH offers *weak forward secrecy* [14]—it protects the session key only when both parties complete the exchange. XZDH protects completed sessions and incomplete sessions that stall long enough to be invalidated by a participant. We discuss this distinction in more detail in Section 7.
6. **(Optional) quantum resistance** [88]: Our DAKEs can optionally be combined with a quantum-resistant key encapsulation mechanism (KEM) to maintain forward secrecy against future quantum adversaries, at the expense of performance. We make this property optional because the secure messaging community is still debating the importance of quantum resistance and which primitives are likely to resist quantum cryptanalysis, while the computational or transmission costs to use these primitives is higher than classical Diffie-Hellman exchanges.
7. **Post-specified peer** [24]: Participants in DAKEZ exchanges learn the identities of their partners during the exchange. For ZDH and XZDH exchanges, only one party begins the protocol with knowledge of the other's claimed identity.

In all configurations, our protocols are universally composable AKEs with offline deniability, contributiveness, and post-specified peers. Given these baseline properties, we present configurations that offer different trade-offs between online deniability, forms of forward secrecy, computational efficiency,[1] and quantum resistance; the best configuration to use depends on the secure messaging environment in question.

3.4 Quantum Transitional Security

With our DAKEs, we address the problem of forward secrecy against quantum adversaries. The purpose of forward secrecy in key exchange protocols is to prevent future adversaries from retroactively compromising recorded sessions. This property is extremely important because it minimizes the impact of compromised long-term keys. However, current deployments of key exchanges with forward secrecy are believed to be vulnerable to quantum cryptanalysis. This leads to the worrying possibility that all communications that are currently protected by "secure" protocols like Signal might be retroactively decrypted by a passive adversary that gains access to a quantum computer in the future; such an event could be potentially devastating to many real users. As quantum computation continues to appear more feasible, it is becoming increasingly clear that we must defend against this potential threat as quickly as possible. However, implementing protocols that are fully resistant to all quantum attacks remains challenging and expensive. Moreover, the security (even classically) of "quantum-resistant" cryptosystems remains suspect, relative to our confidence in classical schemes based on widely examined hardness assumptions.

An interesting class of key exchanges are those providing *quantum transitional security*—classical authentication and quantum-resistant confidentiality [88]. Assuming that the quantum resistance assumptions hold, these schemes preserve forward secrecy against future quantum adversaries. Additionally, these hybrid protocols maintain classical security in the event that the quantum resistance assumptions fail. The downside of these schemes is that they must be replaced with fully

1 We preferentially optimize our constructions based on computational efficiency rather than other performance metrics, such as network transmission size.

quantum-resistant schemes after quantum adversaries appear. Google recently tested the deployment of a quantum transitionally secure key exchange in TLS [68].

DAKEZ, ZDH, and XZDH optionally provide quantum transitional security by incorporating a passively secure (IND-CPA) quantum-resistant KEM. Because there are many different quantum-resistant KEMs with varying security properties, performance, and usage restrictions, we provide generic "placeholders" in our DAKEs. An implementer is free to incorporate the KEM of their choice when using our schemes. We return to the practical issues surrounding this choice in Section 8.1.

4 Cryptographic Primitives

The primary barrier to adoption of strongly deniable DAKEs like Φ_{dre}, RSDAKE, and Spawn is poor performance caused by the use of inefficient *dual-receiver encryption* (DRE) and *ring signatures* in the standard model. Dual-receiver encryption [38] is similar to ordinary public-key encryption, except that messages are encrypted for two recipient public keys. The message can be decrypted by *either* corresponding private key, and it is verifiable that decrypting with either key produces the same result. DRE is similar to the more well-known notion of broadcast encryption [47], except that DRE does not require centralized generation of private keys, and broadcast encryption does not provide any verifiability guarantees. Ring signatures [85] are similar to ordinary digital signatures, except that messages are signed by a set of potential signers called a *ring*. Anyone with knowledge of a private key corresponding to *any* public key in this ring can produce the ring signature, and it is not possible to determine which key was used.

These two primitives are very useful in deniable key exchanges; ring signatures provide deniable authentication, and DRE can assist with online transcript forgery. Ring signatures with a carefully selected ring can convince a verifier of the authenticity of a message (due to a non-transferable belief that the signer cannot know certain keys) while allowing the signer to plausibly deny authorship (because the signature could have been produced using other keys in the ring). DRE can be used to allow a simulator to recover the shared secret and use it to forge subsequent messages even when outsourcing the generation of ephemeral secrets to an adversary; this technique enables online deniability by eliminating a potential protocol for producing evidence of an exchange.

The best known constructions of DRE and ring signatures with the appropriate security properties in the standard model are very inefficient. Additionally, these schemes are constructed using complex primitives such as cryptographic pairings. Pairing-based schemes rely on relatively obscure security assumptions that are the subject of recently discovered attacks [64], and lack efficient and rigorous implementations.[2] In this section, we specify DRE and ring signature primitives, the security properties that we require for our DAKEs, and efficient ROM-based constructions of these primitives.

4.1 Notation

All of our definitions are implicitly given with respect to a security parameter λ. As a notational convenience, we write $r \xleftarrow{\$} \{0,1\}^\lambda$ to denote that r is set to a random value with the appropriate form and length for the context. The length of r is set to some linear function of λ, as needed. When defining concrete two-party protocols, we denote the initiator and the responder as $\mathcal{I}$ and $\mathcal{R}$, respectively. We write "$\mathcal{P}$" (with quotes) to denote an implementation-defined identifier for the party $\mathcal{P}$.

4.2 Dual-Receiver Encryption with Associated Data

While it is possible to construct our protocols using traditional DRE [38], we can reduce transmission sizes by incorporating Rogaway's notion of associated authenticated data [86] in the primitive. We define a dual-receiver encryption with associated data (DREAD) cryptosystem to consist of the following functions:

- **DRGen**(s): a key generation function. DRGen produces a key pair (pk, sk) for use with the scheme. s represents the seed used to generate the key pair and may be omitted to imply $s \xleftarrow{\$} \{0,1\}^\lambda$.
- **DREnc**(pk_1, pk_2, m, Φ, r): an encryption function. DREnc encrypts a message m with associated data Φ under two public keys pk_1 and pk_2. If pk_1 and pk_2 are valid public keys, then DREnc produces a ciphertext γ. Otherwise, DREnc returns the special value $\perp$. The output of DREnc is consistent across invocations with the same (pk_1, pk_2, m, Φ, r) as input and varies when r is changed. Omitting r implies $r \xleftarrow{\$} \{0,1\}^\lambda$.
- **DRDec**$(pk_1, pk_2, sk, \Phi, \gamma)$: a decryption function. If (pk_1, sk) ***or*** (pk_2, sk) was generated by DRGen, $\gamma = \text{DREnc}(pk_1, pk_2, m, \Phi, r)$ for some r and m, and $\gamma \neq \perp$, then DRDec returns m. In all other cases, DRDec returns $\perp$ with overwhelming probability.

2 The most popular pairing libraries, the PBC library [73] and RELIC [7], have published warnings against their use in production. The MIRACL [27] library offers support for Type-I and Type-III pairings that is described as experimental.

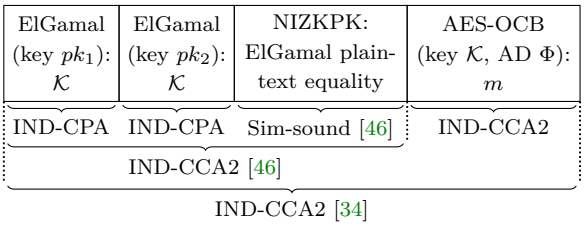

Fig. 1. Our DREAD construction is a hybrid encryption of message m for keys pk_1 and pk_2 with associated data Φ.

The associated data Φ is information that is not transmitted with the ciphertext, but is nonetheless authenticated; the same value must be provided to both DREnc and DRDec in order for the decryption of m to succeed.

Note that the definitions of DREnc and DRDec implicitly rely on the ability to verify that a public key is valid and corresponds to a unique secret key. A scheme satisfying this property is called *admissible* [30], and we restrict our attention to such schemes. Additionally, our key exchange protocols require DREAD instantiations to satisfy two security properties that are naturally extended from the DRE security definitions introduced by Chow et al. [30] by incorporating the Φ parameter where necessary. We require only *soundness* (decryption always produces the same value for both secret keys) and *indistinguishability under adaptive chosen ciphertext attack* (IND-CCA2) from our DREAD scheme. Appendix B details the necessary modifications to the definitions from Chow et al. [30], which merely involve allowing the adversary to select Φ in the security games.

The few explicit constructions of DRE in the literature are almost all designed for the standard model. Since we permit random oracles, we can produce an extremely efficient DREAD scheme based on the Naor-Yung paradigm [77] within a hybrid cryptosystem. Figure 1 depicts our scheme. The ciphertext consists of a symmetric key $\mathcal{K}$ encrypted twice with ElGamal [45] (once to each recipient), a non-interactive zero-knowledge proof of knowledge (NIZKPK) proving that the ElGamal ciphertexts contain the same plaintext, and an authenticated encryption with associated data (AEAD) of the message m. This approach is known to generate a sound DRE system [30]. We select ElGamal because it is a very efficient scheme with the desired security properties, and it can be instantiated without adding new security assumptions to our DAKEs. Our NIZKPK is formed using the Fiat-Shamir transform [48]. When used in the Naor-Yung paradigm, Faust et al. [46] proved that such NIZKPKs are simulation-sound and thus yield IND-CCA2 security when combined with IND-CPA encryption schemes like ElGamal. Since an AEAD must be IND-CCA2 secure, the overall combination of schemes is also IND-CCA2 secure using the proof from Cramer and Shoup [34, Th. 7.2].[3] In practice, the AEAD may be instantiated using AES-256 in OCB or an equivalent unpatented mode [58] using the NIZKPK to derive the nonce. Appendix B shows that the soundness of the NIZKPK proof implies soundness of the DREAD construction.

Our complete DREAD scheme is as follows:

- **Setup**: all users share a group description, $(\mathbb{G}, q, g)$, for use in the ElGamal scheme [45]. g is a generator for the group $\mathbb{G}$ of prime order q. All hash functions used in this paper output values in $\mathbb{Z}_q$.
- **DRGen**(s): keys are generated as in the ElGamal scheme [45]. The resulting public key for a user is $h = g^s$, and the secret key is s.
- **DREnc**(pk_1, pk_2, m, Φ, r): r is interpreted as $r = \mathcal{K}\|k_1\|k_2\|t_1\|t_2$. k_1 and k_2 are used to encrypt $\mathcal{K}$ for each recipient using ElGamal [45], and t_1 and t_2 are used later in the NIZKPK. If pk_1 or pk_2 are outside of $\mathbb{G}$ or are the identity element, DREnc returns $\perp$. Otherwise, the resulting ciphertexts consist of $c_{1i} = g^{k_i}$ and $c_{2i} = pk_i^{k_i} \cdot \mathcal{K}$ for $i \in \{1, 2\}$. The message m is encrypted using the AEAD scheme with key $\mathcal{K}$ and associated data Φ, denoted by $\Theta = \text{AEnc}(\mathcal{K}, m, \Phi)$. The result also includes a NIZKPK of the following statement, proving that the ciphertext is well formed, given in Camenisch-Stadler notation [22]:

$$PK\left\{(k_1, k_2) : c_{11} = g^{k_1} \wedge c_{12} = g^{k_2} \wedge \frac{c_{21}}{c_{22}} = \frac{pk_1^{k_1}}{pk_2^{k_2}}\right\}$$

The party calling DREnc acts as the prover $\mathcal{P}$ for the NIZKPK. $\mathcal{P}$ uses the random values $t_i \in \mathbb{Z}_q$ for $i \in \{1, 2\}$ to compute $T_{11} = g^{t_1}$, $T_{12} = g^{t_2}$, and $T_2 = {pk_1^{t_1}}/{pk_2^{t_2}}$. Next, $\mathcal{P}$ computes the hash:

$$L = H(g\|q\|pk_1\|pk_2\|c_{11}\|c_{21}\|c_{12}\|c_{22}\|T_{11}\|T_{12}\|T_2\|\Phi)$$

where H is a cryptographic hash function modeled by a random oracle.[4] $\mathcal{P}$ then computes $n_i = t_i - L \cdot k_i \pmod q$ for $i \in \{1, 2\}$. DREnc returns the encryption of m, $\gamma = (c_{11}, c_{21}, c_{12}, c_{22}, L, n_1, n_2, \Theta)$.

3 When expressing the ElGamal encryptions of $\mathcal{K}$ and the NIZKPK as the key encapsulation mechanism, AES-OCB as the one-time symmetric key encryption scheme, and the IND-CCA2 security game as in Appendix B, the proof directly applies.

4 If H is instantiated with a concrete hash function that suffers from length-extension or padding attacks [91] (e.g., SHA-2), Φ should be hashed before it is passed as input to H, thereby ensuring that the input is of fixed length. This additional hash is unnecessary if Φ is unused, is already of fixed length, or when using an appropriate hash function (e.g., SHA-3).

– **DRDec**$(pk_1, pk_2, sk_i, \Phi, \gamma)$: the recipient parses γ to retrieve its components. If either public key is outside of $\mathbb{G}$ or is the identity element, or if γ is not of the correct form, then DRDec returns $\perp$. Otherwise, the recipient computes the following three values:

$$T'_{11} = g^{n_1}(c_{11})^L \quad T'_{12} = g^{n_2}(c_{12})^L \quad T'_2 = \frac{pk_1^{n_1}}{pk_2^{n_2}}\left(\frac{c_{21}}{c_{22}}\right)^L$$

The recipient then computes L' using the same hash operation described in DREnc:

$$L' = H(g\|q\|pk_1\|pk_2\|c_{11}\|c_{21}\|c_{12}\|c_{22}\|T'_{11}\|T'_{12}\|T'_2\|\Phi)$$

If $L \neq L'$, DRDec returns $\perp$. Otherwise, the recipient recovers the secret key $\mathcal{K} = {c_{2i}}/{c_{1i}^{sk_i}}$. The recipient can then recover the message m by decrypting the AEAD ciphertext Θ with key $\mathcal{K}$ and associated data Φ, denoted by $m = \text{ADec}(\mathcal{K}, \Theta, \Phi)$. If ADec fails, DRDec returns $\perp$. Otherwise, it returns m.

4.3 Efficient Ring Signatures in the ROM

One way to achieve strong deniability in a DAKE is to use ring signatures, as in RSDAKE and Spawn [93]. Unfortunately, the literature lacks an explicit ring signature construction that provably provides the security properties we require while also being efficient, even though the techniques to do so are known. Schemes introduced prior to the publication of the security definitions from Bender et al. [15] lack the appropriate proofs [1], require unusual primitives that are difficult to use in practice [18, 41, 102], or reduce to a generic form of our approach [2]. Schemes published after the definitions from Bender et al. [15] either focus on adding new features, or offering strong standard model security proofs (at the expense of practicality) [98]. Additionally, "efficient" ring signature schemes typically focus on scalability with respect to ring size [41]. Our DAKEs use small rings with only three potential signers, so scalable schemes are usually not optimized for our setting. There are also a variety of well-known ring signature schemes that operate in slightly different scenarios, at the cost of performance [21, 29, 71, 96].

RSDAKE and Spawn use ring signatures to prove that the signer knows one of three private keys. We can construct an efficient ROM-based ring signature scheme to accomplish this by issuing a *signature of knowledge* (SoK) of one out of three discrete logarithms [22]. An SoK is a non-interactive zero-knowledge proof system demonstrating knowledge of a value. We construct the SoK in our ring signature based on the Schnorr signature scheme [89] and the "OR proof" technique introduced by Cramer et al. [35].[5] The Fiat-Shamir transform [48] makes this proof non-interactive. This approach is far more efficient than the ring signature scheme used in RSDAKE and Spawn [90].

Assume that each public key is of the form $A_i = g^{a_i}$, as in a typical Diffie-Hellman key exchange (where g is the generator for a group $\mathbb{G}$ of prime order q). The SoK over keys (A_1, A_2, A_3) is a proof of the following statement, given in Camenisch-Stadler notation [22]:

$$SKREP\{(a) : g^a = A_1 \vee g^a = A_2 \vee g^a = A_3\}(m)$$

Assuming without loss of generality that the signer knows a_1, the proof proceeds as follows:

1. Generate random values $t_1, c_2, c_3, r_2, r_3 \in \mathbb{Z}_q$.
2. Compute $T_1 = g^{t_1}$.
3. Compute $T_2 = g^{r_2}A_2^{c_2}$ and $T_3 = g^{r_3}A_3^{c_3}$.
4. Compute $c = H(g\|q\|A_1\|A_2\|A_3\|T_1\|T_2\|T_3\|m)$, where H is a hash function modeled by a random oracle and m is the message to "sign".
5. Compute $c_1 = c - c_2 - c_3 \pmod q$.
6. Compute $r_1 = t_1 - c_1 a_1 \pmod q$.

The resulting proof consists of $(c_1, r_1, c_2, r_2, c_3, r_3)$. To verify the proof, the verifier begins by computing $c' = H(g\|q\|A_1\|A_2\|A_3\|g^{r_1}A_1^{c_1}\|g^{r_2}A_2^{c_2}\|g^{r_3}A_3^{c_3}\|m)$. The verifier then checks whether $c' \stackrel{?}{=} c_1 + c_2 + c_3 \pmod q$.

In the general case where the prover knows a secret a_i, they select t_i randomly, compute $T_i = g^{t_i}$, and compute $T_j = g^{r_j}A_j^{c_j}$ for $j \neq i$, proceeding as normal and ultimately computing c_i and r_i. The order of elements passed to H and sent to the verifier must *not* depend on the secret known the prover (otherwise, the key used to produce the proof can be inferred in practice).

We use these processes to define our ring signature scheme, which consists of the following functions:

– **RSig**(A, a, S, m, r): RSig produces an SoK σ, bound to the message m, that demonstrates knowledge of a private key corresponding to one of three public keys. S is the ring of public keys $\{A_1, A_2, A_3\}$ that could possibly have produced the proof. It is required that (A, a) is a Diffie-Hellman keypair in the appropriate group, and $A \in S$. r controls the randomization of the output. The SoK is computed as described above, with r interpreted as $r = t_i\|c_j\|c_k\|r_j\|r_k$, where i is the index of A in S, and j and k are the values in $\{1, 2, 3\}$ such that i, j, and k are distinct and $j < k$. If r is omitted, it is assumed that $r \stackrel{\$}{\leftarrow} \{0,1\}^\lambda$ is used.

5 This technique was subsequently extended [31, 32]; however, the later results are not useful for our application because the statement to prove is fully known when the proof commences.

– **RVrf**(S, σ, m): a verification function. RVrf returns TRUE if the SoK σ is valid, and FALSE otherwise. Correctness requires that RVrf(S, σ, m) = TRUE when σ = RSig(A, a, S, m, r) for any valid inputs.

Bender et al. [15] defined several security properties for ring signature schemes. Our DAKEs require RSig and RVrf to exhibit *anonymity against full key exposure* (the signer remains anonymous within the ring even if all signing keys are subsequently compromised) and *unforgeability with respect to insider corruption* (signatures are unforgeable even if signing keys outside of the ring are adversarially generated). Appendix C includes the definitions of these properties and the security proofs for the RSig/RVrf ring signature scheme.

4.4 Quantum-Resistant KEM

As described in Section 3.4, our protocols provide quantum transitional security against future quantum adversaries. We accomplish this by using a black-box passively secure quantum-resistant KEM modeled by the following functions:

– **QRGen**$_I(s)$: a key generation function for initiators. QRGen$_I$ produces a key pair (PQ_I, SQ_I) for use with the scheme. s represents the seed used to generate the key pair and may be omitted to denote $s \xleftarrow{\$} \{0,1\}^\lambda$.

– **QRGen**$_R(PQ_I, s)$: an encapsulation function used by responders. QRGen$_R$ takes an initiator public key as input and produces a response message Q_R and a shared secret Q_k. s is a random seed that may be omitted to denote $s \xleftarrow{\$} \{0,1\}^\lambda$.

– **QRKey**$_I(SQ_I, Q_R)$: a decapsulation function used by initiators to derive the shared key. If $(PQ_I, SQ_I) \leftarrow$ QRGen$_I()$ and $(Q_R, Q'_k) \leftarrow$ QRGen$_R(PQ_I)$, then QRKey$_I(SQ_I, Q_R)$ produces $Q_k = Q'_k$.

This model permits a variety of quantum-resistant KEMs to be used in our DAKEs. For contributory schemes, QRGen$_R$ uses PQ_I to derive Q_k, and Q_R is a public contribution from the responder. For non-contributory (key transport) schemes, the responder unilaterally determines Q_k, and Q_R securely delivers Q_k to the initiator.

Our protocols perform both a black-box quantum-resistant KEM and a traditional Diffie-Hellman key exchange, then input both shared secrets to a key derivation function. A similar technique is used by CECPQ1 [68]. Key secrecy and freshness is immediately implied by the security of either underlying scheme.

5 DAKEZ

Given the cryptographic primitives defined in Section 4, we can now define our new three-flow DAKE, DAKEZ. Figure 2 depicts the protocol. KDF refers to a secure key derivation function.

Initially, the developer selects a common group $\mathbb{G}$ generated by g with prime order q. The CDH problem should be hard within $\mathbb{G}$. All parties select long-term public keys and distribute them. Initiator $\mathcal{I}$ chooses long-term secret key I and public key g^I. Responder $\mathcal{R}$ chooses long-term secret key R and public key g^R.

A DAKEZ session normally takes place within a higher-level protocol (e.g., XMPP or HTTP). To prevent attacks that rebind the DAKEZ transcript into different contexts, it is prudent to ensure that the DAKEZ session authenticates its context. Given state information Φ associated with the higher-level context, DAKEZ authenticates that both parties share the same value for Φ. Section 8 discusses the contents of this state information in practice. A DAKEZ session proceeds as follows:

1. $\mathcal{I}$ selects ephemeral secrets $i \in \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow$ QRGen$_I()$. $\mathcal{I}$ sends "I", g^i, and PQ_I to $\mathcal{R}$.
2. $\mathcal{R}$ selects ephemeral secrets $r \in \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow$ QRGen$_R(PQ_I)$. $\mathcal{R}$ sends "R", g^r, Q_R, and RSig$(g^R, R, \{g^I, g^R, g^i\}, t)$ to $\mathcal{I}$, where the tag t is t = "0"$\|$"I"$\|$"R"$\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$. $\mathcal{R}$ computes k = KDF$((g^i)^r\|Q_k)$ and securely erases r and Q_k.
3. $\mathcal{I}$ verifies the proof sent by $\mathcal{R}$. $\mathcal{I}$ computes and sends the proof RSig$(g^I, I, \{g^I, g^R, g^r\}, t)$ to $\mathcal{R}$, where the tag t is t = "1"$\|$"I"$\|$"R"$\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$. $\mathcal{I}$ computes Q_k = QRKey$_I(SQ_I, Q_R)$ and uses it to compute the shared secret k = KDF$((g^r)^i\|Q_k)$, then securely erases i, SQ_I, and Q_k.
4. $\mathcal{R}$ verifies the proof sent by $\mathcal{I}$.

The algebraic description of DAKEZ given above omits several important practical considerations that must be handled correctly to produce a secure implementation. We discuss these considerations in Section 8.

Proposition 1 states a simplified security theorem for DAKEZ. A formal theorem and proof sketch appear in Appendix E.

Proposition 1 (*DAKEZ security*)

If the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then DAKEZ provides the security properties listed in Section 3.3 against adaptive corruptions.

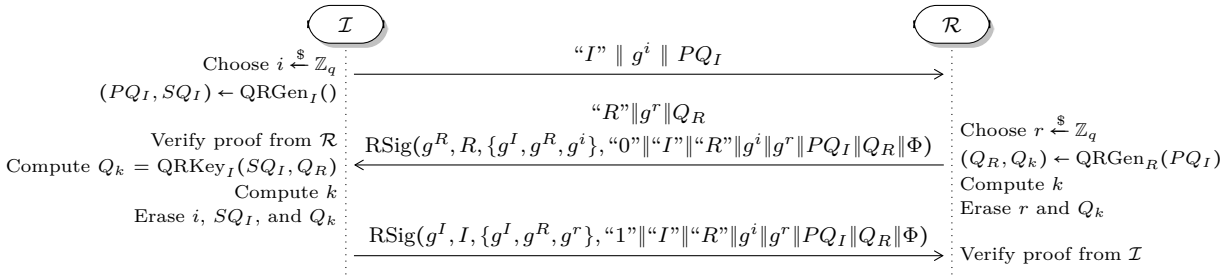

Fig. 2. The DAKEZ protocol. Φ is shared session state. The shared secret is $k = \mathrm{KDF}(g^{ir} \parallel Q_k)$.

6 ZDH

Spawn [93] is a two-flow DAKE with a single post-specified peer [24] (i.e., the initiator $\mathcal{I}$ does not know the identity of its partner a priori, but the responder $\mathcal{R}$ does). In an interactive setting, Spawn provides online deniability for both parties. In non-interactive settings where the adversary is able to guarantee that the first message is honestly generated by the true $\mathcal{I}$, Spawn does not provide online deniability for $\mathcal{R}$ [93]. In these settings, *explicitly* sacrificing online deniability for $\mathcal{R}$ results in a more efficient protocol, ZDH.

6.1 Efficient Spawn in the ROM

Before introducing the construction of ZDH, we first show how to use our DREAD and ring signature schemes from Section 4 to instantiate Spawn far more efficiently than the original construction [93]. While improving the performance of Spawn is not a primary goal of this work, doing so is important for three reasons: it allows us to precisely characterize the design improvements in ZDH, it simplifies the security proof of ZDH, and it provides a baseline performance comparison for ZDH. Figure 3 depicts Spawn$^+$, a contributory instantiation of Spawn using our ROM-based primitives. Unlike DAKEZ, which is secure in the same model as RS-DAKE, the contributiveness of Spawn$^+$ makes it provably secure against stronger adversaries than Spawn.

Initially, the developer selects a common group $\mathbb{G}$ generated by g with prime order q in which the CDH problem is hard. All parties select long-term ElGamal public keys and distribute them. Initiator $\mathcal{I}$ and responder $\mathcal{R}$ generate long-term key pairs (g^I, I) and (g^R, R), respectively, as described in Section 4. A Spawn$^+$ session between $\mathcal{I}$ and $\mathcal{R}$ within a higher-level protocol with shared session state Φ proceeds as follows:

1. $\mathcal{I}$ selects ephemeral secrets $i \in \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \mathrm{QRGen}_I()$. $\mathcal{I}$ sends $(\text{“}I\text{”}, g^i, PQ_I)$ to $\mathcal{R}$. An untrusted server may cache this “prekey” message.
2. $\mathcal{R}$ selects ephemeral secrets $r \in \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \mathrm{QRGen}_R(PQ_I)$. $\mathcal{R}$ then computes the ciphertext $\gamma = \mathrm{DREnc}(g^I, g^R, g^r\|Q_R, t)$, where the tag t is given by $t = \text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|PQ_I$. $\mathcal{R}$ computes $\sigma = \mathrm{RSig}(g^R, R, \{g^I, g^R, g^i\}, t\|\gamma\|\Phi)$. $\mathcal{R}$ sends $(\text{“}R\text{”}, \gamma, \sigma)$ to $\mathcal{I}$. $\mathcal{R}$ computes $k = \mathrm{KDF}((g^i)^r\|Q_k)$ and securely erases r and Q_k. Note that $\mathcal{R}$ can attach an initial message m to this flow by immediately encrypting it with a symmetric cryptosystem keyed with k.
3. $\mathcal{I}$ verifies the proof σ and decrypts γ using I. $\mathcal{I}$ verifies that the decrypted message is of the correct form (e.g., the fields are of the expected length) and that the prekey (g^i, PQ_I) that $\mathcal{I}$ previously sent remains unused. $\mathcal{I}$ computes $Q_k = \mathrm{QRKey}_I(SQ_I, Q_R)$ and the shared secret $k = \mathrm{KDF}((g^r)^i\|Q_k)$, then securely erases i, SQ_I, and Q_k. If an initial message was attached, $\mathcal{I}$ can decrypt the message using k.

