Practical Privacy-Preserving Authentication for SSH

Lawrence Roy[*] Stanislav Lyakhov[*] Yeongjin Jang[*] Mike Rosulek[*]

June 9, 2022

Abstract

Public-key authentication in SSH reveals more information about the participants' keys than is necessary. (1) The server can learn a client's entire set of public keys, even keys generated for other servers. (2) The server learns exactly which key the client uses to authenticate, and can further *prove* this fact to a third party. (3) A client can learn whether the server recognizes public keys belonging to other users. Each of these problems lead to tangible privacy violations for SSH users.

In this work we introduce a new public-key authentication method for SSH that reveals essentially the minimum possible amount of information. With our new method, the server learns only whether the client knows the private key for *some* authorized public key. If multiple keys are authorized, the server does not learn which one the client used. The client cannot learn whether the server recognizes public keys belonging to other users. Unlike traditional SSH authentication, our method is fully deniable. Our new method also makes it harder for a malicious server to intercept first-use SSH connections on a large scale.

Our method supports existing SSH keypairs of all standard flavors — RSA, ECDSA, EdDSA. It does not require users to generate new key material. As in traditional SSH authentication, clients and servers can use a mixture of different key flavors in a single authentication session.

We integrated our new authentication method into OpenSSH, and found it to be practical and scalable. For a typical client and server with at most 10 ECDSA/EdDSA keys each, our protocol requires 9 kB of communication and 12.4 ms of latency. Even for a client with 20 keys and server with 100 keys, our protocol requires only 12 kB of communication and 26.7 ms of latency.

1 Introduction

The Secure Shell (SSH) protocol is used by developers for interacting with remote servers, transmitting files, opening secure tunnels, and updating git repositories. The recommended method for authentication in SSH is public-key authentication [SSH22]. This authentication method requires a client to generate keypairs and register the public keys with the server. The server stores, for each user, a list of authorized keys (*e.g.*, ~/.ssh/authorized_keys).

Figure 1a illustrates how public-key authentication works in SSH. The client may have public keys for many servers, so the client advertises its public keys, one at a time. These key advertisements continue until the server recognizes a public key contained in the user's authorized key list. Finally, the client signs a nonce to prove the ownership of the matching private key, and the authentication is successful if the server can verify the signature.

[*]Oregon State University, {royl,lyakhovs,jangye,rosulekm}@oregonstate.edu. First author supported by a DoE CSGF Fellowship.

1.1 Privacy Attacks Against SSH Authentication

Unfortunately, SSH's authentication protocol leaks more information than required for authentication. First, a server can learn all of the client's public keys — even its keys for another server [Val15a, Val15b, Sie16] — allowing the server to fingerprint clients based on their public keys. Second, a client can check if a (username,public_key) pair is valid for authentication, even without knowing the corresponding secret key, allowing the client to probe the server for authorized users. This behavior of SSH was known to the developer in 2002, has been reported in CVE-2016-20012, but not fixed as of May 2022, for 20 years [Nat16].

Third, a server knows which key has been used in authenticating the current session, allowing the server to track a specific user's usage based on their keys, and also prove to third parties that the user authenticated. Fourth, a malicious server can intercept a client's connection, fooling any user who does not carefully check the server's public key fingerprint upon first use. In the following, we give more detail on each of these attacks

Preliminary: building a key-to-id database. It is possible to build a database that partially maps SSH public keys to pseudonyms (*i.e.*, usernames on online services). This is because public services such as Github and Gitlab make all users' SSH public keys available to the general public. For example, the SSH keys used by a Github user torvalds can be publicly accessed via https://github.com/torvalds.keys. This feature is available for the user's convenience, *e.g.*, anyone can easily authorize a user to their SSH server simply by knowing their Github username. Consequently, it is possible to build a database mapping public keys to pseudonyms by enumerating all usernames on the service [NKS^{+}17, Cox15].

Attack 1: Client De-anonymization. A malicious server may obtain a list of all available public keys of the client, then use this list to reveal the client's identity (pseudonyms on public services). Figure 1b illustrates how this attack works. Specifically, the server simply declines all public keys advertised by the client, so that the client eventually offers all of its public keys.[1] This is because the default behavior of the SSH client [Ope22a] is to continue advertising all keys until authentication succeeds.

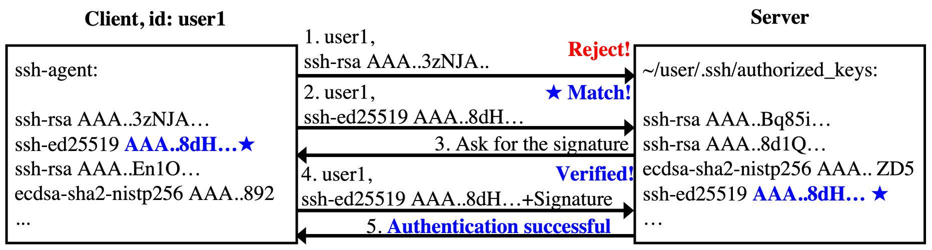
Colluding servers can identify common users, and any server can discover the client's public pseudonyms by consulting a key-to-pseudonym database built from a corpus of publicly available keys [Val15a, Val15b, Sie16].

In particular, Cox [Cox15] built a database containing the public keys of all Github users, using the Github website functionality described above. Later, Valsorda [Val15a] built and publicly deployed a proof-of-concept de-anonymizing SSH server [Val15b], driven from this database. The server would decline every public key offered by the client, until the client exhausted its set of public keys. The server would check the client's keys against the Github key database and print a message containing the client's Github username.

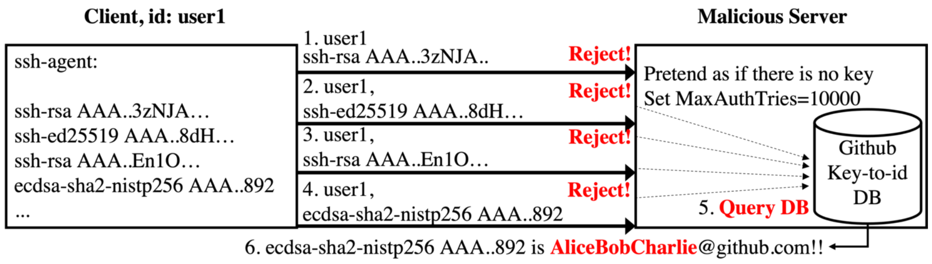
Attack 2: User Probing by the Client. A malicious client can check if a public key is authorized for a username on the server. Such information, in combination with the key-to-pseudonym database, can be used by the client to probe whether a specific user exists on the server. The vulnerability has been acknowledged by comments in the OpenSSH source code [Ope22b] since May of 2002, and assigned CVE-2016-20012, but has not been fixed.

Figure 1c illustrates the attack. In particular, a client may advertise a public key for which it does not know the secret key. The server gives a different response based on whether that key is authorized for the give username.

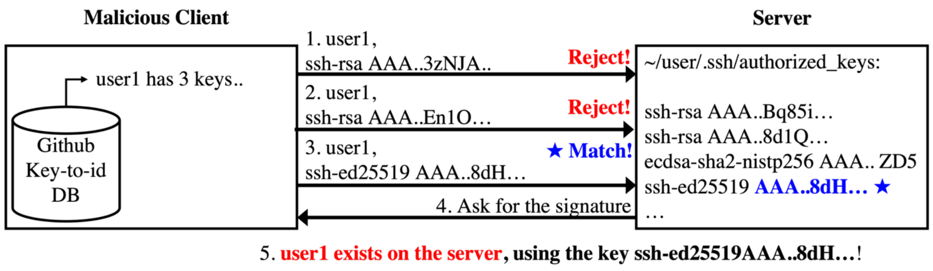
[1] The default behavior of an OpenSSH server considers a rejected public-key advertisement as an authentication failure, and limits the number of such failures to 6 per connection. However, the server can be configured with DEFAULT_AUTH_FAIL_MAX=1000 to ensure that the client can advertise all of its public keys.


Client, id: user1
Server
ssh-agent:
ssh-rsa AAA..3zNJA…
ssh-ed25519 AAA..8dH…★
ssh-rsa AAA..En1O…
ecdsa-sha2-nistp256 AAA..892
...
1. user1, ssh-rsa AAA..3zNJA..
Reject!
2. user1, ssh-ed25519 AAA..8dH…
★ Match!
3. Ask for the signature
4. user1, ssh-ed25519 AAA..8dH…+Signature
Verified!
5. Authentication successful
~/user/.ssh/authorized_keys:
ssh-rsa AAA..Bq85i…
ssh-rsa AAA..8d1Q…
ecdsa-sha2-nistp256 AAA.. ZD5
ssh-ed25519 AAA..8dH… ★
…

(a) Normal Authentication. Client advertises its public keys one by one. When the server recognizes an authorized key, it requests a signature. Authentication succeeds if the signature can be verified.


Client, id: user1
Malicious Server
ssh-agent:
ssh-rsa AAA..3zNJA…
ssh-ed25519 AAA..8dH…
ssh-rsa AAA..En1O…
ecdsa-sha2-nistp256 AAA..892
...
1. user1 ssh-rsa AAA..3zNJA..
Reject!
2. user1, ssh-ed25519 AAA..8dH…
Reject!
3. user1, ssh-rsa AAA..En1O…
Reject!
4. user1, ecdsa-sha2-nistp256 AAA..892
Reject!
Pretend as if there is no key
Set MaxAuthTries=10000
Github Key-to-id DB
5. Query DB
6. ecdsa-sha2-nistp256 AAA..892 is AliceBobCharlie@github.com!!

(b) Attack 1: Client De-anonymization. A malicious server rejects all of the client's public-key advertisement. This causes the client to advertise all of its public keys, under the default client behavior. The server can then use a key-to-id database to identify pseudonyms of the client, *e.g.*, their Github username.


Malicious Client
Server
user1 has 3 keys..
Github Key-to-id DB
1. user1, ssh-rsa AAA..3zNJA..
Reject!
2. user1, ssh-rsa AAA..En1O…
Reject!
3. user1, ssh-ed25519 AAA..8dH…
★ Match!
4. Ask for the signature
~/user/.ssh/authorized_keys:
ssh-rsa AAA..Bq85i…
ssh-rsa AAA..8d1Q…
ecdsa-sha2-nistp256 AAA.. ZD5
ssh-ed25519 AAA..8dH… ★
…
5. user1 exists on the server, using the key ssh-ed25519AAA..8dH…!

(c) Attack 2: User Probing by Client, CVE-2016-20012. A malicious client obtains a victim's username and public-key pair from its key-to-id database. The client guesses or searches for a likely username, then attempts to authenticate to the server by advertising this public-key. The server's response reveals whether that public key is authorized for that username.

Figure 1: Illustration of SSH public-key authentication and attacks. We have not drawn protocol-message arrows for server's rejections of public-key advertisements. Client holds multiple private keys, and the server holds a list of authorized public keys.

Using this basic attack as a primitive, an attacker can reveal the identity of a known username on the server by trying public keys from a database. This attack is especially effective against users who re-use usernames across different services. Additionally, an attacker can often obtain the list of users if the attacker itself has access to the server. In such a case, the attacker may reveal the Github usernames of all accounts on the server.

Attack 3: Tracking and Implicating Users via Key-Usage Patterns. The server knows exactly which public key is used in each successful authentication. A malicious server can use this information to track the usage of individual users or devices based on their keys. As an example, a user may have multiple keys registered for the server under a single username, where each key is associated with a different device (*e.g.*, laptop, desktop, work computer). Then a server can track the usage patterns for specific devices.

Another example is an SSH account shared among an anonymous group. Suppose one would like to build an anonymous group of open source developers that uses git via SSH as their source code repository. Anonymity is not possible in this scenario, since the SSH server learns exactly which key was used for each commit.

Clients authenticate by signing some data under their private key. Signatures are *non-repudiable* meaning that a signature is proof that a *specific user* endorsed a message. The signature produced in an SSH authentication is thus *proof*, verifiable by anyone, that a particular user connected to a server. I.e., a client cannot plausibly deny that it connected to the server. In other online infrastructure (encrypted messaging, email), deniability is understood as a desirable feature, and therefore it is natural to ask whether deniabiilty can be extended to SSH authentication.

Attack 4: Intercepting Connections on First Use. This final attack is an attack on security, not on privacy. Instead of declining every public key offered by the client (as in Attack 1), a malicious server can *accept* every key. If an attacker redirects a client's SSH traffic to such a server, the client will wrongly believe that he/she has connected to a different, desired server. Of course, SSH clients verify the server's public key in order to prevent such an attack. However, a user who follows a *trust on first use (TOFU)* principle may not carefully check the server's public key fingerprint upon the first connection. This leaves the first connection vulnerable to this kind of attack.

1.2 Problem Statement and Goal

The unifying problem in the first three attacks is that a server or a client may obtain more information than is needed for authentication, such as unrelated public keys held by the client, the validity of a username-public-key pair on the server, or the identity of the key (with corresponding proof) used in a successful authentication.

The SSH community and developers are aware of these problems [Val15a, Val15b, Sie16, Ope22b]. These problems remain because blocking these information leaks requires either maintaining site-specific configuration (in the case of Attacks 1 & 2), or fundamentally changing the protocol (Attacks 3 & 4); see §1.3.

Since SSH has become an important part of the Internet infrastructure, it is worth revisiting whether its privacy issues can be completely eliminated. There have been significant advances in cryptographic protocols (specifically, protocols for private set intersection) since SSH was designed. Authentication approaches based on advanced cryptographic techniques — which may have previously seemed far-fetched and prohibitively expensive — may now be truly practical.

What would be the appropriate way to reimagine SSH authentication to resolve these privacy problems? A server should grant access to a client iff *the client holds a secret key corresponding to one of the public keys that the server considers authorized.* If the authentication mechanism reveals more information about the participants' keys than the answer to this question, there is a potential for violating users' privacy.

Our work is motivated by the question:

> *Is it possible for public-key authentication in SSH to reveal only the bare minimum information?*

Our goal is to design an authentication protocol satisfying the following requirements:

Security. The protocol should not reveal information beyond what is strictly required for an authentication decision. Without extra information, three of the aforementioned privacy attacks cannot be carried out. Of course, both clients and servers can learn extra information about each other through other parts of an SSH interaction (*e.g.*, IP address, software version, etc). However, we believe there is no reason for the *public-key authentication mechanism itself* to contribute to privacy violations in SSH, enabling aforementioned attacks.

Drop-In Replacement. Some of the attacks that we consider are *inherent* to the SSH authentication protocol, and can only be fixed by introducing a new authentication protocol. Given this fact, our goal is to mimize the required changes to existing deployments and user experience. Specifically:

1. The protocol should authenticate clients with respect to their *existing* SSH keys — *i.e.*, server/client should not need to change keys or generate new key material to use the new protocol. Client and server can negotiate whether to use the new or old authentication method.
2. Current SSH authentication works seamlessly even when clients and servers hold keys of many different flavors (*e.g.*, RSA, (EC)DSA, EdDSA). The new authentication protocol should also enjoy this property.
3. Clients should be able to benefit from the new protocol without needing to establish and maintain site-specific configuration.

1.3 Existing Mitigations and Their Limitations

We have introduced 4 motivating attacks on SSH authentication. There are several existing techniques to mitigate some of these attacks, which we briefly discuss below. Some of the attacks can be fully mitigated, but only at the cost of site-specific configuration — whereas our proposed protocol addresses all of the privacy problems simultaneously, and "out of the box." Other attacks are more fundamental to the existing SSH authentication protocol and cannot be mitigated without changing the protocol.

Configuration-level fixes. Most SSH clients [Ope22a, PuT22, Mob22] allow users to configure which public keys are advertised to specific sites; this can indeed mitigate Attack 1. This countermeasure requires manual server-specific configuration to be in place before a connection, while our proposed approach completely protects the client's privacy off-the-shelf. Additionally, OpenSSH server has a configurable limit on the number of authentication trials, which is 6 by default (any key advertised by the client counts as an authentication attempt, regardless of whether the server accepts the advertisement). This configuration cannot nullify the attack because the setup is done at the server side. In Attack 1, the malicious party is the server, and can freely change their configuration to launch the attack.

Regarding Attack 2, the SSH protocol allows clients to *optionally* and pre-emptively provide a signature alongside a public key advertisement, rather than advertising a key and proving identity at a later time. In principle, an SSH server could be modified to accept only these kinds of advertisements, so that a client who doesn't know the correct secret key cannot learn whether the server recognizes that key. If both clients and servers employed appropriate configurations (clients advertising only the "correct" keys to a server and including pre-emptive signatures; servers requiring pre-emptive signatures), then Attacks 1 & 2 would be effectively mitigated. To the best of our knowledge, no implementation of SSH provides such a configuration option to the server. In general, pre-emptive signatures have been discouraged in SSH because it requires computational effort (signing) for the client which may be considered wasted when the key is not authorized by the server. Many SSH design decisions were made when RSA and (non-EC) DSA were the only available signature schemes; both of these schemes have expensive signing algorithms. Modern signature schemes based on elliptic curves are several orders of magntidue faster.

We note that our proposed protocol also requires the client to expend effort equivalent to signing under each of its keys. In that sense, our approach would have similar computational cost to the approach where clients & servers modify their configurations as just described. The advantage of our approach would not be in its computational cost, but in the fact that it does not allow anything less than this guarantee of privacy, while requiring no special site-specific configuration for the client, and also addressing the other attacks we consider.

Apart from the impact on Attack 2, if a client provides pre-emptive signatures, the server obtains *non-repudiable proof* that a certain user—even a user of a different service, if the client does not limit its public keys on a per-site basis—has tried to connect. Our proposed approach improves privacy while also providing deniability.

Regarding Attack 3, no amount of client/server configuration can provide client anonymity or deniability (hiding from the server which among the authorized keys was used) since the protocol fundamentally lacks these properties.

As we previously mentioned, clients can prevent Attack 4 by carefully checking the server's public key fingerprint upon first use. Existing SSH authentication can provide no fallback protection to a client who does not verify the server's identity in this way. Looking ahead, our proposed protocol does not eliminate Attack 4, but makes it harder for the adversarial server, even if the client does not verify the server's key fingerprint.

Joint key management. To counter Attack 3, a group of clients can enjoy anonymity by simply sharing a single secret key. However, this is not a viable approach when the authorized users do not know each other's identities. Revocation of a user from the group is also cumbersome under this kind of arrangement.

Prior work on anonymous authentication. Many cryptographic primitives promise a combination of anonymity and authentication. Most notably, *ring signatures* [RST01] and their interactive counterpart *deniable ring authentication* [Nao02] allow a client to prove that it knows the secret key corresponding to *some* public key in a given set of authorized keys, without revealing which key it knows. However, these primitives fundamentally require the client/prover to *know the set of authorized keys*, making them a poor fit for SSH authentication.

Other related primitives like group signatures [Cv91] and anonymous ad-hoc authentication [DKNS04] similarly require the client to know the set of authorized keys. Few methods for "anonymous authentication" also hide the set of authorized keys. One notable exception is a *secret handshake* protocol [BDS+03] (see also [JL07, JKT08, JL09, MPT10]), which hides one party's authentication policy and hides how the other party satisfied the policy. However, authentication policies for secret handshake protocols are expressed in terms of *credentials issued by a known central authority* — not in terms of user-generated keypairs. Furthermore, these protocols all require specialized key material, not simple pre-existing SSH keys. The same limitations are both true of authentication approaches based on attribute-based cryptography [GPSW06, MPR11].

1.4 Our Contributions

Our main result is a practical privacy-preserving public-key authentication method for SSH, with the following features:

Minimum information. Our method leaks *almost* the bare minimum information necessary for authentication. Both parties learn whether the client holds a secret key that corresponds to a public key that the server considers authorized. In addition:

1. The server learns how many keypairs the client has (but not their flavors; *e.g.*, RSA, ECDSA, etc.).
2. The client learns how many public keys of each flavor are authorized (and even less information than this for some flavors).

3. The client learns which of its *valid* keypairs are authorized by the server. I.e., the only way for a client to know whether the server authorizes a public key is by knowing the corresponding secret key.

Compatibility with existing SSH keys. Our method supports all SSH key flavors currently supported by default in OpenSSH: RSA, ECDSA, and EdDSA, which account for 99.7% of SSH keys in use today [Coo21].[2] All parties can use a mixture of key flavors in a single authentication attempt.

Threat model and other security properties. Our security definition considers an adversary who can steal the secret keys of honest users. After doing so, the adversary can of course impersonate the user but all past and future authentication attempts by honest users still reveal only the minimal information described above. This property implies both *forward secrecy* and *deniability* [DDN91, DNS98]. Since the server can simulate its view of the protocol given only the set of authorized keys, the transcript cannot prove anything to an external party. The protocol is also secure against *adaptive* corruptions — *i.e.*, parties can become compromised even during the execution of the authentication protocol.

The server cannot convince the client of a successful authentication unless the server explicitly knows one of the client's public keys. This feature does not completely prevent session interception (as in Attack 4) against a client who does not carefully check the server's key fingerprint upon first use, but it adds a barrier to such an attack. Such an attack can only be targeted to a small number of clients/keys, and not done on a massive scale.

Finally, we prove security in a model where parties can use the same SSH keys for both traditional and privacy-preserving authentication.

Implementation and performance. We built a prototype implementation of our authentication method, as an extension of OpenSSH server/client. Our authentication method is practical and scalable. For a typical client and server, with at most 10 keys each, our protocol requires 9 kB of communication and 12.4 ms of latency for ECDSA/EdDSA keys, or 13 kB of communication and 226 ms of latency for RSA-3072 keys. Even for a client with 20 keys and server with 100 keys, our protocol requires 12 kB of communication and 26.7 ms of latency for ECDSA/EdDSA keys, or 54kB of communication and 300 ms of latency for RSA-3072 keys.

Technical overview. We first introduce a variant of broadcast encryption called **anonymous multi-KEM**. A multi-KEM ciphertext is generated by running $(c, m_1, \ldots, m_n) \leftarrow \mathsf{Enc}(pk_1, \ldots, pk_n)$. Think of the resulting c as a ciphertext addressed to a collection of public keys $pk_1, \ldots, pk_n$, where the owner of pk_i (who knows the matching sk_i) can decrypt c to obtain plaintext m_i. The multi-KEM is *anonymous* if the ciphertext c leaks only the number of recipient public keys, but nothing about their identities.

In our authentication protocol, the server generates a multi-KEM ciphertext c addressed to the set of authorized keys. The client holds a set of secret keys and decrypts c under each one to obtain a set of candidate plaintexts. If one of the client's keys is authorized, then she and the server will now hold a common plaintext. To determine whether this is the case, the parties next run a *private set intersection (PSI)* protocol on their sets of plaintexts. The goal of PSI is for parties to learn the intersection of these sets, but nothing else about these sets. We use a variant of PSI in which the client learns the contents of the intersection — *i.e.*, the client learns which of its keypairs was authorized — while the server learns only whether the intersection was nonempty.

We show how to construct a single anonymous multi-KEM scheme that simultaneously supports all standard SSH key flavors: RSA, (EC)DSA, and EdDSA. We also show how to modify the leading

[2]The other 0.3% of keys are (non-EC) DSA, which our methods can easily support, but which is now deprecated in OpenSSH.

PSI protocol of Rosulek & Trieu [RT21] to allow the server to learn (only) whether the intersection is nonempty.

