Practical Privacy-Preserving Authentication for SSH

Lawrence Roy, Stanislav Lyakhov, Yeongjin Jang,
and Mike Rosulek, *Oregon State University*

https://www.usenix.org/conference/usenixsecurity22/presentation/roy

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the
31st USENIX Security Symposium is
sponsored by USENIX.

Practical Privacy-Preserving Authentication for SSH

Lawrence Roy[*]
Oregon State University

Stanislav Lyakhov
Oregon State University

Yeongjin Jang
Oregon State University

Mike Rosulek
Oregon State University

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 [45]. 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.

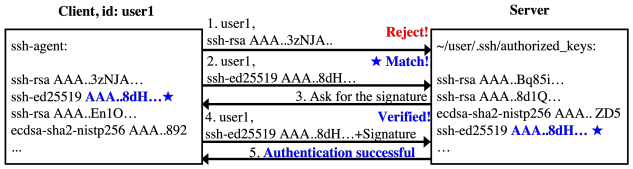
1.1 Privacy Attacks Against SSH Authentication

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 [44, 46, 47] — 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 [35].

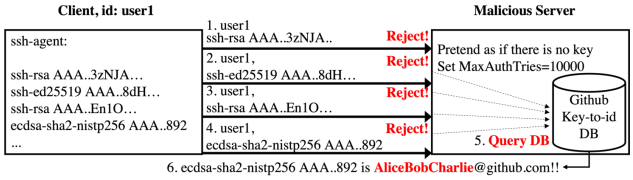
Third, a server knows which key has been used in authenticating the current session, allowing the server to track a specific user's usage based on their keys, and also prove to third parties that the user authenticated. Fourth, a malicious server can intercept a client's connection, fooling any user 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 ex-

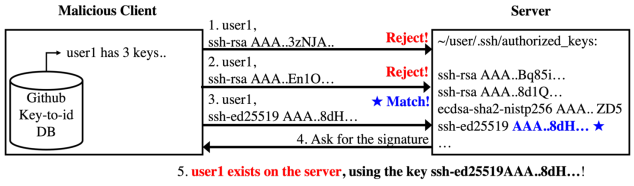
[*]Supported by a DoE CSGF Fellowship.



(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.



(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.



(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.

ample, 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 [10, 36].

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 [39] 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 [44, 46, 47].

In particular, Cox [10] built a database containing the public keys of all Github users, using the Github website functionality described above. Later, Valsorda [46] built and publicly deployed a proof-of-concept de-anonymizing SSH server [47], 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 [40] 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.

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*,

[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.

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 [40, 44, 46, 47]. 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 [31, 39, 41] 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* [42] and their interactive counterpart *deniable ring authentication* [34] 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 [8] and anonymous ad-hoc authentication [16] 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 [2] (see also [24–26, 30]), 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 [23, 29].

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 [9].[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* [17, 18]. Since the server can simulate its view of the protocol given only the set of authorized keys,

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

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 PSI protocol of Rosulek & Trieu [43] 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) [11–13], 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.*, [13, 21]) and has even been used previously as a means of authentication [49]. 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 [4, 27], 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 [28] (sometimes called key-private [3]), 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 [19, 50].

2 Preliminaries

Definition 1. *Let G generate a cyclic group $\mathbb{G}$ of order ℓ. The **gap computational Diffie–Hellman (GapCDH)** assumption [38] 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 de-*

cisional Diffie–Hellman problem. More precisely, every PPT adversary $\mathcal{A}$ has negligible probability to win the game:

$a, b \leftarrow [0, \ell) \cap \mathbb{Z}$
GUESS$(X \in \mathbb{G}, Y \in \mathbb{G}, Z \in \mathbb{G})$:
  return $\text{dlog}_G(X) \cdot \text{dlog}_G(Y) \stackrel{?}{=} \text{dlog}_G(Z) \bmod \ell$
win if $\mathcal{A}^{\text{GUESS}(\cdot)}(G^a, G^b) = G^{ab}$

2.1 Signatures

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

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

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:*

$M := \{\}$
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{SS.Gen}(\mathsf{opts})$
SIGN(m):
  $M := M \cup \{m\}$
  return $\mathsf{SS.Sign}(\mathsf{sk}^*, m)$
$(m, s) \leftarrow \mathcal{A}^{\text{SIGN}(\cdot)}(\mathsf{pk}^*)$
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*

