Proactive Two-Party Signatures for User Authentication

Antonio Nicolosi, Maxwell Krohn, Yevgeniy Dodis, and David Mazières
NYU Department of Computer Science
{nicolosi,max,dodis,dm}@cs.nyu.edu

Abstract

*We study proactive two-party signature schemes in the context of user authentication. A proactive two-party signature scheme (*P2SS*) allows two parties—the client and the server—jointly to produce signatures and periodically to refresh their sharing of the secret key. The signature generation remains secure as long as both parties are not compromised between successive refreshes. We construct the first such proactive scheme based on the discrete log assumption by efficiently transforming Schnorr's popular signature scheme into a* P2SS*. We also extend our technique to the signature scheme of Guillou and Quisquater (GQ), providing two practical and efficient* P2SS*s that can be proven secure in the random oracle model under standard discrete log or RSA assumptions.*

We demonstrate the usefulness of P2SS*s (as well as our specific constructions) with a new user authentication mechanism for the Self-certifying File System (SFS) [28]. Based on a new* P2SS *we call 2Schnorr, the new SFS authentication mechanism lets users register the same public key in many different administrative realms, yet still recover easily if their passwords are compromised. Moreover, an audit trail kept by a secure authentication server tells users exactly what file servers an attacker may have accessed—including even accounts the user may have forgotten about.*

1. Introduction

Until now, little attention has been given to *proactive two-party signature schemes* (P2SSs). In an ordinary two-party signature scheme, a private key is split between two parties, both of whom must approve and participate in the signing of messages. An attacker must compromise both parties to forge signatures on its own. However, the attacker has the entire lifetime of the public key to compromise each of the two parties. Moreover, particularly in the two-party case, the parties' roles may be asymmetric—for instance, a client may have the right to initiate signatures of arbitrary messages, while a server's role is simply to approve and log what has been signed. In such settings, an attacker may gain fruitful advantage from the use of even a single key share, unless some separate mechanism is used for mutual authentication of the two parties. Finally, ordinary two-party signatures offer no way to transfer ownership of a key share from one party to another—as the old owner could neglect to erase the share it should no longer be storing.

Proactive digital signatures allow private key shares to be updated or "refreshed" in such a way that old key shares cannot be combined with new shares to sign messages or recover the private key. While a number of proactive signature protocols have been constructed, most existing protocols are threshold schemes designed for a variable number of parties. Because these threshold schemes require a majority of participants to be honest, they do not scale down to only two parties.

This paper describes 2Schnorr, a proactive signature protocol specifically designed for two parties. 2Schnorr is an efficient protocol that is easy to implement and produces digital signatures compatible with the Schnorr [32] signature scheme. In the random-oracle model, a three-message version of 2Schnorr is provably secure against existential forgeries assuming only that discrete logs are hard. For applications with bounded concurrency, such as user authentication, a two-message version can also be proven secure under the stronger one-more-discrete-log assumption. The technique we describe is equally applicable to the Guillou-Quisquater (GQ) [20] signature scheme, producing two- and three-message 2GQ protocols based on the RSA and one-more-RSA inversion problems, respectively. We will concentrate on 2Schnorr, however, as 2GQ is completely analogous.

P2SS signatures have a natural application to the problem of user authentication, particularly for settings with many administrative realms. Within a large university, for example, it is not uncommon for a user to have five or six different shell accounts on machines in separate research groups. On the web, users typically establish accounts at dozens of different sites over time. Under such circumstances, a user whose private key or other credentials get

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8927.

compromised is unlikely to remember every place at which he needs to update his login information. Some of the sites may even be unavailable at the time the user tries to update them, at which point the user may just give up on the problem until the next time he needs one of the accounts.

Using 2Schnorr, we built a user-authentication mechanism that addresses these challenges for SFS [28]. SFS is a secure, global file system in which users gain transparent access to files from many different administrative realms after logging in with a single password. With the new authentication mechanism, every user has an ordinary Schnorr public signature key on file wherever the user has an account. The corresponding private key is split between the user and an authentication server of the user's choice. If the user's password is ever compromised, he can immediately block further unauthorized access to all of his accounts by updating his password and private key halves on this single authentication server. Moreover, from the server's logs, the user can determine exactly what servers an attacker has accessed, where on the network those accesses came from, and whether the attacker has changed the user's login information at any sites. Thus, even accounts the user may have forgotten about will be brought back to his attention if there is any risk of an attacker having accessed them.