1.5 Other Related Work

PSI Variants. Our protocol is a kind of private set intersection (PSI) where the client cannot include *pk* in its set without also knowing the corresponding *sk*. A closely related PSI variant is authorized PSI (APSI) [DJKT09, DKT10, DT10], where the client cannot include m in its set without also knowing a signature on m from a certificate authority.

In APSI, the protocol implicitly verifies signatures on the client's items, but all of these signatures are with respect to a *single verification key* (belonging to the certificate authority) that all parties know. In the case of RSA signatures, the APSI protocol can take advantage of the algebraic structure of the certificate authority's RSA modulus. Our setting is quite different, since the protocol must authenticate potentially many RSA keys held by the client, each with different moduli *that the server doesn't even know*, since they are part of the client's private input.

In our protocol, the client proves a non-empty intersection by using a PSI where each item has an associated payload. Posession of this payload serves as proof of the non-empty intersection. The idea of associating PSI items with payloads is common (*e.g.*, [FIPR05, DT10]) and has even been used previously as a means of authentication [ZC17]. Our specific combination of MKEM and PSI to authenticate with respect to a set of public keys is novel, to the best of our knowledge.

Multi-Encryption and Broadcast Encryption. In broadcast encryption, a sender addresses a single ciphertext to an ad-hoc group of public keys. Broadcast encryption was first studied in [Kur02, BBS03], where it was observed that there exist techniques that are more efficient than simply encrypting separately to each receiver. Much of subsequent work on broadcast encryption involves other features (*e.g.*, revocation, traitor-tracing) that are orthogonal to our needs.

We use a simple variant of broadcast encryption that we call multi-KEM. Multi-KEMs appear implicitly in most constructions of broadcast encryption, but as a high-level technique and not a well-defined primitive. We require the multi-KEM to be anonymous [LPQ12] (sometimes called key-private [BBW06]), meaning that the ciphertext hides the set of recipients. We require a weaker confidentiality property (infeasibility of total plaintext recovery) than is standard for broadcast encryption, leading to simpler constructions.

One important technique we use in our multi-KEM construction is encoding RSA ciphertexts as outputs of a polynomial; this technique was used previously in constructions of broadcast encryption in [FHH10, ZLZL15].

2 Preliminaries

Definition 1. *Let G generate a cyclic group $\mathbb{G}$ of order ℓ. The* ***gap computational Diffie–Hellman (GapCDH)*** *assumption [OP01] for G states that it is computationally hard to find G^{ab} from G^a and G^b, even with an oracle for solving the decisional Diffie–Hellman problem. More precisely, every PPT adversary $\mathcal{A}$ has negligible probability to win the game:*

```
$a, b \leftarrow [0, \ell) \cap \mathbb{Z}$
$\underline{\textsc{guess}(X \in \mathbb{G}, Y \in \mathbb{G}, Z \in \mathbb{G})}$:
  return $\mathrm{dlog}_G(X) \cdot \mathrm{dlog}_G(Y) \overset{?}{=} \mathrm{dlog}_G(Z) \bmod \ell$
$\textbf{win}$ if $\mathcal{A}^{\textsc{guess}(\cdot)}(G^a, G^b) = G^{ab}$
```

2.1 Signatures

Definition 2. *A **signature scheme** is a collection* SS *of PPT algorithms*

$$\begin{aligned}(\mathsf{pk},\mathsf{sk}) &\leftarrow \mathsf{SS.Gen}(\mathsf{opts})\\ s &\leftarrow \mathsf{SS.Sign}(\mathsf{sk}, m)\\ v &:= \mathsf{SS.Verify}(\mathsf{pk}, m, s)\end{aligned}$$

for $\mathsf{opts} \in \mathsf{SS.OPTS}$*,* $\mathsf{pk}, \mathsf{sk}, m, s \in \{0,1\}^*$*, and* $v \in \{0,1\}$*, satisfying correctness: when these algorithms are executed as above,* $v = 1$ *except with negligible probability.*

Definition 3. *A signature scheme* SS *satisfies **existential unforgeability under chosen message attacks (EUF-CMA)** if for all* $\mathsf{opts} \in \mathsf{SS.OPTS}$*, every PPT adversary* $\mathcal{A}$ *has negligible probability of winning the game:*

```
$M := \{\}$
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{SS.Gen}(\mathsf{opts})$
$\underline{\textsc{sign}(m)}$:
  $M := M \cup \{m\}$
  return $\mathsf{SS.Sign}(\mathsf{sk}^*, m)$
$(m, s) \leftarrow \mathcal{A}^{\textsc{sign}(\cdot)}(\mathsf{pk}^*)$
**win** if $m \notin M \wedge \mathsf{SS.Verify}(\mathsf{pk}^*, m, s)$
```

3 Anonymous Multi-KEM

In this section we introduce our encryption abstraction, called a *multi-KEM.* Multi-KEM allows a sender to generate a ciphertext c addressed to a set of public keys. Each corresponding secret key may decrypt c to a different value. The sender does not need to choose these values, but she learns them when encrypting, as in a typical KEM.

Definition 4. *A **multi-KEM (MKEM)** is a collection* MKEM *of PPT algorithms*

$$\begin{aligned}(\mathsf{pk},\mathsf{sk}) &\leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})\\ (c, r) &\leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\})\\ m &:= \mathsf{MKEM.Msg}(\mathsf{pk}, r)\\ m' &:= \mathsf{MKEM.Dec}(\mathsf{sk}, c)\end{aligned}$$

for $\mathsf{opts} \in \mathsf{MKEM.OPTS}$ *and* $\mathsf{pk}, \mathsf{sk}, c, r, m, m' \in \{0,1\}^*$*, satisfying correctness: no adversary can pick public keys to make decryption fail for an honestly generated key. I.e., for all* $\mathsf{opts} \in \mathsf{MKEM.OPTS}$*, every PPT* $\mathcal{A}$ *has negligible probability of winning the game:*

```
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
$\mathsf{PK} \leftarrow \mathcal{A}(\mathsf{pk})$
$(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}\} \cup \mathsf{PK})$
**win** if $\mathsf{MKEM.Msg}(\mathsf{pk}, r) \neq \mathsf{MKEM.Dec}(\mathsf{sk}, c)$
```

Note that instead of having Enc output the set of plaintext values, we have Enc output some state r, which the sender can further use to determine one receiver's output via $\mathsf{Msg}(\mathsf{pk}, r)$. This choice of syntax simplifies some parts of our protocol.

We require a relatively mild security definition for a MKEM. In our eventual protocol, MKEM plaintexts are used only as inputs to a private set intersection (PSI) protocol. The PSI protocol exposes to the adversary an oracle for verifying guesses of MKEM plaintexts — *i.e.*, the adversary learns no more than whether one of its PSI inputs (guesses) is equal to one of the honest party's MKEM plaintexts. Hence, our security definition requires that *total plaintext recovery* is infeasible, even in the presence of oracles for verifying guesses of plaintexts (from either $\mathsf{MKEM.Dec}$ or $\mathsf{MKEM.Msg}$). We call this security notion **weak chosen ciphertext attack (wCCA)** security.

Definition 5. *A multi-KEM* MKEM *is **secure against weak chosen ciphertext attacks (wCCA)** if for all* $\mathsf{opts} \in \mathsf{MKEM.OPTS}$*, every PPT adversary* $\mathcal{A}$ *has negligible probability of winning the game:*

```
$R :=$ empty
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$

$\underline{\text{ENCRYPT}(\mathsf{PK})\text{:}}$
  $(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}^*\} \cup \mathsf{PK})$
  $R[c] := r$
  return $c$

$\underline{\text{GUESS\_DEC}(c, m)\text{:}}$
  return $\mathsf{MKEM.Dec}(\mathsf{sk}^*, c) \overset{?}{=} m$

$\underline{\text{GUESS\_MSG}(c, \mathsf{pk}, m)\text{:}}$
  if $R[c]$ defined:
    return $\mathsf{MKEM.Msg}(\mathsf{pk}, R[c]) \overset{?}{=} m$

$(c, m) \leftarrow \mathcal{A}^{\text{ENCRYPT,GUESS\_DEC,GUESS\_MSG}}(\mathsf{pk}^*)$
**win** if $R[c]$ defined $\wedge\ \mathsf{MKEM.Dec}(\mathsf{sk}^*, c) = m$
```

Note that adversarially chosen public keys can be input to GUESS_MSG. This models an attack scenario for the eventual protocol, where the adversary may create a public key related to an honest user's key, rather than generating them honestly. Such related public keys may have related MKEM plaintexts. However, including GUESS_MSG in this game guarantees that that checking guesses of these related plaintexts will not be useful for attacking the protocol.

We additionally require that MKEM ciphertexts leak a minimal amount about the set of recipient keys. The nature of the leakage varies by scheme, so we let the leakage function be a parameter of an MKEM scheme. The leakage function parameterizes what a MKEM ciphertext reveals about the *honestly-generated* recipient keys, while we assume that the ciphertext can leak arbitrary information about adversarially chosen keys. The bound on leakage holds even to adversaries who know the secret keys of all honestly generated keypairs, and learn the sender's state value r:

Definition 6. MKEM *is **anonymous except for leakage*** $\mathsf{MKEM.Leak}$ *if there is a PPT simulator* $(\mathsf{AnonSim}, \mathsf{AnonView})$ *such that the following oracles are indistinguishable.*

```
$\mathsf{PK}^* := \{\}$

GENERATE(opts):
    $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
    $\mathsf{PK}^* := \mathsf{PK}^* \cup \{\mathsf{pk}\}$
    return $(\mathsf{pk}, \mathsf{sk})$

ENCRYPT(PK):
    $(c, r) \leftarrow \mathsf{MKEM.Enc}(\mathsf{PK})$
    for $\mathsf{pk} \in \mathsf{PK} \setminus \mathsf{PK}^*$:
        $M[\mathsf{pk}] := \mathsf{MKEM.Dec}(r, \mathsf{pk})$
    return $(c, r, M)$
```

```
$\mathsf{PK}^* := \{\}$
$\mathsf{SK} :=$ empty

GENERATE(opts):
    $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
    $\mathsf{PK}^* := \mathsf{PK}^* \cup \{\mathsf{pk}\}$
    $\mathsf{SK}[\mathsf{pk}] = \mathsf{sk}$
    return $(\mathsf{pk}, \mathsf{sk})$

ENCRYPT(PK):
    $L := \mathsf{MKEM.Leak}(\mathsf{PK})$
    $(c, M, v) \leftarrow \mathsf{AnonSim}(L, \mathsf{PK} \setminus \mathsf{PK}^*)$
    $S := \{\mathsf{SK}[\mathsf{pk}] \mid \mathsf{pk} \in \mathsf{PK} \cap \mathsf{PK}^*\}$
    $r \leftarrow \mathsf{AnonView}(v, S)$
    return $(c, r, M)$
```

3.1 Joint Security

Existing SSH keypairs are essentially signing keys, but our new authentication method requires us to treat them as MKEM keys. In order for the existing uses of these SSH keys to remain valid, we must consider *joint* security of an MKEM and signature scheme using the same keypair.

Definition 7. MKEM *is a **jointly secure multi-KEM and signature scheme (MKEMSS)** if it satisfies both correctness definitions (with the same* Gen*), and is both EUF-CMA and wCCA secure when the adversary is given the oracles from both of those games simultaneously. Formally, every PPT adversary has negligible chance of winning the game:*

```
$R, M := \{\}$
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
// ENCRYPT, GUESS_DEC, GUESS_MSG as in Definition 5
// SIGN as in Definition 3
$(c, m, \sigma) \leftarrow \mathcal{A}^{\text{ENCRYPT,GUESS\_DEC,GUESS\_MSG,SIGN}}(\mathsf{pk}^*)$
win if $[R[c] \text{ defined} \wedge \mathsf{MKEM.Dec}(\mathsf{sk}^*, c) = m]$
    $\vee [m \notin M \wedge \mathsf{MKEM.Verify}(\mathsf{pk}^*, m, \sigma)]$
```

3.2 Instantiations

We describe MKEMSS constructions for the standard SSH key flavors: EdDSA, (EC)DSA, and RSA.

3.2.1 EdDSA

EdDSA [BDL+12] is a particular way of instantiating Schnorr signatures over twisted Edwards curves such as Ed25519. Let G be a point on elliptic curve E that generates a subgroup $\mathbb{G}$ of prime order ℓ. Let f be the cofactor of the curve, and let $M \subseteq f\mathbb{Z}$ the set of exponents (to clear cofactors).

The nonce r is chosen deterministically in EdDSA by evaluating a PRF, $\mathsf{F}\colon \{0,1\}^{2\lambda} \times \{0,1\}^* \to \mathbb{Z}/\ell\mathbb{Z}$. The PRF key h is part of the private key.[3] The Schnorr challenge comes from a random oracle $H\colon E \times E \times \{0,1\}^* \to \mathbb{Z}/\ell\mathbb{Z}$.

[3]Implementations compress the two parts of the private key using a PRG.

```
EdDSA.Gen():
    $a \leftarrow M$
    $h \leftarrow \{0,1\}^{2\lambda}$
    return $(G^a, (a,h))$

EdDSA.Verify$(A, m, (R, s))$:
    return $G^s \overset{?}{=} R + A^{H(R,A,m)}$

EdDSA.Sign$((a,h), m)$:
    $r := \mathsf{F}(h, m)$
    $R := G^r$
    $s := (r + H(R, A, m)a) \bmod \ell$
    return $(R, s)$
```

The corresponding multi-KEM is based on elliptic curve Diffie–Hellman, reusing a single ECDH message for all public keys. Since Enc does not depend on the public keys at all, it trivially satisfies the anonymity definition with no leakage.

```
EdDSA.Enc(PK):
    $r \leftarrow M$
    return $(G^r, r)$

EdDSA.Msg(pk, $r$):
    return $\mathsf{pk}^r$

EdDSA.Dec$((a,h), C)$:
    return $C^a$
```

In Section A.1 we prove the joint security of EdDSA under the GapCDH assumption, using a variant of the well-known proof for Schnorr signatures. A similar proof of joint security for Schnorr and Diffie–Hellman was given in [DLP+12].

Lemma 8. *Any attack $\mathcal{A}$ against the joint security of the MKEMSS* EdDSA *implies an attack $\mathcal{A}'$ against the GapCDH problem. $\mathcal{A}'$ takes approximately twice the computation of $\mathcal{A}$, and*

$$\mathrm{Adv}[\mathcal{A}] \leq \sqrt{q_H\left(\frac{\mathrm{Adv}[\mathcal{A}']}{P^2} + \frac{1}{\ell}\right)} + \frac{q_H q_S}{\ell},$$

where $\mathcal{A}$ makes q_H queries to the random oracle H and requests q_S signatures.

The slack in the concrete security bound is common to security proofs for Schnorr signatures based on the forking-lemma, and can typically be improved by an analysis in the stronger generic group model (GGM) [NSW09].

3.2.2 ECDSA

ECDSA is another signature scheme based on ECC, and hence our multi-KEM for ECDH is essentially the same as the one for EdDSA. Let E, G, and ℓ be the same as above.

```
ECDSA.Gen():
    $a \leftarrow [1, \ell) \cap \mathbb{Z}$
    return $(G^a, a)$

ECDSA.Enc(PK):
    $r \leftarrow [1, \ell) \cap \mathbb{Z}$
    return $(G^r, r)$

ECDSA.Msg$(A, r)$:
    return $A^r$

ECDSA.Dec$(a, C)$:
    return $C^a$

ECDSA.Sign$(a, m)$:
    $k \leftarrow [1, \ell) \cap \mathbb{Z}$
    $r := (G^k)_x \bmod \ell$
    $s := \frac{H(m)+ra}{k} \bmod \ell$
    return $(r, s)$

ECDSA.Verify$(A, m, (r, s))$:
    if $0 \equiv rs \bmod \ell$:
        return 0
    return $r \overset{?}{=} \left(G^{\frac{H(m)}{s}} A^{\frac{r}{s}}\right)_x$
```

Unfortunately, all known proofs of ECDSA's security depend on highly idealized assumptions. Specifically, the conversion operation $(R)_x$ that gets the x-coordinate of a curve point has to be idealized [FKP16]. Brown [Bro02] proved security in the Generic Group Model (GGM); a generic group does not have meaningful x-coordinates, so this implicitly turns $(R)_x$ into a random oracle. Later, Fersch et al. [FKP16] proved security using only an idealized model for $(R)_x$, without the GGM.

A similar joint encryption and signature scheme was proven in the GGM [DLP+12]. We adapt their result to our scheme (proof in Section A.2):

Lemma 9. *If H is collision resistant and zero-finder-resistant, then* ECDSA *is a jointly secure MKEMSS in the GGM.*

3.2.3 RSA

There are several methods for sampling RSA keypairs, and we let the opts argument to Gen specify the method of choice. SSH keypairs use RSASSA-PKCS1-v1_5 signatures [MKJR16], outlined below. To encode the message to be signed, it uses a padding scheme, $\mathsf{PKCS}_N \colon \{0,1\}^* \to \mathbb{Z}/N\mathbb{Z}$, the details of which are unimportant for our purpose.

```
$\underline{\mathsf{RSA.Sign}((N,e,d),m)}$:
    return $\mathsf{PKCS}_N(m)^d$

$\underline{\mathsf{RSA.Verify}((N,e),m,s)}$:
    return $s^e \overset{?}{=} \mathsf{PKCS}_N(m)$
```

It is trivial to construct a KEM for a *single* recipient by simply using bare RSA as a trapdoor function. Padding is both undesirable for anonymity, and unnecessary since the plaintext is uniformly random in $\mathbb{Z}/N\mathbb{Z}$.

```
$\underline{\mathsf{RSA.Enc1}((N,e))}$:
    $r \leftarrow \mathbb{Z}/N\mathbb{Z}$
    return $(r^e \bmod N, r)$

$\underline{\mathsf{RSA.Dec1}((N,e,d),c)}$:
    return $c^d \bmod N$
```

Constructing an anonymous *multi-KEM* is non-trivial. Unlike the Diffie–Hellman approach which works for ECC keys, RSA encryptions depend on the public key, so a multi-KEM must generate separate ciphertexts for each recipient. This creates two problems for anonymity: an individual RSA ciphertext leaks some information about its public key N, since it is a number in $[0, N)$, and RSA.Dec must somehow be told which ciphertext to decrypt for which keypair. We solve the first problem by encoding the ciphertext into an (approximately) uniformly random integer $c' \in [0, 2^{2\lambda\, s(N)})$, by adding some padding p', which is a random multiple of N below $2^{2\lambda\, s(N)}$. Here, $s(N) = \left\lceil \frac{\ell(N)+\lambda}{2\lambda} \right\rceil$ is chosen to lengthen c' enough to be almost uniform, while padding it to be a multiple of 2λ bits long, and $\ell(N)$ is the size of the public key N, so $2^{\ell(N)-1} < N < 2^{\ell(N)}$.

To handle the second problem, we encode the public-key-to-ciphertext mapping in a polynomial. Essentially, the sender generates a polynomial C such that $C(\mathsf{pk}) = c'$ for each key pk and associated ciphertext c'. The coefficients of the polynomial leak nothing about the pk's if the c' values are jointly pseudorandom. However, this would require a very large field since RSA keys and ciphertexts are rather large. Instead, our Multi-KEM sender divides c' into chunks $c_0, \ldots, c_{s(N)-1}$, each of size 2λ bits. She then encodes a polynomial $C(x)$ such that $C(H(\mathsf{pk}, i)) = c_i$ for each chunk c_i, where H is a collision-resistant hash. Dec then evaluates this polynomial at $H(\mathsf{pk}, i)$ for each i, combines the chunks into a ciphertext c', and then decrypts it. We set the chunk size to 2λ bits because H needs to be a collision resistant hash. The result is polynomial operations in a field $\mathbb{F}$ of order very close to $2^{2\lambda}$.

Interpolation of a degree-n polynomial requires $\Theta(n \log^2 n)$ field operations. Instead of a polynomial, it is possible to use any *oblivious key-value store (OKVS)* [GPR+21], a generalization of polynomials. There exist more asymptotically efficient OKVS constructions, but we found simple polynomial interpolation to be sufficiently fast for the small set sizes in our setting.

```
$\underline{\mathsf{RSA.Enc}(\mathsf{PK})}$:
  $S := \{\}$
  $R :=$ empty map
  for $(N, e) \in \mathsf{PK}$:
    $c, r \leftarrow \mathsf{RSA.Enc1}((N, e))$
    $R[(N, e)] := r$
    $c_0, \ldots, c_{s(N)-1} \leftarrow \mathsf{Chk}_N(c)$
    for $i := 0$ to $s(N) - 1$:
      $S := S \cup \{(H(N, e, i), c_i)\}$
  return $\mathsf{interpol}_{\mathbb{F}}(S), R$

$\underline{\mathsf{Chk}_N(c)}$:
  $p \leftarrow [0, 2^{2\lambda\, s(N)}) \cap \mathbb{Z}$
  $p' := p - (p \bmod N)$
  $c' := p' + c$
  for $i := 0$ to $s(N) - 1$:
    $c_i := c' \bmod 2^{2\lambda}$
    $c' := \lfloor c'/2^{2\lambda} \rfloor$
  return $c_0, \ldots, c_{s(N)-1}$

$\underline{\mathsf{RSA.Msg}(\mathsf{pk}, R)}$:
  return $R[\mathsf{pk}]$

$\underline{\mathsf{RSA.Dec}(\mathsf{sk}, C)}$:
  $N, e, d := \mathsf{sk}$
  for $i := 0$ to $s(N) - 1$:
    $c_i := C(H(N, e, i))$
  $c := \mathsf{Unchk}_N(\{c_i\}_i)$
  $m := \mathsf{RSA.Dec1}(\mathsf{sk}, c)$
  return $m$

$\underline{\mathsf{Unchk}_N(c_0, \ldots, c_{s(N)-1})}$:
  $c := \sum_{i=0}^{s(N)-1} 2^{2\lambda i} c_i$
  return $c \bmod N$
```

PKCS signatures lack a security reduction to the RSA assumption, so we cannot prove the joint security of RSA based on $(\cdot)^e \bmod N$ being one-way. However, we can do the next best thing: prove joint security under the assumption that the signature scheme is secure.

Lemma 10. *The EUF-CMA security of the* RSA *signature scheme implies that* RSA *is a jointly secure MKEMSS.*

Recall that joint security requires that the scheme satisfy weak-CCA security (Definition 5). In particular, it should be hard to guess the decapsulation of a KEM ciphertext, even given an oracle for checking such guesses. In the case of RSA, the adversary already has the ability to test whether a guess is correct: To test whether $m = c^d = \mathsf{RSA.Dec1}((N, e, d), c)$ for some guess m, the adversary can simply test whether $m^e = c$, using only public information. This algebraic property of RSA renders the GUESS_DEC and GUESS_MSG oracles redundant, and greatly simplifies the security proof compared to the Diffie-Hellman-based MKEMs. The proof details are defered to the full version.

Finally, we need to show anonymity with respect to a leakage function. For properly generated public keys, Chk_N will produce uniformly random chunks, so C will be a uniformly random polynomial with degree less than $s(\mathsf{PK})$, where $s(\mathsf{PK})$ is the sum of $s(N)$ for all the public keys in PK. That is, only the *combined length of all public keys* needs to be leaked.[4]

Lemma 11. RSA *is an anonymous MKEM with respect to leakage* $\mathsf{RSA.Leak}(\mathsf{PK}) = s(\mathsf{PK})$.

Both proofs for the RSA MKEM are given in in Section A.3.