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

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 MKEM.Dec or 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
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{MKEM.Gen}(\mathsf{opts})$
ENCRYPT(PK):
  $(c, r) \leftarrow \mathsf{MKEM.Enc}(\{\mathsf{pk}^*\} \cup \mathsf{PK})$
  $R[c] := r$
  return c
GUESS_DEC(c, m):
  return $\mathsf{MKEM.Dec}(\mathsf{sk}^*, c) \stackrel{?}{=} m$
GUESS_MSG(c, pk, m):
  if $R[c]$ defined:
    return $\mathsf{MKEM.Msg}(\mathsf{pk}, R[c]) \stackrel{?}{=} m$
$(c, m) \leftarrow \mathcal{A}^{\text{ENCRYPT,GUESS\_DEC,GUESS\_MSG}}(\mathsf{pk}^*)$
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*** MKEM.Leak *if there is a PPT simulator* (AnonSim, 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 $[R[c]$ 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 [5] 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 $H\colon E \times E \times \{0,1\}^* \to \mathbb{Z}/\ell\mathbb{Z}$.

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

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

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 pk^r

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

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 [14].

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*

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

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

Proof. See the full version of this work. □

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) [37].

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.

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

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

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

$\underline{\mathsf{ECDSA.Msg}(A,r):}$
return A^r

$\underline{\mathsf{ECDSA.Dec}(a,C):}$
return C^a

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

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

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 [20]. Brown [7] 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. [20] 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 [14]. We adapt their result to our scheme (proof included in the full version):

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 [33], 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 $s^e \stackrel{?}{=} \mathsf{PKCS}_N(m)$

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)

$\underline{\mathsf{RSA.Enc}(\mathsf{PK}):}$
$S := \{\}$
$R :=$ 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 $c \bmod N$

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)* [22], 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.

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 the full version.

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 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 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 (f,pk) where pk is a key of flavor f.

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

[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.

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

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

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

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

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

$\underline{\mathsf{MixKEM.Msg}((\mathsf{f},\mathsf{pk}),R)}$:
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 $\mathsf{PK_f}$ be defined as in 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 $\mathsf{f.Enc}(\mathsf{PK_f})$. Therefore, the leakage function for MixKEM is $\mathsf{MixKEM.Leak}(\mathsf{PK}) = \{(\mathsf{f},\mathsf{f.Leak}(\mathsf{PK_f})) \mid \mathsf{f} \in F\}$.

The following lemmas are proven in the full version:

Lemma 12. MixKEM *is a jointly secure MKEMSS if every flavor in* FLAVORS *is.*

Lemma 13. MixKEM *is anonymous, assuming that every* $\mathsf{f} \in$ 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, 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.

Keys can be classified into 3 categories with respect to the ideal functionality: (1) A key generated by an honest user

[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 [6] technique.

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

Static variables:
- Sets Σ and Secure; associative arrays $\mathsf{SK}_1, \mathsf{SK}_2, \ldots$

Define predicate:

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

On input $(\texttt{genkey}, \mathsf{opts})$ 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 pk to Secure.
3. Give pk to P_i.

On input $(\texttt{get\_sk}, \mathsf{pk})$ 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 $(\texttt{stealkeys}, P_i)$ from the adversary:
5. Set $\mathsf{Secure} := \mathsf{Secure} \setminus \{\mathsf{pk} \mid \mathsf{SK}_i[\mathsf{pk}] \text{ defined}\}$.
6. Give SK_i to the adversary.

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

On input $(\texttt{verify}, \mathsf{pk}, m, \sigma)$ 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_S & client P_C
On input $(\texttt{auth}_1, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K_\mathsf{S})$ from P_S:
12. If P_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 $(\texttt{auth}_2, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K_\mathsf{C})$ from P_C.
14. Give $|K_\mathsf{C}|$ to P_S.
15. Wait for command $(\texttt{auth}_3, (P_\mathsf{S}, P_\mathsf{C}, \mathit{ssid}), K'_\mathsf{S})$ from P_S.
16. If P_S is corrupt: set $K_\mathsf{S} := K'_\mathsf{S}$ (otherwise ignore K'_S).
17. Compute $A := K_\mathsf{S} \cap K_\mathsf{C} \cap \{\mathsf{pk} \mid \texttt{can\_use}(P_\mathsf{C}, \mathsf{pk})\}$.
18. Give $(A, |K_\mathsf{S}|)$ to P_C.
19. Wait for command $(\texttt{deliver}, \mathit{ssid}, d \in \{0,1\})$ from P_C.
20. Give $d \wedge [A \neq \emptyset]$ to P_S.