Developers should incorporate the safeguards discussed in Section 8 to securely implement Spawn$^+$.

Proposition 2 states a simplified security theorem for Spawn$^+$. A formal theorem and proof sketch appear in Appendix F.

Proposition 2 (*Spawn$^+$ security*)

If the DREAD scheme is sound and IND-CCA2 secure, the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then Spawn$^+$ provides the security properties listed in Section 3.3 against adaptive corruptions.

6.2 The ZDH Protocol

Given the definition of Spawn$^+$, we can now easily discuss the construction of ZDH, a variant that improves performance in non-interactive settings primarily by avoiding the use of DREAD operations to facilitate online deniability for $\mathcal{R}$. ZDH is depicted in the unshaded portions of Figure 4 (the shaded values are used only in XZDH, which we introduce in Section 7).

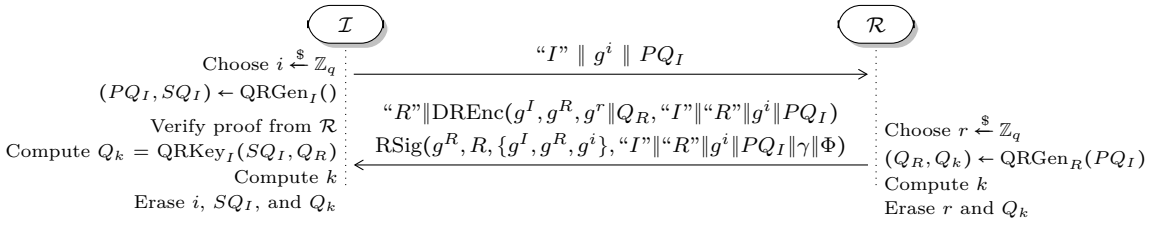

Fig. 3. A Spawn$^+$ exchange. Φ is shared session state. γ is the DREnc output. The shared secret is $k = \text{KDF}(g^{ir} \| Q_k)$.

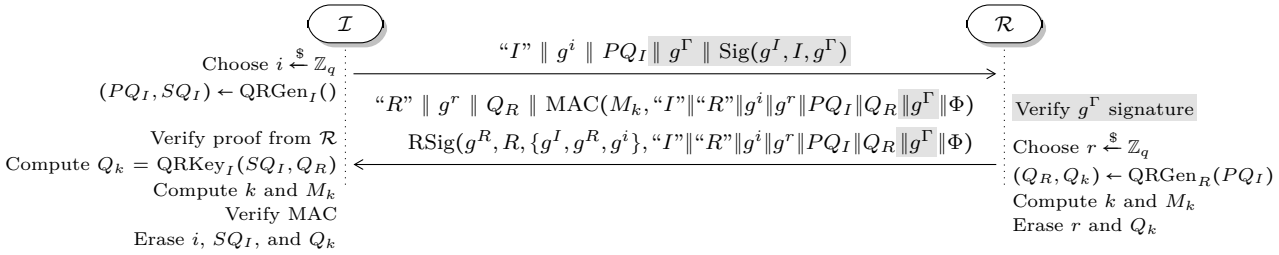

Fig. 4. A ZDH/XZDH exchange. Φ is shared session state. $\kappa = \text{KDF}_1(g^{ir} \| g^{\Gamma r} \| g^{Ir} \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$ and the shared secret is $k = \text{KDF}_3(\kappa)$. Shaded terms are used in XZDH only, and omitted for ZDH. In XZDH, g^Γ is a reusable signed prekey.

The initial ZDH setup is the same as for Spawn$^+$; parties generate and distribute long-term ElGamal public keys in a group $\mathbb{G}$ generated by g with prime order q in which the CDH problem is hard. Within a higher-level protocol with shared session state Φ, a ZDH session between initiator $\mathcal{I}$ and responder $\mathcal{R}$ with respective key pairs (g^I, I) and (g^R, R) proceeds as follows:

1. $\mathcal{I}$ selects ephemeral secrets $i \in \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}_I()$. $\mathcal{I}$ sends $(\text{"}I\text{"}, g^i, PQ_I)$ to $\mathcal{R}$. An untrusted server may cache this "prekey" message.
2. $\mathcal{R}$ selects ephemeral secrets $r \in \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$. $\mathcal{R}$ then derives the shared keys $\kappa = \text{KDF}_1((g^i)^r \| (g^I)^r \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and $k = \text{KDF}_3(\kappa)$, where KDF_1, KDF_2, and KDF_3 are key derivation functions modeled by a random oracle. $\mathcal{R}$ securely erases r and Q_k. Let the tag t be defined by $t = \text{"}I\text{"}\|\text{"}R\text{"}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi$. $\mathcal{R}$ computes $mac = \text{MAC}(M_k, t)$, where $\text{MAC}(key, msg)$ refers to a key-only unforgeable [49] message authentication code function for message msg with key key. $\mathcal{R}$ computes $\sigma = \text{RSig}(g^R, R, \{g^I, g^R, g^i\}, t)$. $\mathcal{R}$ sends $(\text{"}R\text{"}, g^r, Q_R, mac, \sigma)$ to $\mathcal{I}$. Note that $\mathcal{R}$ can attach a message m to this flow by immediately encrypting it with a symmetric cryptosystem keyed with k.
3. $\mathcal{I}$ verifies σ. $\mathcal{I}$ verifies that (g^i, PQ_I) is an unused prekey previously sent by $\mathcal{I}$. $\mathcal{I}$ computes $Q_k = \text{QRKey}_I(SQ_I, Q_R)$, $\kappa = \text{KDF}_1((g^r)^i \| (g^r)^I \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and $k = \text{KDF}_3(\kappa)$. $\mathcal{I}$ verifies the MAC and securely erases i, SQ_I, and Q_k. If a message was attached, $\mathcal{I}$ can decrypt it using k.

Developers should incorporate the safeguards discussed in Section 8 to securely implement ZDH.

Proposition 3 states a simplified security theorem for ZDH. A formal theorem and proof sketch appear in Appendix G.

Proposition 3 (*ZDH security*)

If the MAC is weakly unforgeable under chosen message attack [13], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then ZDH provides the security properties listed in Section 3.3 against adaptive corruptions when the response is sent by an honest party.

6.3 Design Discussion

The primary difference between ZDH and Spawn$^+$ is that ZDH does not use DREAD to enable $\mathcal{R}$ to derive the shared secret during online deniability attacks. ZDH also uses a MAC to achieve full key confirmation for $\mathcal{I}$ [49]. Without this MAC, ZDH would be vulnerable to an identity misbinding attack [67]; an active adversary $\mathcal{P}$ that replaced the ring signature with one for the ring $\{g^I, g^P, g^i\}$ would cause $\mathcal{I}$ to believe that it was communicating with $\mathcal{P}$ while sharing a key with $\mathcal{R}$.

In order to derive the shared secret for Spawn$^+$ from the exchanged information, a party must either know $\mathcal{R}$'s ephemeral secret, or $\mathcal{I}$'s ephemeral secret *and* either of the long-term secrets. In order to derive the shared secret for ZDH from the exchanged information, a party must know either $\mathcal{R}$'s ephemeral secret, or $\mathcal{I}$'s ephemeral and long-term secrets. The important difference is that in ZDH, a party that knows $\mathcal{R}$'s long-term secret and $\mathcal{I}$'s ephemeral secret cannot derive the key. This is pre-

cisely the knowledge available to $\mathcal{R}$ when it is trying to simulate $\mathcal{I}$ for an interactive third party. For this reason, the protocol explicitly lacks online deniability for $\mathcal{R}$, just like non-interactive Spawn and Spawn$^+$. This concession causes the prohibition of corrupt responders in Proposition 3, but also enables ZDH's efficient design.

ZDH's derivation of κ is similar to the derivation of the shared secret in 3DH [78]. However, 3DH also includes g^{iR} as input to its KDF. The purpose of this term in 3DH is to authenticate $\mathcal{R}$, but ZDH accomplishes this using a ring signature instead. Moreover, including this term in ZDH would break online deniability for $\mathcal{I}$, as it does in 3DH, because an interactive third party that chooses i and coerces $\mathcal{I}$ into sending g^i can derive the shared secret while preventing $\mathcal{I}$ from doing so.

7 XZDH

Spawn, 3DH, Spawn$^+$, and ZDH all share a similar weakness: they are two-flow protocols with weak forward secrecy. This makes them vulnerable to an attack where an active adversary modifies the first flow from $\mathcal{I}$ to use an adversarially controlled ephemeral key, captures and drops the response from $\mathcal{R}$, and then compromises $\mathcal{I}$'s long-term secret key [14]. $\mathcal{I}$ will never see the message, and the adversary will be able to decrypt it. Moreover, since long-term keys are usually meant to last for years, a long time may pass between $\mathcal{R}$ sending the message and the adversary compromising $\mathcal{I}$'s key. In practice, this attack requires a powerful adversary.

Signal's X3DH protocol [75] somewhat mitigates this weakness in 3DH by introducing the notion of *signed prekeys*. In contrast to the one-time prekeys used by 3DH, signed prekeys are signed by long-term secret keys and are reusable. Each user maintains one signed prekey on the prekey server, which is changed on a roughly weekly basis [75]. The combination of a one-time prekey and a signed prekey is called a *prekey bundle*. X3DH incorporates a Diffie-Hellman exchange between $\mathcal{I}$'s signed prekey and $\mathcal{R}$'s ephemeral key, and makes one-time prekeys optional. The benefit is that the aforementioned attack is thwarted if $\mathcal{I}$ changes its signed prekey and erases the old one before being corrupted by the adversary. Because the prekey is signed, it cannot be adversarially altered, and $\mathcal{I}$ controls the timing of key erasure. However, the use of signed prekeys in X3DH severely harms its offline deniability: transcripts can be forged only by one of the alleged participants, since the forger must complete a Diffie-Hellman exchange between $\mathcal{I}$'s signed prekey and $\mathcal{R}$'s long-term key.

XZDH is a variant of ZDH incorporating signed prekeys. $\mathcal{I}$'s signed prekey is g^Γ, with corresponding secret key $\Gamma \in \mathbb{Z}_q$. $\mathcal{I}$ uploads g^Γ to the prekey server alongside an existentially unforgeable digital signature [55] for g^Γ created using I as the signing key, denoted by $\mathrm{Sig}(g^I, I, g^\Gamma)$. The public verification function $\mathrm{SVerif}(g^I, m, \xi)$ returns TRUE if ξ is a valid signature for message m created using I, and FALSE otherwise.

XZDH is identical to ZDH, except that it modifies the derivation of κ to be $\kappa = \mathrm{KDF}_1(g^{ir} \| g^{\Gamma r} \| g^{Ir} \| Q_k)$ and includes g^Γ in its MAC and RSig messages, as depicted in Figure 4. The first flow in Figure 4 represents the complete prekey bundle that $\mathcal{R}$ downloads from the server. g^Γ is reused across sessions, but $\mathcal{I}$ replaces it regularly (e.g., once a week). XZDH provides online deniability, the same forward secrecy characteristics as X3DH, and the same offline deniability as ZDH—anyone can forge an XZDH transcript between $\mathcal{I}$ and $\mathcal{R}$ given only g^I, g^R, and a prekey bundle containing g^Γ and $\mathrm{Sig}(g^I, I, g^\Gamma)$. These values are all publicly distributed.

Proposition 4 states a simplified security theorem for XZDH.

Proposition 4 (*XZDH security*)

If the MAC is weakly unforgeable under chosen message attack [13], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in $\mathbb{G}$, then XZDH provides the security properties listed in Section 3.3 against adaptive corruptions when the response is sent by an honest party.

A formal theorem and proof sketch appear in Appendix H.

8 Secure Messaging Integration

This section covers several practical considerations involved in securely implementing DAKEZ, Spawn$^+$, ZDH, and XZDH for use in secure messaging tools.

All elements in the ciphertexts must be of fixed length to prevent potentially exploitable parsing confusion. The identifiers for the parties (e.g., "I" and "R") may be cryptographic hashes of usernames, or they may be the long-term public keys. Identifiers can also include device codes to differentiate between devices owned by a user. In general, the problem of supporting multi-device conversations is difficult and mostly orthogonal. Any multi-device technique that works for 3DH or X3DH should also be applicable to our DAKEs; the most common techniques in practice—replicating one key or generating per-device keys [9]—are directly applicable.

While we used generic groups in our definitions, the primitives should be implemented using elliptic curve cryptography for maximum performance. Parties must also ensure that the Diffie-Hellman contributions they receive are in the expected group and are not the identity element. Otherwise, the implementation may be vulnerable to identity-element [40], small-subgroup [69], or invalid-curve [6] attacks.

As discussed in Section 6, interactive DAKEZ and Spawn$^+$ provide online deniability for both $\mathcal{I}$ and $\mathcal{R}$, while ZDH and XZDH do not. Consequently, developers should consider a hybrid approach where interactive DAKEZ is attempted first, and non-interactive ZDH or XZDH is used as a fallback option. This hybrid approach enables non-interactive messaging and minimizes the use of one-time prekeys, but sacrifices online deniability for $\mathcal{R}$ when $\mathcal{I}$ is willing to use the fallback option.

Using our DAKEs non-interactively requires an untrusted central server to store prekeys. If prekey submissions are not authenticated, then malicious users can perform denial-of-service attacks. To preserve the deniability of the overall protocol, one-time prekeys should never be digitally signed. The best approach is to authenticate prekey uploads using a DAKEZ exchange between the uploader and the server, which preserves deniability. As an added safeguard, the server can require a ZKPK of the private keys associated with the prekeys.

All of our DAKEs permit authenticating the shared session state Φ as part of the exchange. Theoretically, this ensures that the DAKE is "bound" to a session of the messaging protocol. The higher-level protocol should always include its implicit or explicit session identifier in Φ to perform this binding. In practice, Φ also allows both sides to cryptographically verify some beliefs they have about the session. For example, in an application that assigns some attribute to users before a conversation (e.g., a networked game in which players take on specific roles), the expected attributes (expressed in fixed length) should be included in Φ. If the DAKE succeeds, then the participants know that they agree about the attribute values.

Implementations of our DAKEs should follow standard secure coding practices for cryptographic software. Cryptographic operations with secret data should be performed in constant time. Secret values, including intermediaries, should be stored in locked private memory pages, and ephemeral keys should be securely erased in a manner guaranteed to persist through compiler optimizations. All hash functions should be domain separated using protocol-specific null-terminated prefixes.

Deployments of our DAKEs should always include tools to forge key exchanges; this improves plausible deniability in practice. Specifically, implementations of RSig from Section 4.3 should use the same code for honest and forged authentication. To accomplish this in constant time, developers can use conditional move operations to copy the appropriate keys into memory regions for the calculations, and again to move the results into the appropriate positions in the proof.

8.1 Incorporating Quantum Resistance

Selecting a quantum-resistant KEM to use in an implementation is challenging. There is not yet a scheme that indisputably dominates the others in terms of security, efficiency, and deployability. Additionally, many schemes are covered by actively enforced patents, which may further restrict the available options.

Two of the most prominent schemes are the supersingular isogeny Diffie-Hellman (SIDH) [60] and the "New Hope" [4] protocols. New Hope has faster implementations, and relies on older hardness assumptions. However, there is disagreement about the vulnerability of lattice-based cryptography to quantum cryptanalysis [28]. SIDH relies on newer hardness assumptions, but has a lower communication cost. SIDH also has a very simple parameter selection process to achieve a chosen security level. Due to its relative novelty, attacks against SIDH are still being discovered [52].[6] In Section 10, we evaluate the performance of SIDH and New Hope in relation to the core DAKEs.

Developers should consider all of these factors when selecting a quantum-resistant KEM. Note that aside from their effect on message sizes, the choice of KEM is independent of other design choices, and the KEM can be changed as a protocol evolves.

9 Implementation

Although many DAKEs have been described in the literature, very few of them have ever been implemented. If a DAKE is designed with the intention of protecting real users, then producing a proof-of-concept implementation has several benefits. In addition to enabling experiments to quantify real-world performance, the implementation process often reveals practical hurdles limiting deployability that cannot be identified from a detached security analysis, necessitating improved designs. For these reasons, we developed a prototype implemen-

6 The attack described by Galbraith et al. [52] does not apply here because we use SIDH for ephemeral-ephemeral exchanges.

tation of our DAKEs in C using production-ready cryptographic libraries targeting a 128-bit security level. Our implementation employs all of the secure coding techniques mentioned in Section 8, and supports both New Hope and SIDH as instantiations of the generalized KEM functions in Section 4.4.

The remainder of this section provides implementation details. Section 9.1 lists the third-party libraries we used, Section 9.2 describes our additions to the elliptic curve library, and Section 9.3 lists our chosen primitives.

9.1 Libraries

We used version 1.1.0e of the OpenSSL library in our implementation for its cryptographic randomness source and its AEAD primitives. We used the New Hope implementation with AVX2 extensions from Alkim et al. [4], and the AMD64 SIDH library from Costello et al. [33]. Our implementation uses the twisted Edwards curve associated with Ed25519 [17] for our group operations. We used the `ed25519-donna` library [76] with SSE2 extensions due to its speed and portability. We pack curve points generated by the library into 32 bytes, which includes the y-coordinate and a parity bit of the x-coordinate, for transmission and storage. When unpacked, the points can be multiplied by the subgroup order and compared to the identity element to ensure that they are in the correct subgroup and not on the twist, although this is not required for the security of Ed25519 signatures [17]. We used the Keccak Code Package [8] for its SHA-3 implementation. To hash curve points, we hash the packed 32 bytes with SHA-3.

9.2 `Ed25519-donna` Additions

Although Curve25519 [16] is used in Signal [79] and is generally well-regarded in the secure messaging community, existing libraries cannot be used to implement our DAKEs without modification. X25519 [16] libraries typically store points in Montgomery form, which discards the sign and complicates ElGamal implementations. Ed25519 [17] libraries store points on a birationally equivalent twisted Edwards curve, but do not provide independent point addition or scalar multiplication functions. Consequently, we made several additions to `ed25519-donna` to implement our prototype.

We implemented constant-time scalar multiplication with variable bases using the windowed double-and-add-always algorithm. We select random scalars using the X25519 secret generation procedure [16]. We also adapted existing code to produce constant-time implementations of point negation, scalar subtraction, conditional memory copies, and scalar equality tests.

9.3 Primitive Instantiations

To instantiate our DREAD scheme from Section 4.2, we used OpenSSL's implementation of AES256-OCB with a 16-byte tag and a 15-byte nonce. We use the first 15 bytes of the L hash derived in the NIZKPK as the nonce for OCB mode, since its probability of repeating is negligible. We select $\mathcal{K}$ by multiplying the Ed25519 base point by a random scalar.[7] To derive the AEAD key from $\mathcal{K}$, we pack the point into its 32-byte representation and use SHA3-256 as a KDF. We also use SHA3-256 as the hash H used to compute c in RSig.

Our implementations support arbitrary caller-defined user identifiers and shared session state Φ. Our ZDH and XZDH implementations use KMAC256 [63] as the MAC, since it is based on SHA-3. Although all of our fields are of fixed length, we implemented the complete KMAC256 padding scheme to reflect the overhead of general-purpose KMAC libraries.

10 Performance Evaluation

When deciding whether or not to include a DAKE in a secure messaging application, one of the most important considerations for developers is the expected performance impact on both users and infrastructure. We performed an analysis of our implementation to precisely quantify this impact by answering the following questions for DAKEZ, Spawn$^+$, ZDH, and XZDH:

- What is the computational overhead for generating long-term keys and participating in key exchanges?
- How large are the long-term public keys?
- How large are the prekeys (where applicable)?
- How much data is transferred during a key exchange?
- How does the performance of our DAKEs compare to other state-of-the-art AKEs?
- What is the overhead of quantum resistance?

We compared the performance of our DAKEs to four AKEs with weaker security properties—ECDH, 3DH [78], X3DH [75], and SIGMA-R [67]—and the three existing DAKEs with the same properties—Φ_{idre}, RS-DAKE, and Spawn [93]. (Φ_{idre} [93] is a more efficient instantiation of Φ_{dre} [42].)

7 This is not a safe method for selecting a random point in general, since the discrete log of the point is known. The method is safe here because the result is only input into a KDF.

Table 1. Comparison of DAKE features, computational performance, and size requirements

	ECDH	3DH	X3DH	SIGMA-R	Φ_{idre}	RSDAKE	Spawn	DAKEZ	Spawn$^+$	ZDH	XZDH
Offline Deniable	●	●	○	○	●	●	●	●	●	●	●
Online Deniable	●	-	-	-	●	●	○	●	○	○	○
Authenticated	-	●	●	●	●	●	●	●	●	●	●
Non-Interactive	●	●	●	-	-	-	●	-	●	●	●
Forward Secrecy	-	-	○	●	●	●	-	●	-	-	○
Proof Model	SM	ROM	ROM	ROM	SM	SM	SM	ROM	ROM	ROM	ROM
Public Key Generation [ms]	-	0.0228 (0.0012)	0.0240 (0.0013)	0.0240 (0.0012)	0.40 (0.01)	206 (8)	206 (4)	0.0440 (0.0016)	0.0429 (0.0016)	0.0441 (0.0018)	0.0444 (0.0017)
Exchange [ms]	0.1733 (0.0033)	0.4229 (0.0050)	0.5533 (0.0056)	0.3478 (0.0048)	13 (2)	6 630 (50)	3 390 (20)	1.094 (0.014)	1.3683 (0.0082)	0.778 (0.013)	0.9217 (0.0069)
Flows	2	2	2	4	9	3	2	3	2	2	2
Public Key [B]	-	32	32	32	415	395	992	32	32	32	32
Prekey [B]	-	32	32+96	-	-	-	938	-	32	32	32+96
Exchange [B]	64	80	80	272	5 140	7 598	73 763	464	512	304	304

● = provides property; ○ = partially provides property; - = does not provide property / not applicable; SM = standard model; ROM = random oracle model. Standard deviations are in parentheses. "Forward secrecy" is the strong variant [14] (all schemes have weak forward secrecy). Prekeys are listed as (one-time)+(signed) sizes.

3DH, X3DH, ZDH, and XZDH are all designed for non-interactive messaging (e.g., text messaging). Our ZDH and XZDH schemes use ROM-based ring signatures to achieve strong deniability, which 3DH and X3DH lack. Φ_{idre}, RSDAKE, Spawn, and DAKEZ are all designed for interactive communication. Our DAKEZ scheme is similar to RSDAKE, except that it avoids the need for digital signatures, and uses ROM-based ring signatures. Φ_{idre} provides strong deniability using DRE in the standard model, but it requires an excessive number of network round trips to do so.

In order to fairly compare the high-level designs, we implemented ECDH, 3DH, X3DH, and SIGMA-R using the primitives described in Section 9: group operations over the Ed25519 twisted Edwards curve, SHA3-256 for hashing and key derivation, and KMAC256 for message authentication. We used EdDSA to sign X3DH and XZDH prekeys. We implemented SIGMA-R as described by Di Raimondo et al. [36] using EdDSA for signatures, and without the identity protection mechanism in the OTR protocol [3]. For Φ_{idre}, RSDAKE, and Spawn, we used the primitives selected in UG15 [93], which are secure in the standard model. Our experiments use 8-byte identifiers for users, and no higher-level shared session data. We include identifiers in the transmission costs for non-interactive protocols.

We performed key exchanges using all 11 protocols on one Intel Skylake core pinned to 4.0 GHz with 8 MB of L3 cache and Intel Turbo Boost disabled. For each protocol, we measured the thread-specific CPU time required to perform a key exchange and, independently, to generate a long-term public key. We gathered 100,000 measurements for each protocol except for Φ_{idre}, RSDAKE, and Spawn, where the computational costs restricted us to 1,000 measurements each. We also measured the size of the long-term public keys, prekeys (where applicable), and the total key exchange traffic (including one-time prekeys) for each protocol. Table 1 compares the schemes and presents our results. The "partial" forward secrecy for X3DH and XZDH in the table denotes the characteristics described in Section 7.

Table 1 shows the dramatic difference between the protocol classes. The schemes without authentication and strong deniability (left) require less than 1 ms to generate long-term keys or complete key exchanges. In contrast, the strongly deniable schemes from UG15 [93] (middle) are prohibitively more expensive: RSDAKE and Spawn require several *seconds* for exchanges and require kilobytes of transmission, while Φ_{idre} requires over four network round trips. Despite the computational efficiency of Φ_{idre}, it is often the slowest protocol in practice due to network round-trip time dominating all other costs in typical networks. DAKEZ, Spawn$^+$, ZDH, and XZDH (right) provide authentication and strong deniability while nearly matching the weaker schemes' performance: our DAKEs require roughly 1 ms of CPU time and only slightly more communication than SIGMA-R. The one-time prekeys are 32 bytes for all non-interactive schemes except Spawn, which uses 938 bytes. The signed prekeys for X3DH and XZDH are 96 bytes.

Our results show that DAKEZ, ZDH, and XZDH are the first schemes to offer strong deniability with only slightly more overhead than AKEs used in popular secure messaging protocols. While comparing Spawn to

Table 2. Quantum resistance performance overhead

Scheme	Δ Time [ms]	PQ_I [bytes]	Q_R [bytes]
New Hope	+0.0542 (0.0041)	+1824	+2048
SIDH	+63.8 (1.5)	+768	+768

Spawn$^+$ demonstrates the value of ROM-based primitives, DAKEZ, ZDH, and XZDH are all more efficient. For non-interactive settings, XZDH provides better forward secrecy than ZDH with very small overhead.

To quantify the overhead of quantum-resistant exchanges, we used the same experimental setup to measure SIDH [33] and New Hope [4]. Table 2 shows the cost to use the schemes in our DAKEs in terms of the added key exchange time and the sizes of PQ_I and Q_R. New Hope requires less computation than the DAKEs themselves, but it also adds almost 4 KiB of data transmission to the overall exchange. SIDH requires under 2 KiB of extra data, but it adds over 60 ms of computation. In real secure messaging deployments, the infrequency of DAKE operations (compared to message deliveries) likely makes the overhead of either scheme negligible. In XZDH, using a scheme for which static-ephemeral exchanges are safe enables reuse of PQ_I as part of the signed prekey, which further reduces costs.

11 KCI Attacks

One aspect of online deniability that is often overlooked in the literature is the relationship between strongly deniable key exchange protocols and key compromise impersonation attacks. A KCI attack begins when the long-term secret key of a user of a vulnerable DAKE is compromised. With this secret key, an adversary can impersonate *other* users *to* the owner of the key. DAKEs offering online deniability, such as Φ_{idre}, RSDAKE, Spawn, DAKEZ, Spawn$^+$, ZDH, and XZDH, are inherently vulnerable to key compromise impersonation attacks. Moreover, "vulnerability" in this context is actually a desirable property.

In theory, a user who claims to cooperate with a judge may justifiably refuse to reveal their long-term secret key because it would make them vulnerable to a KCI attack. The design of strongly deniable DAKEs makes it impossible for the user to provide proof of communication to the judge without also revealing their long-term secret key. This is the primary benefit and motivation of this class of key exchanges: they prevent a judge and informant from devising a protocol wherein the judge is given cryptographic proof of communication while the informant suffers no repercussions.

However, this scenario may be mostly theoretical. The more common case in practice may be the one in which the judge has access to the user's long-term secret keys. A typical real-world example is when a user is forced to surrender and unlock their mobile device with a secure messaging application installed; American and Canadian border agents currently exercise this power over travelers [56, 57]. In this situation, the KCI "vulnerability" also becomes an asset. We use non-interactive Spawn$^+$ as an example in the following discussion, but the ideas can be extended to the other DAKEs.