The next section describes SFS and related work in user authentication and proactive signature schemes. Section 3 describes the 2Schnorr protocol and gives a proof of security. Section 4 describes the implementation of our user-authentication mechanism in SFS. Section 5 reports on the performance of 2Schnorr and our user-authentication scheme. Section 6 concludes.

2. Related Work

A vast number of systems have dealt with the problem of user authentication. This section describes SFS and the motivation for a new SFS user authentication mechanism. We then highlight a few other systems that have tackled user authentication on a large scale. Finally, we discuss related work in cryptography.

2.1. SFS Overview

SFS is a secure network file system designed for decentralized control and easy sharing of files across organizational boundaries. In SFS's administrative model, servers are grouped into administrative *realms* that recognize the same set of authorized users. Realms can be as large as an entire campus or as small as a single server behind a DSL line. While a simple mechanism allows one realm to "import" or recognize users from another, realms in general need not trust each other, coordinate with each other, or even know of each other's existence.

Each SFS user may have accounts in many different administrative realms. From a single client machine, users can simultaneously access servers in multiple realms. The SFS client itself has no notion of belonging to a particular realm. (In fact, SFS has no client-side configuration options that would differentiate one client from another.) Users simply access files based on whatever realms they belong to. If a user accesses a file on a server the client has never heard of, an "automounting" mechanism causes the file to spring into existence before the access completes.

SFS users have public signature keys which they register with any realms in which they have accounts. User authentication consists of digitally signing an authentication request with the corresponding private key. Each user runs a program, *sfsagent*, that attempts to authenticate her to every file server she accesses. In this way, by registering the same public key in every administrative realm, a user can transparently access files from multiple realms without worrying about administrative boundaries. Unfortunately, if a user's private key is ever compromised, the user may have to update her public key in a large number of realms. The mechanism described in this paper makes it considerably more difficult to compromise a user's key.

SFS comes bundled with a remote execution utility, *rex*, with similar functionality to the popular *ssh* [37]. Between the file system and *rex*, any SFS user authentication mechanism can cover a large fraction of the day-to-day network accesses people make to their servers.

2.2. User Authentication

Of widely used network file systems, SFS's goals are probably most similar to those of AFS [22]. AFS is a file system designed to work over the wide-area network. AFS has been particularly successful in large organizations—for instance permitting the user community of an entire university to share access to the same file systems. Unfortunately, AFS does not adapt as well to settings with many different administrative realms. AFS's security is based on the centralized Kerberos [33] authentication system in which a central authority manages all of the accounts and servers in a given administrative realm. Cross-realm authentication is possible, but requires cooperation from realm administrators. Thus, users must typically type a separate password for each realm in which they wish to access servers. Since the central Kerberos server stores a secret that is effectively equivalent to the user's password, it is inadvisable for users to have the same password in different Kerberos realms.

The SSH remote login tool supports a mode of authentication based on public keys. The user registers his private key with an agent process on the local machine, and stores the corresponding public key in a file `.ssh/authorized_keys` in his home directory on the server. SSH public key authentication is very convenient. Users therefore typically end up copying their `authorized_keys` file to all of their different accounts. Unfor-

tunately, changing public keys requires many accounts to be updated, and users are likely to forget to update accounts on infrequently used machines.

Perhaps most relevant to P2SS are the various hardware-based user-authentication systems. As smart cards and other physical security devices gain more computational power, it will become increasingly practical for them to compute digital signatures. Such configurations will be even more desirable if they can keep an audit trail of all signed messages in case the device is stolen or otherwise compromised. P2SS schemes enable such scenarios, while additionally allowing users to recover from compromised devices without changing their public keys. To compromise a user's public key permanently, an attacker would need to break the user's hardware device (or steal a backup of the user's share) *and* compromise the centralized signature server before the user had an opportunity to recover from the first event.

2.3. Related Cryptography

Related work in cryptography includes proactive cryptosystems, two-party signature schemes, and in particular several specialized two-party signature schemes designed to remain secure under various models of device compromise.