3.2.4 Mixing Key Flavors

SSH allows users to authenticate themselves with many different keypair flavors. To achieve the same property, our authentication protocol requires a *single* multi-KEM where encryptions can be

[4]Adversarial public keys can be malformed so that RSA.Enc1 does not generate a uniformly random element of $\mathbb{Z}/N\mathbb{Z}$, e.g. by picking an e that is not coprime to $\lambda(N)$. Recall that MKEM ciphertexts need not hide anything about adversarially generated keys.

addressed to a mixture of different key flavors. We build such a multi-flavor MKEM by simply concatenating a separate MKEM ciphertext for each key flavor.

The mixed-flavor multi-KEM (which we call MixKEM) is parameterized by a set $\mathsf{FLAVORS}$ of supported key flavors. The key generation of MixKEM expects a particular flavor as one of its options, and keys in the MixKEM scheme are of the form $(\mathsf{f}, \mathsf{pk})$ where pk is a key of flavor f.

$$\mathsf{MixKEM.OPTS} = \left\{ (\mathsf{f}, \mathsf{opts}) \;\middle|\; \begin{array}{l} \mathsf{f} \in \mathsf{FLAVORS}, \\ \mathsf{opts} \in \mathsf{f.OPTS} \end{array} \right\}$$

```
MixKEM.Gen((f, opts)):
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{f.Gen}(\mathsf{opts})$
  return $(\mathsf{f}, \mathsf{pk}), (\mathsf{f}, \mathsf{sk})$

MixKEM.Sign((f, sk), m):
  return $\mathsf{f.Sign}(\mathsf{sk}, m)$

MixKEM.Verify((f, pk), m, s):
  return $\mathsf{f.Verify}(\mathsf{pk}, m, s)$

MixKEM.Dec((f, sk), C):
  if $C[\mathsf{f}]$ undefined:
    return $\bot$
  return $\mathsf{f.Dec}(\mathsf{sk}, C[\mathsf{f}])$

MixKEM.Enc(PK):
  $F := \{\mathsf{f} \mid (\mathsf{f}, \mathsf{pk}) \in \mathsf{PK}\}$
  $C, R :=$ empty map
  for $\mathsf{f} \in F$:
    $\mathsf{PK}_\mathsf{f} := \{\mathsf{pk} \mid (\mathsf{f}, \mathsf{pk}) \in \mathsf{PK}\}$
    $c, r \leftarrow \mathsf{f.Enc}(\mathsf{PK}_\mathsf{f})$
    $C[\mathsf{f}] := c$
    $R[\mathsf{f}] := r$
  return $C, R$

MixKEM.Msg((f, pk), R):
  return $\mathsf{f.Msg}(\mathsf{pk}, R[\mathsf{f}])$
```

Regarding anonymity, we must characterize what information $\mathsf{MixKEM.Enc}(\mathsf{PK})$ leaks about the public keys in PK. Let F and PK_f be defined as in $\mathsf{MixKEM.Enc}$. Clearly, $\mathsf{MixKEM.Enc}(\mathsf{PK})$ leaks F (the set of flavors present), and it also leaks any information from each flavor's $\mathsf{f.Enc}(\mathsf{PK}_\mathsf{f})$. Therefore, the leakage function for MixKEM is $\mathsf{MixKEM.Leak}(\mathsf{PK}) = \{(\mathsf{f}, \mathsf{f.Leak}(\mathsf{PK}_\mathsf{f})) \mid \mathsf{f} \in F\}$.

In Section A.4 we prove the following:

Lemma 12. MixKEM *is a jointly secure MKEMSS if every flavor in* $\mathsf{FLAVORS}$ *is.*

Lemma 13. MixKEM *is anonymous, assuming that every* $\mathsf{f} \in \mathsf{FLAVORS}$ *is, with advantage is bounded by the total advantage against all the individual flavors' anonymities.*

MixKEM is subject to some tradeoffs between efficiency and leakage. For example, $\mathsf{MixKEM.Enc}$ could be made to always generate ciphertexts for some set of commonly used flavors, thereby not leaking whether they are present in PK. Key flavors beyond RSA (including EdDSA and ECDSA) could be encoded into a single polynomial,[5] which would leak no more than the total size of all ciphertexts. Our choice of MixKEM was motivated largely by simplicity. Finally, note that ECDSA keys can be instantiated over a variety of different curves, and each curve correpsonds to a different MKEM flavor.

4 Security Definition

We present our formal security definition in the form of an ideal functionality in the UC framework, in Figure 2. The functionality is somewhat complicated and subtle, so we provide intuitive explanations of its main features below.

Keys. The functionality's genkey command generates and logs a keypair to model the local process of key generation by honest parties. We consider an adversary who is capable of stealing honest users' secret keys; this is modeled by the functionality's stealkeys command.

[5] Encoding into a polynomial requires the ciphertexts to be pseudorandom bit strings. This could be achieved for EC-based schemes, *e.g.*, with the Elligator [BHKL13] technique.

```
Parameters:
  • Parties $P_1, P_2, \dots$
  • A signature scheme $\mathsf{SS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.
  • Function $\mathcal{L}$ characterizing leakage on server's set.

Static variables:
  • Sets $\Sigma$ and $\mathsf{Secure}$; associative arrays $\mathsf{SK}_1, \mathsf{SK}_2, \dots$

Define predicate:
  $$\text{can\_use}(P_i, \mathsf{pk}) = \begin{cases} \mathsf{SK}_i[\mathsf{pk}] \text{ defined}, & P_i \text{ honest} \\ \mathsf{pk} \notin \mathsf{Secure}, & P_i \text{ corrupt} \end{cases}$$

On input $(\text{genkey}, \mathsf{opts})$ from party $P_i$:
  1. Do $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(\mathsf{opts})$ and set $\mathsf{SK}_i[\mathsf{pk}] := \mathsf{sk}$.
  2. If $P_i$ is honest: add $\mathsf{pk}$ to $\mathsf{Secure}$.
  3. Give $\mathsf{pk}$ to $P_i$.

On input $(\text{get\_sk}, \mathsf{pk})$ from the adversary:
  4. If $\mathsf{SK}_i[\mathsf{pk}]$ is defined for some $i$, then give that $\mathsf{SK}_i[\mathsf{pk}]$ to the adversary.

// simulator can send this command only when the real-world
// adversary compromises $P_i$'s storage
On input $(\text{stealkeys}, P_i)$ from the adversary:
  5. Set $\mathsf{Secure} := \mathsf{Secure} \setminus \{\mathsf{pk} \mid \mathsf{SK}_i[\mathsf{pk}] \text{ defined}\}$.
  6. Give $\mathsf{SK}_i$ to the adversary.

On input $(\text{sign}, \mathsf{pk}, m)$ from $P_i$:
  7. If $\neg\text{can\_use}(P_i, \mathsf{pk})$: abort.
  8. Add $(\mathsf{pk}, m)$ to $\Sigma$.
  9. Give $\mathsf{Sign}(\mathsf{SK}_i[\mathsf{pk}], m)$ to $P_i$.

On input $(\text{verify}, \mathsf{pk}, m, \sigma)$ from any party:
  10. If $\mathsf{pk} \in \mathsf{Secure}$ and $(\mathsf{pk}, m) \notin \Sigma$: respond false.
  11. Otherwise respond with $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$.

// authentication attempt between server $P_\mathsf{S}$ & client $P_\mathsf{C}$
On input $(\text{auth}_1, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K_\mathsf{S})$ from $P_\mathsf{S}$:
  12. If $P_\mathsf{C}$ is corrupt, give leakage to the adversary:
      $$\Big(\mathcal{L}(K_\mathsf{S}),\ \{\mathsf{pk} \in K_\mathsf{S} \mid \forall i : \mathsf{SK}_i[\mathsf{pk}] \text{ undefined}\}\Big).$$
  13. Wait for command $(\text{auth}_2, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K_\mathsf{C})$ from $P_\mathsf{C}$.
  14. Give $|K_\mathsf{C}|$ to $P_\mathsf{S}$.
  15. Wait for command $(\text{auth}_3, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K'_\mathsf{S})$ from $P_\mathsf{S}$.
  16. If $P_\mathsf{S}$ is corrupt: set $K_\mathsf{S} := K'_\mathsf{S}$ (otherwise ignore $K'_\mathsf{S}$).
  17. Compute $A := K_\mathsf{S} \cap K_\mathsf{C} \cap \{\mathsf{pk} \mid \text{can\_use}(P_\mathsf{C}, \mathsf{pk})\}$.
  18. Give $(A, |K_\mathsf{S}|)$ to $P_\mathsf{C}$.
  19. Wait for command $(\text{deliver}, \mathit{ssid}, d \in \{0,1\})$ from $P_\mathsf{C}$.
  20. Give $d \wedge [A \neq \emptyset]$ to $P_\mathsf{S}$.
```

Figure 2: Ideal functionality $\mathcal{F}_{\mathsf{new\text{-}auth}}$ defining the security of our new public-key authentication method.

Keys can be classified into 3 categories with respect to the ideal functionality: (1) A key generated by an honest user is initially considered **secure** and stored in the set Secure (line 2). (2) A secure key becomes **stolen** when the adversary calls the stealkeys command on the owner of that key. (3) Parties can invoke the functionality's commands on keys that were not generated by honest parties. We call such keys as **unregistered**, and they are treated as adversarially generated.

The functionality uses a predicate can_use to decide whether a user is allowed to use a key for authentication or signing.

- Honest users can only use keys that they generated honestly, regardless of whether they are *secure* or *stolen*.
- Corrupt users can only use *stolen* or *unregistered* keys, but not *secure* keys.

In our security proof, we restrict our focus to simulators that call stealkeys *only* when the real-world adversary compromises a party's actual key storage. Hence stealkeys in the ideal world captures key compromise in the real-world, and stealkeys is the only way for an adversary to gain an advantage in the real world, with respect to the can_use predicate. In the *ideal* world, knowledge of the sk values offers no advantage to an adversary. We therefore allow the ideal-world simulator to learn these sk values (via the get_sk command), which is helpful in our security proof. Again, we emphasize that giving all sk values to the ideal-world adversary does not help that adversary authenticate or forge signatures under more keys, if they don't also send a stealkeys command.

Authentication. A server P_{S} and client P_{C} can perform an authentication session using a sequence of auth commands. Each party provides a set of public keys: $K_{\mathsf{S}}, K_{\mathsf{C}}$ respectively. The client learns the intersection $A = K_{\mathsf{S}} \cap K_{\mathsf{C}}$ (line 17-18). If the client is corrupt, then it learns further leakage on the server's set K_{S}, as well as the unregistered keys in K_{S} (line 12). Leaking the set of unregistered keys is necessary for our security proof, but it does no harm to honest users since their keys are always registered. The server learns only $|K_{\mathsf{C}}|$ (line 14) and whether the intersection A is nonempty (line 20).

We say that the client "successfully authenticates" under a key if that key is in the set A. A client can only authenticate under keys for which it satisfies the can_use (line 17).

If the intersection is nonempty, the client can make the server think that the intersection is empty (line 19-20, $d = 0$) — this relaxation of correctness is needed to model our eventual protocol. However, lying in this way is not beneficial for the client with respect to authentication. The client can never make an empty intersection seem nonempty.

Signing. We model a setting where users can use the same keypairs both for our new authentication protocol and for traditional authentication as well. Since traditional authentication uses a simple challenge-response protocol and uses keypairs for signing, it suffices for our functionality to provide a way for parties to sign and to verify signatures with their keypairs (sign and verify commands). Honest parties will always use the functionality to sign and verify.

If a key is *secure* with respect to the functionality, then the functionality's verify command will reject signatures on messages that weren't originally generated by the key's owner (lines 8,10).[6] In short, if a client P_{C} honestly generates its keypair, and an adversary has not stolen its secret key, then P_{C} is the only party that generate signatures (on *new* messages) that verify properly.

The functionality does not provide any particular unforgeability guarantee for stolen or unregistered keys. Instead, it simply runs the signature scheme's Verify algorithm, so that the real and ideal worlds match (line 11).

[6]If the key owner generates a signature σ on m, then the functionality does not rule out the possibility of an adversary generating a *different* signature σ on the same m. This corresponds to *weak* unforgeability, and such a relaxation is necessary because ECDSA is only weakly unforgeable.

Key agreement. In our envisioned application within SSH, client and server first perform key agreement and then authenticate each other. Hence, our authentication protocol can safely assume that a secure point-to-point channel already exists between client and server. Our protocol can be executed within this secure channel.[7] This means that our ideal functionality does not need to deal with the complexities of defining key agreement — *i.e.*, giving a common random key to both parties iff the client is authorized — it merely needs to give the server the answer to whether the client is authorized.

Other properties. Invoking stealkeys does not allow the adversary to learn whether the newly-stolen keys were used in any *past* authentication attempts, by either the client or server. In other words, our protocol is fully **deniable** for both parties.

Another interesting property is that a server cannot convince the client that authentication has succeeded, unless the server *explicitly knows* (and commits to) one of the client's public keys. This property makes it harder (though not impossible) for a corrupt server to intercept SSH connections intended for another server, as in Attack 4 that we describe in Section 1. Such an attacker would need to target specific users/keys, and would not be able to easily intercept connections on a much larger scale.

5 Main Protocol

Our authentication protocol follows the high-level outline presented in Section 1.4. Namely, the server encrypts a multi-KEM ciphertext to the set of authorized public keys. The client decrypts this ciphertext under each of its secret keys. Finally, the parties perform a private set intersection (PSI), using the plaintexts that they obtained from the multi-KEM. The resulting intersection is non-empty if and only if the client holds an authorized secret key.

We require a flavor of PSI in which the client learns the contents of the intersection, and the server can learn whether the intersection was non-empty. However, it does no harm if the client can *choose* whether to prove that the intersection was non-empty — choosing not to do so only prevents authentication from succeeding. Later in Section 6 we describe how to construct an efficient PSI protocol with this feature. In Figure 3 we formally define the security of this PSI variant, as an ideal functionality in the UC framework.

The formal details of our authentication protocol are given in Figure 4. For technical reasons, the parties perform the PSI on a set of $\langle \mathsf{pk}, m \rangle$ pairs rather than plaintext values alone.

5.1 Security Proof

Theorem 14. *The protocol in Figure 4 is a UC-secure protocol realizing ideal functionality* $\mathcal{F}_{\textsf{new-auth}}$ *(Figure 2) against adaptive adversaries, assuming that* MKEM *is anonymous (Definition 6) and a jointly secure multi-KEM and signature scheme (Definition 7).*

Proof. We sketch a proof here and defer the full details to Appendix B. There are two important cases for the simulator, depending on who is corrupted when the auth session starts.

Case of honest server, corrupt client: In this case, the simulator obtains leakage on the honest server's set of public keys, as well as all its unregistered public keys. It generates a dummy ciphertext c by calling $\mathsf{MKEM.AnonSim}$ on that leakage. $\mathsf{AnonSim}$ is from the MKEM anonymity definition, which we use to show that these dummy ciphertexts are indistinguishable from the real ones. If an honest server is corrupted adaptively during an auth session, then the simulator must provide a dummy internal state for the server. In this case the server's state consists of the r-value from the ciphertext. The simulator generates an r-value using the $\mathsf{AnonView}$ algorithm from the anonymity definition.

[7] We assume that parties will incorporate a transcript of the key agreement session as part of their session id *ssid* to further bind our authentication protocol to their secure channel.

```
Behavior:
  1. Await command $(\text{input}, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), M_{\mathsf{C}})$ from $P_{\mathsf{C}}$.
  2. Give $|M_{\mathsf{C}}|$ to $P_{\mathsf{S}}$.
  3. Await command $(\text{input}, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), M_{\mathsf{S}})$ from $P_{\mathsf{S}}$.
  4. Compute $I = M_{\mathsf{S}} \cap M_{\mathsf{C}}$ and give $(I, |M_{\mathsf{S}}|)$ to $P_{\mathsf{C}}$.
  5. Await command $(\text{deliver}, ssid, d \in \{0,1\})$ from $P_{\mathsf{C}}$.
  6. Give $d \wedge [I \neq \emptyset]$ to $P_{\mathsf{S}}$.
```

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{psi+}}$ for PSI-with-emptiness.

```
On command $(\text{genkey}, \mathsf{opts})$ to party $P_i$:
  1. $P_i$: Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(\mathsf{opts})$ and set $\mathsf{SK}_i[\mathsf{pk}] := \mathsf{sk}$.
     // adversary learns $\mathsf{SK}_i$ when it compromises $P_i$'s storage.
  2. $P_i$: Output $\mathsf{pk}$.

On command $(\text{sign}, \mathsf{pk}, m)$ to party $P_i$:
  3. $P_i$: If $\mathsf{SK}_i[\mathsf{pk}]$ not defined: abort.
  4. $P_i$: Run $\mathsf{Sign}(\mathsf{SK}_i[\mathsf{pk}], m)$ and return the result.

On command $(\text{verify}, \mathsf{pk}, m, \sigma)$ to party $P_i$:
  5. $P_i$: Run $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ and return the result.

On command $(\text{auth}_1, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), K_{\mathsf{S}})$ to party $P_{\mathsf{S}}$:
  6. $P_{\mathsf{S}}$: Generate $(c, r) \leftarrow \mathsf{Enc}(K_{\mathsf{S}})$ and send $c$ to $P_{\mathsf{C}}$.
  7. $P_{\mathsf{C}}$: Await command $(\text{auth}_2, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), K_{\mathsf{C}})$ and set:
       $$M_{\mathsf{C}} := \left\{ \left\langle \mathsf{pk}, \mathsf{Dec}(\mathsf{SK}_{\mathsf{C}}[\mathsf{pk}], c) \right\rangle \;\middle|\; \begin{matrix} \mathsf{pk} \in K_{\mathsf{C}} \text{ and} \\ \mathsf{SK}_{\mathsf{C}}[\mathsf{pk}] \text{ defined} \end{matrix} \right\}.$$
  8. $P_{\mathsf{C}}$: Send $(\text{input}, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), M_{\mathsf{C}})$ to $\mathcal{F}_{\mathsf{psi+}}$.
  9. $P_{\mathsf{S}}$: Receive $|M_{\mathsf{C}}|$ from $\mathcal{F}_{\mathsf{psi+}}$ and output it.
 10. $P_{\mathsf{S}}$: Await command $(\text{auth}_3, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), -)$ and set:
       $$M_{\mathsf{S}} := \left\{ \langle \mathsf{pk}, \mathsf{Msg}(r, \mathsf{pk}) \rangle \;\middle|\; \mathsf{pk} \in K_{\mathsf{S}} \right\}.$$
 11. $P_{\mathsf{S}}$: Send $(\text{input}, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), M_{\mathsf{S}})$ to $\mathcal{F}_{\mathsf{psi+}}$.
 12. $P_{\mathsf{C}}$: Receive output $(I, |M_{\mathsf{S}}|)$ from $\mathcal{F}_{\mathsf{psi+}}$ and output:
       $$\left( \{ \mathsf{pk} \mid \exists m : \langle \mathsf{pk}, m \rangle \in I \}, |M_{\mathsf{S}}| \right)$$
 13. $P_{\mathsf{C}}$: Await command $(\text{deliver}, ssid, d)$ and forward it to $\mathcal{F}_{\mathsf{psi+}}$.
 14. $P_{\mathsf{S}}$: Receive output $e$ from $\mathcal{F}_{\mathsf{psi+}}$ and output it.
```

Figure 4: Our anonymous authentication protocol.

Later, the corrupt client will provide a set of $\langle \mathsf{pk}, m\rangle$ pairs as input to $\mathcal{F}_{\mathsf{psi+}}$. The simulator's main task is to check which of these $\langle \mathsf{pk}, m\rangle$ pairs is "correct" — *i.e.*, whether m is the correct decryption of c with respect to key pk. The set of pk's having correct decryption values is what the simulator sends to $\mathcal{F}_{\mathsf{new\text{-}auth}}$ as the corrupt client's extracted input.

The simulator checks the correctness of a $\langle \mathsf{pk}, m\rangle$ pair in different ways depending on the status of pk:

- If pk is registered, the simulator calls get_sk to learn the corresponding sk, and computes the correct m as $\mathsf{Dec}(\mathsf{sk}, c)$.
- If pk is unregistered, then $\mathsf{MKEM.AnonSim}$ already provided the correct decryption value when generating the dummy ciphertext.

When a key pk is in Secure, this models a key registered to an honest party, whose secret key has not yet been stolen by the real-world adversary. We further use the joint MKEMSS security of MKEM to argue that the adversary cannot predict a "correct" decryption with respect to such a secure pk, and neither can it generate a signature forgery under such a key. Without knowing correct decryptions under pk, the corrupt client cannot authenticat under pk.

Case of corrupt server, honest client: In this case there is no protocol message from the client to simulate in an auth interaction, and no persistent state held by the honest client to simulate in the event of an adaptive corruption. The only job of the simulator is to extract the corrupt server's input (a set of keys) to send to the $\mathcal{F}_{\mathsf{new\text{-}auth}}$ functionality. The simulator observes the server's protocol message c and then later observes the server's PSI input, a set of $\langle \mathsf{pk}, m\rangle$ pairs. As before, the main task of the simulator is to determine which of these pairs is "correct."

- If pk is registered by the functionality (secure or stolen), then it was honestly generated. The simulator can learn the corresponding sk (via get_sk) and obtain the correct m as $\mathsf{Dec}(\mathsf{sk}, c)$.
- If pk is not registered by the functionality, then an honest client will not attempt to authenticate under it. So the simulator can safely ignore these keys. $\square$

The keys from the server's PSI input that are associated with correct decryption values comprise the $\mathcal{F}_{\mathsf{new\text{-}auth}}$ input extracted by the simulator.

6 PSI variant

Our authentication protocol requires a variant of PSI in which the client learns the contents of the intersection, and then the client can (optionally) prove to the server that the intersection was non-empty. Our setting involves relatively small input sets (*e.g.*, a few hundred items each, at the most). The leading PSI protocol for sets of this size — in terms of both communication and running time — is due to Rosulek and Trieu [RT21] (hereafter RT21). We adapt the RT21 protocol to provide the proof-of-nonempty intersection property, to instantiate the ideal functionality in Figure 3. Here we simply sketch the main ideas of our simple modification. The details and formal proof are deferred to Appendix C.

Nearly all PSI protocols, including RT21, use the **oblivious PRF (OPRF)** paradigm of [FIPR05]. The parties first run an OPRF protocol, in which the sender learns a PRF seed k, and a receiver learns $\mathsf{F}(k, x)$ for each x in its set, where F is a PRF. The sender learns nothing about the x values. To obtain a PSI protocol, the OPRF sender sends $\mathsf{F}(k, y)$ for every y in its set. The receiver can determine which items are in the intersection by identifying matching PRF outputs. PRF outputs of items not in the intersection look random to the receiver.

In order to provide proof of nonempty intersection, we modify the protocol as follows. The OPRF sender will send $h^* = H(s)$ to the client, where s is random and H is a collision-resistant hash. Suppose the output of F is divided into two halves $\mathsf{F}(k, x) = \mathsf{F}_1(k, x) \| \mathsf{F}_2(k, x)$. Then instead of sending $\{\mathsf{F}(k, x) \mid x \in X\}$ as before, the sender sends pairs $\{\langle \mathsf{F}_1(k, x), \mathsf{Enc}(\mathsf{F}_2(k, x), s)\rangle \mid x \in X\}$.