11.1 KCI Attacks Against Spawn$^+$

The security of Spawn$^+$ does not require trusting the central server used to distribute prekeys. However, if we allow a scenario in which the user's keys have been compromised but the central server has not, then we can achieve better plausible deniability. The user may ask the central server in advance to assist with a forged conversation, casting doubt on all conversations conducted by the judge using the compromised device.

If the judge attempts to act as $\mathcal{I}$ in a conversation using the compromised device, then the user (or a trusted accomplice with access to the long-term secret I) can impersonate $\mathcal{R}$ by executing RSig with g^I and I instead of g^R and R. In practice, the user (or accomplice) simply needs to run the protocol honestly, but pretend to be $\mathcal{R}$ in their response to the prekey.

If the judge attempts to act as the responder $\mathcal{R}$ of a conversation using the compromised device, then we can somewhat improve the situation, but we cannot offer full deniability. The user must ask the central server to return a false prekey for $\mathcal{I}$ that was generated by the user or their trusted accomplice, and to redirect all traffic to the associated forging device. This false prekey must be returned to the judge when they request one. The user can derive the shared secret k by decrypting the DREAD ciphertext using I instead of R. In practice, the judge can always bypass this forgery attempt by obtaining a legitimate prekey for $\mathcal{I}$ and using this to respond using R. This is a fundamental limitation of Spawn$^+$ that also applies to Spawn, and is conjectured to be insurmountable by a two-flow non-interactive protocol [93]. The design of ZDH in Section 6.2 explicitly acknowledges this limitation to improve performance.

11.2 Limiting or Preventing KCI Attacks

If the KCI vulnerability is undesirable, it is possible to make all of our DAKEs more resilient to it while maintaining their deniability properties. To do

so, a protocol like Spawn$^+$ can be altered to include long-term "forger" keys for all participants. For example, initiator $\mathcal{I}$ would distribute both g^I and g^{F_I} as its public key, where g^{F_I} is $\mathcal{I}$'s public forging key, and F_I is the associated secret key. The second flow in the Spawn$^+$ key exchange is then altered so that the ciphertext and proof are computed as follows: $\gamma = \text{DREnc}(g^I, g^{F_R}, g^r \| Q_R, \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I)$, and $\sigma = \text{RSig}(g^R, R, \{g^{F_I}, g^R, g^i\}, \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I \| \gamma \| \Phi)$.

In general, this transformation changes all long-term public keys in the protocol that are not used in the "honest" case to reference the forging keys instead. This alteration allows the forging keys to be stored more securely than the "honest" public keys; since the forging keys are not needed for normal operation, they may be stored offline (e.g., on paper in a vault). Alternatively, if a user (or developer) is more concerned about preventing KCI attacks than providing online deniability, the forging secret keys can be destroyed immediately after generation; this will sacrifice online deniability for the user, but also prevent KCI attacks against them.

Counterintuitively, implementing this option for users can actually provide both benefits in practice. Consider a secure messaging application that asks users whether or not they would like to save forging keys during setup. Even if most users select the default option to securely erase the forging keys, thereby preventing them from performing the online forgery techniques described above, a judge does not generally know the choice of a particular user. Consequently, a judge that engages in a conversation using a compromised device is given two explanations: either the conversation is genuine, or the owner of the device was one of the users that elected to store the forgery keys and they are using those keys to forge the conversation. The result is that a degree of plausible deniability is preserved, even though most users in this scenario become immune to KCI attacks.

The same general transformation works for the other DAKEs described in this paper. Note that trust establishment (e.g., physical exchange of key fingerprints) must cover both keys in this scheme.

12 Conclusion

Due to practical weaknesses with existing DAKEs, the most popular "deniable" secure messaging protocols still lack the strongest deniability notions considered by cryptographers. In this work, we proposed DAKEZ, ZDH, and XZDH as a means to bridge this gap. Our DAKEs retain the most important security and deployability properties for modern secure messaging environments while providing provably strong deniability. We presented concrete instantiations that overcome the limitations preventing previous DAKEs from being widely used. With networks growing increasingly hostile and security breaches becoming commonplace, our need to maintain control over our information has never been more important. It is our hope that the substantially improved DAKEs presented here will eventually be deployed in popular messaging protocols, and subsequently provide strong deniability for real-world users.

Acknowledgments

The authors would like to thank Makulf Kohlweiss and the anonymous reviewers for their insightful comments and feedback, Alfredo Rial Duran for his exceptionally detailed verification of our security proofs, Peter Schwabe for his comments on contributiveness and New Hope, and David Jao for his comments on SIDH. We gratefully acknowledge the support of NSERC for grants RGPIN-2017-03858 and STPGP-463324-14.

References

[1] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. 1-out-of-n Signatures from a Variety of Keys. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 415–432. Springer, 2002.

[2] Ben Adida, Susan Hohenberger, and Ronald L Rivest. Ad-Hoc-Group Signatures from Hijacked Keypairs. In *in DIMACS Workshop on Theft in E-Commerce*, 2005.

[3] Chris Alexander and Ian Goldberg. Improved User Authentication in Off-The-Record Messaging. In *Workshop on Privacy in the Electronic Society*, pages 41–47. ACM, 2007.

[4] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange—A New Hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343. USENIX Association, 2016.

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[6] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, and Scott Vanstone. Validation of Elliptic Curve Public Keys. In *Public Key Cryptography—PKC 2003*, pages 211–223. Springer, 2003.

[7] Diego de Freitas Aranha and Conrado Porto Lopes Gouvêa. RELIC is an Efficient LIbrary for Cryptography, 2009. URL https://github.com/relic-toolkit/relic. Accessed 2017-08-11.

[8] Gilles Van Assche. Keccak Code Package, 2013. URL https://github.com/gvanas/KeccakCodePackage. Accessed 2017-08-11.

[9] Erinn Atwater and Urs Hengartner. Shatter: Using Threshold Cryptography to Protect Single Users with Multiple Devices. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 91–102. ACM, 2016.

[10] Mihir Bellare and Phillip Rogaway. Entity Authentication and Key Distribution. In *Advances in Cryptology–CRYPTO'93*, pages 232–249. Springer, 1993.

[11] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[12] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In *Annual International Cryptology Conference*, pages 26–45. Springer, 1998.

[13] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of the Cipher Block Chaining Message Authentication Code. *Journal of Computer and System Sciences*, 61(3): 362–399, 2000.

[14] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated Key Exchange Secure Against Dictionary Attacks. In *Advances in Cryptology–EUROCRYPT*, pages 139–155. Springer, 2000.

[15] Adam Bender, Jonathan Katz, and Ruggero Morselli. Ring Signatures: Stronger Definitions, and Constructions without Random Oracles. In *Theory of Cryptography*, pages 60–79. Springer, 2006.

[16] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography—PKC 2006*, pages 207–228. Springer, 2006.

[17] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2): 77–89, 2012.

[18] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 416–432. Springer, 2003.

[19] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-Record Communication, or, Why Not To Use PGP. In *Workshop on Privacy in the Electronic Society*, pages 77–84. ACM, 2004.

[20] Colin Boyd, Wenbo Mao, and Kenneth G Paterson. Key Agreement using Statically Keyed Authenticators. In *Applied Cryptography and Network Security*, pages 248–262. Springer, 2004.

[21] Emmanuel Bresson, Jacques Stern, and Michael Szydlo. Threshold Ring Signatures and Applications to Ad-hoc Groups. In *Annual International Cryptology Conference*, pages 465–480. Springer, 2002.

[22] Jan Camenisch and Markus Stadler. Efficient Group Signature Schemes for Large Groups. In *Annual International Cryptology Conference*, pages 410–424. Springer, 1997.

[23] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 136–145. IEEE, 2001.

[24] Ran Canetti and Hugo Krawczyk. Security Analysis of IKE's Signature-based Key-Exchange Protocol. In *Advances in Cryptology–CRYPTO'02*, pages 143–161. Springer, 2002.

[25] Ran Canetti and Hugo Krawczyk. Universally Composable Notions of Key Exchange and Secure Channels. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 337–351. Springer, 2002.

[26] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally Composable Security with Global Setup. In *Theory of Cryptography Conference*, pages 61–85. Springer, 2007.

[27] CertiVox. MIRACL Cryptographic SDK, 2012. URL https://github.com/miracl/MIRACL. Accessed 2017-08-11.

[28] Sanjit Chatterjee, Neal Koblitz, Alfred Menezes, and Palash Sarkar. Another Look at Tightness II: Practical Issues in Cryptography. *IACR Cryptology ePrint Archive*, 2016:360, 2016.

[29] Sherman SM Chow, Siu-Ming Yiu, and Lucas CK Hui. Efficient Identity Based Ring Signature. In *International Conference on Applied Cryptography and Network Security*, pages 499–512. Springer, 2005.

[30] Sherman SM Chow, Matthew Franklin, and Haibin Zhang. Practical Dual-Receiver Encryption. In *Cryptographers' Track at the RSA Conference*, pages 85–105. Springer, 2014.

[31] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Improved OR Composition of Sigma-Protocols. In *Theory of Cryptography Conference*, pages 112–141. Springer, 2016.

[32] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Online/Offline OR Composition of Sigma Protocols. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 63–92. Springer, 2016.

[33] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Advances in Cryptology*. Springer, 2016.

[34] Ronald Cramer and Victor Shoup. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[35] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *Annual International Cryptology Conference*, pages 174–187. Springer, 1994.

[36] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Secure Off-the-Record Messaging. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages 81–89. ACM, 2005.

[37] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Deniable Authentication and Key Exchange. In *Conference on Computer and Communications Security*, pages 400–409. ACM, 2006.

[38] Theodore Diament, Homin K Lee, Angelos D Keromytis, and Moti Yung. The Dual Receiver Cryptosystem and Its Applications. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 330–343. ACM, 2004.

[39] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[40] Roger Dingledine. Tor security advisory: DH handshake flaw, 2005. URL http://archives.seul.org/or/announce/Aug-2005/msg00002.html. Accessed 2017-08-11.

[41] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. Anonymous Identification in Ad Hoc Groups. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 609–626. Springer, 2004.

[42] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. Composability and On-Line Deniability of Authentication. In *Theory of Cryptography*, pages 146–162. Springer, 2009.

[43] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-Malleable Cryptography. In *SIAM Journal on Computing*, pages 542–552, 1998.

[44] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent Zero-Knowledge. In *Symposium on Theory of Computing*, pages 409–418. ACM, 1998.

[45] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[46] Sebastian Faust, Markulf Kohlweiss, Giorgia Azzurra Marson, and Daniele Venturi. On the Non-malleability of the Fiat-Shamir Transform. In *International Conference on Cryptology in India*, pages 60–79. Springer, 2012.

[47] Amos Fiat and Moni Naor. Broadcast Encryption. In *Annual International Cryptology Conference*, pages 480–491. Springer, 1993.

[48] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology–CRYPTO'86*, pages 186–194. Springer, 1987.

[49] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi. Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 452–469, 2016.

[50] Marc Fischlin and Sogol Mazaheri. Notions of Deniable Message Authentication. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*, WPES '15, pages 55–64. ACM, 2015. ISBN 978-1-4503-3820-2. 10.1145/2808138.2808143.

[51] Marc Fischlin and Cristina Onete. Relaxed Security Notions for Signatures of Knowledge. In *International Conference on Applied Cryptography and Network Security*, pages 309–326. Springer, 2011.

[52] Steven D Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In *Advances in Cryptology–ASIACRYPT*, pages 63–91. Springer, 2016.

[53] Juan A Garay, Philip MacKenzie, and Ke Yang. Strengthening Zero-Knowledge Protocols Using Signatures. In *Eurocrypt*, volume 2656, pages 177–194. Springer, 2003.

[54] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. In *Advances in Cryptology–EUROCRYPT*, pages 295–310. Springer, 1999.

[55] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17(2): 281–308, 1988.

[56] Mark Gollom. Alain Philippon phone password case: Powers of border agents and police differ, 2015. URL http://www.cbc.ca/news/1.2983841. Accessed 2017-08-11.

[57] Loren Grush. A US-born NASA scientist was detained at the border until he unlocked his phone, 2017. URL https://www.theverge.com/2017/2/12/14583124/. Accessed 2017-08-11.

[58] Viet Tung Hoang, Jonathan Katz, and Alex J Malozemoff. Automated Analysis and Synthesis of Authenticated Encryption Schemes. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 84–95. ACM, 2015.

[59] Dennis Hofheinz, Jörn Müller-Quade, and Rainer Steinwandt. Initiator-Resilient Universally Composable Key Exchange. In *European Symposium on Research in Computer Security*, pages 61–84. Springer, 2003.

[60] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[61] Shaoquan Jiang and Reihaneh Safavi-Naini. An Efficient Fully Deniable Key Exchange Protocol. In *Financial Cryptography and Data Security*. Springer, 2008.

[62] Jonathan Katz. Efficient and Non-Malleable Proofs of Plaintext Knowledge and Applications. In *Advances in Cryptology–EUROCRYPT*, pages 211–228. Springer, 2003.

[63] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 Derived Functions. *NIST Special Publication*, 800:185, 2016.

[64] Taechan Kim and Razvan Barbulescu. Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case. In *Advances in Cryptology–CRYPTO'16*, pages 543–571. Springer, 2016.

[65] Neal Koblitz and Alfred J Menezes. The random oracle model: a twenty-year retrospective. *Designs, Codes and Cryptography*, 77(2-3):587–610, 2015.

[66] Hugo Krawczyk. SKEME: A Versatile Secure Key Exchange Mechanism for Internet. In *Network and Distributed System Security Symposium*, pages 114–127. IEEE, 1996.

[67] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003.

[68] Adam Langley. Intent to Implement and Ship: CECPQ1 for TLS, 2016. URL https://groups.google.com/a/chromium.org/forum/#!topic/security-dev/DS9pp2U0SAc. Accessed 2017-08-11.

[69] Chae Hoon Lim and Pil Joong Lee. A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup. In *Advances in Cryptology—CRYPTO '97*, pages 249–263. Springer-Verlag, 1997.

[70] Yehuda Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 394–403. IEEE, 2003.

[71] Joseph K Liu, Victor K Wei, and Duncan S Wong. Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups. In *Australasian Conference on Information Security and Privacy*, pages 325–335. Springer, 2004.

[72] Luke Rosiak. Here's Cryptographic Proof That Donna Brazile Is Wrong, WikiLeaks Emails Are Real, 2016. URL http://dailycaller.com/2016/10/21/heres-cryptographic-proof-that-donna-brazile-is-wrong-wikileaks-emails-are-real/. Accessed 2017-08-11.

[73] Ben Lynn. The Pairing-Based Cryptography Library, 2006. URL https://crypto.stanford.edu/pbc/. Accessed 2017-08-11.

[74] Marry Madden. Americans' Attitudes About Privacy, Security and Surveillance, 2015. URL http://www.pewinternet.org/2015/05/20/americans-attitudes-about-privacy-security-and-surveillance/. Accessed 2017-08-11.

[75] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol, 2016. URL https://whispersystems.org/docs/specifications/x3dh/. Accessed 2017-08-11.

[76] Andrew Moon. Implementations of a fast Elliptic-curve Digital Signature Algorithm, 2012. URL https://github.com/floodyberry/ed25519-donna. Accessed 2017-08-11.

[77] Moni Naor and Moti Yung. Public-key Cryptosystems Provably Secure against Chosen Ciphertext Attacks. In *Proceedings of 22nd Annual ACM Symposium on Theory of Computing*, pages 427–437. ACM, 1990.

[78] Open Whisper Systems. Simplifying OTR deniability, 2013. URL https://www.whispersystems.org/blog/simplifying-otr-deniability. Accessed 2017-08-11.

[79] Open Whisper Systems. Open Whisper Systems, 2013. URL https://www.whispersystems.org/. Accessed 2017-08-11.

[80] Open Whisper Systems. Open Whisper Systems partners with WhatsApp to provide end-to-end encryption, 2014. URL https://www.whispersystems.org/blog/whatsapp/. Accessed 2017-08-11.

[81] Open Whisper Systems. Open Whisper Systems partners with Google on end-to-end encryption for Allo, 2016. URL https://whispersystems.org/blog/allo/. Accessed 2017-08-11.

[82] Open Whisper Systems. Facebook Messenger deploys Signal Protocol for end to end encryption, 2016. URL https://whispersystems.org/blog/facebook-messenger/. Accessed 2017-08-11.

[83] OTR Development Team. Off-the-Record Messaging Protocol version 3, 2016. URL https://otr.cypherpunks.ca/Protocol-v3-4.1.1.html. Accessed 2017-08-11.

[84] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm, 2016. URL https://whispersystems.org/docs/specifications/doubleratchet/. Accessed 2017-08-11.

[85] Ronald L Rivest, Adi Shamir, and Yael Tauman. How to Leak a Secret. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 552–565. Springer, 2001.

[86] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM, 2002.

[87] Sven Schäge. TOPAS: 2-Pass Key Exchange with Full Perfect Forward Secrecy and Optimal Communication Complexity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1224–1235. ACM, 2015.

[88] John M Schanck, William Whyte, and Zhenfei Zhang. Circuit-extension handshakes for Tor achieving forward secrecy in a quantum world. *Proceedings on Privacy Enhancing Technologies*, 2016(4):219–236, 2016.

[89] Claus-Peter Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[90] Hovav Shacham and Brent Waters. Efficient Ring Signatures without Random Oracles. In *Public Key Cryptography*, pages 166–180. Springer, 2007.

[91] Gene Tsudik. Message Authentication with One-Way Hash Functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.

[92] Nik Unger. *Deniable Key Exchanges for Secure Messaging*. PhD thesis, University of Waterloo, 2015.

[93] Nik Unger and Ian Goldberg. Deniable Key Exchanges for Secure Messaging. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1211–1223. ACM, 2015.

[94] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure Messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, 2015.

[95] Shabsi Walfish. *Enhanced Security Models for Network Protocols*. PhD thesis, New York University, 2008.

[96] Shangping Wang, Rui Ma, Yaling Zhang, and Xiaofeng Wang. Ring signature scheme based on multivariate public key cryptosystems. *Computers & Mathematics with Applications*, 62(10):3973–3979, 2011.

[97] Weiqiang Wen, Libin Wang, and Min Xie. One-Round Deniable Key Exchange with Perfect Forward Security. Technical Report 2014/904, Cryptology ePrint Archive, 2014. URL https://eprint.iacr.org/2014/661.

[98] Hu Xiong, Zhiguang Qin, and Fagen Li. A Taxonomy of Ring Signature Schemes: Theory and Applications. *IETE Journal of Research*, 59(4):376–382, 2013.

[99] Andrew Chi-Chih Yao and Yunlei Zhao. OAKE: A New Family of Implicitly Authenticated Diffie-Hellman Protocols. In *Conference on Computer and Communications Security*, pages 1113–1128. ACM, 2013.

[100] Kazuki Yoneyama and Kazuo Ohta. Ring Signatures: Universally Composable Definitions and Constructions. *Information and Media Technologies*, 2(4):1038–1051, 2007.

[101] Taek-Young Youn, Changhoon Lee, and Young-Ho Park. An efficient non-interactive deniable authentication scheme based on trapdoor commitment schemes. *Computer Communications*, 34(3):353–357, 2011.

[102] Fangguo Zhang and Kwangjo Kim. ID-Based Blind Signature and Ring Signature from Pairings. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 533–547. Springer, 2002.

A Online Deniability Attacks

In this work, online deniability is generally discussed in an abstract manner that can obscure the implications. This appendix describes two illustrative types of attacks—coercive judges and malicious participants—against two popular secure messaging protocols lacking online deniability—OTRv3 and Signal. Our DAKEs are immune to these attacks because the coerced or malicious participant can simulate the victim; this invalidates the trustworthiness of any transcript that they help to generate, since they may be secretly impersonating the other user in the conversation.

Because the purpose of this appendix is to present technical details, we assume that the reader is familiar with the OTRv3 [83], X3DH [75], and double ratchet [84] specifications. The examples adopt the notation of the relevant specifications. For simplicity, the examples omit irrelevant protocol and verification steps.

A.1 Coercive Judges

In this attack, an online judge coerces a protocol participant into interactively proving that messages were authored by a victim, without compromising long-term secrets. The participant shares a secure channel with the judge. Deviations from the protocol are detectable. The same approach works for both OTRv3 and Signal:

1. The AKE is completed between the victim and the judge. The coerced participant provides the authenticating information necessary to cause the victim to believe they are communicating with the participant, rather than the judge.
2. The judge conducts the resulting conversation normally, with no need for the participant other than relaying ciphertexts. The judge can be certain that all messages they exchange are with the victim.

A.1.1 OTRv3

In this example, Judson is the judge, Alice is the victim, and Bob is the coerced participant. Alice and Bob are the entities described in the OTRv3 specification [83].

1. Bob establishes a connection to Alice.
2. Judson picks random values r and x.
3. Judson sends $\text{AES}_r(g^x)$ and $\text{HASH}(g^x)$ to Alice through Bob.
4. Alice replies with g^y.
5. Judson computes M_B normally, forces Bob to produce $\text{sig}_B(M_B)$, and verifies the signature.
6. Judson continues normally and sends r, $\text{AES}_c(X_B)$, and $\text{MAC}_{m2}(\text{AES}_c(X_B))$ to Alice through Bob.
7. Alice sends $\text{AES}_{c'}(X_A)$ and $\text{MAC}_{m2'}(\text{AES}_{c'}(X_A))$.
8. Judson now shares $s = g^{xy}$ with Alice and continues the OTR session, relaying ciphertexts through Bob.

The reverse attack, where Judson coerces Alice into establishing a session with Bob, is nearly identical.

A.1.2 Signal

In this example, Judson is the judge, Alice is the coerced participant, and Bob is the victim. Alice, Bob, and the server are the entities in the X3DH specification [75].

1. Alice contacts the server and receives Bob's prekey bundle: IK_B, SPK_B, $\text{Sig}(IK_B, \text{Encode}(SPK_B))$, and (without loss of generality) OPK_B.
2. Judson generates a key pair with public key EK_A.
3. Judson forces Alice to compute and reveal $DH1 = \text{DH}(IK_A, SPK_B)$. Optionally, Judson may ask for a zero-knowledge proof of correctness for $DH1$.
4. Judson sends EK_A to Bob through Alice.
5. Judson computes $DH2$, $DH3$, and $DH4$ using the secret for EK_A. Judson then computes $SK = \text{KDF}(DH1 \| DH2 \| DH3 \| DH4)$, which is also known by Bob. Judson now continues the Signal session normally, relaying ciphertexts through Alice.

Note that while this attack does not reveal IK_A to Judson, it does slightly weaken the security of future exchanges with IK_A and SPK_B; this is alleviated when SPK_B is replaced or one-time prekeys are used by Bob.

The reverse attack, where Judson coerces Bob into communicating with Alice, is more complex because Judson must generate the prekeys. In that case, the risks to Bob can be mitigated by using a distributed key generation scheme [54] to jointly generate the prekeys. When Alice selects a bundle, Bob sends the associated shares to Judson, who reconstructs the secrets. For bundles selected by non-victims, Judson sends the associated shares to Bob, who proceeds unmonitored.

A.2 Malicious Users

In this attack, a malicious participant interacts with a purpose-built third-party service during a conversation with a victim. Assuming that the public keys of the participant, victim, and service are known and validated, the participant and service did not collude to forge the transcript, and the service does not have direct access to the connection for the conversation, the participant is able to produce non-repudiable proof of message authorship by the victim. It is possible to do this while

protecting the participant's long-term keys, preventing the service from reading or modifying the conversation, preventing forgery by any individual party or outsider, allowing the participant to selectively disclose only a portion of the conversation, and preventing a forward secrecy breach when only the service is compromised.

The overall attack involves using the service to produce a trustworthy log of the protocol transcript (the exact network data transmitted to and from the participant), and enabling the participant to selectively publish ephemeral keys for ciphertexts to reveal portions of the conversation transcript (the plaintext messages exchanged in the session).

The basic approach is the same for both OTRv3 and Signal: the service maintains the keys for the session and otherwise behaves honestly, while the participant handles sending and receiving ciphertexts. Both the participant and service digitally sign protocol messages and, when necessary, they use secure multiparty computation (SMC) to allow the participant to conduct the conversation. Alternatively, a different trustworthy logging mechanism, like Intel SGX secure enclaves [5], can be used instead.

The protocol begins by performing the attack against the AKE from Section A.1 in such a way that the participant is given proof of the victim's identity and the service cannot select a specific ephemeral secret.
When the participant receives a message:

1. The participant signs the protocol message.
2. The participant and service perform SMC to verify the authenticity of the protocol message and decrypt the message for the participant (but not the service).
3. The service signs the participant's signature and as much of the protocol message as possible without being given enough data to decrypt it.

When the participant wants to send a message:

1. The participant and service use SMC to generate the encrypted and authenticated protocol message.
2. The participant signs the protocol message.
3. The service signs the participant's signature and as much of the protocol message as possible.

While the conversation is ongoing, before erasing ephemeral keys needed to decrypt messages, the service encrypts them under a public key generated by the participant at the start of the protocol, and stores them.
When the participant wants to incriminate the victim:

1. The participant asks the service to terminate.
2. The service sends the encrypted ephemeral keys it has stored, and any currently active keys, to the participant. The service then terminates the session.
3. The participant publishes the signatures, proving that the protocol transcript is real and unmodified.
4. The participant can selectively reveal messages by decrypting and publishing the old ephemeral keys needed to decrypt and authenticate target messages.

A.2.1 Protocol Specifics

When the service performs the attack from Section A.1 on either OTRv3 or Signal, the ephemeral key is produced using distributed key generation [54] and the participant's share is revealed to the service.

For OTRv3, the service reveals r, $m1$, c', and $m1'$ to the participant to verify the victim. For Signal, the participant uses the signed prekey to verify the victim.

When receiving a message in OTRv3, SMC is used to check the MAC; the victim keeps the ciphertext and received MAC secret, and the service keeps mk secret. To decrypt, the service verifiably reveals the AES-CTR keystream for the received ctr using SMC, and the participant uses it to privately decrypt the ciphertext. In Signal, the service simply reveals the message key. In both cases, the service only assists with decryption after signing the participant's signature on the protocol message. It is not necessary for the service to verify the participant's signature, since cheating will be detected during verification of the incriminating transcript.

When sending a message in OTRv3 or Signal, the participant generates a symmetric key. The parties then use SMC to encrypt and authenticate a message (known to the participant) with the message key (known to the service), with the resulting ciphertext being encrypted under the participant's symmetric key.

Because the service reveals message keys for Signal, it is not necessary for it to store encrypted versions for later release; this is only needed for expired Diffie-Hellman secrets in OTRv3.

B DREAD Security

In Section 4.2, we defined DREAD schemes and the security properties that we require from them. These properties can be easily derived from those given by Chow et al. [30] by incorporating the associated data parameter Φ where needed:

– **Soundness**[8]: ciphertexts decrypt to the same value (including $\perp$) even when the keys used by DREnc are not honestly generated, or the ciphertext is not pro-

8 Chow et al. [30] refer to the analogous property for DRE as *strong soundness*.

duced by DREnc at all. Concretely, any probabilistic polynomial time (PPT) adversary has negligible advantage in the following game:

1. The adversary produces a ciphertext γ, associated data Φ, and two public keys pk_1 and pk_2.
2. Let sk_1 and sk_2 be the unique secret keys associated with pk_1 and pk_2.
3. The adversary wins if $\text{DRDec}(pk_1, pk_2, sk_1, \Phi, \gamma) \neq \text{DRDec}(pk_1, pk_2, sk_2, \Phi, \gamma)$.

Note that this definition requires the cryptosystem used in the DREAD construction to be *admissible*, as defined in Section 4.2. ElGamal is admissible [30].