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

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).

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.

Behavior:

1. Await command $(\texttt{input}, (P_S, P_C, ssid), M_C)$ from P_C.
2. Give $|M_C|$ to P_S.
3. Await command $(\texttt{input}, (P_S, P_C, ssid), M_S)$ from P_S.
4. Compute $I = M_S \cap M_C$ and give $(I, |M_S|)$ to P_C.
5. Await command $(\texttt{deliver}, ssid, d \in \{0,1\})$ from P_C.
6. Give $d \wedge [I \neq \emptyset]$ to P_S.

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

On command $(\texttt{genkey}, \texttt{opts})$ to party P_i:

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

On command $(\texttt{sign}, \mathsf{pk}, m)$ 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 $(\texttt{verify}, \mathsf{pk}, m, \sigma)$ to party P_i:

5. P_i: Run $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ and return the result.

On command $(\texttt{auth}_1, (P_S, P_C, ssid), K_S)$ to party P_S:

6. P_S: Generate $(c, r) \leftarrow \mathsf{Enc}(K_S)$ and send c to P_C.
7. P_C: Await command $(\texttt{auth}_2, (P_S, P_C, ssid), K_C)$ and set:
   $$M_C := \left\{ \left\langle \mathsf{pk}, \mathsf{Dec}(\mathsf{SK}_C[\mathsf{pk}], c) \right\rangle \;\middle|\; \begin{array}{c} \mathsf{pk} \in K_C \text{ and} \\ \mathsf{SK}_C[\mathsf{pk}] \text{ defined} \end{array} \right\}.$$
8. P_C: Send $(\texttt{input}, (P_S, P_C, ssid), M_C)$ to $\mathcal{F}_{\mathsf{psi+}}$.
9. P_S: Receive $|M_C|$ from $\mathcal{F}_{\mathsf{psi+}}$ and output it.
10. P_S: Await command $(\texttt{auth}_3, (P_S, P_C, ssid), -)$ and set:
    $$M_S := \left\{ \left\langle \mathsf{pk}, \mathsf{Msg}(r, \mathsf{pk}) \right\rangle \;\middle|\; \mathsf{pk} \in K_S \right\}.$$
11. P_S: Send $(\texttt{input}, (P_S, P_C, ssid), M_S)$ to $\mathcal{F}_{\mathsf{psi+}}$.
12. P_C: Receive output $(I, |M_S|)$ from $\mathcal{F}_{\mathsf{psi+}}$ and output:
    $$\Big(\{ \mathsf{pk} \mid \exists m : \langle \mathsf{pk}, m \rangle \in I \}, |M_S| \Big)$$
13. P_C: Await command $(\texttt{deliver}, ssid, d)$ and forward it to $\mathcal{F}_{\mathsf{psi+}}$.
14. P_S: Receive output e from $\mathcal{F}_{\mathsf{psi+}}$ and output it.

Figure 4: Our anonymous authentication protocol.

5.1 Security Proof

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

[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.

[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.

Proof. We sketch a proof here and defer the details to the full version. 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 MKEM.AnonSim on that leakage. 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 AnonView algorithm from the anonymity definition.

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 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. □

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 [43] (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 the full version of this work.

Nearly all PSI protocols, including RT21, use the **oblivious PRF (OPRF)** paradigm of [21]. 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 the full version. We also prove security against adaptive corruption, while RT21's original proof considers only static corruption.

Theorem 15. *The modified RT21 PSI protocol UC-securely realizes the $\mathcal{F}_{\mathsf{psi+}}$ functionality (Figure 3) against adaptive adversaries, in the ideal cipher + random oracle model, assuming a suitable 2-message key agreement scheme exists.*

See the full version 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 [1].

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 [6] to encode elliptic curve elements as uniform bit strings. We observe that a different technique of Möller [32] results in elliptic-curve-based key agreement at roughly half the computational cost. The details are given in the full version.

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 [43], 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 [48].