The receiver can use the F_1-values to identify the intersection as before. For any x in the intersection, she can decrypt the associated ciphertext with the key $\mathsf{F}_2(k, x)$ to recover s, discarding x from the intersection if $H(s) \neq h^*$. In this way, the receiver learns s if and only if the intersection is nonempty, so her knowledge of r can serve as proof of a nonempty intersection.

The formal description of the modified protocol, and a proof of the following theorem, are provided in Appendix C. We also prove security against adaptive corruption, while RT21's original proof considers only static corruption.

Theorem 15. *The protocol in Figure 8 UC-securely realizes the $\mathcal{F}_{\textsf{psi+}}$ functionality (Figure 3) against adaptive adversaries, in the ideal cipher + random oracle model, assuming a suitable 2-message key agreement scheme exists.*

See Section C for the precise criteria needed for the key agreement scheme. There we also show a suggested scheme that satisfies these properties under a variant of the Strong Diffie–Hellman assumption [ABR01].

Other improvements. One of the main components in the RT21 protocol is a key agreement protocol whose messages are pseudorandom bit strings. In order to support elliptic-curve Diffie-Hellman key agreement, the suggested key agreement uses the *Elligator* technique [BHKL13] to encode elliptic curve elements as uniform bit strings. We observe that a different technique of Möller [Möl04] results in elliptic-curve-based key agreement at roughly half the computational cost. The details are given in Section C.1.

7 Implementation and Evaluations

Implementation. We implemented our protocol in C++ and integrated it into both the client and server of OpenSSH version 8.2p1[8]. We implemented the Multi-KEMs for RSA, ECDSA, and EdDSA as described in Section 3, using OpenSSH and libsodium. We also adapted the implementation of the RT21 PSI protocol [RT21], with the modifications described in Section 6. Namely, we added the proof-of-nonempty-intersection feature, and also incorporated an improved technique for the underlying key agreement. The implementation of the RT PSI protocol uses Rijndael as a 256-bit ideal cipher and SHA-256 as a random oracle.

OpenSSH delegates sensitive signing operations to a separate ssh-agent daemon process, which provides a signing oracle to the SSH client. Since our protocol uses SSH keys as KEM keys, we added an additional KEM decryption interface to ssh-agent. The remainder of the protocol is implemented in the SSH client/server processes (*i.e.*, ssh and sshd). Upon publication, we will make the source code available on GitHub under the same BSD license that OpenSSH uses.

7.1 Experimental Setup

Hardware. All experiments use two desktop machines with 32-core AMD Threadripper 2990WX running at 3.0Ghz, running Ubuntu 20.04 LTS with 32GB DRAM. We use one machine as SSH server and the other for running many SSH clients. While the SSH server utilizes multiple cores for handling multiple clients, we do not use multiple cores to parallelize our authentication protocol.

Network. We ran microbenchmarks over the loopback device to focus on computation time. To simulate realistic network conditions in a macrobenchmark, we used the tc traffic control utility to add 42.5 ms latency: the average of local (20ms within US west coast) and distant (65ms between east and west coast) latencies reported in [Won22].

Keys. We performed SSH authentication on a range of key configurations. For our microbenchmarks, we considered sets of keys that were **RSA-only** (RSA-3072, which is the default RSA key

[8]Our implementation is available at https://github.com/osu-crypto/PSIPK-ssh

size in OpenSSH), **EdDSA-only**, and **ECDSA-only**. Hence, we explore the effect of key flavor on our protocol's performance. In our macrobenchmark we used a **mixture of keys**: 92% RSA, 7% EdDSA, and 1% ECDSA, to model realistic proportions of key flavors according to Github statistics [Coo21]. We also present macrobenchmark results with 100% EdDSA keys to demonstrate optimal performance.

We tested clients and servers with different numbers of keys: We considered clients with 5 (normal user) and 20 (heavy user) keys. We considered servers with 10 (private server), 100 (mid-sized git repository), and 1000 (popular git repository) keys.

Reported numbers. For each test case, we report the average over 10 executions. For comparison, we also measure the cost of vanilla SSH public-key authentication under the same network/system setup.

7.2 Evaluation Results

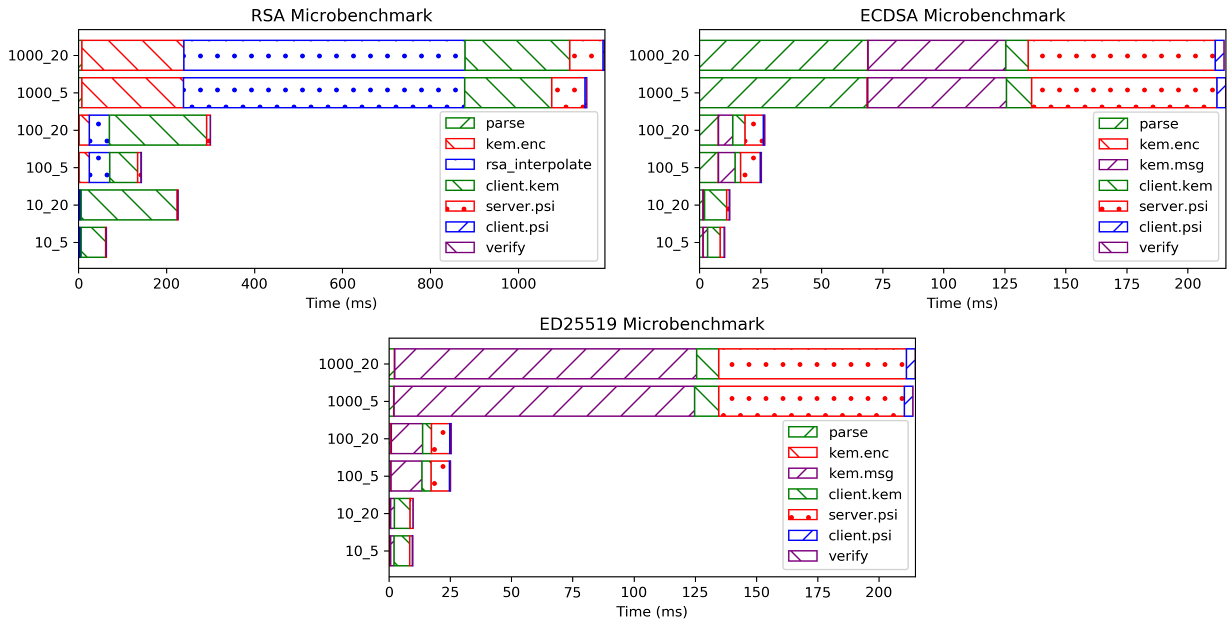


Figure 5: Microbenchmark result per each key setup. For each type of key, we vary the number of keys at the client side for 5 and 20, and we also vary number of authorized keys at the server side for 10, 100, and 1000.

Microbenchmark: performance breakdown. We conducted a microbenchmark of our protocol to investigate the computation time required for each step. Figure 5 (in appendix) shows the results.

Legend. We divided the protocol's computational tasks into the following phases, and measured the time taken by each: First, the server parses the authorized user's keys file (parse), and encrypts the KEM ciphertexts (kem.enc). It must interpolate a polynomial containing all of the RSA ciphertexts (rsa_interpolate). The KEM messages are then computed by the server (kem.msg). After receiving the KEM ciphertexts from the server, the client decrypts them using its private keys and generates a PSI polynomial (client.kem). The server evaluates this polynomial, and generates a challenge for the client (server.psi). Finally, the client solves this challenge (client.psi), and the server verifies the solution (verify). A test case named "X_Y" indicates that the server has X number of authorized keys and the client has Y number of available private keys for authentication.

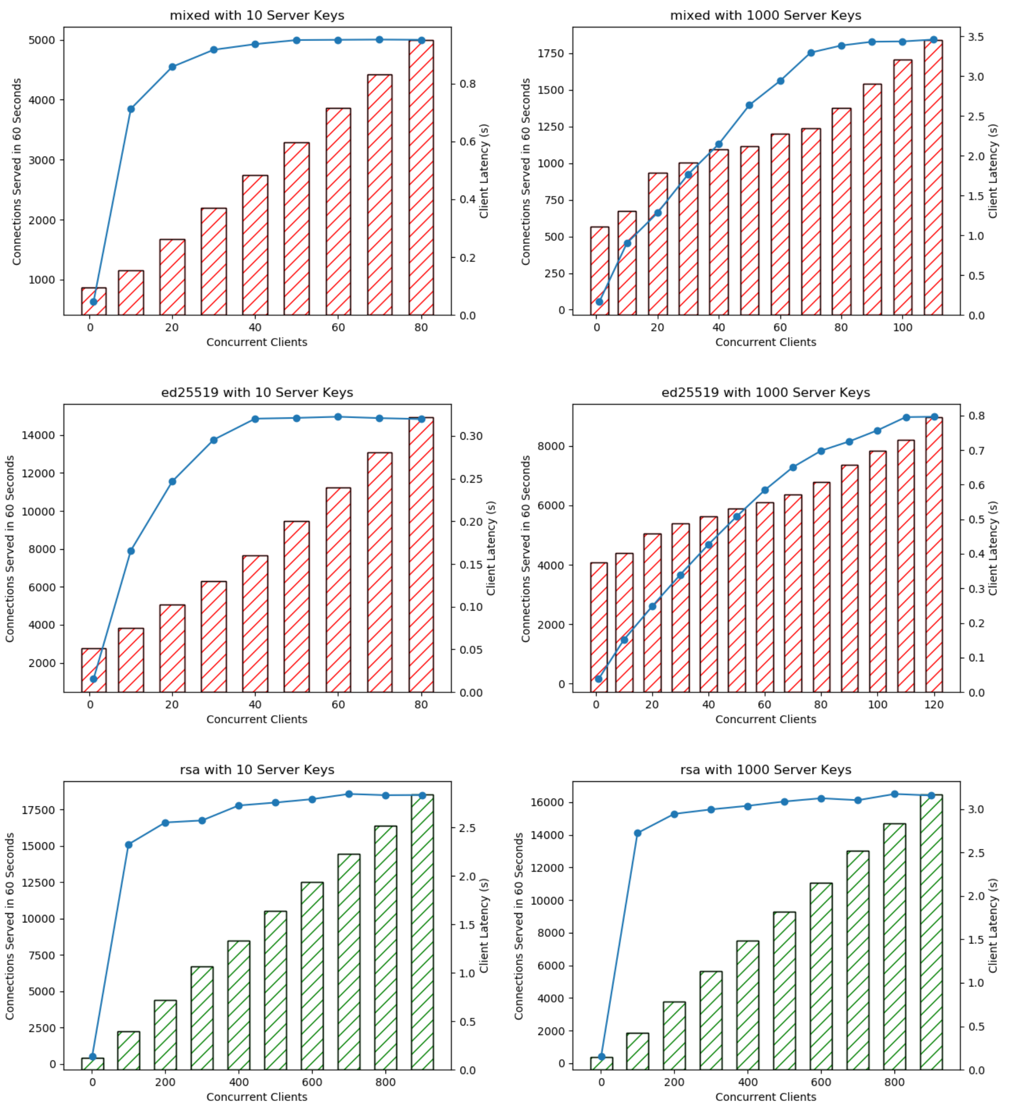


Figure 6: Macrobenchmark result for using mixed keys from the Github key statistics [Coo21] and using only EdDSA keys, for 10 and 1000 keys on the server. In all cases, we use 3 keys on the client side. Each graph shows the number of connections processed in minutes (line, left Y-axis) and average latency that each client suffers for the SSH authentication (bar, right Y-axis), by increasing number of concurrently connecting clients (X-axis). Top two graphs are for mixed keys, middle two graphs are for EdDSA. The bottom two graphs show the performance of vanilla authentication when using RSA keys.

e.g., 1000_20 refers to a server with 1000 keys and a client with 20 keys, corresponding to a heavy user authenticating to a very popular git repository.

RSA. RSA keys are the slowest among three key flavors. For a huge number of keys on the server, *e.g.*, 1000_5 and 1000_20, authentication takes 1,155 ms and 1,196 ms, respectively, while authentication with fewer than 100 keys needs less than 300 ms. For the server, kem.enc and rsa_interpolate took the majority of computation time. Because the time increases linearly with the number of authorized keys on the server, these tasks dominate when the server has many keys. For the client, the client.kem task increases proportionally to number of keys from both server and client. In the case of relatively many keys for the client relative to the server, the client.kem cost overwhelms the computing time, as shown on 100_20, 10_20, and 10_5.

Auth-type	Key-conf	RSA	ECDSA	EdDSA
PSI	100_20	54188	12116	12174
PSI	10_20	13340	8972	9228
Vanilla	10_5	12742	10036	9572
Vanilla	10_1	9642	8582	8242

Table 1: Total communication (in bytes) for a successful authentication, under various key configurations and key flavors. Note that we do not test 100 keys on the server for vanilla authentication because communication cost does not depend on the server's set of keys in vanilla authentication.

ECDSA and EdDSA. Both ECDSA and EdDSA are significantly faster than RSA, and the performance characteristics of these two are very similar. Even with 1000 keys on the server, the authentication finishes in less than 216 ms. With fewer than 100 keys, it finishes in less than 27 ms, which is comparable to a typical network delay. Thus, the effect on the user would be negligible. For these two key types, server-side computation time dominates the entire execution time. The difference between ECDSA and EdDSA is that ECDSA requires more time to parse the key. However, EdDSA requires more time on kem.msg, resulting in less than 1% time difference for 1000_20: 214.83 ms *vs.* 214.89 ms.

Communication cost. The new protocol incurs not only computational overhead but also overhead in communication. We measure the communication cost of our protocol and compare it to vanilla SSH authentication. Table 1 shows the results. With 10 keys on the server and 20 keys on the client, communication overhead for all key flavors is negligible when compared to the vanilla authentication with 5 keys on the client. When increasing the server-side key number to 100, the message size for both ECDSA and EdDSA does not increase much in size (from 9 kB to 12 kB). However, RSA requires 54 kB of message transfer, which incurs around 4.25 times more in communication when compared to the vanilla 10_5 case. For vanilla with five client keys, we use the 5th key for authentication, and thereby include four trials of failed public key probing.

Macrobenchmark: server authentication throughput. We macrobenchmark our protocol, measuring the authentication throughput (in reqs/min, at the server side) and latency (in seconds, at the client side). Figure 6 shows the results. Below, we report throughput and latency at the maximum throughput and compare it to the vanilla SSH authentication.

For mixed key setup (mixed according to Github statistics), a server with 10 keys can process up to 5,003 reqs/min (83.4 reqs/sec), with clients observing up to 0.83 second of latency. A server with 1000 keys can process upto 1,869 reqs/min (31.1 reqs/sec), with clients observing up to 3.45 seconds of latency. For pure EdDSA, a server with 10 keys can process up to 14,964 reqs/min (249.4 reqs/sec), with clients observing up to 0.24 second of latency. A server with 1000 keys can process upto 8,981 reqs/min (149.6 reqs/sec) while clients may observe 0.79 seconds of latency. As we observed in microbenchmark, RSA keys are much slower than ECDSA/EdDSA keys, and that is also consistent in the macrobenchmark.

We compare this result with the throughput/latency of the vanilla SSH authentication. For pure RSA keys setup, a server with 10 keys can process up to 18,560 reqs/min (309.3 reqs/sec). This is 3.7 times faster than our protocol with mixed keys, but only 24% faster than our protocol with EdDSA keys. A server with 1000 RSA keys can process up to 16,500 reqs/min (275.0 reqs/sec), which is 8.8 times faster than our protocol with mixed keys, but only 85% faster than ours with EdDSA keys.

In conclusion, our protocol runs comparable to the vanilla SSH authentication when used with ECDSA/EdDSA keys.

8 Discussions

In this section we discuss security, privacy, and usability issues arising from integrating our protocol into SSH.

Passphrase-protected SSH keys. SSH clients allow users to protect keys with a passphrase, which must be entered interactively before that key is used for authentication. In a standard authentication, the client software can collect the passphrase from the user only if the server requests authentication under that key. In our authentication method, the client effectively makes authentication attempts under all of its keys. Therefore, a user may need to enter passphrases to *all* keys while running our authentication method. Thankfully, ssh-agent can be configured to only require a passphrase once during the life of the ssh-agen process (*e.g.*, per reboot).

Integrating to Git/SSH. Our protocol assumes that the server has identified the set of authorized keys at the time of authentication (*e.g.*, from ~/.ssh/authorized_keys). Not all applications may be compatible with this requirement.

We illustrate the issue using Github as an example. When committing changes to GitHub, the SSH connection is always made to git@github.com. The client reports the name of the repository only after the SSH authentication, as git@github.com:username/repository. In other words, *every* Github user is authorized to connect to username git.

This is not problematic for standard SSH authentication because the server identifies the client from its public key. In contrast, our new protocol would require the server to encrypt a KEM message to the set of all (73 million as of November 2021 [DMR22]) Github users, which is prohibitively expensive.

In order to integrate our protocol with systems like Github, the server would need to learn the repository name *before* the client authentication step. We believe that the *SSH username*, which is indeed sent to the server before cient authentication, is a natural way to convey this information. For example, a client who opts into the new authentication method could use an SSH connection to, say, repositoryname@new.github.com or username.repository@new.github.com. All other users could continue to be supported via SSH connections to git@github.com. Github users could configure which of these two git URL styles is presented to them on the Github website. Repository owners could choose which flavors of authentication to support when connecting to their repositories.

Downgrading attacks and Trust on First Use. A mischevious server can simply claim to not support our privacy-enhancing protocol. When connecting to such a server, the client is forced to downgrade to a less private, conventional authentication method. Clients should be vigilant about such downgrade attacks, which completely undermine the protection of our protocol. The same trust-on-first-use (TOFU) policy for authenticating the server can be applied to this behavior — *e.g.*, the client software can report an error if the server supported privacy-preserving authentication in the past but now claims to not support it, similar to the error when a server's public key has changed relative to the known_hosts file.

Size of key-sets. Our protocol leaks an upper bound on the size of both the client's and server's set of keys. This leakage is another avenue for fingerprinting, although carrying much less identifying information. Still, users may wish to mitigate this leakage by padding their key sets with dummy items, up to some fixed size — *e.g.*, the next power of two.

Server-side probing. A server can choose to run our authentication protocol with a strict subset of the authorized keys. By varying the subset across repeated authentication attempts, the server could de-anonymize the client's choice of key via a binary search.

However, this attack leads to user-visible authentication failures, and it requires a client to repeatedly retry after such failures. We leave open the problem of whether our protocol could be extended to notify clients of extreme changes in the server's set of keys.

One indication of a probing server may be its use of a very large set of keys. Our protocol reveals the size of the server's set to the client, just before the client decides whether to deliver output to the server. In principle a client could be configured to refuse connection to a server with a suspiciously high number of authorized keys.

Other authentication methods. SSH supports a lightweight certificate system for authentication, but supporting it is well beyond our scope. Certificates introduce an extra level of indirection: the server knows the root signing key but not the keys of individual users, so the protocol would need to verify two steps of the trust chain. SSH also supports hardware-token-based keys. These tokens support only signing, and not KEM decryption, making them incompatible with our approach.

References

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, April 2001.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[BBS03] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. Randomness re-use in multi-recipient encryption schemeas. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 85–99. Springer, Heidelberg, January 2003.

[BBW06] Adam Barth, Dan Boneh, and Brent Waters. Privacy in encrypted content distribution using private broadcast encryption. In Giovanni Di Crescenzo and Avi Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 52–64. Springer, Heidelberg, February / March 2006.

[BDL$^+$12] 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, September 2012.

[BDS$^+$03] Dirk Balfanz, Glenn Durfee, Narendar Shankar, Diana K. Smetters, Jessica Staddon, and Hao-Chi Wong. Secret handshakes from pairing-based key agreements. In *2003 IEEE Symposium on Security and Privacy*, pages 180–196. IEEE Computer Society Press, May 2003.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

[Bro02] Daniel R. L. Brown. Generic groups, collision resistance, and ECDSA. Contributions to IEEE P1363a, February 2002. Updated version for "The Exact Security of ECDSA." Available from http://grouper.ieee.org/groups/1363/.

[Coo21] Matt Cooper. Improving Git protocol security on GitHub, 2021. https://github.blog/2021-09-01-improving-git-protocol-security-github/.

[Cox15] Ben Cox. Auditing GitHub users' SSH key quality. Blog post. https://blog.benjojo.co.uk/post/auditing-github-users-keys, 2015.

[Cv91] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, April 1991.

[DDN91] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography (extended abstract). In *23rd ACM STOC*, pages 542–552. ACM Press, May 1991.

[DJKT09] Emiliano De Cristofaro, Stanislaw Jarecki, Jihye Kim, and Gene Tsudik. Privacy-preserving policy-based information transfer. In Ian Goldberg and Mikhail J. Atallah, editors, *PETS 2009*, volume 5672 of *LNCS*, pages 164–184. Springer, Heidelberg, August 2009.

[DKNS04] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. Anonymous identification in ad hoc groups. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 609–626. Springer, Heidelberg, May 2004.

[DKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 213–231. Springer, Heidelberg, December 2010.

[DLP$^+$12] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefler. On the joint security of encryption and signature in EMV. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 116–135. Springer, Heidelberg, February / March 2012.

[DMR22] DMR. GitHub Statistics, User Counts, Facts & News (2022), 2022. https://expandedramblings.com/index.php/github-statistics/.

[DNS98] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent zero-knowledge. In *30th ACM STOC*, pages 409–418. ACM Press, May 1998.

[DT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 143–159. Springer, Heidelberg, January 2010.

[FHH10] Chun-I Fan, Ling-Ying Huang, and Pei-Hsiu Ho. Anonymous multireceiver identity-based encryption. *IEEE Transactions on Computers*, 59(9):1239–1249, 2010.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[FKP16] Manuel Fersch, Eike Kiltz, and Bertram Poettering. On the provable security of (EC)DSA signatures. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1651–1662. ACM Press, October 2016.

[GPR$^+$21] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event, August 2021. Springer, Heidelberg.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 89–98. ACM Press, October / November 2006. Available as Cryptology ePrint Archive Report 2006/309.

[GS98] Venkatesan Guruswami and Madhu Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. In *39th FOCS*, pages 28–39. IEEE Computer Society Press, November 1998.

[JKT08] Stanislaw Jarecki, Jihye Kim, and Gene Tsudik. Beyond secret handshakes: Affiliation-hiding authenticated key exchange. In Tal Malkin, editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 352–369. Springer, Heidelberg, April 2008.

[JL07] Stanislaw Jarecki and Xiaomin Liu. Unlinkable secret handshakes and key-private group key management schemes. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 270–287. Springer, Heidelberg, June 2007.

[JL09] Stanislaw Jarecki and Xiaomin Liu. Private mutual authentication and conditional oblivious transfer. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 90–107. Springer, Heidelberg, August 2009.

[Kur02] Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In David Naccache and Pascal Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, Heidelberg, February 2002.

[KY07] Aggelos Kiayias and Moti Yung. Cryptographic hardness based on the decoding of reed-solomon codes. Cryptology ePrint Archive, Report 2007/153, 2007. https://eprint.iacr.org/2007/153.

[LPQ12] Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Anonymous broadcast encryption: Adaptive security and efficient constructions in the standard model. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 206–224. Springer, Heidelberg, May 2012.

[MKJR16] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[Mob22] Mobatek. MobaXterm, 2022. https://mobaxterm.mobatek.net/.

[Möl04] Bodo Möller. A public-key encryption scheme with pseudo-random ciphertexts. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer, Heidelberg, September 2004.