– **Dual-receiver IND-CCA2 security**: any PPT adversary has negligible advantage in the following game against a sound scheme[9]:

1. The challenger produces $(pk_i, sk_i) \leftarrow \text{DRGen}()$ for $i \in \{1, 2\}$ and sends (pk_1, pk_2) to the adversary.
2. The adversary is given decryption oracle access for $\text{DRDec}(pk_1, pk_2, sk_1, \cdot, \cdot)$. The adversary may perform a polynomially bounded number of DREnc calls, oracle requests, and other operations.
3. The adversary chooses two messages, m_1 and m_2, of equal length, and associated data Φ. The adversary sends (m_1, m_2, Φ) to the challenger.
4. The challenger chooses $b \xleftarrow{\$} \{1, 2\}$ and sends $\gamma \leftarrow \text{DREnc}(pk_1, pk_2, m_b, \Phi)$ to the adversary.
5. The adversary may perform a polynomially bounded number of calls to DREnc, oracle requests, and other operations.
6. The adversary outputs a guess bit b'.
7. The adversary wins if $b = b'$ and it did not query the oracle for $\text{DRDec}(pk_1, pk_2, sk_1, \Phi, \gamma)$ in step 5.

It is easy to see that the DREAD construction in Section 4.2 satisfies the soundness property. If the NIZKPK verifies, then except with negligible probability, both ElGamal ciphertexts must encode the same value $\mathcal{K}$ due to the soundness of the NIZKPK scheme. Consequently, both sk_1 and sk_2 must cause ADec to return the same value. Otherwise, decryption with either key returns $\perp$.

The proof of dual-receiver IND-CCA2 security for the DREAD construction in Section 4.2 can be derived from the proofs given by Faust et al. [46] and by Cramer and Shoup [34] as described in Section 4.2.

9 The soundness property simplifies the definition of the game by eliminating the need for two decryption oracles [30].

C Ring Signature Security

In Section 4.3 we defined a ROM-based ring signature scheme based on an SoK of one out of three discrete logarithms. We require this RSig/RVrf scheme to satisfy two security properties defined by Bender et al. [15]:

– **Anonymity against full key exposure** [15, Def. 4]: it is not possible to determine which secret key was used to produce the ring signature, even if all secret keys are revealed. Concretely, any PPT adversary has negligible advantage in the following game:

1. The challenger generates n key pairs (PK_i, SK_i) where n is a polynomial of the security parameter. Let PK be the set of all public keys, and SK be the set of all secret keys.
2. The adversary is given PK and access to an oracle $\mathcal{O}_{\text{RSig}}(\cdot, \cdot, \cdot)$ such that $\mathcal{O}_{\text{RSig}}(\bar{S}, \bar{m}, \bar{i})$ returns $\text{RSig}(PK_{\bar{i}}, SK_{\bar{i}}, \bar{S}, \bar{m})$ where we require $PK_{\bar{i}} \in \bar{S}$.
3. The adversary outputs a message m, distinct indices i and j, and a ring S for which $PK_i, PK_j \in S$. The adversary is given SK.
4. The challenger chooses $b \xleftarrow{\$} \{i, j\}$ and sends $\sigma \leftarrow \text{RSig}(PK_b, SK_b, S, m)$ to the adversary.
5. The adversary outputs b' and wins if $b = b'$.

– **Unforgeability with respect to insider corruption** [15, Def. 7]: it is not possible to produce an illegitimate ring signature, even with access to legitimate signatures that were produced using adversarially controlled keys in their rings. Concretely, any PPT adversary wins the following game with negligible probability:

1. The challenger generates n key pairs (PK_i, SK_i) where n is a polynomial of the security parameter. Let PK be the set of all public keys, SK be the set of all secret keys, and $C \leftarrow \varnothing$ be the set of corrupted users.
2. The adversary is given PK.
3. The adversary is given access to an oracle $\mathcal{O}_{\text{RSig}}(\cdot, \cdot, \cdot)$ such that $\mathcal{O}_{\text{RSig}}(\bar{S}, \bar{m}, \bar{i})$ returns $\text{RSig}(PK_{\bar{i}}, SK_{\bar{i}}, \bar{S}, \bar{m})$ where we require $PK_{\bar{i}} \in \bar{S}$.
4. The adversary is given access to a corruption oracle $\mathcal{O}_{\text{corr}}(\cdot)$ such that $\mathcal{O}_{\text{corr}}(\bar{i})$ returns $SK_{\bar{i}}$ and sets $C \leftarrow C \cup \{PK_{\bar{i}}\}$.
5. The adversary outputs (S, σ, m) and wins if $\text{RVrf}(S, \sigma, m) = \text{TRUE}$, the adversary never queried $\mathcal{O}_{\text{RSig}}(S, m, \cdot)$, and $S \subseteq PK \setminus C$.

Some previous work has been done on proving the security of ring signatures in the universal composability framework. Fischlin and Onete [51] presented ideal

functionalities for SoKs, and Yoneyama and Ohta [100] defined a ring signature functionality that is provably equivalent to Bender's security properties [15]. However, neither of these schemes is defined for recent frameworks with more accurate modeling of cross-session state (e.g., long-lived public keys). Defining ring signature functionalities for more realistic frameworks is a substantial task that is beyond the scope of this work. The remainder of this appendix presents security theorems for the RSig/RVrf scheme following the game-based definitions.

Theorem 1 (*Anonymity of RSig/RVrf*)

If the SoK produced by RSig is zero-knowledge, then the RSig/RVrf ring signature scheme provides anonymity against full key exposure.

The proof of Theorem 1 directly follows from the security assumption. Since the SoK is zero-knowledge, the adversary learns nothing from the signing oracle or from the private keys. The construction of the SoK in Section 4.3 is known to be zero-knowledge in the ROM [22].

In fact, the RSig/RVrf construction fulfills an even stronger property than we require: the signer remains anonymous even if the secret keys are revealed at the start of the game. As far as we know, this stronger property has no name in the ring signature literature.

Theorem 2 (*Unforgeability of RSig/RVrf*)

If the discrete logarithm problem is hard in $\mathbb{G}$, then the RSig/RVrf ring signature scheme provides unforgeability with respect to insider corruption.

It is known that SoKs produced by the Fiat-Shamir heuristic [48] are *weakly simulation-extractable* [46]: for any PPT adversary with access to a proof simulator that can produce a valid new proof (one not retrieved from the simulator), there is an efficient extractor that can retrieve a witness for that proof. Using this extractor, the reduction for Theorem 2 becomes simple: given an adversary that can forge ring signatures, include a blinded group element in the ring, then extract a witness from the forged proof to compute the discrete logarithm of that group element with non-negligible probability.

D DAKE Security Proof Techniques

In this appendix, we outline the techniques that we use to sketch the security proofs for DAKEZ, Spawn$^+$, ZDH, and XZDH in Appendices E, F, G, and H, respectively.

D.1 Related Models and Frameworks

Several techniques have been suggested for modeling and proving deniability properties of DAKEs. Di Raimondo et al. first formalized the notion of DAKEs [37]. Recently, Fischlin and Mazaheri [50] proposed weaker deniability notions that can characterize the properties of SIGMA, 3DH, and X3DH. Dodis et al. [42], hereafter referred to as DKSW09, pointed out that the notions of online and offline deniability have natural parallels in security proofs within the universal composability (UC) framework introduced by Canetti [23]. UC is a framework for instantiating security models and using them to prove the security of protocols. A protocol that is secure in a UC-based model is guaranteed to retain its security properties under arbitrary compositions, even when arbitrary protocols are run concurrently.

A UC security proof involves defining an *ideal functionality* that describes a protocol with self-evident security properties. This ideal functionality is executed by a trusted authority that protocol participants, and an adversary, interact with. The functionality defines the possible interactions between the parties and the authority; in the case of the adversary, messages sent to the authority model adversarial control over protocol execution, and messages sent to the adversary represent information disclosures. A UC security proof involves showing that any adversary attacking a particular "real" protocol (without a trusted authority) can be used to construct an attack against the ideal functionality, thereby showing that their security properties are equivalent. A protocol with this property is said to *UC-realize* the functionality. To show this property, proofs demonstrate that no external environment can distinguish between an adversary interacting with the real protocol and a simulator interacting with the ideal functionality. This environment can communicate with the adversary (or simulator), control the inputs of the protocol participants, and read their outputs, but it cannot directly view or interact with messages exchanged between the participants. In contrast, the adversary is given complete control over messages transmitted between participants. The adversary (or simulator) can also corrupt parties, which provides complete control over their future interactions, and reveals all memory state that has not been erased (if memory erasure is permitted by the model). The external environment is notified when parties are corrupted.

Internally, the computational model of UC is defined in terms of interactive Turing machines with "secure" and "insecure" tapes, and the ability to invoke

other machines as subroutines. Our proof sketches are not expressed in terms of these internal modeling details; we refer the interested reader to Canetti's definitions for the formalization [23]. We generally assume that the reader is familiar with the UC framework.

Canetti et al. defined the generalized universal composability (GUC) framework for the purpose of proving strong deniability properties [26]. The GUC framework is an extension of UC in which all machines (including the ideal functionality and the external environment) are granted access to *shared functionalities* that persist between protocol sessions. This provides a natural way to model features like a PKI. In previous work, we used the GUC framework to sketch security proofs for RSDAKE and Spawn [93].

D.2 The GUC Framework

Following DKSW09 [42] and UG15 [93], we set up our security proofs in the GUC framework. The GUC framework corrects a deficiency in the basic UC framework: in many protocols, some state information persists across sessions (e.g., long-term public keys). This information should be available to the external environment, because it could be used to distinguish between simulators and real protocols. This persistent state is captured by shared functionalities.

For maximum usefulness, we rely on traditional game-based security proofs for the primitives that we use within the GUC framework; this makes it easy to substitute primitives with other well-known constructions, while also providing strong composability guarantees for our DAKEs. Unlike UG15, we instantiate a GUC-based security model including random oracles.

Security proofs in the GUC framework follow four steps: define an ideal functionality with the desired security properties, define a "real" protocol, construct a simulator $\mathcal{S}$ attacking the ideal functionality based on an adversary $\mathcal{A}$ attacking the real protocol, and show that the two scenarios are externally indistinguishable given the security assumptions. The main participants in a protocol are called the *principal parties*. In the real setting, these parties use their inputs to exchange messages over a network controlled by $\mathcal{A}$, then generate outputs. In the ideal setting, these parties are *dummy parties* that simply forward inputs and outputs to and from the ideal functionality over a secure channel.

To simplify the process of producing GUC security proofs, Canetti et al. [26] introduced the external-subroutine universal composability (EUC) framework. EUC is equivalent to GUC, except that it constrains the scheme to a single shared functionality $\bar{\mathcal{G}}$ (an ideal functionality meeting this definition is called $\bar{\mathcal{G}}$-*subroutine respecting*). Moreover, in the EUC framework it is only necessary to consider a single session of the challenge protocol. Canetti et al. [26] proved that EUC security is equivalent to GUC security for $\bar{\mathcal{G}}$-subroutine respecting protocols [26, Th. 2.1], greatly reducing the complexity of proofs. Notably, this surprising result means that a $\bar{\mathcal{G}}$-subroutine respecting protocol that *EUC-realizes* an ideal functionality also *GUC-realizes* that ideal functionality when it is run concurrently with itself and with arbitrary external protocols. We use this result to sketch our proofs within the EUC framework (and therefore extend our results to the GUC framework).

D.3 Proof Notation and Setup

We adopt standard notational conventions for interactive Turing machines in the EUC/GUC framework. Ideal functionalities include $\mathcal{F}$ in their name. The external environment, denoted by $\mathcal{Z}$, attempts to distinguish between a simulator $\mathcal{S}$ attacking the ideal functionality and a real adversary $\mathcal{A}$ attacking a real protocol. We denote an idealized party interacting with an ideal functionality in the EUC/GUC framework as $\overline{P}$. We denote the corresponding party in a real protocol as $\mathcal{P}$. We denote the corresponding party simulated by $\mathcal{S}$ for $\mathcal{A}$ as $P^{(s)}$. As a notational convenience, we write $\overline{P}$ within the context of a EUC/GUC message to denote a label for the party. Similarly, we write $\mathcal{P}$ in a real protocol message to denote the same label for the party.

The GUC framework permits more general interactions between entities, but due to the aforementioned equivalence of the frameworks, we can focus solely on EUC interactions while effectively producing GUC proofs. In summary, in the EUC framework, $\mathcal{Z}$ is permitted to securely communicate with $\mathcal{A}$ (or $\mathcal{S}$), control the inputs of every principal party $\mathcal{P}$ (or $\overline{P}$), and read their outputs. $\mathcal{A}$ is given control over the interactions of every principal party $\mathcal{P}$ (in the real setting) or $P^{(s)}$ (when being simulated by $\mathcal{S}$), while $\mathcal{S}$ is permitted to interact with the ideal functionality in the prescribed manner. Both $\mathcal{A}$ and $\mathcal{S}$ are able to corrupt principal parties, and these corruptions are reported to $\mathcal{Z}$. The ideal functionality, $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{Z}$ are all permitted to interact with with the shared functionality in the prescribed manner.

In our security proof sketches, we define code for interactive Turing machines in event-based C-like pseudocode. We use the keyword `return` to mean that message processing immediately ceases; we combine `return`

with **if** to express publicly known constraints on message values. We adopt the common notion of "delayed messages" to mean that the ideal functionality gives $\mathcal{S}$ control over the timing and success of the message delivery by sending a message to $\mathcal{S}$, and delivering the original message only upon receipt of a delivery instruction message from $\mathcal{S}$.

The execution of a system of interactive Turing machines takes place sequentially (i.e., only one machine is active at any point in time) [23]. Our proof sketches follow the execution semantics defined by Canetti and Krawczyk for real protocols [25, Fig. 1] and ideal processes [25, Fig. 2], which unambiguously specify the activation order of machines. We specifically note that these semantics allow principal parties in real protocols to send one message and locally output a value in the same activation (after which $\mathcal{Z}$ becomes active), and they also allow ideal functionalities to send multiple messages in an activation (if the functionality sends a message to $\mathcal{S}$ then $\mathcal{S}$ becomes active, otherwise the previously activated party becomes active).

We write $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$ to denote the shared functionality depicted in Algorithm 1, which is used by all of our proof sketches. This shared functionality models two types of cross-session state: the registration and distribution of long-term public keys, and a collection of domain-separated random oracles. We accomplish this by merging together two shared functionalities that were previously defined in the literature: $\bar{\mathcal{G}}_{krk}^{\mathcal{F}}$ and $\bar{\mathcal{G}}_{ro}$. $\bar{\mathcal{G}}_{krk}^{\mathcal{F}}$ refers to the *key registration with knowledge* PKI shared functionality defined by DKSW09 [42, Fig. 2], which distributes public keys to all parties, but reveals the corresponding secret keys only to corrupted owners and the ideal functionality $\mathcal{F}$. $\bar{\mathcal{G}}_{ro}$ refers to the random oracle shared functionality defined by Walfish [95, Fig. 2.4]. $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$ essentially combines $\bar{\mathcal{G}}_{krk}^{\mathcal{F}}$ with n copies of $\bar{\mathcal{G}}_{ro}$ in order to explicitly model the notion of domain separation. It is necessary to combine these functionalities into one so that the EUC framework applies.

$\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$ is parameterized with a set of interactive Turing machines (ITMs) that are permitted to retrieve secret keys. Corrupt parties can always retrieve their own secret keys. In our proofs, we will permit secret keys to be retrieved by honest parties running the real key exchange protocol (so that they can perform the protocol without revealing their secret keys to $\mathcal{Z}$) and by the ideal functionality. This restricts honest parties to safe usage of their long-term secret keys. Borrowing a notational convenience from Walfish [95, §3.3], we always implicitly parameterize $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$ with the ideal functionality we are attempting to realize.

Algorithm 1 The shared functionality $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$

Parameterized by an implicit security parameter λ, a set of interactive Turing machines $\mathcal{F}$, a number of random oracles n, and a group $\mathbb{G}$ generated by g with prime order q.

on $(\texttt{register})$ **from** $\mathcal{P}$:
  if ($\mathcal{P}$ is corrupt) **return**
  if (there is a record $(\texttt{key}, \mathcal{P}, \cdot, \cdot)$) **return**
  $SK \xleftarrow{\$} \mathbb{Z}_q$
  $PK \leftarrow g^{SK}$
  Record $(\texttt{key}, \mathcal{P}, PK, SK)$

on $(\texttt{register}, SK)$ **from** $\mathcal{P}$:
  if ($\mathcal{P}$ is not corrupt) **return**
  if (there is a record $(\texttt{key}, \mathcal{P}, \cdot, \cdot)$) **return**
  $PK \leftarrow g^{SK}$
  Record $(\texttt{key}, \mathcal{P}, PK, SK)$

on $(\texttt{retrieve}, \mathcal{Q})$ **from** $\mathcal{P}$:
  if (there is a record $(\texttt{key}, \mathcal{Q}, PK, SK)$) {
    Send $(\texttt{pubkey}, \mathcal{Q}, PK)$ to $\mathcal{P}$
  } **else** {
    Send $(\texttt{pubkey}, \mathcal{Q}, \perp)$ to $\mathcal{P}$
  }

on $(\texttt{retrievesecret}, \mathcal{Q})$ **from** $\mathcal{P}$:
  if (($\mathcal{P}$ is honest) $\wedge$ ($\mathcal{P}$'s code is $\notin \mathcal{F}$)) **return**
  if (($\mathcal{P}$ is corrupt) $\wedge$ ($\mathcal{P} \neq \mathcal{Q}$)) **return**
  if (there is a record $(\texttt{key}, \mathcal{Q}, PK, SK)$) {
    Send $(\texttt{seckey}, \mathcal{Q}, PK, SK)$ to $\mathcal{P}$
  } **else** {
    Send $(\texttt{seckey}, \mathcal{Q}, \perp, \perp)$ to $\mathcal{P}$
  }

on $(\texttt{ro}, i, x)$ **from** $\mathcal{P}$:
  if ($i \notin [1, n]$) **return**
  if (there is a record $(\texttt{ro}, i, x, v)$) {
    Send $(\texttt{ro}, v)$ to $\mathcal{P}$
  } **else** {
    $v \xleftarrow{\$} \{0,1\}^\lambda$
    Record $(\texttt{ro}, i, x, v)$
    Send $(\texttt{ro}, v)$ to $\mathcal{P}$
  }

Note $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$ does not provide the ability to "reprogram" random oracle results or to extract random oracle queries. This differs from traditional random oracle models, but follows the GUC-based functionality defined by Walfish [95]. The main reason for omitting these features is that simulators in the UC (and EUC/GUC) framework must be straight-line simulatable, since $\mathcal{S}$ is

given only black-box access to $\mathcal{Z}$ and has no control over it [70]. This is an intuitive result, since $\mathcal{Z}$ effectively represents concurrent protocols, and these protocols have direct access to the same random oracle. In general, $\mathcal{Z}$ can perform random oracle queries on its own, and transfer the results to $\mathcal{A}$ or $\mathcal{S}$ to hide the queries and avoid the possibility of reprogramming. However, indistinguishability proofs have no such restrictions, and reductions may rewind $\mathcal{Z}$ or reprogram the random oracle, since these reductions can *internally* execute $\mathcal{Z}$ and $\mathcal{S}$. This technique can be used to employ the Fiat-Shamir heuristic in protocols within the GUC framework by performing the standard reduction [11] involving programming the random oracle [95, Th. 5.10]. We stress that all of our simulators in Appendices E, F, G, and H are straight-line simulatable and these simulators do not reprogram the random oracle. Our simulators do not require the extraction of witnesses from NIZKPKs or SoKs, and thus our protocols do not require straight-line extractable zero-knowledge proofs (e.g., Ω-protocols [53]).

While we do not include explicit reductions in our proof sketches, our indistinguishability proofs note when indistinguishability depends on the security assumptions of the primitives. In these cases, it is easy to construct reductions that attack the assumptions by internally executing $\mathcal{S}$ and a $\mathcal{Z}$ with distinguishing advantage, programming values as necessary to insert the reduction question, and then using the results of $\mathcal{Z}$ to complete the attack.

E Proof of DAKEZ Security

To prove the security of DAKEZ within the GUC framework, we must select an ideal functionality that represents a DAKE with our desired features and security properties. Unfortunately, the ideal functionalities defined by DKSW09 ($\mathcal{F}_{keia}^{\text{IncProc}}$) and UG15 ($\mathcal{F}_{post-keia}^{\text{IncProc}}$) do not capture all of our desired properties, so we must define a new functionality. We introduce and discuss this new functionality in Section E.1. In Section E.2, we formally define DAKEZ with an interface matching the ideal functionality. We then present our security theorem and an overview of our proof strategy in Section E.3. Section E.4 relates the protocol properties in Section 3.3 to the ideal functionality definition. Finally, we describe our actual proof sketch in Section E.5 and subsequent sections of this appendix.

E.1 Ideal Functionality for DAKEZ

E.1.1 Contributiveness

In their original analysis of universally composable key exchange protocols, Canetti and Krawczyk [25] (hereafter CK02) proposed an ideal functionality, $\mathcal{F}_{ke}$, that sends a randomly selected session key to the participants. CK02 noted that this functionality cannot be realized by protocols like two-flow Diffie-Hellman, since the simulator must "commit" to a shared secret without knowledge of the session key selected by the ideal functionality. Non-static adversarial corruptions can distinguish between real and ideal protocols by corrupting ephemeral state prior to the final flow, using it to compute the shared secret, and comparing the result to the key chosen by the ideal functionality. Hofheinz et al. [59] (hereafter HMS03) later showed that $\mathcal{F}_{ke}$ can in fact never be realized in the presence of adaptive adversaries. There are two simple ways to overcome this problem: modify the ideal functionality to allow the simulator to dictate the shared secret when a party is corrupted (the approach taken by DKSW09, HMS03, and UG15), or move part of the simulator into a protocol-specific *non-information oracle* (the approach taken by CK02).

A non-information oracle $\mathcal{N}$ is a probabilistic interactive Turing machine that interacts with another machine $\mathcal{M}$ and then produces local output. The "non-information" property requires that the local output of $\mathcal{N}$ is computationally indistinguishable from random from the perspective of $\mathcal{M}$, and independent of all messages exchanged between $\mathcal{N}$ and $\mathcal{M}$ [25]. This construct can be used as part of an ideal functionality to make notions of key exchange realizable in the UC framework. Specifically, a properly designed non-information oracle can provide information to the simulator $\mathcal{S}$, allowing it to provide the "commitments" necessary to simulate a real protocol, while using secret internal state to provide a session key to the ideal functionality. The computational indistinguishability of the local output (which becomes the session key) from random guarantees that $\mathcal{S}$ (and thus $\mathcal{A}$) cannot compromise the shared secret key. Since the shared secret should not be hidden from the protocol participants, these internal secrets are exposed to $\mathcal{S}$ if it corrupts one of the participating ideal parties. CK02 defined a relaxed ideal functionality, $\mathcal{F}_{wke}^{\mathcal{N}}$, parameterized by a non-information oracle $\mathcal{N}$, that is realized by a two-party Diffie-Hellman key exchange. The security achieved by realizing this ideal functionality is equivalent to the older notion of SK-security [23].

HMS03 later noted that non-information oracles can be used to capture the common notion of contributive-

ness [59]. This notion, which they refer to as *initiator resilience*, prevents the initiator of the key exchange from predetermining the value of the shared secret. As in a Diffie-Hellman key exchange, the responder can still completely determine the value of the secret by selecting their contribution appropriately.

E.1.2 Deniability

DKSW09 [42] noted that universal composability can be used to prove that a key exchange protocol is deniable. If a protocol realizes an appropriately chosen key exchange ideal functionality, then the resulting simulatability properties imply both offline and online deniability; the simulator $\mathcal{S}$ acts as the forger (in the offline case) or the misinformant (in the online case), and the distinguishing environment $\mathcal{Z}$ acts as the judge.

DKSW09 proposed the definition of a key exchange functionality with an "incriminating abort", which can be realized by efficient DAKEs that leak non-simulatable messages to active adversaries willing to cause session failures. This weakness is not a significant concern in practice, and it prevents the need to use unrealistically expensive cryptographic primitives. The simulator is permitted to ask the ideal functionality to abort the exchange, which prevents the delivery of the shared secret to at least one party. After aborting the protocol, the simulator can obtain some non-simulatable information that betrays involvement in the protocol by one of the parties. The model parameterizes the functionality with a protocol-specific "incriminating procedure" called IncProc that generates this incriminating information from one of the long-term secret keys. We adopted this approach in our security proof sketches for RSDAKE and Spawn [93].

E.1.3 Functionality Construction

Algorithm 2 depicts $\mathcal{F}^{+}_{post\text{-}keia}$, our new ideal functionality. This functionality incorporates the HMS03 model of contributiveness [59], the DKSW09 model of deniability with incriminating abort [42], and the UG15 model of post-specified peers [93]. Consequently, the functionality is parameterized by both a non-information oracle $\mathcal{N}$ and an incrimination procedure IncProc.