Proactive signatures. Basic multi-party signature schemes remain secure only as long as a sufficient number of parties remain uncompromised. Unfortunately, the longer the lifetime of a public key, the more realistic the threat that enough parties will be compromised to render the public key completely insecure. *Proactive cryptosystems* [29, 21] address the problem by allowing a potentially unbounded number of compromises, so long as not too many of them happen simultaneously. Specifically, there is an efficient share *update* protocol which allows parties to refresh the way in which they share the secret key. As long as a bounded number of servers are compromised between any two successive refreshes, the system remains secure.

Most proactive signatures are threshold schemes, which assume an honest majority of players. Such schemes only function in the case of $n \geq 3$ players. In fact, most threshold techniques and even definitions (e.g., robustness) are inapplicable to the two-party setting.

Two-party signature schemes. Building an ordinary two-party signature scheme is trivial. Given a secure one-party signature algorithm, let the client and server each generate its own key pair—call them $\langle K_c, K_c^{-1} \rangle$ and $\langle K_s, K_s^{-1} \rangle$, respectively. The two-party public key then consists of the two public keys $\langle K_c, K_s \rangle$. A two-party signature of a message m just consists of two independent signatures of m using K_c^{-1} and K_s^{-1}. The first signature can only be produced by the client, and the second only by the server.

Most previous work on two-party signatures has therefore focused on the problem of generating signatures that are compatible with existing one-party algorithms. Such two-party schemes allow systems to interoperate with verifiers that cannot be updated to understand new signature types. While 2Schnorr and 2GQ are the first two-party schemes compatible with Schnorr and GQ, they are hardly the first schemes to interoperate with standard one-party algorithms. Bellare and Sanhdu [6] and MacKenzie and Reiter [26] consider several flavors of two-party generation of the RSA (full domain hash) signatures (building on some previous less formal work, e.g. [10, 18]). The schemes are simple, elegant, and in most cases reducible to the basic RSA assumption. MacKenzie and Reiter [27] also give a protocol for two-party generation of DSA signatures [16]. Two-party signatures can also be viewed as a special case of general secure two-party computation [36].

Resisting compromise. Two-party signatures are also related to the notion of *key-insulated signature schemes* [13]. In this model, a server helps the client update its secret key from period to period. Though public keys remain the same across updates, signatures reflect the period in which they were created. Thus, a verifier can reject signatures produced in periods during which the client was known to be compromised. Within a given period, the client performs all signatures on its own. This has the advantage of not requiring the server's cooperation on each signature, but for our application we specifically *want* all signatures to go through the server for approval and logging. Involving the server on each signature also allows for immediate recovery from a compromised password, without the need to notify all verifiers of compromised periods.

Recent work of Itkis and Reyzin [24] on intrusion-resilient signatures combines the properties of key-insulated and proactive signatures. However, as in the key insulated model, the server only helps the client update its secret key from one time-period to the next; all the signing is done by the client alone. In the P2SS model, the actual secret does not change from one time period to the next; only the sharing of the secret changes.

The most closely related work was proposed in [25], where MacKenzie and Reiter extend their schemes from [26] to allow for *delegation* of password-checking services. Their paper describes a protocol for a novel hardware-based user-authentication mechanism. In their model, the server is almost stateless, the device contains a password protected private key, yet an attacker who captures the device cannot mount an off-line password guessing attack. By contrast, the authentication system described in this paper assumes a stateful server and is designed to work with stateless clients—as in the case of a user with only a password who wants to sit down at a new workstation and access all of her files.

Embedded in the MacKenzie and Reiter protocol is actually a fully general-purpose P2SS version of RSA. We

believe that their RSA algorithm could be used to build a user-authentication system similar to the one described in this paper. Similarly, our 2Schnorr and 2GQ schemes could replace RSA within their protocols. We note, however, that in the case of RSA, proactivization costs some efficiency because of the need to employ techniques from [17] to share the secret exponent over a much larger modulus than $\phi(n)$.

In both 2Schnorr and 2GQ, the share update protocol is very simple: a client simply sends a random element of an appropriate group to the server over a secure channel. Of course, this does not mean that proactivization is generally simple in the two-party case. Indeed, there seems to be no way to "proactivize" the trivial double signature two-party scheme. The question of *generic* proactive two-party signatures is not as trivial as in the non-proactive case. Other than the two-party RSA scheme in [25], previous two-party signatures do not appear to "proactivize" in as simple a manner as 2Schnorr and 2GQ.