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 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 [9]. 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

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. *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`.

ECDSA and EdDSA. Both ECDSA and EdDSA are significantly faster than RSA, and the performance characteristics

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

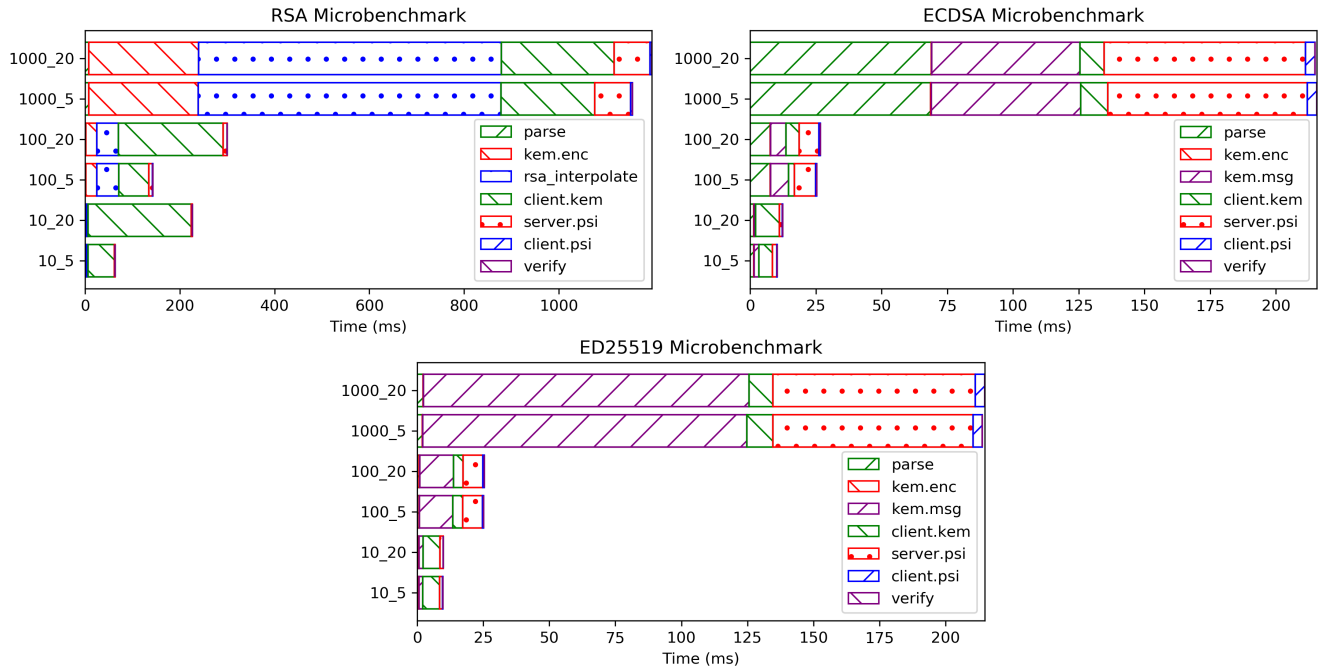


Figure 5: Microbenchmark result per each key setup. For each type of key, we vary the number of keys at the client side for 5 and 20, and we also vary number of authorized keys at the server side for 10, 100, and 1000.

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.