The core of $\mathcal{F}^{+}_{post\text{-}keia}$ is the computation and delivery of a shared secret key. $\mathcal{F}^{+}_{post\text{-}keia}$ expects two parties to declare participation in the protocol with `initiate` and `establish` messages.[10] These parties are thereafter

10 Walfish [95, Fig. 3.5], DKSW09 [42, Fig. 3], and UG15 [93, Alg. 1] all make a mistake in the registration messages by requiring the secret keys SK_I and SK_R as input. In UC/EUC/GUC,

Algorithm 2 Ideal functionality $\mathcal{F}^{+}_{post\text{-}keia}$

$\mathcal{F}^{+}_{post\text{-}keia}$ proceeds as follows, running on security parameter λ, in the $\bar{\mathcal{G}}^{\mathcal{F},n,\mathbb{G},q,g}_{krkro}$-hybrid model, with parties $\overline{P_1},\ldots,\overline{P_n}$ and an adversary $\mathcal{S}$. The functionality is parameterized by a non-information oracle $\mathcal{N}$, and an incrimination procedure IncProc. When initializing, $\mathcal{F}^{+}_{post\text{-}keia}$ invokes $\mathcal{N}$ with fresh randomness.

on interaction with $\mathcal{N}$:
  Allow $\mathcal{S}$ to communicate with $\mathcal{N}$ by forwarding messages between them. If at any point $\overline{I}$ or $\overline{R}$ is corrupted or $\overline{R}$ is "aborted" while $\mathcal{N}$ has produced local output, send the complete state and output of $\mathcal{N}$ to $\mathcal{S}$.

on $(\texttt{initiate}, sid, \overline{I}, \Phi, aux)$ **from** $\overline{P} \in \{\overline{P_1},\ldots,\overline{P_n}\}$:
  if ($\overline{I}$ is defined) **return**
  Denote $\overline{P}$ as $\overline{I}$
  Mark $\overline{I}$ as "active"
  Send $(\texttt{initiate}, sid, \overline{I}, \Phi)$ to $\mathcal{S}$

on $(\texttt{establish}, sid, \overline{R}, \Phi)$ **from** $\overline{P} \in \{\overline{P_1},\ldots,\overline{P_n}\}$:
  if (an `establish` message was previously received) **return**
  if (($\overline{I}$ is undefined) || ($\overline{I}$ is not "active")) {
    Resume processing once $\overline{I}$ is "active"
  }
  Denote $\overline{P}$ as $\overline{R}$
  Mark $\overline{R}$ as "active"
  Send $(\texttt{establish}, sid, \overline{R}, \Phi)$ to $\mathcal{S}$

on $(\texttt{ok}, sid, k)$ **from** $\mathcal{S}$:
  if (a key tuple (sid, κ) has been recorded) **return**
  if (($\overline{I}$ is undefined) || ($\overline{R}$ is undefined)) **return**
  if (($\overline{I}$ is corrupt) **&&** ($\overline{R}$ is corrupt)) **return**
  if (($\overline{I}$ is uncorrupted) **&&** ($\overline{R}$ is uncorrupted)
  ↪ **&&** ($\overline{R}$ is "active")) { $\kappa \xleftarrow{\$} \{0,1\}^{\lambda}$ }
  else if (($\overline{I}$ is corrupt) **&&** ($\overline{R}$ is "active")) {
  ↪ Let κ denote the local output of $\mathcal{N}$ }
  else { $\kappa \leftarrow k$ }
  Record key tuple (sid, κ)

on $(\texttt{deliver}, sid, t, p')$ **from** $\mathcal{S}$:
  if (no key tuple (sid, κ) has been recorded) **return**
  if (a `set-key` message was already sent to P_t) **return**
  if (IncProc was previously executed) **return**
  if (($t \notin \{\overline{I}, \overline{R}\}$) || ($t$ is not "active")) **return**
  Let $p \in \{\overline{I}, \overline{R}\}$ such that $p \neq t$
  if (($p \neq p'$) **&&** (p' is uncorrupted)) **return**
  Send $(\texttt{set-key}, sid, p', \kappa)$ to P_t
  if (two `set-key` messages have been sent) Halt

on $(\texttt{abort}, sid)$ **from** $\mathcal{S}$:
  if ($\overline{I}$ is "active") Send delayed $(\texttt{abort}, sid, \overline{I})$ to $\overline{I}$
  if ($\overline{R}$ is "active") {
    Mark $\overline{R}$ as "aborted"
    Send delayed $(\texttt{abort}, sid, \overline{R})$ to $\overline{R}$
  }

on $(\texttt{incriminate}, sid)$ **from** $\mathcal{S}$:
  if (IncProc was previously executed) **return**
  if (($\overline{R}$ is "aborted") **&&** ($\overline{I}$ is "active")
  ↪ **&&** ($\overline{R}$ is uncorrupted)) {
    Send $(\texttt{retrievesecret}, \overline{R})$ to $\bar{\mathcal{G}}^{\mathcal{F},n,\mathbb{G},q,g}_{krkro}$, retrieving SK_R
    Execute IncProc$(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$
  }

referred to as the initiator $\overline{I}$ and responder $\overline{R}$, respectively. Like DKSW09 [42, Fig. 3], we assume that all parties in the protocol have registered secret keys with the shared functionality before beginning a protocol session. $\mathcal{F}^{+}_{post\text{-}keia}$ ensures that the initiator is always defined first by delaying the processing of an `establish` message until a `initiate` message is seen. This simplifies simulator construction. Once both roles are defined, the simulator $\mathcal{S}$ is permitted to control delivery of the shared secret to $\overline{I}$ and $\overline{R}$. $\mathcal{S}$ sends a `ok` message to indicate that the shared key has become fixed. $\mathcal{S}$ can then individually choose to deliver the key to the ideal parties by sending `deliver` messages. We model post-specified peers by allowing $\mathcal{S}$ to specify the identity of the remote party in `deliver` messages. $\mathcal{S}$ may only specify "incorrect" remote identities if it has corrupted the corresponding parties. As in CK02 [24], the *aux* parameter in the `initiate` and `establish` messages contains auxiliary routing information. Real protocols use this information to deliver messages to the other party even though their logical identity is not known at the start of the protocol. The ideal functionality simply ignores *aux*. The protocol-specific details of the message routing (e.g., local broadcasts, message pools, or central servers) are independent of the security analysis [24]. When both parties receive the key, $\mathcal{F}^{+}_{post\text{-}keia}$ halts.

Contributiveness is provided by placing restrictions on $\mathcal{S}$'s ability to set the value of the session key. If $\mathcal{S}$ does not corrupt either party before fixing the session key with an `ok` message, $\mathcal{F}^{+}_{post\text{-}keia}$ selects a key completely at random. However, if $\mathcal{S}$ corrupts only $\overline{I}$, its influence over the key is still restricted. In this case, $\mathcal{F}^{+}_{post\text{-}keia}$ draws the value of the session key from $\mathcal{N}$. Although $\mathcal{S}$ is granted unrestricted interaction with $\mathcal{N}$, the non-information property of $\mathcal{N}$ prevents $\mathcal{S}$ from learning or controlling the key. Only if $\mathcal{S}$ corrupts $\overline{R}$ before sending an `ok` message is it given complete control over the session key. In all cases, corruption of either party provides knowledge of the session key; $\mathcal{F}^{+}_{post\text{-}keia}$ transmits the internal state and output of $\mathcal{N}$ to $\mathcal{S}$ upon corruption of either participant.

To permit realization of the functionality, $\mathcal{F}^{+}_{post\text{-}keia}$ also provides an incriminating abort procedure through IncProc. If $\mathcal{S}$ sends an `abort` message to $\mathcal{F}^{+}_{post\text{-}keia}$ before the functionality halts, the key exchange can no longer fully complete. $\mathcal{F}^{+}_{post\text{-}keia}$ models this by internally labeling $\overline{I}$ and $\overline{R}$ (when defined) as "active" or "aborted". The session key cannot be delivered to an "aborted" party using a `deliver` message. For $\mathcal{F}^{+}_{post\text{-}keia}$, only $\overline{R}$ is guaranteed to be aborted; it is still possible for $\overline{I}$ to output a result. When aborting the protocol, $\mathcal{S}$ can choose to deliver notifications of the abort to active parties independently. If both parties have been defined, $\overline{R}$ remains uncorrupted, and $\mathcal{S}$ has aborted the protocol, $\mathcal{S}$ can send an `incriminate` message to trigger invocation of `IncProc`, allowing it to receive protocol-specific incriminating messages.

One subtle interaction between the features is that we must allow $\mathcal{S}$ to derive the session key when the protocol is aborted. This case models the real-world situation in which the adversary has altered a message flow to incorporate adversarially controlled ephemeral state. While an authenticated key exchange must detect this alteration and prevent completion of the protocol (modeled in $\mathcal{F}^{+}_{post\text{-}keia}$ by the abort procedure), the attack may allow the adversary to derive the shared secret after it has become fixed, but before verification occurs.

E.2 DAKEZ in the GUC Framework

We originally defined DAKEZ in Section 5. However, to prove the security of DAKEZ in the GUC framework, it is necessary to define the DAKEZ program in terms of the $\mathcal{F}^{+}_{post\text{-}keia}$ interface. Specifically, we must define an interactive Turing machine that completes a DAKEZ key exchange in a way that is indistinguishable from dummy parties forwarding the same inputs to $\mathcal{F}^{+}_{post\text{-}keia}$. The details of the protocol are the same as in Figure 2, but we must change the interface. Algorithm 3 contains the adapted program. The program is implicitly parameterized with the group $\mathbb{G}$, q, and g.

After receiving its input, the DAKEZ program in Algorithm 3 determines whether it is playing the role of the initiator or the responder. We write "**on** (m) **to** $\mathcal{P}$" to mean that the given function is executed when a message of the form m is received, and the party is playing the role of $\mathcal{P}$. When evaluating the form of a message, group elements are checked to ensure that they are in $\mathbb{G}$ and are not the identity element. All variables shown in Algorithm 3 are scoped to their containing function unless they are explicitly persisted using the

ideal functionalities are executed with *dummy parties* that simply pass inputs received from $\mathcal{Z}$ to the functionality. Since one purpose of using a shared functionality to model the PKI is to hide honest parties' secret keys from $\mathcal{Z}$, this prevents $\mathcal{Z}$ from starting the protocols. The correct solution, as we noted in Section D.3, is to give the ideal functionality access to the secret keys for the purpose of invoking IncProc. Walfish and DKSW09 both do this, but superfluously and erroneously also require SK_I and SK_R as inputs from the principal parties.

Algorithm 3 Real protocol DAKEZ

on activation with input $(\texttt{initiate}, sid, \mathcal{I}, \Phi, aux)$:
  Record that we are the initiator, $\mathcal{I}$
  Retrieve PK_I and SK_I from shared functionality
  Record PK_I, SK_I, sid, and Φ
  Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}_I()$
  Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I$ using aux for routing

on activation with input $(\texttt{establish}, sid, \mathcal{R}, \Phi)$:
  Record that we are the responder, $\mathcal{R}$
  Retrieve PK_R and SK_R from shared functionality
  Record PK_R, SK_R, sid, and Φ
  Set state to $\texttt{await-}\psi_1$

on $(\mathcal{P} \| g^p \| PQ_P)$ **to** $\mathcal{R}$ **in state** $\texttt{await-}\psi_1$:
  Record $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_P)$
  Record $\mathcal{P}$, g^p, PQ_P, g^r, and Q_R
  Retrieve PK_P from shared functionality
  Let t = "0" $\| \mathcal{P} \| \mathcal{R} \| g^p \| g^r \| PQ_P \| Q_R \| \Phi$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_P, PK_R, g^p\}, t)$
  Record $k = \text{KDF}((g^p)^r \| Q_k)$
  Erase r and Q_k
  Send $\psi_2 = \mathcal{R} \| g^r \| Q_R \| \sigma$ to $\mathcal{P}$
  Set state to $\texttt{await-}\psi_2$

on $(\mathcal{P} \| g^p \| Q_P \| \sigma)$ **to** $\mathcal{I}$:
  Retrieve PK_P from shared functionality
  Let t_1 = "0" $\| \mathcal{I} \| \mathcal{P} \| g^i \| g^p \| PQ_I \| Q_P \| \Phi$
  if $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t_1)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Compute $Q_k = \text{QRKey}_I(SQ_I, Q_P)$
  Compute $k = \text{KDF}((g^p)^i \| Q_k)$
  Erase i, SQ_I, and Q_k
  Let t_2 = "1" $\| \mathcal{I} \| \mathcal{P} \| g^i \| g^p \| PQ_I \| Q_P \| \Phi$
  Send $\psi_3 = \text{RSig}(PK_I, SK_I, \{PK_I, PK_P, g^p\}, t_2)$ to $\mathcal{P}$
  Locally output $(\texttt{set-key}, sid, \mathcal{P}, k)$ and halt

on (σ) **to** $\mathcal{R}$ **in state** $\texttt{await-}\psi_2$:
  Let t = "1" $\| \mathcal{P} \| \mathcal{R} \| g^p \| g^r \| PQ_P \| Q_R \| \Phi$
  if $(\neg(\text{RVrf}(\{PK_P, PK_R, g^r\}, \sigma, t_1)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
  }
  Locally output $(\texttt{set-key}, sid, \mathcal{P}, k)$ and halt

on unknown or invalid message:
  Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
  Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

Algorithm 4 $\text{IncProc}_{\text{DAKEZ}}(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{"}I\text{"}, \text{"}R\text{"}, \Phi, g^i, PQ_I)$ **from** $\mathcal{S}$:
  Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
  Let t = "0" $\| \text{"}I\text{"} \| \text{"}R\text{"} \| g^i \| g^r \| PQ_I \| Q_R \| \Phi$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
  Compute $\psi = \text{"}R\text{"} \| g^r \| Q_R \| \sigma$
  Send $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi, g^r, r, Q_R, Q_k)$ to $\mathcal{S}$

"Record" statement (e.g., the variable t in one function is not the same as the t in another function).

We define the shared functionality for DAKEZ, $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$, to be $\bar{\mathcal{G}}^{\text{DAKEZ},3,\mathbb{G},q,g}_{krkro}$. All calls to the key derivation function $\text{KDF}(x)$ are modeled by the first random oracle; a message $(\texttt{ro}, 1, x)$ is sent to $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$, which then replies with the result. In a similar manner, the random oracles needed to model the hash functions within the RSig and DREAD schemes (thereby making their security proofs applicable) are provided by the second and third random oracles in $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$, respectively.

E.3 Proof Strategy

In this section, we describe a strategy for proving the security of DAKEZ by showing that it GUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$. We denote the three message flows of DAKEZ, as shown in Algorithm 3, as ψ_1, ψ_2, and ψ_3. Since $\mathcal{F}^{+}_{post\text{-}keia}$ is parameterized by additional procedures, we must define an incrimination procedure and non-information oracle for DAKEZ. Algorithm 4 depicts the DAKEZ incrimination procedure, which simply computes ψ_2 as an honest responder would. In practice, the presence of IncProc allows an adversary to prove that a party is willing to respond to a key exchange request from an entity with a particular claimed (but unauthenticated) identity. For most applications, this has no real-world impact on the security or privacy of the protocol.

Algorithm 5 depicts the non-information oracle for DAKEZ. The general construction of this oracle follows the approach of HMS03 [59] for Diffie-Hellman based protocols. $\mathcal{N}_{QRDH}$ internally generates both ephemeral keys and sends the public parts to $\mathcal{M}$. $\mathcal{M}$ can then accept or reject the proposed keys by sending a `complete` message. If the keys are accepted (by sending a `complete` message with ok = TRUE), $\mathcal{N}_{QRDH}$ completes the exchange and locally outputs the shared secret. If $\mathcal{M}$ rejects the keys, $\mathcal{N}_{QRDH}$ discards them and accepts one half of the ephemeral key exchange from $\mathcal{M}$ (in the α and β parameters). The local output of $\mathcal{N}_{QRDH}$ in this case is the shared secret of

Algorithm 5 A non-information oracle $\mathcal{N}_{QRDH}$

on $(\texttt{setup}, \mathbb{G}, q, g)$ **from** $\mathcal{M}$:
  if (a `setup` message was already received) **return**
  Read $(\mathbb{G}, q, g)$ as group $\mathbb{G}$, prime order q, generator g
  Generate $i \xleftarrow{\$} \mathbb{Z}_q$ and $r \xleftarrow{\$} \mathbb{Z}_q$
  Generate $(PQ_I, SQ_I) \leftarrow \text{QRGen}_I()$
  Generate $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
  Record $\mathbb{G}$, q, g, i, r, SQ_I, and Q_k
  Send $(\texttt{exchange}, g^i, g^r, PQ_I, Q_R)$ to $\mathcal{M}$

on $(\texttt{complete}, ok, \alpha, \beta)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (already output a key) **return**
  if (ok is TRUE) {
    Compute $k = \text{KDF}((g^i)^r \parallel Q_k)$
  } **else** {
    if $((\alpha \notin \mathbb{G}) \parallel (\alpha$ is identity element$))$ **return**
    if (β not generated by QRGen_I) **return**
    Generate new $r \xleftarrow{\$} \mathbb{Z}_q$
    Generate new $(Q_R, Q_k) \leftarrow \text{QRGen}_R(\beta)$
    Compute $k = \text{KDF}(\alpha^r \parallel Q_k)$
  }
  Locally output k

on $(\texttt{prove}, p, S, m)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (a `complete` message has been received) **return**
  if $(p \notin \{1, 2\})$ **return**
  if $(p = 1)$ { Let $x \leftarrow i$ } **else** { Let $x \leftarrow r$ }
  if $(g^x \notin S)$ **return**
  Compute $\sigma = \text{RSig}(g^x, x, S, m)$
  Send $(\texttt{proof}, \sigma)$ to $\mathcal{M}$

an exchange with $\mathcal{M}$. This option is necessary in the event that the adversary corrupts the initiator in the exchange; $\mathcal{N}$ grants the simulator the ability to complete the exchange on behalf of the remaining honest simulated party. $\mathcal{N}$ also provides a facility to generate RSig messages using either ephemeral key held by $\mathcal{N}_{QRDH}$.

While we do not formally prove it, $\mathcal{N}_{QRDH}$ is clearly a non-information oracle for appropriately chosen cryptographic groups. The only information revealed to $\mathcal{M}$ are public Diffie-Hellman contributions and public values produced by QRGen_I and QRGen_R. The only input from $\mathcal{M}$ is α and β when the initial exchange has been rejected. In this case, $\mathcal{M}$ never receives enough information to complete the exchange. The release of RSig proofs in response to `prove` messages releases no information other than the possession of a key in the set S by design. Consequently, distinguishing the output of $\mathcal{N}_{QRDH}$ from random would require $\mathcal{M}$ to break the computational Diffie-Hellman assumption in $\mathbb{G}$ (in order to determine the correct input to the random oracle) or the zero-knowledge property of the SoK in RSig.

We now present the security theorem for DAKEZ:

Theorem 3 (***Classical security of DAKEZ***)

If the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then DAKEZ GUC-realizes $\mathcal{F}^{+}_{post\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$-hybrid model with adaptive security for $\text{IncProc}_{\text{DAKEZ}}$ and non-information oracle $\mathcal{N}_{QRDH}$.

We use the erasure model defined by DKSW09 [42], which allows participants in the protocol to erase private state information, preventing it from being leaked in the event of a subsequent corruption. Since the capability to securely erase RAM contents is generally accepted in practice, this weakening of the model does not admit any actual attacks. The construction of the simulator for the proof of Theorem 3 can be used as guidance for practitioners seeking to implement real-world key exchange forgery tools (as a means to improve the plausible deniability of higher level protocols). In general, the simulator simulates both parties involved in the key exchange honestly. To produce the RSig proofs, the simulator uses the ephemeral keys of the remote party in the case when the adversary has not corrupted either participant, or the compromised long-term keys if a participant has been corrupted. The simulator uses the non-information oracle to simulate the calculation of the shared secret key, and uses its access to the non-information oracle's internal state to construct the simulated memory contents of corrupted parties. The simulator only uses IncProc if the first message, ψ_1, is altered by the adversary.

E.4 Relationship to Security Properties

Now that we have defined the ideal functionality that represents the features of DAKEZ and stated the associated security theorem, we can more rigorously define the security properties in Section 3.3 in terms of the GUC framework and discuss how they are captured by our proof sketch. Each property either follows from the definition of $\mathcal{F}^{+}_{post\text{-}keia}$, or from the proof of Theorem 3:

1. **Universally composable AKE**: $\mathcal{F}^{+}_{post\text{-}keia}$ provides mutual authentication, key secrecy, and key freshness. Mutual authentication ensures that $\mathcal{S}$ (and thus $\mathcal{A}$) cannot cause $\overline{I}$ (resp. $\overline{R}$) to output an uncorrupted partner identifier other than $\overline{R}$ (resp.

$\overline{I}$). Key secrecy and freshness ensure that if $\overline{I}$ or $\overline{R}$ is uncorrupted and outputs a key κ and a partner identifier $\overline{P}$, and $\overline{P}$ is uncorrupted, then $\mathcal{Z}$ cannot distinguish κ from $\kappa' \xleftarrow{\$} \{0,1\}^\lambda$.

2. **Offline deniability**: It is possible to construct a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between $\mathcal{S}$ interacting with $\mathcal{F}^+_{post\text{-}keia}$ and $\mathcal{A}$ interacting with DAKEZ after the following sequence of events: $\mathcal{Z}$ selects a party $\overline{P} \in \{\overline{I}, \overline{R}\}$, allows the key exchange to complete unmodified, corrupts both $\overline{I}$ and $\overline{R}$, and asks $\mathcal{S}$ to reveal $\overline{P}$'s ephemeral keys. Such a simulator can be constructed from the proof of Theorem 3 by simulating the "honest" case and then exposing the contents of the non-information oracle to $\mathcal{Z}$.
3. **Online deniability**: $\mathcal{Z}$ cannot distinguish between $\mathcal{S}$ interacting with $\mathcal{F}^+_{post\text{-}keia}$ and $\mathcal{A}$ interacting with DAKEZ, given the restrictions in Theorem 3.
4. **Contributiveness / Initiator-resilience**: When $\overline{R}$ is uncorrupted, any key output by $\overline{R}$ is computationally independent of any values chosen by $\mathcal{Z}$. This property follows from the design of $\mathcal{F}^+_{post\text{-}keia}$, which either outputs a bit string chosen uniformly at random or the local output of the non-information oracle when $\overline{R}$ is uncorrupted.
5. **Forward secrecy**: In the strong form [14]: if a party $\overline{P}$ outputs a key κ and a partner identifier $\overline{P'}$, then $\mathcal{Z}$ can never distinguish κ from $\kappa' \xleftarrow{\$} \{0,1\}^\lambda$ unless $\overline{P}$ or $\overline{P'}$ was corrupted before the corresponding session completed. In the weak form of the property, this is only true if $\mathcal{S}$ also did not abort the session (and thus $\mathcal{A}$ did not modify any messages).
6. **Post-specified peer**: The `initiate` and `establish` messages do not identify the intended communication partner.

E.4.1 Quantum Transitional Security

Note that Theorem 3 only refers to classical security (i.e., it ignores quantum adversaries). Transitionally secure key exchanges cannot realize authenticated key exchange functionalities like $\mathcal{F}^+_{post\text{-}keia}$ against quantum adversaries because, by definition, they fail to provide the necessary authentication properties. However, a more traditional analysis makes it clear that DAKEZ is transitionally secure. Because the KDF is modeled by a random oracle, any passive adversary that can derive the session key using only long-term secret keys and a transcript of the exchange must be able to derive the KDF input and send it to $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$ within an `ro` message. This KDF input includes Q_k, the shared secret derived from the QRGen_R and QRKey_I functions. Since the adversary does not have access to ephemeral state ($\mathcal{I}$ and $\mathcal{R}$ already erased SQ_I and Q_k from their memory), the adversary must be able to derive Q_k using only PQ_I and Q_R (and the unrelated secrets I and R). Therefore, this adversary can break the key secrecy property of the quantum-resistant KEM, which we assume is not possible. A similar argument holds for the other DAKEs presented in this work.

E.5 Proof of Theorem 3

(Sketch) To show that DAKEZ GUC-realizes $\mathcal{F}^+_{post\text{-}keia}$, it suffices to show that DAKEZ EUC-realizes $\mathcal{F}^+_{post\text{-}keia}$. DAKEZ EUC-realizes $\mathcal{F}^+_{post\text{-}keia}$ if and only if, for any PPT adversary $\mathcal{A}$ attacking DAKEZ, there exists a PPT adversary $\mathcal{S}$ attacking $\mathcal{F}^+_{post\text{-}keia}$ such that any $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$-externally constrained environment $\mathcal{Z}$ cannot distinguish between the real and simulated conditions.

Like most proofs in UC-based models, we will construct a simulator $\mathcal{S}$ that executes $\mathcal{A}$ internally, simulating the real protocol flows that $\mathcal{A}$ expects based on conditions in the ideal environment. For any ideal party $\overline{P}$, $\mathcal{S}$ simulates a party $P^{(s)}$ for $\mathcal{A}$. All parties know the shared protocol parameters used to instantiate $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$: a group $\mathbb{G}$ of prime order q with generator g. To achieve the required indistinguishability property, we need to show two things: $\mathcal{Z}$ can derive no useful information from sessions other than the one under consideration, and $\mathcal{Z}$ cannot distinguish between the challenge protocols in the context of the current session. To guarantee the latter condition, we must show that, irrespective of the actions performed by $\mathcal{A}$ under the instruction of $\mathcal{Z}$, the outputs of the main parties of $\mathcal{F}^+_{post\text{-}keia}$ are equal to those of DAKEZ, corrupted parties provide memory consistent with all other observations, and the protocol flows within the joint view of $\mathcal{A}$ and $\mathcal{Z}$ are consistent with the outputs of the main parties.

Section E.6 describes the simulator construction, and Section E.7 presents the indistinguishability proof.

E.6 Simulator Construction

E.6.1 Communications between $\mathcal{A}$ and $\mathcal{Z}$

Any data sent to $\mathcal{S}$ from $\mathcal{Z}$ are copied to the input of $\mathcal{A}$. Likewise, any output from $\mathcal{A}$ is sent to $\mathcal{Z}$ by $\mathcal{S}$.

E.6.2 General reactions to actions by $\mathcal{A}$

If $\mathcal{A}$ sends any messages within the simulated environment that are unrelated to DAKEZ, they are ignored (as they would be in a real network environment). If

$\mathcal{A}$ delays delivery of a message flow, $\mathcal{S}$ simply waits for the flow to be delivered before continuing. This leaves $\mathcal{A}$ with few possible actions of consequence: it can alter any of the message flows it perceives (this is equivalent to delaying a message and sending a different one in its place), and it can corrupt simulated parties. Our model allows $\mathcal{A}$ to corrupt parties before the protocol begins, after ψ_1 has been sent, after ψ_2 has been sent, or after ψ_3 has been sent (i.e., we tolerate fully adaptive corruptions). When $\mathcal{A}$ corrupts a simulated party, $\mathcal{S}$ corrupts the corresponding ideal party in order to construct the expected state history. If $\mathcal{A}$ causes a corrupted simulated party to output a message, $\mathcal{S}$ causes the corresponding ideal party to output the same message.

E.6.3 Initialization

When $\mathcal{S}$ first initializes, it sends a $(\texttt{setup}, \mathbb{G}, q, g)$ message to $\mathcal{N}$ through $\mathcal{F}^{+}_{post\text{-}keia}$, and waits to receive a $(\texttt{exchange}, g^i, g^r, PQ_I, Q_R)$ message in response. The group details sent by $\mathcal{S}$ correspond to the protocol instantiation attacked by $\mathcal{A}$. $\mathcal{S}$ makes a note of the ephemeral keys in the exchange message for later use in the simulation.

E.6.4 Receipt of initiate message from $\mathcal{F}^{+}_{post\text{-}keia}$

When $\mathcal{S}$ receives $(\texttt{initiate}, sid, \overline{I}, \Phi_I)$ from $\mathcal{F}^{+}_{post\text{-}keia}$, it honestly constructs a ψ_1 message from $I^{(s)}$ with the help of the non-information oracle $\mathcal{N}$. $\mathcal{S}$ computes ψ_1 = "I" $\|$ g^i $\|$ PQ_I using the g^i and PQ_I values previously received from $\mathcal{N}$ and sends ψ_1 through $\mathcal{A}$ as if it were broadcast by $I^{(s)}$. $\mathcal{S}$ also records the value Φ_I for later reference.

E.6.5 Receipt of establish message from $\mathcal{F}^{+}_{post\text{-}keia}$

When $\mathcal{S}$ receives an establishment message $(\texttt{establish}, sid, \overline{R}, \Phi_R)$ from $\mathcal{F}^{+}_{post\text{-}keia}$, it checks to see the circumstances of the simulated ψ_1 message transmission. Since $\mathcal{F}^{+}_{post\text{-}keia}$ only sends an establish message after it has already sent an initiate message, ψ_1 is guaranteed to have been sent and received in the simulated environment (either by $\mathcal{S}$ in response to an initiate message or by $\mathcal{A}$ from a corrupted party).

$\mathcal{S}$ parses ψ_1 to recover g^i and PQ_I. If ψ_1 is not of the correct format, or it fails to validate (e.g., if "I" is not a valid identity), then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{post\text{-}keia}$ and delivers the resulting abort message to $\overline{R}$ immediately. $\mathcal{S}$ withholds the abort message to $\overline{I}$.

$\mathcal{S}$ records Φ_R for later reference.