[MPR11] Hemanta K. Maji, Manoj Prabhakaran, and Mike Rosulek. Attribute-based signatures. In Aggelos Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 376–392. Springer, Heidelberg, February 2011.

[MPT10] Mark Manulis, Bertram Poettering, and Gene Tsudik. Taming big brother ambitions: More privacy for secret handshakes. In Mikhail J. Atallah and Nicholas J. Hopper, editors, *PETS 2010*, volume 6205 of *LNCS*, pages 149–165. Springer, Heidelberg, July 2010.

[MRR21] Ian McQuoid, Mike Rosulek, and Lawrence Roy. Batching base oblivious transfers. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021*, pages 281–310, Cham, 2021. Springer International Publishing.

[Nao02] Moni Naor. Deniable ring authentication. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 481–498. Springer, Heidelberg, August 2002.

[Nat16] National Vulnerability Database. CVE-2016-20012 detail, 2016. https://nvd.nist.gov/vuln/detail/CVE-2016-20012.

[NKS$^+$17] Matus Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 162–175, 2017.

[NSW09] Gregory Neven, Nigel Smart, and Bogdan Warinschi. Hash function requirements for schnorr signatures. *Journal of Mathematical Cryptology*, 3(1):69–87, 2009. Other identifier: 2001023.

[OP01] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, February 2001.

[Ope22a] OpenSSH. OpenSSH, 2022. https://www.openssh.com/.

[Ope22b] OpenSSH. OpenSSH-Portable, 2022. https://github.com/openssh/openssh-portable/blob/master/auth2-pubkey.c#L280-L286.

[PuT22] PuTTY. Download PuTTY, 2022. https://www.putty.org/.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, December 2001.

[RT21] Mike Rosulek and Ni Trieu. Compact and malicious private set intersection for small sets. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1166–1181, New York, NY, USA, 2021. Association for Computing Machinery.

[Sie16] Chris Siebenmann. Your SSH keys are a (potential) information leak, 2016. https://utcc.utoronto.ca/~cks/space/blog/tech/SSHKeysAreInfoLeak.

[SSH22] SSH.COM. What is SSH public key authentication?, 2022. https://www.ssh.com/academy/ssh/public-key-authentication.

[Val15a] Fillipo Valsorda. SSH whoami.filippo.io. Blog post. https://blog.filippo.io/ssh-whoami-filippo-io/, 2015.

[Val15b] Fillipo Valsorda. whoami.filippo.io: an ssh server that knows who you are. Github repository. https://github.com/FiloSottile/whoami.filippo.io, 2015.

[Won22] WonderNetwork. Global Ping Statistics, 2022. https://wondernetwork.com/pings.

[ZC17] Yongjun Zhao and Sherman S. M. Chow. Are you the one to share? Secret transfer with access structure. *PoPETs*, 2017(1):149–169, January 2017.

[ZLZL15] Fu-Cai Zhou, Mu-Qing Lin, Yang Zhou, and Yu-Xi Li. Efficient anonymous broadcast encryption with adaptive security. *KSII Transactions on Internet and Information Systems (TIIS)*, 9(11):4680–4700, 2015.

A Security Proofs for Multi-KEMS

A.1 EdDSA

Lemma 8. *Any attack $\mathcal{A}$ against the joint security of the MKEMSS* EdDSA *implies an attack $\mathcal{A}'$ against the GapCDH problem. $\mathcal{A}'$ takes approximately twice the computation of $\mathcal{A}$, and*

$$\mathrm{Adv}[\mathcal{A}] \leq \sqrt{q_H\left(\frac{\mathrm{Adv}[\mathcal{A}']}{P^2}+\frac{1}{\ell}\right)}+\frac{q_H q_S}{\ell},$$

where $\mathcal{A}$ makes q_H queries to the random oracle H and requests q_S signatures.

Proof. We give a hybrid proof. Note that $\mathsf{Dec}(\mathsf{sk}, C) = m$ if and only if G^{sk}, C, m are a DH tuple, which is something that a simulator can check using the GUESS oracle of the GapCDH game. In the hybrid, replace the oracles in the MKEMSS game with the following simulations, where $\mathsf{pk}^* = A = G^a, B = G^b$ are sampled at the start with $a, b \leftarrow M$:

```
GUESS_MSG($C, \mathsf{pk}, m$):
  return $m \in \mathbb{G} \wedge$ GUESS($\mathsf{pk}^f, C^f, m^f$)

GUESS_DEC($C, m$):
  return GUESS_MSG($C, \mathsf{pk}^*, m$)
```

```
$X$ := empty map
ENCRYPT($\mathsf{PK}$):
  $x \leftarrow \mathbb{Z}/\ell\mathbb{Z}$
  $C := B\,G^x$
  $X[C] := x$
  return $C$
```

```
SIGN($m$):
  if $m$ already signed:
    return old signature
  $s, h \leftarrow \mathbb{Z}/\ell\mathbb{Z}$
  $R := G^s A^{-h}$
  $H(R, A, m) := h$
  return $(R, s)$
```

CHECK clears the cofactor so that the curve points are in the prime order subgroup, which is required to use GUESS. This loses the cofactor information from C and m, but the former was cleared anyway because $f \mid a$, well the latter is only important because if $m \notin \mathbb{G}$ then $m \neq C^a \in \mathbb{G}$. The challenge ciphertexts output by ENCRYPT are rerandomizations of B, to force $\mathcal{A}$ to solve the GapCDH instance if it manages to decrypt any. SIGN simulates the signatures by sampling the random oracle output h in advance, then computing the R that will make the signature verify. These simulations behave identically to the real MKEMSS oracles, assuming that the hash value $H(R, A, m)$ was not already evaluated. There have been at most q_H previous queries to H with this m, so this bad event gives the adversary an advantage of at most $q_H q_S/\ell$, by the union bound.

Now, we give a reduction, but to a slightly modified GapCDH assumption where a and b continue to be sampled from M. Define a new adversary $\mathcal{A}'$, and let $\mathrm{Adv}_M[A']$ be the advantage of $\mathcal{A}'$ against this modified GapCDH game. $\mathcal{A}'$ runs $\mathcal{A}$, and uses its output to attack the GapCDH challenge. $\mathcal{A}'$ can check for correct signatures using Verify, and for correct decryptions using CHECK. If $\mathcal{A}$ returns neither, we have made no progress — abort the attack. If $\mathcal{A}$ outputs a correct decryption m of a ciphertext C, then $\mathcal{A}'$ solves the GapCDH instance as $B^a = C^a A^{-x} = m A^{-x}$ where $x = X[C]$. Finally, if $\mathcal{A}$ returns only a forged signature (R, s) of a message m, we need to rewind to attempt extraction.

Rewind back to the first evaluation of $h = H(R, A, m)$, for the R and m output by $\mathcal{A}$. Resample it as $h' \leftarrow \mathbb{Z}/\ell\mathbb{Z}$, then run $\mathcal{A}$ again starting from here, resampling subsequent oracle queries as well. Again, if $\mathcal{A}$ outputs a correct decryption, $\mathcal{A}'$ can solve for B^a. If $\mathcal{A}$ outputs another forged signature

(R, s') for the same message m, and $h' \neq h$, then the private key can be extracted as $a = \frac{s'-s}{h'-h} \bmod \ell$, and the GapCDH instance solved by computing B^a. Otherwise, $\mathcal{A}'$ aborts the attack.

There are two exclusive cases where $\mathcal{A}$ succeed. Let P_{MKEM} be the probability that $\mathcal{A}$ finds a correct decryption, and P_{SS} be the probability that $\mathcal{A}$ fails to find a correct decryption, but successfully forges a signature. Then $\mathrm{Adv}[\mathcal{A}] = P_{MKEM} + P_{SS}$, and $\mathrm{Adv}_M[\mathcal{A}'] = P_{MKEM} + P_{SS}P_{MKEM} + P_R$, where P_R is the probability that the rewinding was successful and $\mathcal{A}'$ uses the two forged signatures to solve for a. By the general forking lemma [BN06], we have $P_R \geq P_{SS}\big(\frac{P_{SS}}{q_H} - \frac{1}{\ell}\big)$. Then,

$$\begin{aligned}
\mathrm{Adv}_M[\mathcal{A}'] &\geq P_{MKEM} + P_{SS}P_{MKEM} + \frac{P_{SS}^2}{q_H} - \frac{1}{\ell} \\
&\geq \frac{P_{MKEM}}{q_H}\,\mathrm{Adv}[\mathcal{A}] + \frac{P_{SS}P_{MKEM}}{q_H} + \frac{P_{SS}^2}{q_H} - \frac{1}{\ell} \\
&= \frac{P_{MKEM}}{q_H}\,\mathrm{Adv}[\mathcal{A}] + \frac{P_{SS}\,\mathrm{Adv}[\mathcal{A}]}{q_H} - \frac{1}{\ell} \\
&= \frac{(\mathrm{Adv}[\mathcal{A}])^2}{q_H} - \frac{1}{\ell}
\end{aligned}$$

Finally, we need to lower bound $\mathrm{Adv}[A']$, the success probability against the original GapCDH assumption. We know that there is probabilty P^2 of a and b being in $M \bmod \ell$, and in this case A' will succeed with probability $\mathrm{Adv}_M[\mathcal{A}']$. Therefore, $\mathrm{Adv}[A'] \geq P^2\,\mathrm{Adv}_M[\mathcal{A}']$. Rearrange to get the stated upper bound on $\mathrm{Adv}[A]$. □

A.2 ECDSA

Lemma 9. *If H is collision resistant and zero-finder-resistant, then* ECDSA *is a jointly secure MKEMSS in the GGM.*

Proof. The result proved in [DLP+12] used the related ECIES encryption scheme, and proved that the KEM key was indistinguishable from random, rather than just being unguessable. In ECIES, indistinguishability is achieved by hashing the KEM key: $\mathsf{ECIES.Dec}(a, C) = \mathsf{KDF}((C^a)_x)$. The assumptions [DLP+12] requires for KDF are not given explicitly, but a random oracle should be sufficient. The assumptions they require for H are the ones given in the statement of this lemma. Under these assumptions, they prove that the adversary's advantage against the joint signature and encryption scheme ECIES+ECDSA is at most $O(q_G^2/\ell)$ plus its advantage against H, where q_G is the number of group operations performed.

Note that, unlike our $\mathsf{ECDSA.Dec}$, $\mathsf{ECIES.Dec}$ only depends on the x-coordinate so both C and C^{-1} give the same key. Although earlier in their paper they point out that this means they need to slightly weaken the IND-CCA definition so that the adversary isn't allowed to asked for the decryption of either C or C^{-1}, they do not address this distinction in their proof. While their proof is likely still sound, since this only requires a few small modifications to fix, it clearly shows that a modified scheme ECIES' where $\mathsf{Dec}(a, C) = \mathsf{KDF}(C^a)$ uses the whole curve point and not just the x-coordinate would also be secure. We prove that our scheme is a jointly secure MKEMSS by giving a reduction to the joint IND-CCA and EUF-CMA security of ECIES'.

Let $\mathcal{A}$ be an adversary against the MKEMSS game for ECDSA, and construct from it new adversaries $\mathcal{A}'$ and $\mathcal{A}''$ against the IND-CCA and EUF-CMA games of ECIES', respectively. $\mathcal{A}'$ will succeed when $\mathcal{A}$ would successfully guess a decryption, and $\mathcal{A}''$ will succeed when $\mathcal{A}$ would successfully forge a signature. Both $\mathcal{A}$ and $\mathcal{A}'$ run $\mathcal{A}$, passing through the public key A and the SIGN oracle. ENCRYPT are simulated as follows.

```
$X :=$ empty map
ENCRYPT($\mathsf{PK}$):
  $x \leftarrow \mathbb{Z}/\ell\mathbb{Z}$
  $C := B\,G^x$
  $X[C] := x$
  return $C$
```

Here, B is the challenge ciphertext given by the IND-CCA game, or just a uniformly sampled curve point for $\mathcal{A}''$. Though it rerandomizes to concentrate the attack on B, ENCRYPT is indistinguishable from the real ENCRYPT oracle. Finally, $\text{CHECK}(C, m) = \big(\text{DECRYPT}(C) \overset{?}{=} \mathsf{KDF}(m)\big)$, using the ECIES' decryption oracle, since $\mathsf{ECIES}'.\mathsf{Dec}(a, C) \overset{?}{=} \mathsf{KDF}(m)$ is equivalent to $C^a \overset{?}{=} m$.[9] In IND-CCA, DECRYPT isn't allowed to be used on the challenge ciphertext B. However, $\mathcal{A}$ has negligible probability of guessing B, because ENCRYPT's behavior is statistically independent of B. Therefore, $\mathcal{A}$ can't tell the difference between these simulated oracles and the real ones provided by the MKEMSS security game.

Finally, if $\mathcal{A}$ outputs a correct decryption m of some C output by ENCRYPT, $\mathcal{A}'$ can solve for $B^a = C^a A^{-x} = mA^{-x}$, where $x = X[C]$, and check whether in matches the decryption $\mathsf{KDF}(B^a)$ that it must distinguish from random. If $\mathcal{A}$ outputs a forged signature, $\mathcal{A}''$ can pass this on to the EUF-CMA game. Either way, $\mathcal{A}'$ and $\mathcal{A}''$ have advantage negligibly distant from the chance that $\mathcal{A}$ breaks the corresponding part of the joint scheme. □

A.3 RSA

Lemma 10. *The EUF-CMA security of the* RSA *signature scheme implies that* RSA *is a jointly secure MKEMSS.*

Proof. Let $\mathcal{A}$ be a PPT adversary against MKEMSS security. We construct a new adversary $\mathcal{A}'$ that has similar runtime and almost identical advantage to $\mathcal{A}$, but against EUF-CMA instead of MKEMSS. Let $\mathsf{pk}^* = (N, e)$ be the public key under attack. First, $\mathcal{A}'$ picks a uniformly random message $m^* \in \{0,1\}^\lambda$ and finds $c^* = \mathsf{PKCS}_N(m^*)$. The reduction will force $\mathcal{A}$ to find the signature $s^* = {c^*}^d$ for m^* if it successfully decrypts a KEM message encrypted with pk.

Run $\mathcal{A}^{\text{ENCRYPT},\text{GUESS\_DEC},\text{GUESS\_MSG},\text{sign}}(\mathsf{pk})$, where the oracles ENCRYPT, GUESS_DEC, GUESS_MSG are simulated as follows. First, we observe that the GUESS_DEC, GUESS_MSG oracles give no additional power to the adversary. Since MKEM decryption is essentially just the RSA trapdoor inverse, a plaintext guess can be verified by applying RSA trapdoor in the forward direction. Namely,

```
GUESS_MSG($C, \mathsf{pk} = (N, e), m$):
  for $i := 0$ to $s(N) - 1$:
    $c_i := C(H(N, e, i))$
  $c := \mathsf{Unchk}_N(\{c_i\}_i)$
  return $c \overset{?}{=} m^e \bmod N$

GUESS_DEC($C, m$):
  return GUESS_MSG($C, \mathsf{pk}^*, m$)
```

The simulated ENCRYPT oracle works almost as normal, but with the call to $\mathsf{Enc1}(\mathsf{pk}^*)$ for the target public key replaced with ENCRYPT1, which uses rerandomization to turn c^* into many independent ciphertexts. Each will still be uniformly random, because c^* is in the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^\times$ with all but negligible probability.

[9] Unless a KDF collision is found, but that has negligible probability.

```
$\underline{\textsc{encrypt1}()}$:
  $r \leftarrow \mathbb{Z}/N\mathbb{Z}$
  $c := c^* r^e \bmod N$
  $\mathcal{R}[c] := r$
  return $c$
```

Now, for $\mathcal{A}$ to win the MKEMSS game, it must either forge a signature (m, s), which immediately lets $\mathcal{A}'$ attack EUF-CMA, or find a decryption m under the target key sk of some C generated by ENCRYPT. $\mathcal{A}'$ extracts a signature forgery as follows:

```
// given $(m, s, C)$ such that $\mathsf{RSA.Dec}(\mathsf{sk}^*, C) = m$
for $i := 0$ to $s(N) - 1$:
  $c_i := C(H(N, e, i))$
$c := \mathsf{Unchk}_N(\{c_i\}_i)$
$r := \mathcal{R}[c \bmod N]$
output $s^* = m/r \bmod N$
```

Recall that c was originally generated as $c = c^* r^e \bmod N = \mathsf{PKCS}_N(m^*) r^e \bmod N$. Hence s^* is the correct PKCS signature of the random message m^*. □

Lemma 11. RSA *is anonymous with leakage* $\mathsf{RSA.Leak}(\mathsf{PK}) = s(\mathsf{PK})$*. The distinguisher has an advantage of at most* $2^{-\lambda-2} t_E + \left|2^{-2\lambda}|\mathbb{F}| - 1\right| t_S$*, where* t_E *and* t_S *are the totals of* $|\mathsf{PK}|$ *and* $s(\mathsf{PK})$ *over all calls to* ENCRYPT(PK)*, respectively.*

Proof. We first need to define a simulator that generates a dummy ciphertext given only the leakge L and the set of adversarially chosen keys. The adversarial keys PK_{adv} are easy to handle: we address RSA.Enc1 ciphertexts to them as normal. We don't know the honest keys — we only know from the leakage L that we should produce a polynomial of degree up to L. Therefore, the simulator samples C as uniformly random polynomial with degree less than L that interpolates through the correct points for the adversarial keys.

```
$\underline{\mathsf{RSA.AnonSim}(L, \mathsf{PK}_{adv})}$:
  $S := \emptyset$
  $M :=$ empty map
  // honestly construct Enc1 ciphertexts for keys in $\mathsf{PK}_{adv}$
  for $(N, e) \in \mathsf{PK}_{adv}$:
    $c, r \leftarrow \mathsf{RSA.Enc1}((N, e))$
    $M[(N, e)] := r$
    $c_0, \ldots, c_{s(N)-1} \leftarrow \mathsf{Chk}_N(c)$
    for $i := 0$ to $s(N) - 1$:
      $S := S \cup \{(H(N, e, i), c_i)\}$

  // randomly increase number of interpolation points to $L$
  while $|S| < L$:
    $(a, b) \leftarrow \mathbb{F}$
    $S := S \cup \{(a, b)\}$

  $C := \mathsf{interpol}_{\mathbb{F}}(S)$
  return $(C, M, view = (C, M))$
```

Later AnonView may be asked to produce an R-value for this ciphertext that "explains" its decryptions with respect to honest keypairs. AnonView is given the secret keys for those honest keypairs when this happens. Hence, it can simply decrypt the given ciphertext under each of those secret keys. These plaintexts would have been the only randomness in a real-world ciphertext.

```
$\underline{\mathsf{RSA.AnonView}(view = (C, M), \mathcal{S})\text{:}}$
  // $M$ already contains the decryptions under the adversarial keys
  $R := M$
  for $(N, e, d) \in \mathcal{S}$:
    $R[(N, e)] := \mathsf{RSA.Dec}((N, e, d), C)$
  return $R$
```

We now give a hybrid proof that the real world, where (C, R, M) are generated with ENCRYPT, is indistinguishable from the ideal world, where they are generated by AnonSim and AnonView. Intermediate hybrids run in exponential time, but the bounds in the proof hold even with respect to unbounded adversaries.

1. Start with the real world, and in ENCRYPT separate the honestly generated keys $\mathsf{PK}_{hon} = \mathsf{PK} \cap \mathsf{PK}^*$ from the adversarial keys $\mathsf{PK}_{adv} = \mathsf{PK} \setminus \mathsf{PK}^*$. Treat the adversarial keys as in the real world. For the honest keys $(N, e) \in \mathsf{PK}_{hon}$, compute r and c in reverse order. That is, first sample $c \leftarrow [0, N) \cap \mathbb{Z}$ and then compute r such that $r^e = c \bmod N$ using an exponential-time algorithm. This is indistinguishable from the real world because $(\cdot)^e$ is a bijection on $\mathbb{Z}/N\mathbb{Z}$.

2. In the previous hybrid, for honest keys in PK_{hon} we first sample c, then calculate r and $\mathsf{Chk}_N(c)$. In the next hybrid, we first sample $c_0, \ldots, c_{s(N)-1} = \mathsf{Chk}_N(c)$ uniformly (each component from the range $[0, 2^{2\lambda})$), then compute $c = \mathsf{Unchk}_N(c_0, \ldots, c_{s(N)-1})$, then solve for r.

   Chk_N and Unchk_N can be viewed as bijections between $[0, 2^{2\lambda})^{s(N)}$ and $[0, 2^{2\lambda s(N)})$, but Chk_N does not induce a uniform distribution over its range. Indeed, sometimes the c' value that it generates can overflow the range. Let $M = 2^{2\lambda s(N)} \bmod N$. The probability of that bad event ($c' \geq 2^{2\lambda s(N)}$) is $\frac{M}{2^{2\lambda s(N)}}(1 - \frac{M}{N})$, since it occurs iff $p \geq 2^{2\lambda s(N)} - M$ and $c \geq M$. This quadratic is maximized at $M = N/2$, so it is upper bounded by $\frac{N^2}{2^{2\lambda s(N)+2} N} < 2^{-\lambda-2}$. Taking a union bound over all calls to Chk gives the first term of the distinguisher bound in the lemma statement.

   Suppose we abort when the bad event occurs, then Chk_N induces a uniform distribution over its range. In this case, sampling the output of Chk_N first and then solving for its input (with Unchk_N) induces an identical distribution.

3. In this hybrid, sample the c_i's uniformly in $\mathbb{F}$, instead of $[0, 2^{2\lambda})$. The distinguisher has advantage at most $\frac{\left||\mathbb{F}|-2^{2\lambda}\right|}{\max(|\mathbb{F}|, 2^{2\lambda})} \leq \frac{\left||\mathbb{F}|-2^{2\lambda}\right|}{2^{2\lambda}} = \left|2^{-2\lambda}|\mathbb{F}| - 1\right|$, since sampling from $\mathbb{F}$ and $[0, 2^{2\lambda}) \cap \mathbb{Z}$ can only be distinguished when the sample from the bigger set is outside the subset of values that correspond to the smaller set. Multiply this advantage by t_S, to handle all the c_i.

4. In the previous hybrid, we interpolate a polynomial C through a set of points corresponding to honest $\mathsf{RSA.Enc1}$ ciphertexts for adversarial keys, plus a random collection of other points, so that there are L total points of interpolation. Then for each honest key (N, e), we compute its $\mathsf{RSA.Enc1}$ ciphertext c with Unchk_N, and solve for its plaintext r-value by inverting RSA in exponential time.