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

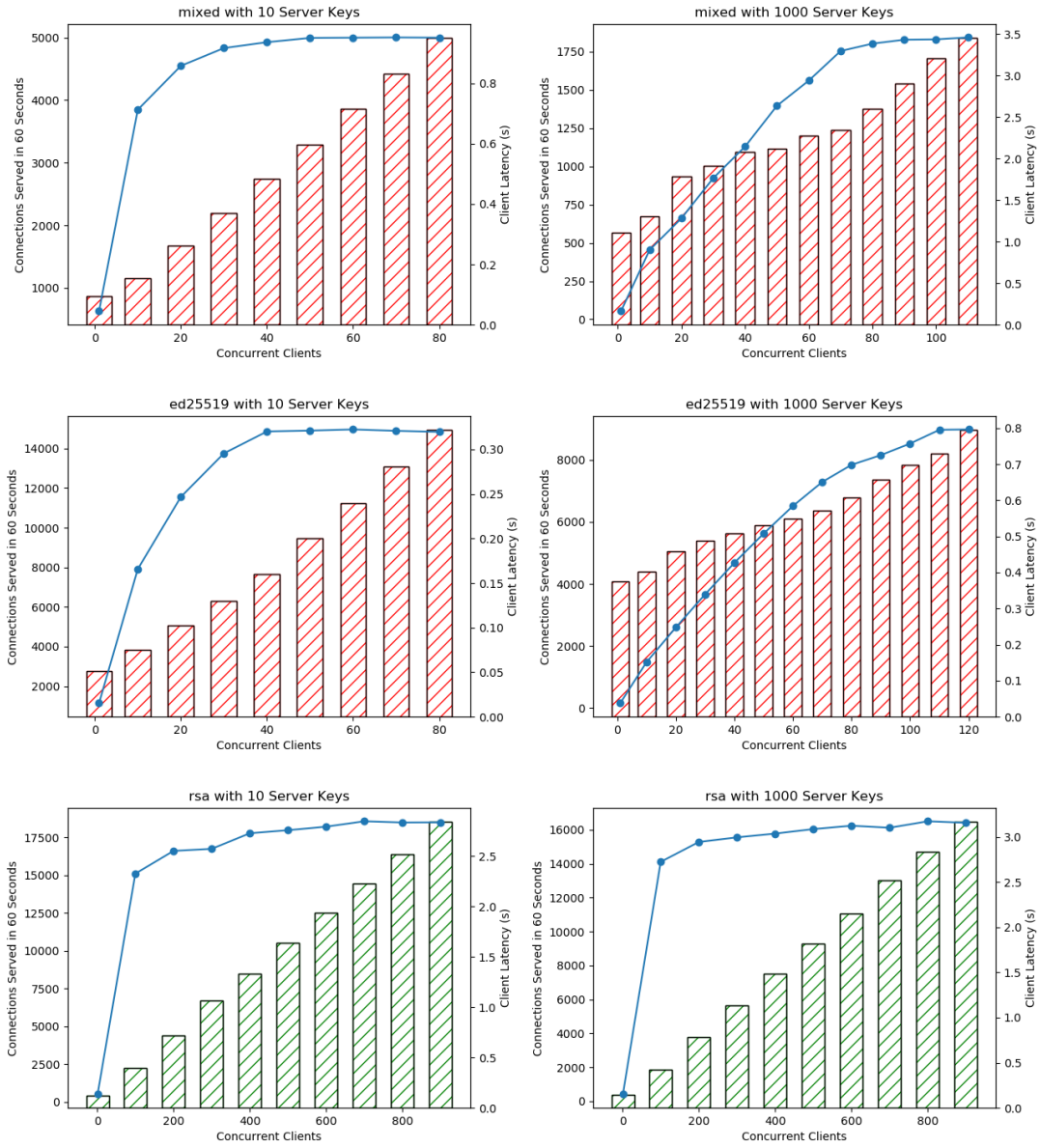


Figure 6: Macrobenchmark result for using mixed keys from the Github key statistics [9] and using only EdDSA keys, for 10 and 1000 keys on the server. In all cases, we use 3 keys on the client side. Each graph shows the number of connections processed in minutes (line, left Y-axis) and average latency 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.

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 [15]) 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

[1] M. Abdalla, M. Bellare, and P. Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In D. Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, Apr. 2001.

[2] D. Balfanz, G. Durfee, N. Shankar, D. K. Smetters, J. Staddon, and H.-C. 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.

[3] A. Barth, D. Boneh, and B. Waters. Privacy in encrypted content distribution using private broadcast encryption. In G. Di Crescenzo and A. Rubin, editors, *FC 2006*, volume 4107 of *LNCS*, pages 52–64. Springer, Heidelberg, Feb. / Mar. 2006.

[4] M. Bellare, A. Boldyreva, and J. Staddon. Randomness re-use in multi-recipient encryption schemeas. In Y. Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 85–99. Springer, Heidelberg, Jan. 2003.

[5] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sept. 2012.

[6] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In A.-R. Sadeghi, V. D. Gligor,

and M. Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, Nov. 2013.

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

[8] D. Chaum and E. van Heyst. Group signatures. In D. W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, Apr. 1991.