If ψ_1 is valid, then $\mathcal{S}$ then constructs a message ψ_2 from $R^{(s)}$ in response to ψ_1. The mechanism for constructing ψ_2 depends on how ψ_1 was generated:

- If $\mathcal{S}$ previously created ψ_1 in response to an initiate message, then $\mathcal{S}$ uses the non-information oracle $\mathcal{N}$ to construct ψ_2. $\mathcal{S}$ requests a forged proof under $I^{(s)}$'s ephemeral keys by sending $(\texttt{prove}, 1, \{g^I, g^R, g^i\}, t)$, where the tag t is given by t = "0"$\|$"I"$\|$"R"$\|g^i\|g^r\|PQ_I\|Q_R\|\Phi_R$, to $\mathcal{N}$ and waiting for a message $(\texttt{proof}, \sigma_R)$ in response. $\mathcal{S}$ then constructs ψ_2 = "R" $\|$ g^r $\|$ Q_R $\|$ σ_R.
- If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ uses its access to corrupt $\overline{I}$ to retrieve $SK_I = I$ from $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$ using a rectrievesecret message. $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I)$. Since $\overline{I}$ is corrupted, $\mathcal{F}^{+}_{post\text{-}keia}$ immediately sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ sends $(\texttt{ok}, sid, 0)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing it to record the output from $\mathcal{N}$ as the shared key. $\mathcal{S}$ uses the newly generated values g^r and Q_R to construct ψ_2 = "R" $\|$ g^r $\|$ Q_R $\|$ σ_R. $\mathcal{S}$ calculates σ_R as in the previous case, except that it uses the long-term keypair (g^I, I) to produce the proof.
- If $\mathcal{S}$ previously created a message ψ_1' but $\psi_1 \neq \psi_1'$, then $\mathcal{A}$ has altered the message in transit. $\mathcal{S}$ constructs ψ_2 through the use of IncProc. $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, but withholds delivery of the resulting abort messages to $\overline{I}$ and $\overline{R}$. It then sends $(\texttt{incriminate}, sid)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing an instance of IncProc to be invoked. Using the values parsed from ψ_1, $\mathcal{S}$ sends $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{"}I\text{"}, \text{"}R\text{"}, \Phi_R, g^i, PQ_I)$ to IncProc and receives $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi_2, g^r, r, Q_R, Q_k)$ in response.

$\mathcal{S}$ then sends ψ_2 through $\mathcal{A}$ as if $R^{(s)}$ sent it to $I^{(s)}$.

E.6.6 Receipt of ψ_2 by uncorrupted $I^{(s)}$

When uncorrupted $I^{(s)}$ receives message ψ_2 claiming to be from $P^{(s)}$, $\mathcal{S}$ checks to see if $I^{(s)}$ has previously broadcast a message ψ_1. If not, then the message ψ_2 is ignored. $\mathcal{S}$ then parses ψ_2 to extract "P", g^p, Q_P, and the proof σ_P. If ψ_2 is not of the correct form, or if σ_P is not a correct proof matching Φ_I and the ψ_1 sent by $I^{(s)}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{post\text{-}keia}$ and delivers the resulting abort message to $\overline{I}$ immediately, while withholding any abort message sent to $\overline{P}$.

If $I^{(s)}$ has previously broadcast a message ψ_1 and ψ_2 is valid, then $\mathcal{S}$ constructs message ψ_3 to send from $I^{(s)}$ to $P^{(s)}$ and outputs a shared key. The private key that

$\mathcal{S}$ uses to construct the proof ψ_3, and the shared key that it outputs, depends on the state of the simulation:

– If $\mathcal{A}$ has previously corrupted $P^{(s)}$, then $\mathcal{S}$ must have previously corrupted $\overline{P}$ (since $\mathcal{S}$ corrupts ideal parties corresponding to simulated parties corrupted by $\mathcal{A}$). In this case, $\mathcal{S}$ retrieves $SK_P = P$ from $\bar{\mathcal{G}}^{\text{DAKEZ}}_{krkro}$ using a `retrievesecret` message and uses this key to compute $\psi_3 = \text{RSig}(g^P, P, \{g^I, g^P, g^p\}, t)$, where the tag t is given by $t = \text{“1”}\|\text{“}I\text{”}\|\text{“}P\text{”}\|g^i\|g^p\|PQ_I\|Q_P\|\Phi_I$.
$\overline{I}$ is expected to output a session key corresponding to the one negotiated between $I^{(s)}$ and $P^{(s)}$. If $\mathcal{S}$ previously simulated a ψ_2 message from a party $R^{(s)}$ and $R^{(s)} \neq P^{(s)}$, then $\mathcal{S}$ issues a $(\texttt{abort}, sid)$ message to $\mathcal{F}^{+}_{post\text{-}keia}$ and withholds the resulting `abort` messages. In any case, $\mathcal{F}^{+}_{post\text{-}keia}$ receives the internal state of $\mathcal{N}$ from $\mathcal{S}$, allowing it to acquire the ephemeral keys i and SQ_I used to generate ψ_1 (this occurs because either $\overline{R}$ has been aborted, or $\overline{R}$ is corrupt). $\mathcal{S}$ computes $k = \text{KDF}((g^p)^i \parallel \text{QRKey}_I(SQ_I, Q_P))$ and sends $(\texttt{ok}, sid, k)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing it to record key tuple (sid, k).

– Otherwise, $P^{(s)}$ is uncorrupted. Because RSig is unforgeable with respect to insider corruption, the only way for σ_P to be valid in this situation is if the sender of the message knows the long-term private key of one of the parties, or the ephemeral key of $I^{(s)}$. Additionally, σ_P must have been computed during this session because its validity depends on ψ_1. Due to the hardness of the discrete log problem in the group (guaranteeing the secrecy of the keys) and the uniqueness of ψ_1, this is only possible if $P^{(s)} = R^{(s)}$ and ψ_2 was previously generated by $\mathcal{S}$. $\mathcal{S}$ forges the proof ψ_3 using $R^{(s)}$'s ephemeral key held by the non-information oracle $\mathcal{N}$. $\mathcal{S}$ sends $(\texttt{prove}, 2, \{g^I, g^R, g^r\}, t)$, where the tag t is given by $t = \text{“1”}\|\text{“}I\text{”}\|\text{“}R\text{”}\|g^i\|g^r\|PQ_I\|Q_R\|\Phi_I$, to $\mathcal{N}$ through $\mathcal{F}^{+}_{post\text{-}keia}$ and receives $(\texttt{proof}, \psi_3)$ in response. $\mathcal{S}$ sends $(\texttt{ok}, sid, 0)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing it to record a random session key.

$\mathcal{S}$ sends $(\texttt{deliver}, sid, \overline{I}, \overline{P})$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing $\overline{I}$ to emit the proper shared secret.

E.6.7 Receipt of ψ_3 by uncorrupted $R^{(s)}$

When uncorrupted $R^{(s)}$ receives message ψ_3 from $I^{(s)}$, $\mathcal{S}$ first checks to ensure that $R^{(s)}$ has previously received a message ψ_1 from $I^{(s)}$ and that it sent a response ψ_2. If either of these conditions do not hold, then the message is ignored. $\mathcal{S}$ then verifies the proof in ψ_3.

If the proof is invalid, does not match Φ_R or the ψ_2 message previously sent by $R^{(s)}$, or fails to verify, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{post\text{-}keia}$ and delivers the resulting `abort` message to $\overline{R}$ immediately. $\mathcal{S}$ withholds the `abort` message sent to $\overline{I}$.

If the proof is valid, then $\mathcal{S}$ also causes $\overline{R}$ to output a key. Since RSig is unforgeable with respect to insider corruption, it is only possible to reach this state if $I^{(s)}$ is corrupt or if the exchange has completed honestly; in all situations, $\mathcal{S}$ has already caused $\mathcal{F}^{+}_{post\text{-}keia}$ to record a shared key. $\mathcal{S}$ sends $(\texttt{deliver}, sid, \overline{R}, \overline{I})$ to $\mathcal{F}^{+}_{post\text{-}keia}$, causing $\overline{R}$ to emit the proper shared secret.

E.6.8 Transmission of ψ_1 by corrupted $I^{(s)}$

When $\mathcal{S}$ has not yet received an `initiate` message from $\mathcal{F}^{+}_{post\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $I^{(s)}$ to issue message ψ_1, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\overline{I}$ to send $(\texttt{initiate}, sid, \overline{I}, \perp, \perp)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, but ignores the resulting `initiate` message sent by $\mathcal{F}^{+}_{post\text{-}keia}$.

E.6.9 Transmission of ψ_2 by corrupted $R^{(s)}$

When $\mathcal{S}$ has not yet received an `establish` message from $\mathcal{F}^{+}_{post\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $R^{(s)}$ to issue message ψ_2, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\overline{R}$ to send $(\texttt{establish}, sid, \overline{R}, \perp)$ to $\mathcal{F}^{+}_{post\text{-}keia}$, but ignores the resulting `establish` message sent by $\mathcal{F}^{+}_{post\text{-}keia}$.

E.6.10 Constructing state for corrupted parties

When $\mathcal{A}$ corrupts a party in the simulated environment, $\mathcal{S}$ corrupts the corresponding party in the ideal environment. If $\mathcal{A}$ causes corrupted parties to output values, $\mathcal{S}$ outputs these values from the corresponding ideal parties. In addition, $\mathcal{S}$ must provide $\mathcal{A}$ with a simulated historical state for corrupted parties.

If $\mathcal{A}$ corrupts the party known as $I^{(s)}$ after an `initiate` message has been received, but before $I^{(s)}$ has received ψ_2, then $\mathcal{S}$ uses its access to $\mathcal{N}$ to provide the random coins i, and SQ_I used to construct ψ_1. If $I^{(s)}$ already received ψ_2, then $\mathcal{S}$ uses its corruption of $\overline{I}$ to provide the session key k that $\overline{I}$ already output.

If $\mathcal{A}$ corrupts the party known as $R^{(s)}$ after it has already received ψ_1, then it must provide the session key k expected to be stored in $R^{(s)}$'s memory. If $R^{(s)}$ has already received ψ_3, then $\overline{R}$ has already output session key k, and so $\mathcal{S}$ can directly provide this value. If $R^{(s)}$ has not yet received ψ_3, then $\mathcal{S}$ uses its access to $\mathcal{N}$ to obtain k (in the event that ψ_1 was sent by a corrupt $I^{(s)}$), or the random coins r and Q_k that, together with ψ_1, can be used to compute k.

E.7 Proof of Indistinguishability

We now turn to the task of proving that $\mathcal{S}$ acting on $\mathcal{F}^{+}_{post\text{-}keia}$ is indistinguishable from $\mathcal{A}$ acting on DAKEZ. To do this, we divide all possible behaviors of $\mathcal{A}$ into several cases. For each case, we show that the protocol flows generated by $\mathcal{S}$ are indistinguishable from those generated by DAKEZ, outputs from $\mathcal{F}^{+}_{post\text{-}keia}$ are indistinguishable from those from DAKEZ, and that the simulated memory states of corrupted parties are indistinguishable from those of real parties.

E.7.1 The honest case

This case occurs when $\mathcal{A}$ does not alter any messages or corrupt $I^{(s)}$ or $R^{(s)}$ until after the session concludes.

All three messages are generated by the combination of $\mathcal{S}$ and $\mathcal{N}$ honestly (i.e., exactly how they would be generated by the parties in a real DAKEZ session), with the exception of the proofs. The proofs are not signed by the long-term secret keys of the parties, as in a real interaction. Instead, they are produced by $\mathcal{N}$ using the ephemeral key of the opposite party. However, because RSig is anonymous against full key exposure, $\mathcal{Z}$ cannot distinguish the proofs produced by $\mathcal{S}$ from those produced in a real interaction, even when it corrupts the long-term keys of $I^{(s)}$ and $R^{(s)}$ after the session.

If the shared session state provided to the parties by $\mathcal{Z}$ differs (i.e., $\Phi_I \neq \Phi_R$), then $I^{(s)}$ will immediately abort when ψ_2 is delivered. This output is the same as in the real protocol because the SoK in ψ_2 will be bound to the wrong message.

When $\Phi_I = \Phi_R$, the output from $\overline{I}$ and $\overline{R}$ in the ideal environment includes the correct identity of the conversation partner, as well as a shared secret k randomly generated by $\mathcal{F}^{+}_{post\text{-}keia}$. These are the expected party identities from the real interaction, so the only possible way for $\mathcal{Z}$ to distinguish between real and simulated outputs is by examining k. Since i and r are erased by real parties before they return output, $\mathcal{A}$ cannot access these values, even when corrupting $I^{(s)}$ and $R^{(s)}$ after the protocol concludes. Therefore, any ability to distinguish between challenge protocols based on the choice of k would mean that $\mathcal{Z}$ could distinguish between k and $\mathrm{KDF}(g^{ir})$. Since KDF is modeled by the random oracle in $\bar{\mathcal{G}}^{\mathrm{DAKEZ}}_{krkro}$, this is only possible if $\mathcal{Z}$ can break the CDH assumption in $\mathbb{G}$ and send the message $(\mathtt{ro}, 1, g^{ir})$ to $\bar{\mathcal{G}}^{\mathrm{DAKEZ}}_{krkro}$, which we assume is not possible.

E.7.2 Alteration of ψ_1

This situation occurs when ψ_1 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ are corrupted when ψ_1 is delivered.

When ψ_1 is altered, $\mathcal{S}$ generates ψ_2 from $R^{(s)}$ using IncProc. The definition of IncProc involves honestly generating ψ_2 using the long-term secret key of $\overline{R}$, so this flow is indistinguishable from a real message. Likewise, the memory state of $R^{(s)}$ is indistinguishable from the real situation because IncProc provides $\mathcal{S}$ with the random coins used to generate the ephemeral keys in ψ_2 and the session key k.

$\mathcal{S}$ causes the protocol to abort, but does not deliver abort messages to either party. If $\mathcal{A}$ allows ψ_2 to be delivered to $I^{(s)}$, then $I^{(s)}$ will abort. This matches the output of real interactions because $I^{(s)}$ expects ψ_2 to include a proof incorporating the true g^i and PQ_I values sent by $I^{(s)}$ in ψ_1. The only way for the simulated and real situations to differ is if $\mathcal{A}$ somehow alters ψ_2 so that it is a valid response. Since RSig is unforgeable with respect to insider corruption, this is not possible.

E.7.3 Alteration of ψ_2

This situation occurs when ψ_2 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ are corrupted when ψ_2 is delivered. $\mathcal{S}$ causes $I^{(s)}$ to immediately abort when it receives an altered ψ_2. As mentioned previously, $I^{(s)}$ will always abort because the proof in the altered ψ_2 message cannot be correct due to RSig being unforgeable with respect to insider corruption.

E.7.4 Indistinguishability under corruptions

This situation occurs when either party is corrupted at a time before the times covered by the previous cases.

The only difference between the normal operation of $\mathcal{S}$ and this case is the secret key used to compute the proofs in messages generated by $\mathcal{S}$, and the mechanism for generating the shared secret keys. Whereas $\mathcal{S}$ normally uses the ephemeral signing keys i and r stored in $\mathcal{N}$ to produce the proofs in the messages, the keys generated by $\mathcal{N}$ might not be used when a party is corrupted before sending its first message. In these cases, $\mathcal{S}$ instead makes use of the long-term secret key of the corrupted party to produce the proofs. Again, these message flows are indistinguishable from real flows due to the anonymity against full key exposure property of RSig. To output the correct session keys, $\mathcal{S}$ extracts the internal state of $\mathcal{N}$ to complete key exchanges with corrupted parties. In all cases, the outputs of the proto-

col are indistinguishable because the uncorrupted party effectively completes the key exchange honestly.

If both simulated parties are corrupted, then indistinguishability is trivial. $\mathcal{S}$ never generates any messages, and so they cannot be used by $\mathcal{Z}$ to detect simulation. The outputs of corrupted parties are copied to the outputs of the corresponding ideal parties, so this is also not useful to $\mathcal{Z}$.

In all cases of corruption, $\mathcal{S}$ provides the expected memory state for the corrupted party—the set of random coins used to generate ephemeral signing keys, and possibly some shared secret keys (depending on which party is corrupted and when). In all cases, these values are indistinguishable from real values because the parties are effectively simulated honestly.

E.7.5 Data from other sessions

Since we are considering the security of DAKEZ in the EUC model, we must also consider the usefulness of information collected by $\mathcal{Z}$ from other protocol sessions. No information from other sessions can be used to assist $\mathcal{A}$ with the generation of false message flows: ψ_1 is generated using no long-term information, and both ψ_2 and ψ_3 require computation of an SoK bound to the contents of ψ_1. Since RSig is unforgeable with respect to insider corruption, collecting SoKs from other sessions does not allow $\mathcal{A}$ to produce proofs correctly bound to the session under attack. □

F Proof of Spawn⁺ Security

In this appendix, we sketch a security proof for Spawn⁺. As in Appendix E, we define an ideal functionality in Section F.1, define the real protocol in Section F.2, present our security theorem in Section F.3, and end with our proof sketch in Section F.4 and subsequent sections of this appendix.

F.1 Ideal Functionality

UG15 sketched a security proof for Spawn in a GUC-based standard model [93]. However, the ideal functionality used in the sketch, $\mathcal{F}_{1psp-keia}^{\text{IncProc}}$, suffers from the same problems we discussed in Section E.1, and thus it cannot be used to prove the security of Spawn⁺. Specifically, $\mathcal{F}_{1psp-keia}^{\text{IncProc}}$ does not capture the notion of contributiveness. Using the same techniques as Section E.1, we can construct a new ideal functionality that describes a two-flow, single post-specified peer, contributory, and strongly deniable key exchange protocol. Algorithm 6 depicts this functionality, $\mathcal{F}_{1psp-keia}^{+}$.

Algorithm 6 Ideal functionality $\mathcal{F}_{1psp-keia}^{+}$

$\mathcal{F}_{1psp-keia}^{+}$ proceeds as follows, running on security parameter λ, in the $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$-hybrid model, with parties $\overline{P_1},\ldots,\overline{P_n}$ and an adversary $\mathcal{S}$. The functionality is parameterized by a non-information oracle $\mathcal{N}$, and an incrimination procedure IncProc. When initializing, $\mathcal{F}_{1psp-keia}^{+}$ invokes $\mathcal{N}$ with fresh randomness.

on interaction with $\mathcal{N}$:
  Allow $\mathcal{S}$ to communicate with $\mathcal{N}$ by forwarding messages between them. If at any point $\overline{I}$ or $\overline{R}$ is corrupted while $\mathcal{N}$ has produced local output, send the complete state and local output of $\mathcal{N}$ to $\mathcal{S}$.

on $(\texttt{solicit}, sid, \overline{I}, \Phi, aux)$ **from** $\overline{P} \in \{\overline{P_1},\ldots,\overline{P_n}\}$:
  if ($\overline{I}$ is defined) **return**
  Denote $\overline{P}$ as $\overline{I}$ and mark $\overline{I}$ as "active"
  Send $(\texttt{solicit}, sid, \overline{I}, \Phi)$ to $\mathcal{S}$

on $(\texttt{establish}, sid, \overline{I}, \overline{R}, \Phi)$ **from** $\overline{P} \in \{\overline{P_1},\ldots,\overline{P_n}\}$:
  if (an establish message was previously received) **return**
  if ($\overline{I}$ is undefined) Resume processing once $\overline{I}$ is defined
  Denote $\overline{P}$ as $\overline{R}$ and mark $\overline{R}$ as "active"
  Send $(\texttt{establish}, sid, \overline{I}, \overline{R}, \Phi)$ to $\mathcal{S}$

on $(\texttt{set-key}, sid)$ **from** $\mathcal{S}$:
  if (a key tuple (sid, κ) has been recorded) **return**
  if (($\overline{I}$ is undefined) || ($\overline{R}$ is undefined)) **return**
  if (($\overline{I}$ is corrupt) **&&** ($\overline{R}$ is corrupt)) **return**
  if (($\overline{I}$ is uncorrupted) **&&** ($\overline{R}$ is uncorrupted)
  ↪ **&&** ($\overline{I}$ is "active")) { $\kappa \xleftarrow{\$} \{0,1\}^\lambda$ }
  else if (IncProc was previously executed) {
  ↪ Let κ denote the local output of IncProc }
  else if ($\overline{I}$ is corrupt) {
  ↪ Let κ denote the local output of $\mathcal{N}$ }
  else { Halt }
  Send $(\texttt{set-key}, sid, \overline{I}, \overline{R}, \kappa)$ to $\overline{R}$
  Record key tuple (sid, κ)

on $(\texttt{finish}, sid, \overline{P})$ **from** $\mathcal{S}$:
  if (no key tuple (sid, κ) has been recorded) **return**
  if ($\overline{I}$ is not "active") **return**
  if ($\overline{P}$ is uncorrupted) {
    Send $(\texttt{set-key}, sid, \overline{I}, \overline{R}, \kappa)$ to $\overline{I}$ and halt
  } **else** {
    Send the state of $\mathcal{N}$ to $\mathcal{S}$
    Wait for $(\texttt{mismatch-key}, sid, k)$ from $\mathcal{S}$
    if ($k = \kappa$) Halt
    Send $(\texttt{set-key}, sid, \overline{I}, \overline{P}, k)$ to $\overline{I}$ and halt
  }

on $(\texttt{abort}, sid)$ **from** $\mathcal{S}$:
  if (($\overline{I}$ is not defined) ∨ ($\overline{R}$ is not defined)) **return**
  Mark $\overline{I}$ as "aborted"
  Send delayed $(\texttt{abort}, sid, \overline{I})$ to $\overline{I}$
  Send delayed $(\texttt{abort}, sid, \overline{R})$ to $\overline{R}$

on $(\texttt{incriminate}, sid)$ **from** $\mathcal{S}$:
  if (IncProc was previously executed) **return**
  if (($\overline{I}$ is "aborted") **&&** ($\overline{R}$ is "active")
  ↪ **&&** ($\overline{R}$ is uncorrupted)) {
    Send $(\texttt{retrievesecret}, \overline{R})$ to $\bar{\mathcal{G}}_{krkro}^{\mathcal{F},n,\mathbb{G},q,g}$, retrieving SK_R
    Execute IncProc$(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$
  }

$\mathcal{F}^{+}_{1psp\text{-}keia}$ shares similarities with both $\mathcal{F}^{\text{IncProc}}_{1psp-keia}$ and $\mathcal{F}^{+}_{post\text{-}keia}$. An initiating party $\overline{I}$ sends a `solicit` message that is answered by a responder $\overline{R}$ with an `establish` message. When both parties are defined, the simulator $\mathcal{S}$ tells $\mathcal{F}^{+}_{1psp\text{-}keia}$ to generate a session key and notify $\overline{R}$ using a `set-key` message. $\mathcal{S}$ can then send a `finish` message to send the key to $\overline{I}$. If $\mathcal{S}$ corrupts a party $\overline{P}$, it can cause $\overline{I}$ to erroneously report $\overline{P}$ (rather than $\overline{R}$) as a conversation partner with an arbitrary session key. However, it is not possible for $\mathcal{S}$ to accomplish this without also causing the parties to derive different session keys (leading to termination if high-level protocols incorporate implicit or explicit key confirmation [49]).[11]

$\mathcal{F}^{+}_{1psp\text{-}keia}$ also includes an incriminating abort procedure. Unlike $\mathcal{F}^{+}_{post\text{-}keia}$, where the $\mathcal{R}$ is forced to abort, $\mathcal{F}^{+}_{1psp\text{-}keia}$ forces $\mathcal{I}$ to abort when IncProc is called. The incrimination procedure in $\mathcal{F}^{+}_{1psp\text{-}keia}$ also has another important difference: rather than sending the fixed session key to $\mathcal{S}$, it is instead designed to send it directly to $\mathcal{F}^{+}_{1psp\text{-}keia}$ as local output on $\mathcal{F}^{+}_{1psp\text{-}keia}$'s subroutine tape. This is important for appropriately modeling forward secrecy, as defined in Section E.4. Because $\mathcal{F}^{+}_{1psp\text{-}keia}$ can permit an uncorrupted party (specifically, $\overline{R}$) to output a key after IncProc is invoked (unlike $\mathcal{F}^{+}_{post\text{-}keia}$, which prevents this possibility), IncProc must not reveal this key to $\mathcal{S}$; doing so would trivially break forward secrecy. Instead, any attack against forward secrecy must exploit the specification of the real protocol by using leaked information from $\mathcal{N}$ or IncProc. Looking ahead, we will specify the attack that limits Spawn^{+} and ZDH to weak forward secrecy, and differentiate the forward secrecy of XZDH, in Section H.1.

F.2 Spawn^{+} in the GUC Framework

We originally defined Spawn^{+} in Section 6.1. Algorithm 7 redefines Spawn^{+} in terms of the $\mathcal{F}^{+}_{1psp\text{-}keia}$ interface. The program is implicitly parameterized with the group $\mathbb{G}$, q, and g. The adaptation is very similar to the approach we used for DAKEZ in Algorithm 3: the parties retrieve the necessary long-term keys from the shared functionality, the initiator broadcasts its initial message using the routing information in aux, and the responder replies in the usual manner.

We define the shared functionality for the proof, $\bar{\mathcal{G}}^{\text{Spawn}^{+}}_{krkro}$, to be $\bar{\mathcal{G}}^{\text{Spawn}^{+},3,\mathbb{G},q,g}_{krkro}$. We model the key derivation function KDF(x) using the first random oracle in

Algorithm 7 Real protocol Spawn^{+}

on activation with input $(\texttt{solicit}, sid, \mathcal{I}, \Phi, aux)$:
- Record that we are the initiator, $\mathcal{I}$
- Retrieve PK_I and SK_I from shared functionality
- Record PK_I, SK_I, sid, and Φ
- Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}_I()$
- Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I$ using aux for routing

on activation with input $(\texttt{establish}, sid, \mathcal{I}, \mathcal{R}, \Phi)$:
- Record that we are the responder, $\mathcal{R}$
- Retrieve PK_R and SK_R from shared functionality
- Retrieve PK_I from shared functionality
- Record $\mathcal{I}$, PK_I, PK_R, SK_R, sid, and Φ

on $(\mathcal{P} \| g^i \| PQ_I)$ **to** $\mathcal{R}$:
- **if** $(\mathcal{P} \neq \mathcal{I})$ Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
- Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
- Let $t_1 = \mathcal{I} \| \mathcal{R} \| g^i \| PQ_I$
- Compute $\gamma = \text{DREnc}(PK_I, PK_R, g^r \| Q_R, t_1)$
- Let $t_2 = t_1 \| \gamma \| \Phi$
- Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t_2)$
- Compute $k = \text{KDF}((g^i)^r \| Q_k)$
- Erase r and Q_k
- Send $\psi_2 = \mathcal{R} \| \gamma \| \sigma$ to $\mathcal{I}$
- Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{R}, k)$ and halt

on $(\mathcal{P} \| \gamma \| \sigma)$ **to** $\mathcal{I}$:
- Retrieve PK_P from shared functionality
- Let $t = \mathcal{I} \| \mathcal{P} \| g^i \| PQ_I$
- **if** $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t \| \gamma \| \Phi)))$ {
  - Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
- }
- Compute $g^p \| Q_P \leftarrow \text{DRDec}(PK_I, PK_P, SK_I, \Phi, \gamma)$
- Compute $Q_k = \text{QRKey}_I(SQ_I, Q_P)$
- Compute $k = \text{KDF}((g^p)^i \| Q_k)$
- Erase i, SQ_I, and Q_k
- Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{P}, k)$ and halt

on unknown or invalid message:
- Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
- Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

11 This attack also applies to Spawn. The original security proof sketch given by Unger [92, Section 3.8.4.2] erroneously excludes this possibility.

$\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$, and use the second and third random oracles to model the hashes in the RSig and DREAD schemes.

F.3 Proof Strategy

Theorem 4 states the security theorem for Spawn$^+$.

Theorem 4 (*Classical security of Spawn$^+$*)

If the DREAD scheme is sound and IND-CCA2 secure, the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then Spawn$^+$ GUC-realizes $\mathcal{F}^+_{1psp\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$-hybrid model with adaptive security for $\text{IncProc}_{\text{Spawn}^+}$ and non-information oracle $\mathcal{N}_{QRDH}$.

Note that Theorem 4 allows for fully adaptive adversaries due to the use of a non-information oracle, whereas the proof for Spawn only defended against semi-adaptive adversaries [93].

The core idea of our proof for Theorem 4 is nearly identical to the security proof for DAKEZ. The incriminating procedure that we use within $\mathcal{F}^+_{1psp\text{-}keia}$, $\text{IncProc}_{\text{Spawn}^+}$, is shown in Algorithm 8. $\text{IncProc}_{\text{Spawn}^+}$ simply honestly calculates the second message flow from $\overline{R}$, using SK_R to produce the authenticating proof. The non-information oracle $\mathcal{N}_{QRDH}$ can be directly reused in the proof, since Spawn$^+$ uses the same ephemeral key structure as DAKEZ. Where the details of the sketch are identical to the proof for DAKEZ, we refer the reader to the DAKEZ proof sketch in Appendix E.