   If the secret key d for (N, e) were available at the time of this last step, then we could instead compute r in polynomial time as $r = c^d \bmod N$. But this corresponds exactly to the situation in the ideal world interaction. Namely, C is generated first, and then later when the honest secret keys are available, we compute their associated plaintext values. □

A.4 Mixed-flavor KEM

Lemma 12. MixKEM *is a jointly secure MKEMSS if every flavor in* $\mathsf{FLAVORS}$ *is.*

Proof. Let $\mathcal{A}$ be an adversary against MixKEM for some generation options $(\mathsf{f}, \mathsf{opts}) \in \mathsf{MixKEM.FLAVORS}$. Construct an adversary $\mathcal{A}'$ against the security of the MKEMSS f. $\mathcal{A}'$ will run $\mathcal{A}$, passing through the public key and the oracles CHECK and SIGN. The last oracle, ENCRYPT, is simulated as:

```
ENCRYPT($\mathsf{PK}$):
  $(C, R) \leftarrow \mathsf{MixKEM.Enc}(\mathsf{PK})$
  $\mathsf{PK_f} := \{\mathsf{pk} \mid (\mathsf{f}, \mathsf{pk}) \in \mathsf{PK}\}$
  $(C[\mathsf{f}], r) \leftarrow \mathsf{f}.\text{ENCRYPT}(\mathsf{PK_f})$
  return $C$
```

It is almost the same as the real ENCRYPT oracle for MixKEM, except that it generates the ciphertext for the key flavor f using the encryption oracle from the security game for f. Therefore, if $\mathcal{A}$ manages to guess a decryption then $\mathcal{A}'$ can immediately output that decryption and win the security game against f. Similarly, if $\mathcal{A}$ manages to forge a signature with pk then $\mathcal{A}'$ can immediately use that signature to win. The simulated oracles are indistinguishable from the real ones, so $\mathcal{A}$ and $\mathcal{A}'$ have the same advantage against their respective games. □

Lemma 13. MixKEM *is anonymous, assuming that every* $\mathsf{f} \in \mathsf{FLAVORS}$ *is, with advantage is bounded by the total advantage against all the individual flavors' anonymities.*

Proof. Simulate the ciphertexts by running the simulators of the individual multi-KEM flavors. The leakage L tells us what flavors are needed, as well as their individual leakages.

```
$\mathsf{MixKEM}.\mathcal{S}(L, \mathsf{PK}_{adv})$:
  $F := \{\mathsf{f} \mid (\mathsf{f}, L_\mathsf{f}) \in L\}$
  $C :=$ empty map
  for $\mathsf{f} \in F$:
    $\mathsf{PK_f} := \{\mathsf{pk} \mid (\mathsf{f}, \mathsf{pk}) \in \mathsf{PK}_{adv}\}$
    find unique $L_\mathsf{f}$ s.t. $(\mathsf{f}, L_\mathsf{f}) \in L$
    $c \leftarrow \mathsf{f}.\mathcal{S}(L_\mathsf{f}, \mathsf{PK_f})$
    $C[\mathsf{f}] := c$
  return $C$
```

Number the flavors in $\mathsf{FLAVORS}$ as $\mathsf{f}_1, \ldots, \mathsf{f}_N$. We give a hybrid proof with $N+1$ hybrids — one change for each flavor. Let hybrid 0 be the real world, where C is generated with ENCRYPT. In hybrid i, the ciphertexts $C[\mathsf{f}_1], \ldots, C[\mathsf{f}_i]$ are sampled with the simulator $\mathsf{f}.\mathcal{S}$, while the remaining ciphertexts $C[\mathsf{f}_{i+1}], \ldots, C[\mathsf{f}_N]$ are still real encryptions from $\mathsf{f.Enc}$. The change from hybrid $i-1$ to i consists of replacing a real ciphertext $C[\mathsf{f}_i]$ with its simulation, and so reduces to the anonymity of f_i. Hybrid N is the ideal world, where C is generated with $\mathsf{MixKEM}.\mathcal{S}$. □

A.5 Adaptive Security

To prove adaptive security of our authentication protocol, we use an adaptive variant of wCCA security (and hence joint MKEMSS security) of an MKEM. Recall that in MKEM-wCCA security, the adversary can request challenge ciphertexts encrypted to the challenge public key (among other keys). The adversary wins the game by predicting $\mathsf{Dec}(\mathsf{sk}^*, c)$ for one of these challenge ciphertexts c.

Challenge ciphertexts are generated as $(c, r) \leftarrow \mathsf{Enc}(\cdots)$. In the adaptive variant of this game, the adversary can ask for any challenge ciphertext's associated r to be revealed. Of course, this makes it trivial to compute $\mathsf{Dec}(\mathsf{sk}^*, c) = \mathsf{Msg}(\mathsf{pk}^*, r)$ by the correctness of the MKEM. So the game no longer considers this c a valid challenge ciphertext — *i.e.*, the adversary wins by predicting $\mathsf{Dec}(\mathsf{sk}^*, c)$ for one of the challenge ciphertext *whose corresponding r was not revealed.*

Definition 16. *A multi-KEM* MKEM *is* ***adaptively secure against chosen ciphertext attacks*** *if for all* $\mathsf{opts} \in \mathsf{MKEM.OPTS}$*, every PPT adversary $\mathcal{A}$ has negligible probability of winning the game:*

```
$R :=$ empty
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$

$\underline{\textsc{encrypt}(\mathsf{PK})}$:
  $(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}^*\} \cup \mathsf{PK})$
  $R[c] := r$
  return $c$

$\underline{\textsc{guess\_dec}(c, m)}$:
  return $\mathsf{MKEM.Dec}(\mathsf{sk}^*, c) \overset{?}{=} m$

$\underline{\textsc{guess\_msg}(c, \mathsf{pk}, m)}$:
  if $R[c]$ defined:
     return $\mathsf{MKEM.Msg}(\mathsf{pk}, R[c]) \overset{?}{=} m$

 ┌──────────────────────┐
 │ $\underline{\textsc{open}(c)}$:             │
 │   $r = R[c]$                 │
 │   $R[c] :=$ undefined        │
 │   return $r$                 │
 └──────────────────────┘

$(c, m) \leftarrow \mathcal{A}^{\textsc{encrypt},\textsc{guess\_dec},\textsc{guess\_msg},\boxed{\textsc{open}}}(\mathsf{pk}^*)$
win if $R[c]$ defined $\wedge$ $\mathsf{MKEM.Dec}(\mathsf{sk}^*, c) = m$
```

Adaptive **joint security** of a MKEMSS is defined analogously, following the pattern of Definition 7.

Static security implies adaptive: Any scheme satisfying the static MKEM definition also satisfies the adaptive definition, although with some security loss.

Define the **concurrency** of an adversary in the adaptive wCCA game as the maximum number of valid challenge ciphertexts at any time — *i.e.*, the maximum number of values stored in the map R at any time.

Lemma 17. *If an MKEM scheme satisfies static wCCA security (Definition 5) then it also satisfies the adaptive variant. For any adversary $\mathcal{A}$, its advantage in attacking the adaptive game is bounded by* $\mathrm{Adv}[\mathcal{A}] \leq N \cdot \mathrm{Adv}[\mathcal{A}']$*, where $\mathcal{A}'$ is an adversary with roughly double the running time of $\mathcal{A}$, attacking the static game, and N is an upper bound on the concurrency of $\mathcal{A}$.*

Proof. Let $\mathcal{A}$ be an adversary attacking the adaptive security game. For each challenge ciphertext c requested during the game, assign it a label $\mathsf{lbl}(c) \in \{1, \ldots, N\}$ as follows: Start the game with a pool $\{1, \ldots, N\}$ of available labels. When a challenge ciphertext is generated, assign it the smallest label in the pool, and remove that label from the pool. When the adversary calls OPEN on any challenge ciphertext c, return its label to the pool. This process always assigns a unique label to all challenge ciphertexts, if N is indeed an upper bound on the concurrency of the adversary.

Recall that $\mathcal{A}$ wins the adaptive wCCA game if it predicts $\mathsf{Dec}(\mathsf{sk}^*, c)$ for an active challenge ciphertext c. We now construct $\mathcal{A}'$, an adversary in the static wCCA game satisfying $\mathrm{Adv}[\mathcal{A}'] \geq \frac{1}{N}\,\mathrm{Adv}[\mathcal{A}]$. Our $\mathcal{A}'$ proceeds by first guessing the label $\mathsf{lbl}^* \leftarrow \{1, \ldots, N\}$ of the challenge ciphertext that $\mathcal{A}$ will use to win the game. $\mathcal{A}'$ will run $\mathcal{A}$, ensuring that $\mathcal{A}$'s view is independent of the guess lbl^*, and that it is distributed identically to the adaptive game. Then we will show that $\mathcal{A}'$ wins in the static game whenever $\mathcal{A}$ wins in the adaptive game using a ciphertext with label lbl^*. This will establish the desired bound.

$\mathcal{A}'(\mathsf{pk}^*)$ internally runs $\mathcal{A}(\mathsf{pk}^*)$ and manages its oracle queries as follows:

- When $\mathcal{A}$ calls $\textsc{encrypt}(\mathsf{PK})$, determine the label of the resulting ciphertext. If it will have label lbl^*, then call our own $\textsc{encrypt}(\mathsf{PK})$ oracle and return the result. If it will not have label lbl^*, then generate the challenge ciphertext ourselves as $(c, r) \leftarrow \mathsf{Enc}(\mathsf{PK})$; store $R[c] = r$.

- When $\mathcal{A}$ calls $\textsc{guess\_dec}(c, m)$: If $\mathsf{lbl}(c) \neq \mathsf{lbl}^*$ then we know $R[c]$ and can compute the result as $m \stackrel{?}{=} \mathsf{Msg}(\mathsf{pk}^*, R[c])$. If $\mathsf{lbl}(c) = \mathsf{lbl}^*$ (or if c has no label) then forward this query to our own $\textsc{guess\_dec}$ oracle and return the result. This equals $m \stackrel{?}{=} \mathsf{Dec}(\mathsf{sk}^*, c)$ by the correctness of the MKEM.

- When $\mathcal{A}$ calls $\textsc{guess\_msg}(c, \mathsf{pk}, m)$: If $\mathsf{lbl}(c) \neq \mathsf{lbl}^*$ then we know $R[c]$ and can compute the result as $m \stackrel{?}{=} \mathsf{Msg}(\mathsf{pk}^*, R[c])$. If $\mathsf{lbl}(c) = \mathsf{lbl}^*$ (or if c has no label) then forward this query to our own $\textsc{guess\_msg}$ oracle and return the result.

In this way, $\mathcal{A}'$ manages everything to do with non-lbl^* challenge ciphertexts, but lets the external static wCCA security game manage the lbl^* challenge ciphertexts.

What happens when $\mathcal{A}$ calls its $\textsc{open}$ oracle on a challenge ciphertext c? If $\mathsf{lbl}(c) \neq \mathsf{lbl}^*$ then $\mathcal{A}'$ knows the corresponding $R[c]$ and can return it as the oracle response. The problem is when $\mathcal{A}'$ requests that a lbl^*-labeled ciphertext be opened.

To handle this, we modify how lbl^*-labeled challenge ciphertexts are generated. Whenever $\mathcal{A}$ calls $\textsc{encrypt}$ and the result will have label lbl^*, save the current state of $\mathcal{A}$ and then toss a random coin b. This b will represent a guess of whether this particular challenge ciphertext will be the "winning one" or it will be eventually $\textsc{open}$'ed (the two possibilities are mutually exclusive).

If $b = 0$ then handle this ciphertext c locally (as if it did not have label lbl^*). If $\mathcal{A}$ later asks for this ciphertext to be $\textsc{open}$'ed or $\mathcal{A}$ terminates the game without winning, then we can respond correctly since we know the corresponding r. If instead, $\mathcal{A}$ later satisfies the winning predicate for the adaptive game using this c, then it does not represent a win for $\mathcal{A}'$ in the static game, because c is not actually a challenge ciphertext in the static game. In this case, rewind to the saved state and completely resample a fair coin b. Proceed as usual, with fresh randomness.

If $b = 1$ then delegate the handling of this ciphertext c to the static-wCCA game oracles, as described above. If $\mathcal{A}$ later satisfies the winning predicate for its adaptive game using c, then the same guess will satisfy the winning predicate for $\mathcal{A}'$ in the static game. If instead, $\mathcal{A}$ later asks for $\textsc{open}(c)$ then $\mathcal{A}'$ cannot respond correctly, since c was generated by the external static-wCCA game which does not provide a way to obtain the corresponding r. In this case, rewind to the saved state and completely resample a fair coin b. Proceed as usual, with fresh randomness. We should similarly rewind if $\mathcal{A}$ terminates the game without winning or opening c, since $b = 1$ corresponds to a guess that $\mathcal{A}$ will win the game using this ciphertext. Rewinding only affects the internal simulation of $\mathcal{A}$ but is not visible to the static wCCA game in which $\mathcal{A}'$ plays. After rewinding, the ciphertext c is still considered a challenge ciphertext in the external static wCCA game; however, $\mathcal{A}'$ will never refer to it again.

The important features of this strategy for $\mathcal{A}'$ are:

- Even after any amount of rewinding, $\mathcal{A}$'s view is distributed identically to its view in the adaptive game. This is because each rewind uses fresh randomness.
- $\mathcal{A}$'s view is independent of b, and therefore the probability that $\mathcal{A}'$ will need to rewind is $1/2$, for each time b is sampled.
- There is only one challenge ciphertext with label lbl^* at a time. This implies that "rewindings" are not nested or recursive. The only possible rewinding is to the point where the *most recent lbl^*-labeled challenge ciphertext is generated.*

From these observations, we have that $\Pr[\mathcal{A}' \text{ wins}] \geq \frac{1}{N} \Pr[\mathcal{A} \text{ wins}]$, as desired. The expected running time of $\mathcal{A}'$ is on average twice that of $\mathcal{A}$, since each challenge ciphertext leads to 1 rewinding in expectation.

Alternatively, we can have $\mathcal{A}'$ abort if a single call to ENCRYPT leads to more than λ consecutive rewindings. This leads to an $\mathcal{A}'$ with *strict* polynomial time at most λ times that of $\mathcal{A}$, and $\mathrm{Adv}[\mathcal{A}'] \geq \frac{1}{N} \mathrm{Adv}[\mathcal{A}] - q2^{-\lambda}$, where $\mathcal{A}$ calls ENCRYPT at most qN times. □

B Proof of Main Protocol

We prove our authentication protocol secure against *adaptive* adversaries. This proof uses the adaptive variant of our MKEM-CCA security definition, presented in Section A.5. In short, we modify the game in Definition 5 to provide a way for the adversary to adaptively reveal the r value for challenge ciphertexts.

In Lemma 17 we showed that the static CCA property implies the adaptive one (with some security loss). Our main security theorem below is stated in terms of the static CCA definition.

Theorem 14. *The protocol in Figure 4 is an adaptively UC-secure protocol realizing ideal functionality $\mathcal{F}_{\textsf{new-auth}}$ (Figure 2), assuming that* MKEM *is anonymous (Definition 6) and joint-MKEMSS secure (Definition 7). The advantage of any distinguisher $\mathcal{A}$ between the real protocol and its simulation is bounded by*

$$\mathrm{Adv}[\mathcal{A}] \leq 2^{-2\lambda-1} N_{\mathrm{auth}}^2 + \sum_{\mathsf{pk}} \max(N_{\mathsf{pk}}, 1)\,\mathrm{Adv}[\mathcal{A}'] + \mathrm{Adv}[\mathcal{A}''] + N_{\mathrm{auth}}\,\mathrm{Adv}[\mathcal{A}'''],$$

where $\mathcal{A}'$, $\mathcal{A}''$, and $\mathcal{A}'''$ are adversaries against the joint security, anonymity, and correctness of MKEM*, respectively. Here, N_{auth} is the total PSI input set size across all evaluations of* auth*, and N_{pk} is the maximum number of* concurrent auth *attempts in which honest servers use the public key* pk.

Simulator. We start by describing the simulator. Only the auth protocol is non-trivial to simulate — the simulator never even finds out when the signature-related commands are executed. The pseudocode for the auth simulator is given in Figure 7. The simulator needs to play many roles to fake the protocol's execution. For clarification, every action is annotated by which entity the simulator is acting as.

Although we prove adaptive security, the simulator is still separated into the two main cases: honest server and honest client. Each simulates the behavior of the honest party, as seen by the corrupted party. To handle all cases for when corruption might occur, the high level control flow of the simulator works as follows.

```
when $(\text{auth}_1, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), \cdot)$ starts:
  $P_{\mathsf{C}}$: get leakage $(L, \mathsf{PK}_{unreg})$ from $\mathcal{F}_{\textsf{new-auth}}$
       $(c, M^*, view) \leftarrow \mathsf{MKEM.AnonSim}(L, \mathsf{PK}_{unreg})$

       // if $P_{\mathsf{S}}$ adaptively corrupted during this auth session,
       // run this subroutine to simulate $P_{\mathsf{S}}$'s internal state
       ON_CORRUPT($K_{\mathsf{S}}$):
         $\mathsf{SK} := \{\}$
         for $\mathsf{pk} \in K_{\mathsf{S}}$:
  $\mathcal{A}$:       send $(\text{get\_sk}, \mathsf{pk})$ to $\mathcal{F}_{\textsf{new-auth}}$; receive $\mathsf{sk}$
           $\mathsf{SK} := \mathsf{SK} \cup \{\mathsf{sk}\}$
         $r \leftarrow \mathsf{AnonView}(view, \mathsf{SK})$
  $P_{\mathsf{S}}$:     give $r$ and $K_{\mathsf{S}}$ to the adversary
  $P_{\mathsf{S}}$: send $c$ to $P_{\mathsf{C}}$

  $\mathcal{F}_{\textsf{psi+}}$: receive $M_{\mathsf{C}}$ from $P_{\mathsf{C}}$
       for $\langle \mathsf{pk}, m \rangle \in M_{\mathsf{C}}$:
  $\mathcal{A}$:     send $(\text{get\_sk}, \mathsf{pk})$ to $\mathcal{F}_{\textsf{new-auth}}$; receive $\mathsf{sk}$
         if $\mathsf{sk} \neq \bot$: $M^*[\mathsf{pk}] = \mathsf{MKEM.Dec}(\mathsf{sk}, c)$
       $\tilde{K}_{\mathsf{C}} = \{\mathsf{pk} \mid \langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \in M_{\mathsf{C}}\}$
  $P_{\mathsf{C}}$: send $(\text{auth}_2, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), \tilde{K}_{\mathsf{C}})$ to $\mathcal{F}_{\textsf{new-auth}}$

  $P_{\mathsf{C}}$: receive $A$ and $|K_{\mathsf{S}}|$ from $\mathcal{F}_{\textsf{new-auth}}$
  $\mathcal{F}_{\textsf{psi+}}$: send $|K_{\mathsf{S}}|$ and $\{\langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \mid \mathsf{pk} \in A\}$ to $P_{\mathsf{C}}$

  $\mathcal{F}_{\textsf{psi+}}$: receive $(\text{deliver}, ssid, d)$ from $P_{\mathsf{C}}$
  $P_{\mathsf{C}}$: send $(\text{deliver}, ssid, d)$ to $\mathcal{F}_{\textsf{new-auth}}$
```

(a) Corrupt client; honest server.

```
when $P_{\mathsf{S}}$ sends $c$ to $P_{\mathsf{C}}$ for session $ssid$:
  $P_{\mathsf{S}}$: send $(\text{auth}_1, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), \emptyset)$ to $\mathcal{F}_{\textsf{new-auth}}$
  $P_{\mathsf{S}}$: receive $|K_{\mathsf{C}}|$ from $\mathcal{F}_{\textsf{new-auth}}$
  $\mathcal{F}_{\textsf{psi+}}$: send $|K_{\mathsf{C}}|$ to $P_{\mathsf{S}}$

  $\mathcal{F}_{\textsf{psi+}}$: receive $M_{\mathsf{S}}$ from $P_{\mathsf{S}}$
       $M^* :=$ empty
       for $(\mathsf{pk}, m) \in M_{\mathsf{S}}$:
  $\mathcal{A}$:     send $(\text{get\_sk}, \mathsf{pk})$ to $\mathcal{F}_{\textsf{new-auth}}$; receive $\mathsf{sk}$
         if $\mathsf{sk} \neq \bot$: $M^*[\mathsf{pk}] := \mathsf{MKEM.Dec}(\mathsf{sk}, c)$
       $\tilde{K}_{\mathsf{S}} := \{\mathsf{pk} \mid \langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \in M_{\mathsf{S}}\}$
  $P_{\mathsf{S}}$: send $(\text{auth}_3, (P_{\mathsf{S}}, P_{\mathsf{C}}, ssid), \tilde{K}_{\mathsf{S}})$ to $\mathcal{F}_{\textsf{new-auth}}$

  $P_{\mathsf{S}}$: receive $e \in \{0, 1\}$ from $\mathcal{F}_{\textsf{new-auth}}$
  $\mathcal{F}_{\textsf{psi+}}$: send $e$ to $P_{\mathsf{S}}$
```

(b) Corrupt server; honest client.

Figure 7: Simulators for adaptive security of Figure 4. The simulator plays a number of roles in the protocol, so each action is annotated by who the simulator is pretending to be when it takes the action.

- When an auth interaction starts, check the corruption status of both the client and the server. Nothing needs to be done when they are both honest.
- When one of the parties gets corrupted, we need to generate a state to send to the adversary. The easiest way to do this is to run the corresponding simulator from Figure 7 from the start. That is, simulate the protocol transcript that would have occurred if that party had been corrupted since the start of auth, but followed the honest protocol anyway.[10] This produces states for both the corrupted party and the simulator of the honest party.
- While only one party is corrupted, run the corresponding simulator in Figure 7.
- If the other party gets adaptively corrupted during the auth session, the simulator needs to simulate an internal state for that party as well. The only place in the protocol where parties maintain nontrivial state is that honest servers store the value r for the duration of an auth session. The simulator description therefore includes a procedure ON_CORRUPT which generates r if needed during that time interval.

[10]During an auth session, the ideal functionality gives various outputs only to corrupt parties. We assume that if a party is adaptively corrupted *during the execution* of an auth session, then it receives this leakage from $\mathcal{F}_{\textsf{new-auth}}$ retroactively.

- Once both parties are corrupted, the simulator only needs to pass messages back and forth between the two corrupted parties.

Corrupt client. We first consider the case of a client who is corrupt at the beginning of an auth session. We show that the real and ideal interactions are indistinguishable, via a sequence of hybrid interactions:

Real interaction: The adversary interacts with honest parties running the protocol, and $\mathcal{F}_{\mathsf{psi+}}$. When an adversary adaptively corrupts a party, it receives a history of their inputs from / outputs to the environment, as well as their private internal state. The only case of a party holding private internal state is an honest server holding state r during an auth session.

Hybrid 0: Same as the real interaction, except that whenever an honest party runs genkey, the resulting pk is added to a set Secure. Whenever the adversary corrupts P_i's storage, remove the corresponding pk's from Secure. Furthermore, whenever an honest party runs $(\text{sign}, \mathsf{pk}, m)$, add (pk, m) to a set Σ. The adversary's view in this hybrid is identical to the real interaction.

Of particular interest in this interaction:

- In every auth session with a corrupt P_{C} and initially honest P_{S}, c is generated honestly as $(c, r) \leftarrow \mathsf{Enc}(K_{\mathsf{S}})$.
- In every auth session with a corrupt P_{C} and initially honest P_{S}, the client's main output from $\mathcal{F}_{\mathsf{psi+}}$ is computed equivalently to the following:

```
for $\mathsf{pk} \in K_{\mathsf{S}}$:
    $M^*[\mathsf{pk}] = \mathsf{Msg}(\mathsf{pk}, r)$
$I := M_{\mathsf{C}} \cap \underbrace{\{\langle \mathsf{pk}, M^*[\mathsf{pk}]\rangle \mid \mathsf{pk} \in K_{\mathsf{S}}\}}_{M_{\mathsf{S}}}$
```

- Each time an honest party runs $(\text{verify}, \mathsf{pk}, m, \sigma)$, the result is computed using $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$.
- If the adversary adaptively corrupts the server during an auth session, the adversary gets to learn the server's internal state r.