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

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

[11] E. De Cristofaro, S. Jarecki, J. Kim, and G. Tsudik. Privacy-preserving policy-based information transfer. In I. Goldberg and M. J. Atallah, editors, *PETS 2009*, volume 5672 of *LNCS*, pages 164–184. Springer, Heidelberg, Aug. 2009.

[12] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 213–231. Springer, Heidelberg, Dec. 2010.

[13] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In R. Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 143–159. Springer, Heidelberg, Jan. 2010.

[14] J. P. Degabriele, A. Lehmann, K. G. Paterson, N. P. Smart, and M. Strefler. On the joint security of encryption and signature in EMV. In O. Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 116–135. Springer, Heidelberg, Feb. / Mar. 2012.

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

[16] Y. Dodis, A. Kiayias, A. Nicolosi, and V. Shoup. Anonymous identification in ad hoc groups. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 609–626. Springer, Heidelberg, May 2004.

[17] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography (extended abstract). In *23rd ACM STOC*, pages 542–552. ACM Press, May 1991.

[18] C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *30th ACM STOC*, pages 409–418. ACM Press, May 1998.

[19] C.-I. Fan, L.-Y. Huang, and P.-H. Ho. Anonymous multireceiver identity-based encryption. *IEEE Transactions on Computers*, 59(9):1239–1249, 2010.

[20] M. Fersch, E. Kiltz, and B. Poettering. On the provable security of (EC)DSA signatures. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1651–1662. ACM Press, Oct. 2016.

[21] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In J. Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, Feb. 2005.

[22] G. Garimella, B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. Oblivious key-value stores and amplification for private set intersection. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event, Aug. 2021. Springer, Heidelberg.

[23] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In A. Juels, R. N. Wright, and S. De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 89–98. ACM Press, Oct. / Nov. 2006. Available as Cryptology ePrint Archive Report 2006/309.

[24] S. Jarecki, J. Kim, and G. Tsudik. Beyond secret handshakes: Affiliation-hiding authenticated key exchange. In T. Malkin, editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 352–369. Springer, Heidelberg, Apr. 2008.

[25] S. Jarecki and X. Liu. Unlinkable secret handshakes and key-private group key management schemes. In J. Katz and M. Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 270–287. Springer, Heidelberg, June 2007.

[26] S. Jarecki and X. Liu. Private mutual authentication and conditional oblivious transfer. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 90–107. Springer, Heidelberg, Aug. 2009.

[27] K. Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In D. Naccache and P. Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, Heidelberg, Feb. 2002.

[28] B. Libert, K. G. Paterson, and E. A. Quaglia. Anonymous broadcast encryption: Adaptive security and efficient constructions in the standard model. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 206–224. Springer, Heidelberg, May 2012.

[29] H. K. Maji, M. Prabhakaran, and M. Rosulek. Attribute-based signatures. In A. Kiayias, editor, *CT-RSA 2011*, volume 6558 of *LNCS*, pages 376–392. Springer, Heidelberg, Feb. 2011.

[30] M. Manulis, B. Poettering, and G. Tsudik. Taming big brother ambitions: More privacy for secret handshakes. In M. J. Atallah and N. J. Hopper, editors, *PETS 2010*, volume 6205 of *LNCS*, pages 149–165. Springer, Heidelberg, July 2010.

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

[32] B. Möller. A public-key encryption scheme with pseudo-random ciphertexts. In P. Samarati, P. Y. A. Ryan, D. Gollmann, and R. Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer, Heidelberg, Sept. 2004.

[33] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, Nov. 2016.

[34] M. Naor. Deniable ring authentication. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 481–498. Springer, Heidelberg, Aug. 2002.

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

[36] M. Nemec, D. Klinec, P. Svenda, P. Sekan, and V. Matyas. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 162–175, 2017.

[37] G. Neven, N. Smart, and B. Warinschi. Hash function requirements for schnorr signatures. *Journal of Mathematical Cryptology*, 3(1):69–87, 2009. Other identifier: 2001023.

[38] T. Okamoto and D. Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In K. Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, Feb. 2001.

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

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

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

[42] R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, Dec. 2001.

[43] M. Rosulek and N. 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.

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

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

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

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

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

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

[50] F.-C. Zhou, M.-Q. Lin, Y. Zhou, and Y.-X. Li. Efficient anonymous broadcast encryption with adaptive security. *KSII Transactions on Internet and Information Systems (TIIS)*, 9(11):4680–4700, 2015.