F.4 Proof of Theorem 4

(Sketch) The general simulator setup is the same as in Appendix E. Specifically, we construct a simulator $\mathcal{S}$ that executes $\mathcal{A}$ internally and simulates parties for $\mathcal{A}$ while interacting with $\mathcal{F}^+_{1psp\text{-}keia}$. We refer to the first flow of Spawn$^+$, as shown in Algorithm 7, as ψ_1, and to the second flow as ψ_2.

Section F.5 describes the simulator construction, and Section F.6 presents the indistinguishability proof.

F.5 Simulator Construction

F.5.1 General handling of $\mathcal{A}$

As in Section E.6, $\mathcal{S}$ allows $\mathcal{A}$ and $\mathcal{Z}$ to communicate, delays messages delayed by $\mathcal{A}$, corrupts ideal parties when $\mathcal{A}$ corrupts the corresponding simulated parties, and replicates output from corrupted parties.

Algorithm 8 $\text{IncProc}_{\text{Spawn}^+}(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I)$ **from** $\mathcal{S}$:

Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
Let $t_1 = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I$
Compute $\gamma = \text{DREnc}(PK_I, PK_R, g^r \| Q_R, t_1)$
Let $t_2 = t_1 \| \gamma \| \Phi$
Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t_2)$
Compute $k = \text{KDF}((g^i)^r \| Q_k)$
Compute $\psi = \text{“}R\text{”} \| \gamma \| \sigma$
Send $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi)$ to $\mathcal{S}$
Locally output k
Halt

F.5.2 Initialization

When $\mathcal{S}$ first initializes, it sends a $(\texttt{setup}, \mathbb{G}, q, g)$ message to $\mathcal{N}$ through $\mathcal{F}^+_{post\text{-}keia}$, and waits to receive a $(\texttt{exchange}, g^i, g^r, PQ_I, Q_R)$ message in response. The group details sent by $\mathcal{S}$ correspond to the protocol instantiation attacked by $\mathcal{A}$. $\mathcal{S}$ makes a note of the ephemeral keys in the exchange message for later use in the simulation.

F.5.3 Receipt of solicit message from $\mathcal{F}^+_{1psp\text{-}keia}$

When $\mathcal{S}$ receives $(\texttt{solicit}, sid, \overline{I}, \Phi_I)$ from $\mathcal{F}^+_{1psp\text{-}keia}$, it honestly constructs a ψ_1 message from $I^{(s)}$ with the help of the non-information oracle $\mathcal{N}$. $\mathcal{S}$ computes $\psi_1 =$ “I” $\|$ g^i $\|$ PQ_I using the g^i and PQ_I values previously received from $\mathcal{N}$ and sends ψ_1 through $\mathcal{A}$ as if it were broadcast by $I^{(s)}$. $\mathcal{S}$ records the value Φ_I for later use.

F.5.4 Receipt of establish message from $\mathcal{F}^+_{1psp\text{-}keia}$

When $\mathcal{S}$ receives an establishment message $(\texttt{establish}, sid, \overline{I}, \overline{R}, \Phi_R)$ from $\mathcal{F}^+_{1psp\text{-}keia}$, it checks to see the circumstances of the simulated ψ_1 message transmission. $\mathcal{S}$ parses ψ_1 to recover g^i and PQ_I. If ψ_1 is not of the correct format, or if the asserted identity is not $I^{(s)}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, delivers the resulting abort to $\overline{R}$ immediately, and withholds the abort message to $\overline{I}$. Otherwise, $\mathcal{S}$ constructs a message ψ_2 from $R^{(s)}$ in response to ψ_1 and causes $\mathcal{F}^+_{1psp\text{-}keia}$ to record a shared secret key. The mechanism for constructing ψ_2 depends on how ψ_1 was generated:

– If $I^{(s)}$ is not corrupt and $\mathcal{S}$ previously created a message ψ_1', but $\psi_1 \neq \psi_1'$, then $\mathcal{A}$ has altered the message in transit. $\mathcal{S}$ constructs ψ_2 through the use of IncProc. $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, but withholds delivery of the resulting abort mes-

sages to $\overline{I}$ and $\overline{R}$. It then sends $(\texttt{incriminate}, sid)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, causing an instance of IncProc to be invoked. Using the values parsed from ψ_1, $\mathcal{S}$ sends $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi_R, g^i, PQ_I)$ to IncProc, causing IncProc to send the message $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi_2)$ in response. IncProc will then privately send the session key to $\mathcal{F}^{+}_{1psp\text{-}keia}$.

– Otherwise, $\mathcal{S}$ constructs the message ψ_2 by producing a ciphertext and forged proof. The method for choosing the ephemeral keys contained within the ciphertext and the keys used to forge the proof depends on the environment in a manner that we will discuss shortly. Given a choice of ephemeral keys for $R^{(s)}$, g^r and Q_R, $\mathcal{S}$ produces a ciphertext $\gamma = \text{DREnc}(PK_I, PK_R, g^r \| Q_R, \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I)$. $\mathcal{S}$ then produces a proof σ_R with tag $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| PQ_I \| \gamma \| \Phi_R$. $\mathcal{S}$ selects ephemeral keys and produces the proof in the following way:
  – If $\mathcal{S}$ previously created ψ_1 in response to a `solicit` message, then $\mathcal{S}$ uses the values of g^r and Q_R retrieved from the non-information oracle $\mathcal{N}$ to produce γ. $\mathcal{S}$ requests a forged proof under $I^{(s)}$'s ephemeral keys by sending $(\texttt{prove}, 1, \{PK_I, PK_R, g^i\}, t)$ to $\mathcal{N}$ and waiting for a message $(\texttt{proof}, \sigma_R)$ in response.
  – If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I)$ to $\mathcal{N}$. Since $\overline{I}$ is corrupted, $\mathcal{F}^{+}_{1psp\text{-}keia}$ immediately sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ uses the newly generated values g^r and Q_R as the ephemeral keys to be contained within γ. $\mathcal{S}$ uses its access to corrupt $\overline{I}$ to retrieve SK_I from $\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$ with a `retrievesecret` message. $\mathcal{S}$ then forges the proof using $\sigma_R = \text{RSig}(PK_I, SK_I, \{PK_I, PK_R, g^i\}, t)$.

  Given γ and σ_R, $\mathcal{S}$ constructs $\psi_2 = \text{“}R\text{”} \parallel \gamma \parallel \sigma_R$.

$\mathcal{S}$ then sends ψ_2 through $\mathcal{A}$ as if $R^{(s)}$ sent it to $I^{(s)}$. $\mathcal{S}$ sends $(\texttt{set-key}, sid)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, causing $\mathcal{F}^{+}_{1psp\text{-}keia}$ to record a shared secret key and causing $\overline{R}$ to output that key. If IncProc was used, $\mathcal{F}^{+}_{1psp\text{-}keia}$ will use its output as the key. Otherwise, $\mathcal{F}^{+}_{1psp\text{-}keia}$ will use the output from $\mathcal{N}$ if $I^{(s)}$ is corrupted, or a fresh random value if $I^{(s)}$ is uncorrupted.

F.5.5 Receipt of ψ_2 by uncorrupted $I^{(s)}$

When uncorrupted $I^{(s)}$ receives message ψ_2 claiming to be from $P^{(s)}$, $\mathcal{S}$ checks to see if $I^{(s)}$ has previously broadcast a message ψ_1. If not, then the message ψ_2 is ignored. $\mathcal{S}$ then parses ψ_2 to extract “P”, the ciphertext γ, and the proof σ_P. If ψ_2 is not of the correct form, or if σ_P is not a correct proof matching Φ_I and the ψ_1 sent by $I^{(s)}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, delivers the resulting `abort` message to $\overline{I}$ immediately, and withholds the `abort` message to $\overline{R}$.

If $I^{(s)}$ has previously broadcast a message ψ_1 and ψ_2 is valid, then $\mathcal{S}$ must cause $\overline{I}$ to output a session key corresponding to the one negotiated between $I^{(s)}$ and $P^{(s)}$. The key depends on the state of the simulation:

– If $\mathcal{A}$ has previously corrupted $P^{(s)}$, then $\mathcal{S}$ must have previously corrupted $\overline{P}$ (since $\mathcal{S}$ corrupts ideal parties corresponding to simulated parties corrupted by $\mathcal{A}$). $\mathcal{S}$ issues a $(\texttt{finish}, sid, \overline{P})$ message to $\mathcal{F}^{+}_{1psp\text{-}keia}$. Since $\overline{P}$ is corrupted, $\mathcal{F}^{+}_{1psp\text{-}keia}$ sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ recovers the ephemeral keys i and SQ_I used to generate ψ_1 from $\mathcal{N}$. $\mathcal{S}$ retrieves SK_P from $\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$ using a `retrievesecret` message, then uses this key to decrypt γ, recovering g^p and Q_P. $\mathcal{S}$ computes $k = \text{KDF}((g^p)^i \parallel \text{QRKey}_I(SQ_I, Q_P))$ and sends $(\texttt{mismatch-key}, sid, k)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, causing $\overline{I}$ to output key k and partner identity $\overline{P}$ with overwhelming probability.

– Otherwise, $P^{(s)}$ is uncorrupted. Because RSig is unforgeable with respect to insider corruption, the only way for σ_P to be valid in this situation is if the sender of the message knows the long-term private key of one of the parties, or the ephemeral key of $I^{(s)}$. Additionally, σ_P must have been computed during this session because its validity depends on ψ_1. Due to the hardness of the discrete log problem in the group (guaranteeing the secrecy of the keys) and the uniqueness of ψ_1, this is only possible if $P^{(s)} = R^{(s)}$ and ψ_2 was previously generated by $\mathcal{S}$. $\mathcal{S}$ sends $(\texttt{finish}, sid, \overline{R})$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, causing $\overline{I}$ to emit an appropriate session key that is shared with $\overline{R}$.

F.5.6 Transmission of ψ_1 by corrupted $I^{(s)}$

When $\mathcal{S}$ has not yet received a `solicit` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $I^{(s)}$ to issue message ψ_1, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\overline{I}$ to send $(\texttt{solicit}, sid, \overline{I}, \perp, \perp)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, but ignores the resulting `solicit` message sent by $\mathcal{F}^{+}_{1psp\text{-}keia}$.

F.5.7 Transmission of ψ_2 by corrupted $R^{(s)}$

When $\mathcal{S}$ has not yet received an `establish` message from $\mathcal{F}^{+}_{1psp\text{-}keia}$, but $\mathcal{A}$ causes a corrupted $R^{(s)}$ to issue message ψ_2, then $\mathcal{S}$ must reflect this in the ideal environment. $\mathcal{S}$ causes $\overline{R}$ to send $(\texttt{establish}, sid, \overline{I}, \overline{R}, \perp)$ to $\mathcal{F}^{+}_{1psp\text{-}keia}$, but ignores the resulting `establish` message sent by $\mathcal{F}^{+}_{1psp\text{-}keia}$.

F.5.8 Constructing state for corrupted parties

When $\mathcal{A}$ corrupts a party in the simulated environment, $\mathcal{S}$ corrupts the corresponding party in the ideal environment. If $\mathcal{A}$ causes corrupted parties to output values, $\mathcal{S}$ outputs these values from the corresponding ideal parties. In addition, $\mathcal{S}$ must provide $\mathcal{A}$ with a simulated historical state for corrupted parties.

If $\mathcal{A}$ corrupts the party known as $I^{(s)}$ after a `solicit` message has been received, but before $I^{(s)}$ has received ψ_2, then $\mathcal{S}$ uses its access to $\mathcal{N}$ to provide the random coins i, and SQ_I used to construct ψ_1. If $I^{(s)}$ already received ψ_2, then $\mathcal{S}$ uses its corruption of $\overline{I}$ to provide the session key k that $\overline{I}$ already output.

If $\mathcal{A}$ corrupts the party known as $R^{(s)}$ after it has already received ψ_1, then $\mathcal{S}$ uses its corruption of $\overline{R}$ to provide the session key k that $\overline{R}$ already output.

F.6 Proof of Indistinguishability

The proof of indistinguishability is similar to the proof described in Section E.7. We show that flows and memory states are indistinguishable in all cases.

F.6.1 The honest case

This situation occurs when $\mathcal{A}$ does not corrupt $I^{(s)}$ or $R^{(s)}$ until after the session concludes, or alter any message flows. The proof of this case is the same as the honest case in Section E.7; since the protocol is executed honestly (with the exception of proof generation), $\mathcal{Z}$ would need to break the anonymity against full key exposure of RSig to identify that the proofs were forged, or $\mathcal{Z}$ would need to break the computational Diffie-Hellman assumption within $\mathbb{G}$ to identify that the shared session key was randomly generated.

F.6.2 Alteration of ψ_1

This situation occurs when ψ_1 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $R^{(s)}$ is corrupted when ψ_1 is delivered. The proof of this case is the same as in Section E.7, except that $\mathcal{Z}$ is given even less information when subsequently corrupting $R^{(s)}$; both the ψ_2 message and session key generated by IncProc are effectively generated honestly, so the messages and output from $R^{(s)}$ are indistinguishable from a real protocol execution. $I^{(s)}$ aborts as expected if ψ_2 is delivered, since RSig is unforgeable with respect to insider corruption.

F.6.3 Alteration of ψ_2

This situation occurs when ψ_2 generated by $\mathcal{S}$ is altered by $\mathcal{A}$ in transit, but neither $I^{(s)}$ nor $P^{(s)}$ (the party named as the communication partner in ψ_2) are corrupted when ψ_2 is delivered. $\mathcal{S}$ causes $I^{(s)}$ to immediately abort when it receives an altered ψ_2. As mentioned previously, $I^{(s)}$ will always abort because the proof in the altered ψ_2 message cannot be correct due to the unforgeability with respect to insider corruption of RSig.

F.6.4 Indistinguishability under corruptions

This situation occurs when either party is corrupted at a time before the times covered by the previous cases. The proof of this case is nearly the same as in Section E.7. When both $I^{(s)}$ and $R^{(s)}$ are corrupted, indistinguishability is trivial because $\mathcal{S}$ can directly replicate the behavior of $\mathcal{A}$. With only a single corrupted party, the other effectively performs the protocol honestly. The exception is when $I^{(s)}$ is corrupted but $R^{(s)}$ is not, in which case $\mathcal{S}$ forges the proof in ψ_2 using the long-term key of $\overline{I}$. This forgery is undetectable by $\mathcal{Z}$ due to the anonymity against full key exposure of RSig, even if $\mathcal{A}$ corrupts all parties after the session. All memory states are consistent with honest execution, as mentioned above.

The only major difference with Section E.7 is that $\mathcal{A}$ can cause $I^{(s)}$ to accept the wrong communication partner without aborting the protocol by leaving $I^{(s)}$ and $R^{(s)}$ uncorrupted, corrupting $P^{(s)}$, and replacing $R^{(s)}$'s ψ_2 message with an honestly generated message from $P^{(s)}$.[12] In this case, $\mathcal{F}^{+}_{1psp\text{-}keia}$ grants $\mathcal{S}$ access to the internal state of $\mathcal{N}$. $\mathcal{S}$ uses this state to compute $\overline{I}$'s output honestly, with the exception of using $\overline{P}$'s long-term secret key to decrypt the DREAD ciphertext. Due to the soundness of the DREAD scheme, this process results in the same key that would have been computed by an honestly behaving $I^{(s)}$, so the key must be the one that $\mathcal{Z}$ expects. Revealing the state of $\mathcal{N}$ and allowing $\mathcal{S}$ to completely dictate the output of $\overline{I}$ is not a problem in this case, since $\mathcal{A}$ effectively controls one of the conversation partners from the perspective of $I^{(s)}$. Leaking $\mathcal{N}$'s state to $\mathcal{S}$ models that the adversary has access to the key shared with $\overline{I}$, and it has full control over the value of the key (since it controls the second ephemeral key contribution).

F.6.5 Data from other sessions

As in Section E.7, no information from other sessions can be used to assist $\mathcal{A}$ with the generation of false message flows: ψ_1 is generated using no long-term infor-

12 As we mentioned in Section F.1, key confirmation can be used in higher-level protocols to alleviate this weakness.

mation, and ψ_2 requires computation of an SoK bound to the contents of ψ_1. Since RSig is unforgeable with respect to insider corruption, collecting SoKs from other sessions does not allow $\mathcal{A}$ to produce proofs correctly bound to the session under attack. Additionally, the IND-CCA2 security of the DREAD scheme implies that it is also non-malleable [12]. This non-malleability prevents $\mathcal{A}$ from modifying DREAD ciphertexts from other sessions to bind them to the session under attack without corrupting one of the parties capable of decrypting the ciphertext. □

G Proof of ZDH Security

It is possible to prove the security of ZDH in nearly the same environment as Spawn$^+$, since the protocols share most properties. In fact, ZDH can GUC-realize the same ideal functionality $\mathcal{F}^+_{1psp\text{-}keia}$ that is GUC-realized by Spawn$^+$. Algorithm 9 defines ZDH (previously defined in Section 6) in terms of the $\mathcal{F}^+_{1psp\text{-}keia}$ interface. The security theorem for ZDH is as follows:

Theorem 5 (*Classical security of ZDH*)

If the MAC is weakly unforgeable under chosen message attack [13], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then ZDH GUC-realizes $\mathcal{F}^+_{1psp\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$-hybrid model with partially adaptive security for $\text{IncProc}_{\text{ZDH}}$ and non-information oracle $\mathcal{N}_{ZDH}$.

There are a few key differences between Theorem 5 and Theorem 4. ZDH uses a protocol-specific incrimination procedure, $\text{IncProc}_{\text{ZDH}}$, that simply provides an honest generation of the second message flow using $\overline{R}$'s long-term secret key. $\mathcal{N}_{ZDH}$ is similar to $\mathcal{N}_{QRDH}$, but also enables calls to MAC with the shared key M_k, and supports adding extra Diffie-Hellman secrets into the KDF. We define the phrase "partially adaptive" in Theorem 5 to mean that $\mathcal{S}$ may corrupt $\overline{I}$ at any time, but $\mathcal{S}$ may only corrupt $\overline{R}$ after a `finish` message has been received by $\mathcal{F}^+_{1psp\text{-}keia}$. In the ideal setting, this means that $\mathcal{S}$ may not send an `establish` message from a corrupted party. In the real setting, this means that $\mathcal{A}$ cannot transmit a response message ψ_2 from a corrupted party. This restriction intentionally prevents the protocol from achieving online deniability for $\mathcal{R}$ in order to allow ZDH to GUC-realize $\mathcal{F}^+_{1psp\text{-}keia}$. We define the shared functionality for the proof, $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$, to be $\bar{\mathcal{G}}^{\text{ZDH},5,\mathbb{G},q,g}_{krkro}$. The first three random oracles in $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ are used to model

Algorithm 9 Real protocol ZDH

on activation with input $(\texttt{solicit}, sid, \mathcal{I}, \Phi, aux)$:
- Record that we are the initiator, $\mathcal{I}$
- Retrieve PK_I and SK_I from shared functionality
- Record PK_I, SK_I, sid, and Φ
- Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}_I()$
- Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I$ using aux for routing

on activation with input $(\texttt{establish}, sid, \mathcal{I}, \mathcal{R}, \Phi)$:
- Record that we are the responder, $\mathcal{R}$
- Retrieve PK_R and SK_R from shared functionality
- Retrieve PK_I from shared functionality
- Record $\mathcal{I}$, PK_I, PK_R, SK_R, sid, and Φ

on $(\mathcal{P} \| g^i \| PQ_I)$ **to** $\mathcal{R}$:
- **if** $(\mathcal{P} \neq \mathcal{I})$ Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
- Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
- Compute $\kappa = \text{KDF}_1((g^i)^r \| (PK_I)^r \| Q_k)$
- Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
- Let $t = \mathcal{I} \| \mathcal{R} \| g^i \| g^r \| PQ_I \| Q_R \| \Phi$
- Compute $mac = \text{MAC}(M_k, t)$
- Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
- Erase r, Q_k, κ, and M_k
- Send $\psi_2 = \mathcal{R} \| g^r \| Q_R \| mac \| \sigma$ to $\mathcal{I}$
- Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{R}, k)$ and halt

on $(\mathcal{P} \| g^p \| Q_P \| mac \| \sigma)$ **to** $\mathcal{I}$:
- Retrieve PK_P from shared functionality
- Let $t = \mathcal{I} \| \mathcal{P} \| g^i \| g^p \| PQ_I \| Q_P \| \Phi$
- **if** $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t)))$ {
  - Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
- }
- Compute $Q_k = \text{QRKey}_I(SQ_I, Q_P)$
- Compute $\kappa = \text{KDF}_1((g^p)^i \| (g^p)^{SK_I} \| Q_k)$
- Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
- Compute $mac' = \text{MAC}(M_k, t)$
- **if** $(mac \neq mac')$ {
  - Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
- }
- Erase i, SQ_I, Q_k, κ, and M_k
- Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{P}, k)$ and halt

on unknown or invalid message:
- Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
- Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

Algorithm 10 $\text{IncProc}_{\text{ZDH}}(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I)$ **from** $\mathcal{S}$:
  Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
  Compute $\kappa = \text{KDF}_1((g^i)^r \| (PK_I)^r \| Q_k)$
  Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
  Let $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^r \| PQ_I \| Q_R \| \Phi$
  Compute $mac = \text{MAC}(M_k, t)$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
  Compute $\psi = \text{“}R\text{”} \| g^r \| Q_R \| mac \| \sigma$
  Send $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi)$ to $\mathcal{S}$
  Locally output k
  Halt

KDF_1, KDF_2, and KDF_3. The last two random oracles model the hashes in the RSig and DREAD schemes.

The proof sketch of Theorem 5 is nearly identical to the one for Spawn$^+$, since the protocols behave similarly in all applicable scenarios. While $\mathcal{F}^+_{1psp\text{-}keia}$ provides some additional features that are not necessary for the proof of Theorem 5, we avoid defining a new ideal functionality in order to clearly show the security relationship between Spawn$^+$ and ZDH.

Algorithm 10 shows the incrimination procedure for ZDH, which is simply an honest generation of a response ψ_2 using $\mathcal{R}$'s long-term secret key to produce the proof. Algorithm 11 shows the non-information oracle used for the proof. $\mathcal{N}_{ZDH}$ is similar to $\mathcal{N}_{QRDH}$, except that it also adds a facility for the simulator to generate and verify a MAC using the shared key derived from the exchange. $\mathcal{N}_{ZDH}$ also permits the caller $\mathcal{M}$ to pass in a set of additional Diffie-Hellman public contributions, C, that are combined with the responder's ephemeral key, r, as part of the input to KDF_1. For ZDH, C contains only the initiator's long-term public key, PK_I.

It is clear that $\mathcal{N}_{ZDH}$ is still a non-information oracle, despite the addition of the new MAC methods and the introduction of the C parameter. As in $\mathcal{N}_{QRDH}$, $\mathcal{M}$ is never given enough information to compute g^{ir} (when ok is TRUE) or α^r (when ok is FALSE). $\mathcal{M}$ is also not given enough information to compute Q_k. Because KDF_1 is modeled by a random oracle, the local output from `complete` is indistinguishable from random without access to these values, which are used as input to KDF_1 in KDF-Val. This holds true even if $\mathcal{M}$ is able to compute the other values concatenated into γ using knowledge of the secrets associated with keys in C, since it cannot compute the entire input to KDF_1. For the same reason, the outputs of KDF_1 and KDF_2 that are passed to MAC in the `authenticate` and `verify` handlers is independent of i, r, and Q_k. Note that the non-information property does *not* depend on the security of MAC, since the MAC key is already computationally independent of the secrets and $\mathcal{M}$ knows m.

Algorithm 11 A non-information oracle $\mathcal{N}_{ZDH}$

$\mathcal{N}_{ZDH}$ copies the `setup` and `prove` commands from $\mathcal{N}_{QRDH}$ (Algorithm 5), slightly modifies its `complete` command, and introduces two new commands.

function KDF-Val(C, r, Q_k)
  $\gamma \leftarrow \varnothing$
  for all $(c \in C)$ **do** {
    if $((c \notin \mathbb{G}) \,\|\, (c \text{ is identity element}))$ Halt
    $\gamma \leftarrow \gamma \| c^r$
  }
  return $\text{KDF}_1(\gamma \,\|\, Q_k)$
}

on $(\texttt{setup}, \mathbb{G}, q, g)$ **from** $\mathcal{M}$:
  Handle the message as in $\mathcal{N}_{QRDH}$

on $(\texttt{prove}, p, S, m)$ **from** $\mathcal{M}$:
  Handle the message as in $\mathcal{N}_{QRDH}$

on $(\texttt{complete}, ok, \alpha, \beta, C)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (already output a key) **return**
  if (ok is TRUE) {
    Compute $k = \text{KDF-Val}(\{g^i\} \cup C, r, Q_k)$
  } **else** {
    Generate new $r \xleftarrow{\$} \mathbb{Z}_q$
    Generate new $(Q_R, Q_k) \leftarrow \text{QRGen}_R(\beta)$
    Compute $k = \text{KDF-Val}(\{\alpha\} \cup C, r, Q_k)$
  }
  Locally output k

on $(\texttt{authenticate}, C, m)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (a `complete` message has been received) **return**
  Compute $k = \text{KDF-Val}(\{g^i\} \cup C, r, Q_k)$
  Compute $mac = \text{MAC}(\text{KDF}_2(k), m)$
  Send $(\texttt{authentication}, mac)$ to $\mathcal{M}$

on $(\texttt{verify}, C, m, mac)$ **from** $\mathcal{M}$:
  if (no `setup` message has been received) **return**
  if (a `complete` message has been received) **return**
  Compute $k = \text{KDF-Val}(\{g^i\} \cup C, r, Q_k)$
  Compute $mac' = \text{MAC}(\text{KDF}_2(k), m)$
  if $(mac' = mac)$ { Send $(\texttt{verified}, \text{TRUE})$ to $\mathcal{M}$ }
  else { Send $(\texttt{verified}, \text{FALSE})$ to $\mathcal{M}$ }

The proof proceeds as follows:

(Sketch) The proof of Theorem 5 is nearly identical to the proof of Theorem 4 in Appendix F. For

this reason, we only highlight the differences in simulator construction between the two sketches. Section G.1 constructs the simulator and Section G.2 demonstrates indistinguishability.

G.1 Simulator Construction

The simulator $\mathcal{S}$ behaves exactly as in Section F.5 except that in the proof for Theorem 5, all references to $\bar{\mathcal{G}}^{\text{Spawn}^+}_{krkro}$ are replaced by references to $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$.