Hybrid 1: Modify the previous hybrid so that the client's $\mathcal{F}_{\mathsf{psi+}}$ output is calculated as:

```
for $\mathsf{pk} \in K_{\mathsf{S}}$:
    $M^*[\mathsf{pk}] = \mathsf{Msg}(\mathsf{pk}, r)$
$I := M_{\mathsf{C}} \cap \{\langle \mathsf{pk}, M^*[\mathsf{pk}]\rangle \mid \mathsf{pk} \in K_{\mathsf{S}} \underline{\setminus \mathsf{Secure}}\}$
```

Additionally, each time an honest party runs $(\text{verify}, \mathsf{pk}, m, \sigma)$, the result is computed as:

```
if $\mathsf{pk} \in \mathsf{Secure}$ and $(\mathsf{pk}, m) \notin \Sigma$: return false
else return $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$
```

Below in Lemma 18 we show that hybrids 0 & 1 are indistinguishable, using a reduction to adaptive joint MKEMSS security. Intuitively, the hybrids are *identical-until-bad*, where the bad event corresponds to an adversary guessing the decryption of an honest ciphertext under a secure key, or forging a signature under a secure key. These are precisely the events that joint MKEMSS security says happen with negligible probability.

Hybrid 2: Modify the previous hybrid so that the client's $\mathcal{F}_{\mathsf{psi+}}$ output is calculated as:

```
for $\mathsf{pk} \in K_\mathsf{S} \setminus \mathsf{Secure}$:
    if $\mathsf{pk}$ unregistered:
        $M^*[\mathsf{pk}] = \mathsf{Msg}(\mathsf{pk}, r)$
    if $\mathsf{pk}$ registered (to party $P_i$):
        $M^*[\mathsf{pk}] = \mathsf{Dec}(\mathsf{SK}_i[\mathsf{pk}], c)$
$I := M_\mathsf{C} \cap \{\langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \mid \mathsf{pk} \in K_\mathsf{S} \setminus \mathsf{Secure}\}$
```

The only difference between the hybrids is how $M^*[\mathsf{pk}]$ is computed for an honestly generated key — whether via Msg or Dec. The adversary's view is identical in these two hybrids by the correctness of the MKEMSS scheme.

Hybrid 3: Modify the previous hybrid as follows. Now c is generated as follows:

$$L = \mathsf{Leak}(K_\mathsf{S})$$
$$(c, M^*, \mathit{view}) \leftarrow \mathsf{AnonSim}(L, K_\mathsf{S} \setminus \{\mathsf{pk} \mid \mathsf{pk} \text{ registered}\})$$

And the client's $\mathcal{F}_{\mathsf{psi+}}$ output is computed as follows:

```
// $M^*$ already initialized above
for $\mathsf{pk} \in K_\mathsf{S} \setminus \mathsf{Secure}$:
    if $\mathsf{pk}$ registered (to party $P_i$):
        $M^*[\mathsf{pk}] = \mathsf{Dec}(\mathsf{SK}_i[\mathsf{pk}], c)$
$I := M_\mathsf{C} \cap \{\langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \mid \mathsf{pk} \in K_\mathsf{S} \setminus \mathsf{Secure}\}$
```

If the adversary adaptively corrupts the honest server between sending c and contacting $\mathcal{F}_{\mathsf{psi+}}$, then the simulator computes r as follows:

$$r \leftarrow \mathsf{AnonView}(\mathit{view}, \{\mathsf{SK}_i[\mathsf{pk}] \mid \mathsf{pk} \in K_\mathsf{S} \text{ registered to } P_i\})$$

The two hybrids are indistinguishable by a straight-forward reduction to the MKEM anonymity property.

Hybrid 4: In the previous hybrid, a pair $\langle \mathsf{pk}, m \rangle$ can only be included in I if: (1) $\mathsf{pk} \notin \mathsf{Secure}$. Note that since P_C is corrupt, $\mathsf{pk} \notin \mathsf{Secure} \Leftrightarrow \text{can\_use}(P_\mathsf{C}, \mathsf{pk})$. (2) $\mathsf{pk} \in K_\mathsf{S}$; (3) $m = \mathsf{Dec}(\mathsf{sk}, c)$ for the sk that corresponds to pk.

Modify the previous hybrid to compute the client's $\mathcal{F}_{\mathsf{psi+}}$ output I as follows:

```
// $M^*$ already initialized above
for all $\mathsf{pk}$ such that $(\mathsf{pk}, \cdot) \in M_\mathsf{C}$:
    if $\mathsf{pk}$ registered (to party $P_i$):
        $M^*[\mathsf{pk}] = \mathsf{Dec}(\mathsf{SK}_i[\mathsf{pk}], c)$

$\tilde{K}_\mathsf{C} = \{\mathsf{pk} \mid \langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \in M_\mathsf{C}\}$
$U := \{\mathsf{pk} \mid \text{can\_use}(P_\mathsf{C}, \mathsf{pk})\}$
$I := \{\langle \mathsf{pk}, M^*[\mathsf{pk}] \rangle \mid \mathsf{pk} \in K_\mathsf{S} \cap \tilde{K}_\mathsf{C} \cap U\}$
```

I is computed in a slightly different, but logically equivalent way, so the adversary's view is identical in the two hybrids.

Ideal interaction. We conclude the proof by observing that Hybrid 4 is identically distributed to the ideal interaction involving the simulator. In particular:

- Server's message c is generated via $\mathsf{AnonSim}$ using the leakage provided by $\mathcal{F}_{\mathsf{new\text{-}auth}}$.

- If the adversary adaptively corrupts the honest server between sending c and contacting $\mathcal{F}_{\mathsf{psi+}}$, the adversary gets r that is computed using $\mathsf{AnonView}$ and the secret keys corresponding to the public keys in K_{S}.

- When the simulator obtains M_{C} from the adversary, it computes $\tilde{K}_{\mathsf{C}}$ and sends it to $\mathcal{F}_{\mathsf{new\text{-}auth}}$. The functionality responds with $K_{\mathsf{S}} \cap \tilde{K}_{\mathsf{C}} \cup \{\mathsf{pk} \mid \text{can\_use}(P_{\mathsf{C}}, \mathsf{pk})\}$, and the simulator can generate I as a function of this response.

- Whenever the adversary corrupts P_i's key storage, the associated keys are removed from Secure, corresponding to the simulator calling stealkeys.

Corrupt server. We first consider the case of a client who is corrupt at the beginning of an auth session. We show that the real and ideal interactions are indistinguishable, via a sequence of hybrid interactions:

Real interaction: The adversary interacts with honest parties running the protocol, and $\mathcal{F}_{\mathsf{psi+}}$. Notably, the honest client's primary output (to the environment) from the auth-protocol is computed by the following sequence:

```
// within honest client:
$M_{\mathsf{C}} := \{\langle \mathsf{pk}, \mathsf{Dec}(\mathsf{SK}_{\mathsf{C}}[\mathsf{pk}], c)\rangle \mid \mathsf{pk} \in K_{\mathsf{C}}\}$
// within $\mathcal{F}_{psi+}$:
$I := M_{\mathsf{C}} \cap M_{\mathsf{S}}$
// within honest client:
output $\{\mathsf{pk} \mid \exists m : \langle \mathsf{pk}, m\rangle \in I\}$
```

Hybrid 0: In the previous hybrid, a value pk is included in the honest client's output iff: (1) pk is registered to the client P_{C}; (2) $\langle \mathsf{pk}, \mathsf{Dec}(\mathsf{sk}, c)\rangle \in M_{\mathsf{S}}$, for the correct sk that corresponds with pk; (3) $\mathsf{pk} \in K_{\mathsf{C}}$. Hence, an equivalent way of computing $\mathcal{F}_{\mathsf{psi+}}$ output A is as follows:

```
$M^* :=$ empty
for each $\langle \mathsf{pk}, m\rangle \in M_{\mathsf{S}}$:
    if $\exists i : \mathsf{SK}_i[\mathsf{pk}]$ defined: $M^*[\mathsf{pk}] := \mathsf{Dec}(\mathsf{SK}_i[\mathsf{pk}], c)$
$\tilde{K}_{\mathsf{S}} := \{\mathsf{pk} \mid \langle \mathsf{pk}, M^*[\mathsf{pk}]\rangle \in M_{\mathsf{S}}\}$
$U := \{\mathsf{pk} \mid \mathsf{SK}_{\mathsf{C}}[\mathsf{pk}] \text{ defined}\} = \{\mathsf{pk} \mid \text{can\_use}(P_{\mathsf{C}}, \mathsf{pk})\}$
output $K_{\mathsf{C}} \cap \tilde{K}_{\mathsf{S}} \cap U$
```

The adversary's view is identical in these two hybrids, since the only difference is a value being computed in a different but logically equivalent way.

Ideal interaction. We conclude by simply observing that the previous hybrid exactly matches what happens in the ideal interaction. In particular, the simulator upon seeing M_{S} computes M^* and $\tilde{K}_{\mathsf{S}}$ as above. Then after the simulator sends an auth$_2$ command to $\mathcal{F}_{\mathsf{new\text{-}auth}}$, it delivers $K_{\mathsf{C}} \cap \tilde{K}_{\mathsf{S}} \cap \{\mathsf{pk} \mid \text{can\_use}(P_{\mathsf{C}}, \mathsf{pk})\}$ to the honest client.

Wrapping up. Overall, we have shown that the real and ideal worlds are indistinguishable. To do so, we have invoked reductions to the MKEM anonymity and *adaptive* joint-MKEMSS security properties. The anonymity property is used only for one hybrid, but in Lemma 18 the adaptive joint-MKEMSS security property is used once for each honestly generated public key. When we invoke adaptive joint-MKEMSS security, we do so with an MKEMSS adversary whose concurrency is bounded by the maximum number of concurrent auth attempts involving that key in our protocol. Applying Lemma 17, we see that such adaptive MKEMSS security reduces to static MKEMSS security, with a multiplicative security loss equal to the concurrency. Combining all of the losses from all hybrid steps finally yields the result and bounds from Theorem 14.

B.1 Supporting Lemma

Lemma 18. *Hybrids 0 & 1 (in the case of initially corrupt client) are indistinguishable if* MKEM *satisfies* adaptive *joint MKEMSS security (Section A.5).*

Proof. In the terminology of code-based games [BR06], these two hybrids are *equivalent-until-bad*, with the bad event being:

- A corrupt client has produced a value $\langle \mathsf{pk}, \mathsf{Dec}(\mathsf{sk}, c)\rangle$ where c was honestly generated by an honest server, $(\mathsf{pk}, \mathsf{sk})$ is honestly generated, and $\mathsf{pk} \in \mathsf{Secure}$ at the time; or
- The adversary has produced a tuple (pk, m, σ) where $\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 1$, pk was generated honestly, $\mathsf{pk} \in \mathsf{Secure}$, and $(\mathsf{pk}, m) \notin \Sigma$ meaning that the honest owner of pk has not yet generated a signature on m.

Note that in the first case, the bad event in Hybrid 0 is defined with respect to computing the client's $\mathcal{F}_{\mathsf{psi+}}$ output when the server is honest. A bad event is not possible (for a given auth session) if the server is adaptively corrupted before the client receives $\mathcal{F}_{\mathsf{psi+}}$ output.

The bad event always happens with respect to a public key pk, so we may consider the bad event to be a union of bad sub-events, each with respect to a different pk (more specifically, with respect to the ith honest keypair generated, for a specific i). The probability of the overall bad event is the sum of probabilities of all bad sub-events. Hence, we focus on bounding the probability that the above bad event happens with respect to a particular keypair, say the ith one.

The reduction to adaptive joint MKEMSS security is rather direct. The reduction algorithm plays as an adversary in the MKEMSS security game. In that game, a challenge keypair $(\mathsf{pk}^*, \mathsf{sk}^*)$ is chosen. The reduction algorithm generates hybrid 0, except it treats the game's challenge keypair $(\mathsf{pk}^*, \mathsf{sk}^*)$ as the ith honest keypair of hybrid 0, and it uses the game's ENCRYPT oracle to generate all MKEM ciphertexts addressed to pk^*. As a result, the reduction algorithm does not know sk^*, and it does not know the value r associated with ciphertexts addressed to pk^*. It must therefore use the oracles provided in the MKEMSS to carry out operations involving these unknown values.

Hybrid 0 uses sk^* only for the following:

- Generating signatures under sk^*.
- Computing $m = \mathsf{Dec}(\mathsf{sk}^*, c)$ values for certain values of c (by an honest client). These m values are ultimately used *only* to form $\langle \mathsf{pk}, m\rangle$ tuples, where the m-components are compared for equality with other strings.
- Totally revealing sk^*, when the simulator makes a corresponding call to stealkeys.

Hybrid 0 uses r values from ciphertexts only for the following:

- Computing $m = \mathsf{Msg}(\mathsf{pk}, r)$ for certain pk values (by an honest server). As above, these m values are ultimately used *only* to form $\langle \mathsf{pk}, m\rangle$ tuples, where the m-components are compared for equality with other strings.
- Revealing r completely (when an honest server is adaptively corrupted).

With the exception of totally revealing sk^*, each of these kinds of operations can be carried out using the oracles of the MKEMSS game. Hence, the reduction algorithm can generate Hybrid 0 while in the role of an adversary in the MKEMSS game, with the MKEMSS challenge keypair corresponding to the ith honest keypair in Hybrid 0, and ciphertexts addressed to pk^* generated by the MKEMSS ENCRYPT oracle. The simulation of Hybrid 0 is exactly faithful, until a stealkeys command means that sk^* should be revealed. We let our reduction algorithm abort in this case.

It suffices to show now that the bad event in Hybrid 0 implies the **win** condition in the MKEMSS game, since the **win** condition happens with negligible probability. First, observe that the bad event with respect to pk^* entails that $\mathsf{pk}^* \in \mathsf{Secure}$. The reduction algorithm aborts only in the case where pk^* would be removed from Secure in Hybrid 0. In other words, the reduction algorithm aborts only when the bad event becomes impossible in the future.

The bad event condition has two clauses. The clause involving forged signatures has a clearly corresponding clause in the definition of the MKEMSS **win** condition. The other clause of the bad event condition refers to an adversary generating a pair $\langle \mathsf{pk}^*, \mathsf{Dec}(\mathsf{sk}^*, c)\rangle$ while c was addressed to pk^*. In this case, c would have been generated by the MKEMSS ENCRYPToracle, and a corresponding $R[c]$ value would be defined. The pair $(c, \mathsf{Dec}(\mathsf{sk}^*, c))$ will satisfy the **win** condition in MKEMSS, provided that $R[c]$ remains defined in the MKEMSS game. $R[c]$ becomes undefined only when the adversary calls OPEN(c). Our reduction algorithm only calls OPEN(c) when an honest server becomes adaptively corrupted and the adversary learns the corresponding r. But if the server has already become adaptively corrupted, then the bad event cannot happen, according to our previous understanding of bad events in Hybrid 0. □

C PSI With Proof of Non-Empty Intersection

C.1 Möller Key Agreement

The RT21 protocol is defined in terms of a 2-message key agreement protocol, where the messages may be sequential. In this work we present the PSI protocol in terms of a special case of key agreement where the messages can be simultenous, since our suggested instantiation via Diffie-Hellman has this property, and it simplifies some notation in the protocol.

Definition 19. KA *is a* ***key agreement (KA)*** *if the following protocol satisfies correctness: K_a must equal K_b, except with negligible probability.*

$$\boxed{\begin{array}{lcl} a \leftarrow \mathsf{KA}.\mathcal{R} & & b \leftarrow \mathsf{KA}.\mathcal{R} \\ A := \mathsf{KA}.\mathsf{msg}_1(a) & \xrightarrow{A} & B := \mathsf{KA}.\mathsf{msg}_2(b) \\ K_a := \mathsf{KA}.\mathsf{key}_1(a, B) & \xleftarrow[B]{} & K_b := \mathsf{KA}.\mathsf{key}_2(b, A) \end{array}}$$

Definition 20. KA *has* ***non-malleable security*** *if every PPT adversary $\mathcal{A}$ has negligible probability of winning the game:*

```
$a \leftarrow \mathsf{KA}.\mathcal{R}$
$A := \mathsf{KA.msg}_1(a)$
$\mathcal{B} := \{\}$

$\underline{\text{MSG}()}$:
  $b \leftarrow \mathsf{KA}.\mathcal{R}$
  $B := \mathsf{KA.msg}_2(b)$
  $\mathcal{B} := \mathcal{B} \cup \{B\}$
  return $B$

$\underline{\text{CHECK}(B, K)}$:
  return $\mathsf{KA.msg}_1(a, B) \overset{?}{=} K$

$(B, K) \leftarrow \mathcal{A}^{\text{MSG,CHECK}}(A)$
**win** if $B \in \mathcal{B} \wedge \mathsf{KA.msg}_1(a, B) \overset{?}{=} K$
```

Definition 21. KA *has* ***pseudorandom responses*** *if* $\mathsf{KA.msg}_2$ *is a PRG. Formally, the following distributions are indistinguishable, for some integer* s*:*

```
$b \leftarrow \mathsf{KA}.\mathcal{R}$
$B = \mathsf{KA.msg}_2(b)$
return $B$
```

```
$B \leftarrow \{0,1\}^s$
return $B$
```

One of our improvements to the RT21 PSI protocol is to use a different choice of underlying key agreement — namely, the elliptic-curve-based key agreement of Möller [Möl04]. We follow [MRR21] in using Möller as a key agreement scheme with pseudorandom responses. Let E be a Montgomery elliptic curve over a field $\mathbb{F}_E$ with almost exactly 2^s elements. Then E has a quadratic twist E', with the property that every x-coordinate either occurs twice in E, twice in E', or once in each. Therefore, $|E| + |E'| = 2(|\mathbb{F}_E| + 1)$.

Möller's idea was to use both E and E' together, and send only the x-coordinate. This way, every x-coordinate in $\mathbb{F}_E$ is possible, and equally likely. Montgomery curves allow the efficient computation of $(B^a)_x$ using only the x-coordinate $(B)_x$. Let $G_0 \in E$ and $G_1 \in E'$ be generators of their respective elliptic curve groups. They should be generators of the whole groups, not just prime order subgroups. Then Möller KA is defined as:

```
$\underline{\mathsf{KA.msg}_1((a,\cdot))}$:          $\mathsf{KA}.\mathcal{R} = [0, \max(|\mathbb{G}|, |\mathbb{G}'|)) \times \{0,1\}$
  $A_0 := G_0^a$                $\underline{\mathsf{KA.msg}_2((b,c))}$:
  $A_1 := G_1^a$                  $B := G_c^b$
  return $(A_0)_x, (A_1)_x$       return $(B)_x$

$\underline{\mathsf{KA.key}_1((a,\cdot),(B)_x)}$:   $\underline{\mathsf{KA.key}_2((b,c),((A_0)_x,(A_1)_x))}$:
  return $(B^a)_x$                return $(A_c^b)_x$
```

The first message, A, consists of a point on each of E and E', while the second message B is randomly selected to be on either E or E'. both parties can compute $(G_c^{ab})_x$, where c indexes the choice of curve made in the second KA message.

Similarly to [MRR21], we need a security assumption that is modified for the presence of two elliptic curves. By analogy with the Strong Diffie–Hellman (SDH) assumption [ABR01], define:

Definition 22. *The* ***2 Group Strong Diffie–Hellman (2G-SDH)*** *assumption for* $G_0 \in \mathbb{G}_0, G_1 \in \mathbb{G}_1$ *states that it is computationally hard to find* G_i^{ab} *from* G_i^a *and* G_i^b *for either* i*, even with an oracle for checking guesses of* X^a *for all* $X \in \mathbb{G}_0 \cup \mathbb{G}_1$*. More precisely, every PPT adversary* $\mathcal{A}$ *has negligible probability to win the game:*

```
$a, b_0, b_1 \leftarrow \big[0, \max(|\mathbb{G}_0|, |\mathbb{G}_1|)\big)$
GUESS($X \in \mathbb{G}_0 \cup \mathbb{G}_1, Y \in \mathbb{G}_0 \cup \mathbb{G}_1$):
  return $X^a \overset{?}{=} Y$
$Z := \mathcal{A}^{\text{GUESS}(\cdot)}(G_0^a, G_1^a, G_0^{b_0}, G_1^{b_1})$
win if $Z \overset{?}{=} G_0^{ab_0} \lor Z \overset{?}{=} G_1^{ab_1}$
```

Lemma 23. *Möller* KA *has non-malleable security under the 2G-SDH assumption. And it has pseudorandom responses if the relative distance from $|\mathbb{F}_E|$ to a power of* 2 *is negligible.*

Proof. Let $\mathcal{A}$ be an adversary against the non-malleable security of KA. We construct a new adversary $\mathcal{A}'$ against the 2G-SDH assumption. Given a challenge $(A_0, A_1, B_0, B_1) = (G_0^a, G_1^a, G_0^{b_0}, G_1^{b_1})$, $\mathcal{A}'$ sets $A = (A_0, A_1)$ and runs $\mathcal{A}(A)$. The MSG and CHECK oracles are simulated as:

```
$R :=$ empty map
MSG():
  $c \leftarrow \{0,1\}$
  $b \leftarrow \big[0, |\mathbb{G}_c|\big)$
  $B := B_c G_c^b$
  $R[(B)_x] := (c, b, B)$
  return $(B)_x$

CHECK($(B)_x, (K)_x$):
  find $X$ where $(X)_x = (B)_x$
  find $Y$ where $(Y)_x = (K)_x$
  return $\left(\begin{array}{c}\text{GUESS}(X,Y) \\ \lor\ \text{GUESS}(X, Y^{-1})\end{array}\right)$
```

MSG works by rerandomizing B_c to get uniformly random elements of the group $\mathbb{G}_c$. While the exponent b is chosen from possibly a smaller set than in the key agreement, the difference is negligible because $\frac{\max(|\mathbb{G}_0|,|\mathbb{G}_1|)}{\min(|\mathbb{G}_0|,|\mathbb{G}_1|)} = 1 + O(\mathbb{F}_E^{-1/2})$ by the Hasse bound.