G.1.1 Receipt of `establish` message from $\mathcal{F}^+_{1psp\text{-}keia}$

This case is identical to the case from Section F.5, except for the mechanism for constructing ψ_2:

- If $I^{(s)}$ is not corrupt and $\mathcal{S}$ previously created a message ψ_1', but $\psi_1 \neq \psi_1'$, then $\mathcal{S}$ behaves as in Section F.5. The only difference is that IncProc returns a properly constructed ψ_2 message for ZDH.
- Otherwise, $\mathcal{S}$ constructs the message ψ_2 by producing a forged proof σ_R with tag t = "I"‖"R"‖g^i‖g^r‖PQ_I‖Q_R‖Φ_R. The method for choosing the ephemeral keys (g^r and Q_R) and the keys used to forge the proof depends on the environment:
  - If $\mathcal{S}$ previously created ψ_1 in response to a `solicit` message, then $\mathcal{S}$ uses the values of g^r and Q_R retrieved from the non-information oracle $\mathcal{N}$. $\mathcal{S}$ requests a forged proof under $I^{(s)}$'s ephemeral keys by sending $(\texttt{prove}, 1, \{g^I, g^R, g^i\}, t)$ to $\mathcal{N}$ and waiting for a message $(\texttt{proof}, \sigma_R)$ in response. $\mathcal{S}$ also requests a MAC by sending $(\texttt{authenticate}, \{PK_I\}, t)$ to $\mathcal{N}$ and waiting for a message $(\texttt{authentication}, mac)$ in response.
  - If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I, \{PK_I\})$ to $\mathcal{N}$. Since $\overline{I}$ is corrupted, $\mathcal{F}^+_{1psp\text{-}keia}$ immediately sends the state of $\mathcal{N}$ to $\mathcal{S}$. $\mathcal{S}$ uses the newly generated values g^r and Q_R as the ephemeral keys for ψ_2. $\mathcal{S}$ uses its access to corrupt $\overline{I}$ to retrieve SK_I from $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ with a `retrievesecret` message. $\mathcal{S}$ forges the proof using $\sigma_R = \text{RSig}(PK_I, SK_I, \{g^I, g^R, g^i\}, t)$. $\mathcal{S}$ also uses the state of $\mathcal{N}$ to compute mac. $\mathcal{S}$ computes $\kappa = \text{KDF}_1((g^i)^r \| (PK_I)^r \| Q_k)$, $M_k = \text{KDF}_2(\kappa)$, and then $mac = \text{MAC}(M_k, t)$.

  Given a choice of ephemeral keys, mac, and σ_R, $\mathcal{S}$ constructs ψ_2 = "R" ‖ g^r ‖ Q_R ‖ mac ‖ σ_R.

As in Section F.5, $\mathcal{S}$ sends ψ_2 through $\mathcal{A}$ as if $R^{(s)}$ sent it to $I^{(s)}$, and then sends $(\texttt{set-key}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$.

G.1.2 Receipt of ψ_2 by uncorrupted $I^{(s)}$

When uncorrupted $I^{(s)}$ receives message ψ_2 claiming to be from $P^{(s)}$, $\mathcal{S}$ checks to see if $I^{(s)}$ has previously broadcast a message ψ_1. If not, then the message ψ_2 is ignored. $\mathcal{S}$ then parses ψ_2 to extract "P", g^p, Q_P, mac_P, and the proof σ_P. If ψ_2 is not of the correct form, σ_P is not a correct proof matching Φ_I and the ψ_1 sent by $I^{(s)}$, or if $P^{(s)}$ is corrupted, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, delivers the resulting `abort` message to $\overline{I}$ immediately, and withholds the `abort` message to $\overline{R}$.

If $I^{(s)}$ has previously broadcast a message ψ_1, ψ_2 is valid, and $P^{(s)}$ is uncorrupted, then $\mathcal{S}$ must cause $\overline{I}$ to output a session key corresponding to the one negotiated between $I^{(s)}$ and $P^{(s)}$. $\mathcal{S}$ proceeds under the assumption that $P^{(s)} = R^{(s)}$ (we argue later that this must be the case). $\mathcal{S}$ uses $\mathcal{N}$ to check the validity of mac_P by sending $(\texttt{verify}, \{PK_I\}, t, mac_P)$, where tag t = "I"‖"R"‖g^i‖g^p‖PQ_I‖Q_P‖Φ_I. If $\mathcal{S}$ receives $(\texttt{verified}, \text{FALSE})$ in response, then it aborts $\overline{I}$ in the same manner as above. Otherwise, $\mathcal{S}$ sends $(\texttt{finish}, sid, \overline{R})$ to $\mathcal{F}^+_{1psp\text{-}keia}$, causing $\overline{I}$ to emit a session key shared with $\overline{R}$.

G.1.3 Transmission of ψ_2 by corrupted $R^{(s)}$

$\mathcal{A}$ is not permitted to transmit ψ_2 messages from corrupted parties, so this case is no longer needed.

G.2 Proof of Indistinguishably

The proof of indistinguishability given in Section F.6 also applies here with several changes. The proof no longer needs to consider corruptions of the responder before transmission of ψ_2, so these cases can be ignored. As in Section F.6, the anonymity against full key exposure of RSig prevents $\mathcal{Z}$ from detecting forgeries of σ_P in ψ_2 by $\mathcal{S}$, even if $\mathcal{A}$ corrupts all parties after the session. Similarly, the CDH assumption on the underlying group prevents $\mathcal{Z}$ from distinguishing the output key in the honest case, and the unforgeability of RSig with respect to insider corruption prevents $\mathcal{Z}$ from producing forged proofs in ψ_2, even with information collected from other sessions.

The only significant difference with the proof in Section F.6 is indistinguishability in the case of an altered ψ_2 received by an uncorrupted $I^{(s)}$. Due to the unforgeability of RSig with respect to insider corruption, any modification to ψ_2 must include a proof σ_P naming a corrupted party $P^{(s)} \neq R^{(s)}$. In this case, $\mathcal{S}$ always aborts $\overline{I}$. This procedure always matches the simulated environment because ψ_2 must be invalid. If ψ_2 includes a valid proof σ_P and $I^{(s)}$ does *not* abort, then this implies

that mac_P is also valid. Due to the weak unforgeability under chosen message attack of MAC, this is only possible if $\mathcal{A}$ knows the MAC key derived from a secret shared between $I^{(s)}$ and $P^{(s)}$, which in turn is only possible (due to the security of the key derivation functions and the hardness of the CDH problem in the underlying group) if $\mathcal{A}$ honestly generated a response from $P^{(s)}$. An honest generation of this type is not permitted due to the "partially adaptive" constraint in Theorem 5, because it corresponds to the completion of the protocol by a corrupted responder. Consequently, $\mathcal{A}$ cannot possibly cause $I^{(s)}$ to output a key when altering ψ_2. □

H Proof of XZDH Security

H.1 Forward Secrecy of DAKEZ, Spawn⁺, and ZDH

We previously claimed that DAKEZ has strong forward secrecy, and that the purpose of XZDH is to improve the forward secrecy properties of ZDH. We can now precisely distinguish between these forward secrecy levels from the perspective of our GUC-based proof sketches.

In Section E.4, we defined strong and weak forward secrecy in terms of ideal GUC processes. In these terms, the proof of Theorem 3 shows that DAKEZ offers strong forward secrecy. Consider the simulator $\mathcal{S}$ interacting with $\mathcal{F}^{+}_{post\text{-}keia}$ as sketched in Section E.6. In order to break strong forward secrecy, $\mathcal{Z}$ must use $\mathcal{S}$ to collect enough information from $\mathcal{N}_{QRDH}$ or $\text{IncProc}_{\text{DAKEZ}}$ to distinguish a key output by $\overline{I}$ or $\overline{R}$ from random without corrupting the entity or its partner. The definition of $\mathcal{F}^{+}_{post\text{-}keia}$ guarantees that no keys can be output when $\text{IncProc}_{\text{DAKEZ}}$ is invoked, so it cannot be used to gather information. Moreover, the only time that $\mathcal{S}$ sends a `deliver` message when a party and its partner are uncorrupted is when both $\overline{I}$ and $\overline{R}$ are uncorrupted, the key was generated by $\mathcal{N}_{QRDH}$, and $\mathcal{N}_{QRDH}$'s proposed exchange was accepted. By the non-information property, $\mathcal{N}_{QRDH}$ yields no information about this key.

In contrast, $\mathcal{F}^{+}_{1psp\text{-}keia}$ only guarantees weak forward secrecy, which prevents $\mathcal{S}$ from aborting the session. IncProc is not useful to $\mathcal{Z}$ because it requires an `abort` command to be invoked. By definition, $\mathcal{N}$ is also not useful, even if its proposed exchange is rejected. Proving Theorem 4 and Theorem 5 therefore shows that Spawn⁺ and ZDH have (at least) weak forward secrecy.

If Spawn⁺ and ZDH do not offer strong forward secrecy, then there should exist an attack in which $\mathcal{Z}$ can issue instructions to $\mathcal{S}$ to learn information about a key output by $\overline{P}$ with partner $\overline{P'}$ without corrupting either $\overline{P}$ or $\overline{P'}$ during the corresponding session. Indeed, such an attack exists when $\mathcal{S}$ is permitted to abort $\mathcal{F}^{+}_{1psp\text{-}keia}$. For simplicity, we will only examine the ZDH case—a similar attack works against Spawn⁺. The attack, which takes place in the setting of Theorem 5 and is the idealized equivalent of the one discussed in Section 7, proceeds as follows:

1. $\mathcal{Z}$ instructs $\overline{I}$ to send a `solicit` message to $\mathcal{F}^{+}_{1psp\text{-}keia}$. This causes $\mathcal{S}$ to simulate a message ψ_1 from $I^{(s)}$.
2. $\mathcal{Z}$ generates $z \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_Z, SQ_Z) \leftarrow \text{QRGen}_I()$. $\mathcal{Z}$ then instructs $\mathcal{S}$ to modify ψ_1 to be "I" $\|$ g^z $\|$ PQ_Z.
3. $\mathcal{Z}$ instructs $\overline{R}$ to send an `establish` message to $\mathcal{F}^{+}_{1psp\text{-}keia}$. This causes $\mathcal{S}$ to abort the protocol, interact with $\text{IncProc}_{\text{ZDH}}$, and simulate the resulting ψ_2 = "R" $\|$ g^r $\|$ Q_R $\|$ mac $\|$ σ_R message from $R^{(s)}$. $\overline{R}$ receives a `set-key` message identifying partner $\overline{I}$ with a key k generated by $\text{IncProc}_{\text{ZDH}}$. Note that $\mathcal{S}$ does not have access to k, since $\text{IncProc}_{\text{ZDH}}$ sends it directly to $\mathcal{F}^{+}_{1psp\text{-}keia}$.
4. $\mathcal{Z}$ instructs $\mathcal{S}$ to deliver ψ_2 to $I^{(s)}$, which causes $\mathcal{S}$ to issue an `abort` command and deliver an `abort` message to $\overline{I}$. The session has now completed.
5. $\mathcal{Z}$ causes $\overline{I}$ to be corrupted, revealing secret key I. $\mathcal{Z}$ can now compute $(g^r)^z$, $(g^r)^I$, and $\text{QRKey}_I(SQ_Z, Q_R)$, which allows it to derive k. $\mathcal{Z}$ can distinguish k from random, even though neither $\overline{I}$ nor $\overline{R}$ were corrupted during the session that $\overline{R}$ outputted k.

After presenting a security theorem for XZDH, we define its forward secrecy in Section H.2.1.

H.2 XZDH Security Theorem

The only difference between XZDH and ZDH is the introduction of signed prekeys in XZDH. Nonetheless, the presence of signed prekeys necessitates some changes to the security model.

Signed prekeys persist across protocol sessions, so they must be modeled as part of the shared functionality. Algorithm 12 depicts $\bar{\mathcal{G}}^{XZDH}$, a shared functionality that combines $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ with a mechanism for sharing signed prekeys. This mechanism is essentially another instance of a key registration with knowledge functionality, $\bar{\mathcal{G}}^{\mathcal{F}}_{krk}$, except with the ability for keys to be replaced with new ones, and distribution of digital signatures produced using the long-term keys.

We originally defined XZDH in Section 7. Algorithm 13 depicts the XZDH program following the

Algorithm 12 The shared functionality $\bar{\mathcal{G}}^{XZDH}$

$\bar{\mathcal{G}}^{XZDH}$ copies all behavior and commands supported by $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ (i.e., it handles `register`, `retrieve`, `retrievesecret`, and `ro` messages identically). In addition, it also supports the following commands:

on (`spk-refresh`) **from** $\mathcal{P}$:
  if (no tuple $(\mathcal{P}, PK, SK)$ is recorded) **return**
  if ($\mathcal{P}$ is corrupt) **return**
  if (tuple $t = (\texttt{spk}, \mathcal{P}, pk, sk, \xi)$ is recorded) Delete t
  $r \xleftarrow{\$} \{0,1\}^\lambda$
  Record tuple $(\texttt{spk}, \mathcal{P}, g^r, r, \text{Sig}(PK, SK, g^r))$

on (`spk-corrupt-refresh`, r) **from** $\mathcal{P}$:
  if (no tuple $(\mathcal{P}, PK, SK)$ is recorded) **return**
  if ($\mathcal{P}$ is uncorrupted) **return**
  if (tuple $t = (\texttt{spk}, \mathcal{P}, pk, sk, \xi)$ is recorded) Delete t
  Record tuple $(\texttt{spk}, \mathcal{P}, g^r, r, \text{Sig}(PK, SK, g^r))$

on (`spk-retrieve`, $\mathcal{P}'$) **from** $\mathcal{P}$:
  if (tuple $t = (\texttt{spk}, \mathcal{P}', pk, sk, \xi)$ is recorded) {
    Send $(\texttt{spk}, \mathcal{P}', pk, \xi)$ to $\mathcal{P}$
  } **else** {
    Send $(\texttt{spk}, \mathcal{P}', \perp, \perp)$ to $\mathcal{P}$
  }

on (`spk-retrievesecret`, $\mathcal{P}'$) **from** $\mathcal{P}$:
  if (($\mathcal{P}$ is honest) $\wedge$ ($\mathcal{P}$'s code is $\notin \mathcal{F}$)) **return**
  if (($\mathcal{P}$ is corrupt) $\wedge$ ($\mathcal{P} \neq \mathcal{P}'$)) **return**
  if (tuple $t = (\texttt{spk}, \mathcal{P}', pk, sk, \xi)$ is recorded) {
    Send t to $\mathcal{P}$
  }

$\mathcal{F}^{+}_{1psp\text{-}keia}$ interface. The program is nearly identical to Algorithm 9, except that it incorporates signed prekey distribution. Differences between Algorithm 13 and Algorithm 9 are shaded in the program listing.

The security theorem for XZDH is as follows:

Theorem 6 (*Classical security of XZDH*)

If the MAC is weakly unforgeable under chosen message attack [13], the RSig/RVrf scheme is anonymous against full key exposure and unforgeable with respect to insider corruption, and the CDH assumption holds in the underlying group, then XZDH GUC-realizes $\mathcal{F}^{+}_{1psp\text{-}keia}$ within the erasure $\bar{\mathcal{G}}^{XZDH}$-hybrid model with partially adaptive security for $\text{IncProc}_{\text{XZDH}}$ and non-information oracle $\mathcal{N}_{ZDH}$.

There are a few differences between Theorem 6 and Theorem 5. The shared functionality, $\bar{\mathcal{G}}^{XZDH}$, enables the use of signed prekeys. The incrimination procedure shown in Algorithm 14, $\text{IncProc}_{\text{XZDH}}$, is identical to

Algorithm 13 Real protocol XZDH

on activation with input $(\texttt{solicit}, sid, \mathcal{I}, \Phi, aux)$:
  Record that we are the initiator, $\mathcal{I}$
  Retrieve PK_I and SK_I from shared functionality
  Retrieve $(\texttt{spk}, \mathcal{I}, pk, sk, \xi)$ from shared functionality
  Record PK_I, SK_I, pk, sid, and Φ
  Record $i \xleftarrow{\$} \mathbb{Z}_q$ and $(PQ_I, SQ_I) \leftarrow \text{QRGen}_I()$
  Broadcast $\psi_1 = \mathcal{I} \| g^i \| PQ_I \| pk \| \xi$ using aux for routing

on activation with input $(\texttt{establish}, sid, \mathcal{I}, \mathcal{R}, \Phi)$:
  Record that we are the responder, $\mathcal{R}$
  Retrieve PK_R and SK_R from shared functionality
  Retrieve PK_I from shared functionality
  Record $\mathcal{I}$, PK_I, PK_R, SK_R, sid, and Φ

on $(\mathcal{P} \| g^i \| PQ_I \| pk \| \xi)$ **to** $\mathcal{R}$:
  if $(\mathcal{P} \neq \mathcal{I})$ Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
  if $(\neg(\text{SVerif}(PK_I, pk, \xi)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{R})$ and halt
  }
  Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
  Compute $\kappa = \text{KDF}_1((g^i)^r \| pk^r \| (PK_I)^r \| Q_k)$
  Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
  Let $t = \mathcal{I} \| \mathcal{R} \| g^i \| g^r \| PQ_I \| Q_R \| pk \| \Phi$
  Compute $mac = \text{MAC}(M_k, t)$
  Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
  Erase r, Q_k, κ, and M_k
  Send $\psi_2 = \mathcal{R} \| g^r \| Q_R \| mac \| \sigma$ to $\mathcal{I}$
  Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{R}, k)$ and halt

on $(\mathcal{P} \| g^p \| Q_P \| mac \| \sigma)$ **to** $\mathcal{I}$:
  Retrieve PK_P from shared functionality
  Retrieve $(\texttt{spk}, \mathcal{I}, pk', sk, \xi)$ from shared functionality
  if $(pk \neq pk')$ Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  Let $t = \mathcal{I} \| \mathcal{P} \| g^i \| g^p \| PQ_I \| Q_P \| pk \| \Phi$
  if $(\neg(\text{RVrf}(\{PK_I, PK_P, g^i\}, \sigma, t)))$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Compute $Q_k = \text{QRKey}_I(SQ_I, Q_P)$
  Compute $\kappa = \text{KDF}_1((g^p)^i \| (g^p)^{sk} \| (g^p)^{SK_I} \| Q_k)$
  Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
  Compute $mac' = \text{MAC}(M_k, t)$
  if $(mac \neq mac')$ {
    Locally output $(\texttt{abort}, sid, \mathcal{I})$ and halt
  }
  Erase i, SQ_I, Q_k, κ, and M_k
  Locally output $(\texttt{set-key}, sid, \mathcal{I}, \mathcal{P}, k)$ and halt

on unknown or invalid message:
  Let $\mathcal{P}$ be our activated role ($\mathcal{I}$ or $\mathcal{R}$)
  Locally output $(\texttt{abort}, sid, \mathcal{P})$ and halt

Algorithm 14 $\text{IncProc}_{\text{XZDH}}(sid, \overline{I}, \overline{R}, PK_I, PK_R, SK_R)$

on $(\texttt{inc}, sid, \mathbb{G}, g, q, \overline{I}, \overline{R}, \text{“}I\text{”}, \text{“}R\text{”}, \Phi, g^i, PQ_I, g^\Gamma)$ **from** $\mathcal{S}$:
- Generate $r \xleftarrow{\$} \mathbb{Z}_q$ and $(Q_R, Q_k) \leftarrow \text{QRGen}_R(PQ_I)$
- Compute $\kappa = \text{KDF}_1((g^i)^r \| (g^\Gamma)^r \| (PK_I)^r \| Q_k)$
- Compute $M_k = \text{KDF}_2(\kappa)$ and $k = \text{KDF}_3(\kappa)$
- Let $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^r \| PQ_I \| Q_R \| g^\Gamma \| \Phi$
- Compute $mac = \text{MAC}(M_k, t)$
- Compute $\sigma = \text{RSig}(PK_R, SK_R, \{PK_I, PK_R, g^i\}, t)$
- Compute $\psi = \text{“}R\text{”} \| g^r \| Q_R \| mac \| \sigma$
- Send $(\texttt{inc}, sid, \overline{I}, \overline{R}, \psi)$ to $\mathcal{S}$
- Locally output k
- Halt

$\text{IncProc}_{\text{ZDH}}$ except for the addition of a signed prekey g^Γ as input, and the use of g^Γ to derive the shared key. Although $\mathcal{N}_{ZDH}$ serves as the non-information oracle, the simulator for XZDH must provide it with signed prekeys so that the correct session keys are derived.

H.2.1 Forward Secrecy of XZDH

XZDH offers forward secrecy that exists somewhere between the traditional “strong” and “weak” definitions. In the context of Theorem 6, the property can now be expressed precisely: if a party $\overline{P}$ outputs a key κ and a partner identifier $\overline{P'}$, then $\mathcal{Z}$ can never distinguish κ from $\kappa' \xleftarrow{\$} \{0,1\}^\lambda$ unless $\overline{P}$ or $\overline{P'}$ was corrupted before the corresponding session completed, or $\mathcal{S}$ aborted the session and $\overline{P'}$ did not subsequently issue an `spk-refresh` message to $\bar{\mathcal{G}}^{XZDH}$.

By definition, $\mathcal{N}_{ZDH}$ cannot assist $\mathcal{Z}$ with breaking forward secrecy. The only viable approach for $\mathcal{Z}$ is to abort the session and derive information from IncProc_{XZDH}, similarly to the attack on strong forward secrecy that we defined in Section H.1. The main difference with XZDH is that $\mathcal{Z}$ needs to compute $g^{\Gamma r}$ in order to provide the correct input to the KDF_1 random oracle. If $\overline{I}$ has not issued an `spk-refresh` message since the end of the session, then $\mathcal{Z}$ can send `spk-retrievesecret` from the corrupted $\overline{I}$ to derive $(g^r)^\Gamma$. However, if the signed prekey has been refreshed, then there is no way for $\mathcal{Z}$ to recover Γ from $\bar{\mathcal{G}}^{XZDH}$, thereby satisfying the forward secrecy property.

H.3 Proof of Theorem 6

(Sketch) The proof of Theorem 6 is nearly identical to the proof of Theorem 5 in Appendix G. For this reason, we only highlight the differences in simulator construction between the two sketches. Section H.4 constructs the simulator and Section H.5 demonstrates indistinguishability.

H.4 Simulator Construction

The simulator $\mathcal{S}$ behaves exactly as in Section G.1 except that in the proof for Theorem 6, all references to $\bar{\mathcal{G}}^{\text{ZDH}}_{krkro}$ are replaced by references to $\bar{\mathcal{G}}^{XZDH}$.

H.4.1 Receipt of `solicit` message from $\mathcal{F}^+_{\mathbf{1psp\text{-}keia}}$

When $\mathcal{S}$ receives $(\texttt{solicit}, sid, \overline{I}, \Phi_I)$ from $\mathcal{F}^+_{1psp\text{-}keia}$, it honestly constructs a ψ_1 message from $I^{(s)}$ with the help of the non-information oracle $\mathcal{N}$. $\mathcal{S}$ requests the current signed prekey for $\overline{I}$ by sending $(\texttt{spk-retrieve}, \overline{I})$ to $\bar{\mathcal{G}}^{XZDH}$, and receiving $(\texttt{spk}, \overline{I}, g^\Gamma, \xi)$ in response. $\mathcal{S}$ computes $\psi_1 = \text{“}I\text{”} \parallel g^i \parallel PQ_I \parallel g^\Gamma \parallel \xi$ using the g^i and PQ_I values previously received from $\mathcal{N}$ and sends ψ_1 through $\mathcal{A}$ as if it were broadcast by $I^{(s)}$. $\mathcal{S}$ also records the value Φ_1 for later reference.

H.4.2 Receipt of `establish` message from $\mathcal{F}^+_{\mathbf{1psp\text{-}keia}}$

This case is mostly the same as the case from Section G.1, with the exceptions noted below.

When checking the validity of ψ_1, $\mathcal{S}$ also examines the signed prekey g^Γ with signature ξ from ψ_1. $\mathcal{S}$ retrieves $PK_I = g^I$, the long-term public key for $\overline{I}$, from $\bar{\mathcal{G}}^{XZDH}$ using a `retrieve` message. If $\text{SVerif}(g^I, g^\Gamma, \xi) \neq \text{TRUE}$, then $\mathcal{S}$ sends $(\texttt{abort}, sid)$ to $\mathcal{F}^+_{1psp\text{-}keia}$, delivers the resulting `abort` to $\overline{R}$ immediately, and withholds the `abort` message to $\overline{I}$.

$\mathcal{S}$ also acts slightly differently when constructing ψ_2:

- If $I^{(s)}$ is not corrupt and $\mathcal{S}$ previously created a message ψ_1', but $\psi_1 \neq \psi_1'$, then $\mathcal{S}$ behaves as in Section G.1 with the exception of passing g^Γ to IncProc as part of the `inc` message. IncProc returns a properly constructed ψ_2 message for XZDH.
- Otherwise, $\mathcal{S}$ constructs the message ψ_2 by producing a forged proof σ_R with tag $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^r \| PQ_I \| Q_R \| g^\Gamma \| \Phi_R$. The method for choosing the ephemeral keys (g^r and Q_R) and the keys used to forge the proof depends on the environment:
  - If $\mathcal{S}$ previously created ψ_1 in response to a `solicit` message, then $\mathcal{S}$ behaves as in Section G.1, except that it sends $(\texttt{authenticate}, \{g^\Gamma, g^I\}, t)$ to $\mathcal{N}$ when requesting mac.
  - If ψ_1 was sent by a corrupted $I^{(s)}$, then $\mathcal{S}$ signals to $\mathcal{N}$ that its transcript has been rejected by sending a message $(\texttt{complete}, \text{FALSE}, g^i, PQ_I, \{g^\Gamma, g^I\})$ to $\mathcal{N}$. $\mathcal{S}$ proceeds to acquire r, $SK_I = I$,

and σ_R as in Section G.1. $\mathcal{S}$ computes $\kappa = \mathrm{KDF}_1((g^i)^r \| (g^\Gamma)^r \| (g^I)^r \| Q_k)$, $M_k = \mathrm{KDF}_2(\kappa)$, and then $mac = \mathrm{MAC}(M_k, t)$.

H.4.3 Receipt of ψ_2 by uncorrupted $I^{(s)}$

This case is mostly the same as the case from Section G.1, with two differences. Firstly, when checking that ψ_2 matches the ψ_1 message sent by $I^{(s)}$, $\mathcal{S}$ first retrieves the latest signed prekey for $\overline{I}$ by sending $(\texttt{spk-retrieve}, \overline{I})$ to $\bar{\mathcal{G}}^{XZDH}$, and receiving $(\texttt{spk}, \overline{I}, g^{\Gamma'}, \xi)$ in response. If g^Γ was the signed prekey transmitted in ψ_1 and $g^\Gamma \neq g^{\Gamma'}$, then $\mathcal{S}$ treats ψ_2 as invalid and aborts $\overline{I}$. Secondly, when checking the validity of mac_P, $\mathcal{S}$ sends $(\texttt{verify}, \{g^\Gamma, g^I\}, t, mac_P)$ to $\mathcal{N}$, where tag $t = \text{“}I\text{”} \| \text{“}R\text{”} \| g^i \| g^p \| PQ_I \| Q_P \| g^\Gamma \| \Phi_I$.

H.5 Proof of Indistinguishably

The proof of indistinguishability given in Section G.2 also applies here with several additional remarks.

The session keys produced by $\mathrm{IncProc}_{XZDH}$ and $\mathcal{N}_{ZDH}$ continue to be indistinguishable to $\mathcal{Z}$ due to the CDH assumption for $\mathbb{G}$, since the inputs to KDF_1 still include terms that $\mathcal{Z}$ cannot compute; the inclusion of $g^{\Gamma r}$ does not negate the computational independence of the random oracle's outputs.

Although ψ_1 now contains long-term information—g^Γ and its signature ξ—these values do not help $\mathcal{Z}$ to distinguish between $\mathcal{S}$ and $\mathcal{A}$. $\mathcal{S}$ ensures that $R^{(s)}$ checks the validity of the signature; if ξ is invalid, $\mathcal{S}$ aborts the session in the same manner as a real responder. If $\mathcal{Z}$ replays a signed prekey or generates one from a corrupted $\overline{I}$ via $\bar{\mathcal{G}}^{XZDH}$, $\mathcal{S}$ still effectively simulates $R^{(s)}$ honestly. Replaying the signed prekey or sending a corrupted one also does not affect the security properties of the MAC or proof in ψ_2.

Finally, $\mathcal{S}$ also aborts $\overline{I}$ when ψ_2 includes an outdated g^Γ value (i.e., $\overline{I}$ has refreshed the signed prekey stored by $\bar{\mathcal{G}}^{XZDH}$ after $\mathcal{S}$ simulated ψ_1). This is precisely the behavior of a real initiator, who always knows which signed prekey is being distributed and who refuses to accept responses using an old key. □