The CHECK oracle is simulated with a complication. It is given only the x-coordinates of curve points, whereas the 2G-SDH GUESS oracle expects full curve points as arguments. However, the two points with the same x-coordinates are always a pair of the form X, X^{-1}. Hence, our simulation of CHECK finds *any* curve point with the given x-coordinate, and queries its GUESS oracle twice to get

```
$P_S$ on input (input, $(P_S, P_C, ssid), M_S$)        $P_C$ in input (input, $(P_S, P_C, ssid), M_C$)

$a \leftarrow \mathsf{KA}.\mathcal{R}$
$A := \mathsf{KA.msg}_1(a)$
                              $\xrightarrow{A}$
                              $\xleftarrow{P}$       $P, \{b_m\}_{m \in M_C} \leftarrow \mathsf{BuildPoly}(M_C)$
output $\deg(P) + 1$ $(= |M_C|)$
$h, K, s \leftarrow \mathsf{GenChallenge}(a, P, M_S)$
// a can now be erased
                              $\xrightarrow{h, K}$
                                                     $I := \left\{ m \in M_C \mid \mathsf{Response}(A, m, b_m, h, K) \overset{?}{\neq} \bot \right\}$
                                                     output $|K|$ $(= |M_S|)$ and $I$
                                                     await command (deliver, $ssid, d \in \{0,1\}$)
                                                     $s' := 0$
                                                     if $d \land \exists m \in I$:
                                                       $s' := \mathsf{Response}(A, m, b_m, h, K)$
                              $\xleftarrow{s'}$
output $s \overset{?}{=} s'$
```

Figure 8: Our PSI protocol. Subroutines GenChallenge, BuildPoly, and Response are defined in Figure 9.

```
$\underline{\mathsf{GenChallenge}(a, P, M_{\mathsf{S}})}$:
  $s \leftarrow \{0,1\}^\lambda$
  $h := \mathsf{Hash}(s)$
  $K :=$ empty map
  for $m \in M_{\mathsf{S}}$:
    $B_m := E^{-1}(m, P(H(m)))$
    $k_m := H'(m, \mathsf{KA.key}_1(a, B_m))$
    $K[k_{m,1}] := s \oplus k_{m,2}$
  return $h, K, s$

$\underline{\mathsf{BuildPoly}(A, M_{\mathsf{C}})}$:
  $S := \{\}$
  for $m \in M_{\mathsf{C}}$:
    $b_m \leftarrow \mathsf{KA}.\mathcal{R}$
    $B_m := \mathsf{KA.msg}_2(b)$
    $S := S \cup \{(H(m), E(m, B_m))\}$
  $P := \mathsf{interpol}_{\mathbb{F}}(S)$
  return $P, \{b_m\}_{m \in M_{\mathsf{C}}}$

$\underline{\mathsf{Response}(A, m, b, h, K)}$:
  $k_m := H'(m, \mathsf{KA.key}_2(b, A))$
  $s' := K[k_{m,1}] \oplus k_{m,2}$
  if $h \overset{?}{\neq} \mathsf{Hash}(s')$:
    return $\perp$
  return $s'$
```

Figure 9: Subroutines of our PSI protocol (Figure 8).

the correct answer. Therefore, $\mathcal{A}$ has the same view as it would when running in the non-malleable security game.

When $\mathcal{A}$ outputs a decryption guess $(B)_x, (K)_x$ that would win the non-malleable security game, fetch $(c, b, B) = R[(B)_x]$. Similarly to CHECK, solve for Y with x-coordinate $(K)_x$ and check which of (B, Y) and (B, Y^{-1}) is a valid decryption pair. Assume w.l.o.g. that this pair is (B, Y). Then $Y\, A_c^{-b} = B_c^a$ solves the 2G-SDH problem.

For pseudorandom responses, notice that picking a uniformly random x-coordinate corresponds to picking a uniformly random pair B, B^{-1} of points, either on the curve or the twist. The only difference is that the choice of curve vs twist will be weighted according to the number of points on each. However, the fraction of x-coordinates on the main curve (relative to the number of x-coordinates in $\mathbb{F}_E$) is negligibly close to $\frac{1}{2}$, by the Hasse bound. Therefore, generating a uniformly random x-coordinate is indistinguishable from the output of $\mathsf{KA.msg}_2$. Finally, a uniformly random element of $\mathbb{F}_E$ has an almost uniformly random bit representation, with statistical distance at most $\frac{||\mathbb{F}_E|-2^s|}{2^s}$, where s is the closest power of 2 to $|\mathbb{F}_E|$. □

C.2 Protocol

Our protocol for PSI with proof of non-empty intersection is given in Figure 8. It is based on the maliciously secure PSI of Rosulek and Trieu [RT21]. The protocol requires several cryptographic primitives. First, we need a key agreement that has non-malleable security and pseudorandom responses. We also use two local random oracles, H and H', and a local ideal cipher E,[11] where

$$H\colon \{0,1\}^* \to \{0,1\}^s$$
$$H'\colon \{0,1\}^* \times \{0,1\}^s \to \{0,1\}^{2\lambda}$$
$$E^{\pm}\colon \{0,1\}^* \times \{0,1\}^s \to \{0,1\}^s.$$

[11] RT21 is proven secure using an ideal permutation rather than ideal cipher. However, like the random oracle model, an ideal permutation is assumed to be *local* to each protocol interaction. In other words, the local ideal permutation model is equivalent to a collection of independent permutations, indexed by the session id. This is not substantively different from the ideal cipher model, treating the session id as the cipher key. Hence, we choose to describe the protocol here using an ideal cipher, since it has the additional property of improving some aspects of the security bound.

```
      $a \leftarrow \mathsf{KA}.\mathcal{R}$
      $A := \mathsf{KA.msg}_1(a)$
      $\underline{\text{ON\_CORRUPT}(M_\mathsf{S})}$:
$P_\mathsf{S}$:   give $M_\mathsf{S}$ and $a$ to the adversary
$P_\mathsf{S}$: send $A$ to $P_\mathsf{C}$

$P_\mathsf{S}$: receive $P$ from $P_\mathsf{C}$
      $M_\mathsf{C} := \{\}$
      for past IC queries $y = E(m, x)$:
        if $y \overset{?}{=} P(H(m))$:
          $M_\mathsf{C} := M_\mathsf{C} \cup \{m\}$
      while $|M_\mathsf{C}| < \deg(P) + 1$: // pad $M_\mathsf{C}$ with dummies
        $m \leftarrow \{0,1\}^{2\lambda}$
        $M_\mathsf{C} := M_\mathsf{C} \cup \{m\}$
$P_\mathsf{C}$: send (input, $(P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), M_\mathsf{C}$) to $\mathcal{F}_{\mathsf{psi+}}$

$P_\mathsf{C}$: receive $(I, |M_\mathsf{S}|)$ from $\mathcal{F}_{\mathsf{psi+}}$
      $h, K, s \leftarrow \mathsf{GenChallenge}(a, P, I)$
      while $|K| < |M_\mathsf{S}|$: // pad $K$ with dummies
        $k \leftarrow \{0,1\}^{2\lambda}$
        $K[k_1] := s \oplus k_2$
      $\underline{\text{ON\_CORRUPT}(M_\mathsf{S})}$:
$P_\mathsf{S}$:   give $M_\mathsf{S}, h, K, s$ to the adversary // $a$ was erased
$P_\mathsf{S}$: send $h, K$ to $P_\mathsf{C}$

$P_\mathsf{S}$: receive $s'$ from $P_\mathsf{C}$
$P_\mathsf{C}$: send (deliver, $\mathit{ssid}, s \overset{?}{=} s'$) to $\mathcal{F}_{\mathsf{psi+}}$
```

(a) Corrupt client and honest server.

```
$P_\mathsf{S}$: receive $|M_\mathsf{C}|$ from $\mathcal{F}_{\mathsf{new\text{-}auth}}$
      $P \leftarrow \mathbb{F}[x]$ with degree less than $|M_\mathsf{C}|$
      $b \leftarrow \mathsf{KA}.\mathcal{R}^{\{0,1\}^*}$
      $\underline{\text{program } E^{-1}(m, y)}$:
        if $y \overset{?}{=} P(H(m))$:
          return $\mathsf{KA.msg}_2(b(m))$
      $\underline{\text{ON\_CORRUPT}(M_\mathsf{C})}$:
      $b_m := b(m), \forall m \in M_\mathsf{C}$
$P_\mathsf{C}$:   give $M_\mathsf{C}$ and $\{b_m\}_{m \in M_\mathsf{C}}$ to the adversary
$P_\mathsf{C}$: send $P$ to $P_\mathsf{S}$

$P_\mathsf{C}$: receive $h, K$ from $P_\mathsf{S}$
      $M_\mathsf{S} := \left\{ m \;\middle|\; \begin{array}{l} \text{adversary queried } H'(m, \cdot) \\ \wedge\, \mathsf{Response}(A, m, b_m, h, K) \overset{?}{\neq} \bot \end{array} \right\}$
      while $|M_\mathsf{S}| < |K|$: // pad $M_\mathsf{S}$ with dummies
        $m \leftarrow \{0,1\}^{2\lambda}$
        $M_\mathsf{S} := M_\mathsf{S} \cup \{m\}$
$P_\mathsf{S}$: send (input, $(P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), M_\mathsf{S}$) to $\mathcal{F}_{\mathsf{psi+}}$

$P_\mathsf{S}$: receive *nonempty* from $\mathcal{F}_{\mathsf{psi+}}$
      if $\mathit{nonempty} \wedge \exists m \in I$:
$P_\mathsf{C}$:   send $\mathsf{Response}(A, m, b_m, h, K)$ to $P_\mathsf{S}$
      else:
$P_\mathsf{C}$:   send 0 to $P_\mathsf{S}$
```

(b) Corrupt server and honest client.

Figure 10: Simulators for adaptive security of Figure 8. The simulator plays a number of roles in the protocol, so each action is annotated by who the simulator is pretending to be when it takes the action.

Strictly speaking, the *ssid* should be passed into all three of these, but we omit this for clarity. The plaintexts and ciphertexts of the idea cipher E are assumed to be in a field $\mathbb{F}$, which needs to have order very close to 2^s. We ignore the difference between $\{0,1\}^s$ and $\mathbb{F}$, as they are assumed to be negligibly different. Finally, we need a collision resistant hash function Hash.

In the protocol we use the notation x_1, x_2 to denote the two halves of a string x (*e.g.*, $k_{m,1}$ and $k_{m,2}$ are the two halves of k_m).

Because we target adaptive security for our protocol, we must be careful that parties erase information from their internal state after it is no longer needed. This becomes important for the case of the value a held by the server.

Note that it would be possible to save a round in our protocol, as sending A could be delayed to go at the same time as h and K. However, we keep the four rounds because it makes no difference to the overall round complexity in the context of our authentication protocol, since A would be sent at the same as sending the MKEM ciphertext. It also allows a small optimization, where the client can compute $\mathsf{KA.key}_2(b_m, A)$ for all $m \in M_\mathsf{C}$ while it is waiting for the server to send K.

Instantiations and Implementation Notes: Following [MRR21], our implementation uses curve25519 as its elliptic curve, which is highly suitable for use in Möller KA. It was designed explicitly to have a secure twist, as well as being secure itself, and its field size $2^{255} - 19$ is extremely close to a power of 2. The polynomial interpolation field $\mathbb{F}$ was chosen to be the first prime bigger than 2^{256}. This difference of one bit between the KA message size and the polynomial size is bridged by concatenating an extra random bit on the end. The hash functions are SHA256, and the ideal cipher is instantiated as $\mathsf{Rijndael256}(\mathsf{SHA256}(m), x)$, using the 256 bit block size and key size variant of the cipher that was standardized as AES.

The table K sent from the server to the client is represented as a sorted list of tuples, to avoid leaking their order. We used a Batcher odd-even sorting network [Bat68] to prevent any timing attacks on this step. The client verifies that K is sorted, and that there are no duplicates.

C.2.1 Security Proof

The security proof follows essentially that of [RT21]. However, they prove only static security for the protocol, whereas we prove adaptive security. Our proof requires the simulator to provide a simulated internal state when a party is corrupted during the protocol execution. This turns out to be rather straight-forward, thanks in part to some careful planning about when parties erase information from their state.

Theorem 15. *The protocol in Figure 8 UC-securely realizes the $\mathcal{F}_{\mathsf{psi+}}$ functionality (Figure 3) against adaptive adversaries, in the ideal cipher + random oracle model, if* KA *has non-malleable security (Definition 20) and pseudorandom responses (Definition 21).*

Proof. As in our proof of Theorem 14, if neither party is initially corrupt, then there is nothing to simulate until one party becomes corrupt. When the first party is corrupted, the simulator can simply run the corresponding simulator from the beginning (as if the party had been semi-honestly corrupt the whole time) until the appropriate point. When the second party becomes corrupt, the simulator must generate an internal state for the newly corrupted party, but after that point there is nothing to simulate.

The formal description of the simlator is given in Figure 10. There are two cases depending on which party is corrupt at the beginning of the protocol execution. At various points throughout the simulator description, it defines a procedure ON_CORRUPT. If the honest party becomes corrupted during the protocol execution, the simulator will execute the most recently defined ON_CORRUPT procedure to generate that party's simulated internal state.

Without loss of generality, we assume that the ideal oracles are never queried if the answer to that query has already been determined previously in the interaction. This includes repeated queries to the random oracles and ideal cipher, and also forward ideal cipher queries where the corresponding backwards query was already made (or vice-versa).

Corrupt client. We consider now the case of an initially corrupt client.

Hybrid 1: Same as the real interaction, except that we log all queries to H' and the ideal cipher E. After receiving polynomial P from the corrupt client, we abort if there is ever a query of the form $H'(m, \mathsf{KA.key}_1(a, E^{-1}(m, P(H(m)))))$, made before the honest server would erase a, and there was no previous ideal cipher query $E(m, x) \to P(H(m))$.

It suffices to show that the probability of aborting is negligible. We do so via the following reduction to the scheme's non-malleability:

> We construct a reduction algorithm which plays in the non-malleability security game, and hence has access to MSG and CHECK oracles. The reduction runs the real protocol interaction, along with the adversary, but instead of choosing a itself, it defines a implicitly as the value chosen in the non-malleability game. If the server is corrupted and the simulator would have to reveal a to the adversary, then abort.

The reduction algorithm further programs the responses to every ideal cipher query $E^{-1}(m, y)$ to be a fresh response from calling its MSG oracle. This causes the output of $E^{-1}(m, y)$ to be added to the game's set $\mathcal{B}$. Since the KA has pseudorandom responses, the protocol messages generated by MSG are indistinguishable from random, and the adversary's view in this reduction is indistinguishable from its real-world view.

When the reduction algorithm needs to simulate the honest server's computation of values of the form $H'(m, \mathsf{KA.key}_1(a, B_m))$, the reduction algorithm cannot compute this directly because it does not have a. So instead, it samples a random value r_m and pretends that r_m is the appropriate response from H'. This change will not have any effect on the adversary's view if we can arrange to program H' to output r_m on this value. This can be arranged using the CHECK oracle. Namely, when any party makes a query $H'(m, z)$, and $\text{CHECK}(B_m, z) = 1$, this means that z indeed is the correct KA key $\mathsf{KA.key}_1(a, B_m)$, so we can program the output to be r_m.

Now, Hybrid 1 artificially aborts upon seeing a query of the form $H'(m, z = \mathsf{KA.key}_1(a, x))$, where $x = E^{-1}(m, P(H(m)))$ but there was never a prior corresponding query to $E(m, x)$. That query to E could only be absent if there was a prior query of the form $E^{-1}(m, P(H(m))) \to x$. But then, x was added to the game's set $\mathcal{B}$ and yet $\text{CHECK}(x, z) = 1$. Hence, the pair (x, z) wins the non-malleability game for the reduction algorithm.

From this we conclude that Hybrid 1's artificial abort happens with negligible probability.

Hybrid 2: Same as the previous hybrid, except that when the simulator receives polynomial P from the corrupt client, we compute the set M_{C} exactly as the simulator does (without padding with dummy items). Later, when the honest server runs $\mathsf{GenChallenge}$, it should compute $k_m = H'(m, \mathsf{KA.key}_1(a, B_m))$ values. We modify this hybrid so that whenever $m \notin M_{\mathsf{C}}$, it samples $k_m \leftarrow \{0,1\}^{2\lambda}$ instead. However, if the server gets compromised before erasing a, pretend this modification didn't happen by going back and recomputing the k_m correctly.

If the server is compromised before a is erased, then the adversary's view is not affected by this change. Otherwise, we have replaced $H'(m, \mathsf{KA.key}_1(a, B_m))$ with a random value, so the change is indistinguishable unless the adversary queries H' at this point. However, since $m \notin M_{\mathsf{C}}$, such a query would trigger the bad event causing the interaction to abort. This can only happen with negligible probability, so the two hybrids are indistinguishable.

Hybrid 3: Pad M_{C} with dummy elements until its size is $\deg(P) + 1$, as in the simulator. These dummy elements have negligible probability of colliding with any values that the interaction checks for membership in M_{C}, so this change is indistinguishable. This modification assumes that $|M_{\mathsf{C}}| \le \deg(P) + 1$, but we can show that this condition is violated only with negligible probability, by reducing to what we call the Random Polynomial Reconstruction assumption (Definition 24).

We appeal to that assumption using the uniformly random set of points $(H(m), E(m, x))$, over all ideal cipher queries $E(m, x)$. For $|M_{\mathsf{C}}|$ to be too large, P would have to go through more than $\deg(P) + 1$ of them, which violate the assumption.

Hybrid 4: Same as the previous hybrid, except abort if $s' = s$ and yet $M_{\mathsf{S}} \cap M_{\mathsf{C}} = \emptyset$. This event happens with negligible probability because: (1) When $M_{\mathsf{S}} \cap M_{\mathsf{C}} = \emptyset$, every k_m value chosen in this hybrid is uniformly random. The k_m values then act as one-time pads to completely hide s from the adversary's view. (2) Without any information on s except for $h = \mathsf{Hash}(s)$, computing s is equivalent to breaking the preimage security of Hash.

Ideal world: In the previous hybrid, each item in $M_{\mathsf{S}} \setminus M_{\mathsf{C}}$ contributes uniformly random values to the map K. Besides that, these items are not used anywhere. In other words, the distribution generated by the previous hybrid can be generated knowing only $I = M_{\mathsf{S}} \cap M_{\mathsf{C}}$, and $|M_{\mathsf{S}}|$. From this we can see that the previous hybrid differs from the ideal interaction only in the artificial aborts. Since the artificial aborts have negligible probability, the previous hybrid is indistinguishable from the ideal interaction.

Corrupt server. We now consider the case where the server is corrupt when the protocol begins.

Hybrid 1: Same as the real interaction, except for the following modifications: Instead of sampling b_m values explicitly, the honest server should lazily sample a random oracle $b \leftarrow \mathsf{KA}.\mathcal{R}^{\{0,1\}^*}$ and set $b_m = b(m)$. This change clearly does not affect the distribution of any values in the interaction, but will become convenient later.

Additionally, generate the polynomial P in "reverse order." That is, first sample a uniformly random polynomial P with degree less than $|M_{\mathsf{C}}|$. Then program the ideal cipher so that $E^{-1}(m, P(H(m))) = \mathsf{KA}.\mathsf{msg}_2(b(m))$. These points on the ideal cipher have negligible probability of interfering with any existing queries, because P is freshly random, and so $P(H(m))$ is random for any m that was previously queried. We now have $P(H(m)) = E(m, \mathsf{KA}.\mathsf{msg}_2(b_m))$ for $m \in M_{\mathsf{C}}$, so P is still the interpolation of the same set as before.

If the $\mathsf{KA}.\mathsf{msg}_2(b(m))$ values were uniformly distributed (and there are no collisions in H leading to a contradictory set of points for interpolation), then we would achieve the same distribution by generating P in the normal way or in this reverse order. Since $\mathsf{KA}.\mathsf{msg}_2(b(m))$ values are merely pseudorandom, this hybrid is merely indistinguishable from the real interaction.

Hybrid 2: Same as the previous hybrid, except remove from the honest client's set I any m where the adversary has not previously queried $H'(m, \cdot)$. Recall that m is included in the intersection if (among other things) the first half of $H'(m, \mathsf{KA}.\mathsf{key}_2(b, A))$ exists as a key in the map K sent by the adversary. If this query to $H'(m, \cdot)$ is a fresh one, then this event happens with negligible probability. Hence, the probability of removing an item from I in this hybrid is negligible, and this hybrid is indisinguishable from the previous one.

As a result of this change, we can rewrite the client's computation of I as follows. First compute M_{S} as in the simulator (but without padding with dummy items) — *i.e.*, the set of all m's for which the adversary previously made a query $H'(m, \cdot)$ and for which $\mathsf{Response}(A, m, b(m), h, K) = 1$. Then set $I = M_{\mathsf{S}} \cap M_{\mathsf{C}}$.

Hybrid 3: Pad M_{S} with random dummy elements until it has the same cardinality as K, just like in the simulator's description. These dummy elements have negligible probability of colliding with anything else, so the change is indistinguishable.

This change assumes that $|M_{\mathsf{S}}| \leq |K|$, which is true with all but negligible probability because otherwise there must be a hash collision. By the pigeonhole principle, if $|M_{\mathsf{S}}| > |K|$ then there must be $m \neq m' \in M_{\mathsf{S}}$ such that m and m' both reference the same entry in K, so $k_{m,1} = k_{m',1}$. Then either $k_{m,2} = k_{m',2}$, implying a full collision in H', or $s = K[k_{m,1}] \oplus k_{m,2} \neq K[k_{m',1}] \oplus k_{m',2} = s'$ and $\mathsf{Hash}(s) = \mathsf{Hash}(s') = h$.

Ideal interaction. In the previous hybrid, the only place M_{C} is used is to compute the honest client's intersection, as $I = M_{\mathsf{C}} \cap M_{\mathsf{S}}$. Thus, the hybrid proceeds identically to the ideal interaction, the only difference being "repackaging" of which computational steps happen in which parts of the interation (*i.e.*, simulator, honest client, ideal functionality). □

Definition 24. *The **Random Polynomial Reconstruction (RPR) assumption** states all PPT adversaries $\mathcal{A}$ win the following game with at most negligible probability, for all (polynomial) n.*

```
for $i := 1$ to $n$:
  $x_i, y_i \leftarrow \mathbb{F}$
$P \leftarrow \mathcal{A}(\{x_i\}_i, \{y_i\}_i)$
win if $|\{i \mid P(x_i) = y_i\}| > \deg(P) + 1$
```

Justification: The standard polynomial reconstruction problem for parameters n, k, t is to find a polynomial P with $\deg(P) < k$ that goes through at least t of n given points (x_i, y_i). Cryptosystems [KY07] have been constructed from a related decision problem. The points are chosen by picking a random polynomial P, evaluating it at n places, then replacing $n - t$ of the resulting y-coordinates with uniform randomness. Thus, the distribution guarantees that a solution exists by constructing the problem around such a solution. Our RPR assumption is particularly difficult in this regard, as the points are not chosen to make the problem solvable, having no structure whatsoever.

As far as parameter selection goes, [KY07] consider two attacks as being important. The first, Guruswami–Sudan list decoding [GS98], fails when $t < \sqrt{kn}$. In our case, $n \geq t = k + 1 \geq 2$, with $k = \deg(P) + 1$ selected by the adversary. For the list decoding algorithm to work,

$$\begin{aligned} t &\geq \sqrt{(t-1)n} \\ \frac{t^2}{t-1} &\geq n \\ t + 1 + \frac{1}{t-1} &\geq n \\ t + 1 &\geq n, \end{aligned}$$

using that t and n are both integers. By a union bound, the existence of a solution P has probability at most $\binom{n}{t}/|\mathbb{F}|$, since any choice of t points gives an interpolation polynomial that is a solution if and only if its highest degree coefficient is zero. Since all the points are random, the coefficients will be random as well, so for each set of t points the probability is $|\mathbb{F}|^{-1}$. If $n = t + 1$, a solution exists with probability at most $n/|\mathbb{F}|$, which is negligible (and much smaller than $2^{-\lambda}$ for our choice of $\mathbb{F}$).

The other attack considered was a simple brute force attack: try the $\binom{n}{t}$ possible choice of t to interpolate a polynomial through, and check if it has degree $t - 2$. As in the above union bound, each guess has probability at most $|\mathbb{F}|^{-1}$, so this attack is also infeasible. Finally, for small t there may be special algorithms, as, e.g., $t = 2$ corresponds to just searching for a collision. To cover these attacks, we conjecture that adversaries with time complexity T can break RPR with probability at most $O(T^2/|\mathbb{F}|)$.

[RT21] also gave a statistical bound for attacking a similar game, where the adversary has to interpret the polynomial through more than just one extra point. Using a bound similar to the union bound we gave above, they show that exceeding $O(n)$ extra interpolation points has negligible probability, even for unbounded adversaries.