3. The 2Schnorr Signing Protocol

This section specifies the 2Schnorr signing protocol and analyzes its security. Like the standard Schnorr signature scheme, 2Schnorr relies on a cryptographic hash function, H, which for the proofs we will assume behaves like a random oracle (a common assumption in cryptographic research, first formalized in [4]. Before going into the details of 2Schnorr, we briefly describe the standard Schnorr scheme.

The Schnorr signature scheme [31] was first proposed as an application of the Fiat-Shamir transformation [14]. It can be instantiated on any group $\mathbf{G}$ of prime order in which the discrete log problem is believed to be hard. Schnorr has been proven secure under the standard notion of *existential unforgeability against an adaptive chosen-message attack* [19] in the Random Oracle Model. The security has been analyzed, among other places, in [32, 30]. For concreteness, we will consider cyclic subgroups of $\mathbf{Z}_p^*$ (for large primes p) of prime order q.

The key generation algorithm produces two large primes p and q such that $q|(p-1)$, and an element g in $\mathbf{Z}_p^*$ of order q. Then it picks a random element x in $\mathbf{Z}_q^*$, and sets $y = g^x \bmod p$. The public key is $\langle p, q, g, y\rangle$, while the corresponding private key is x. The group parameters p, q, g can be safely shared between a community of users, so that y by itself can be thought of as the public key corresponding to private key x. Schnorr additionally relies on a cryptographic hash function, H, mapping arbitrary strings to elements of $\mathbf{Z}_q^*$. We will assume H has been specified as a parameter of the scheme.

To sign a message m, the holder of the private key x picks a random $k \in \mathbf{Z}_q^*$ and sets $r = g^k \bmod p$. It then computes $e = H(m, r)$, $s = k + xe \bmod q$, and outputs the signature $\langle r, s\rangle$. Note that k must be kept secret and chosen anew each time; disclosing or reusing the value of k would allow recovery of the secret key x.

To check whether a given $\langle r, s\rangle$ is indeed a signature for some message m, it suffices to know the corresponding public key $\langle p, q, g, y\rangle$ and verify that $g^s = ry^e \bmod p$, where $e = H(m, r)$.

2Schnorr is a simple two-party proactive variation of the above scheme. We call the two parties the *client* and the *server*. We assume the client is the party that wants the digital signature—it starts with the message and ends with the signature. The server simply wants to log or approve all signed messages.

The main issue in generalizing Schnorr to two parties is that if one party, say the server, could control the choice of one of the secret quantities x or k, or their public counterparts y and r, then the server would gain an advantage over the client and the resulting scheme might not be secure.

Fortunately, the parties need only agree on random values, a task usually referred to as *coin flipping* [8]. This can be implemented by first having each party choose a random share, then exchanging and combining the shares to produce the agreed upon value. Of course, the two parties might not reveal their shares simultaneously. To prevent the second party, say the server, from choosing its share after learning the client's, the server can first "commit" to its share, then "open" the commitment upon receiving the client's share.

The simplest commitment scheme for a random share r is to reveal $G(r)$ for some hash function G that behaves like a random oracle. The client cannot learn anything from $G(r)$, but will be able to check r's consistency at the end of the exchange. G need not be a random oracle, however. Any "extractable" commitment scheme will do (for instance "committing" encryption [12], which can be implemented without random oracles). However, since Schnorr relies on random oracle H anyway, G might as well be a random oracle, too. Note further that G and H take inputs of different lengths, and thus will never be evaluated on the same input. An implementation can therefore use the same cryptographic hash function—e.g., SHA-1 [15]—for both H and G.

We remark that the use of extractable commitment to chose randomness in Schnorr and GQ generalizes to other probabilistic signature schemes in which the security result depends on honestly chosen public randomness (e.g., PSS [5], PFDH [11]).

Key generation. A 2Schnorr public key is just an ordinary Schnorr public key, $\langle p, q, g, y\rangle$. However, the corresponding Schnorr private key, x, is split between two key halves, x_c and x_s, such that $x \equiv x_c + x_s \pmod q$. A public 2Schnorr key can be centrally generated along with two halves for the corresponding private key as follows:

$q \leftarrow$ a large prime
$p \leftarrow$ a larger prime such that $q|(p-1)$
$g \leftarrow$ an element of $\mathbf{Z}_p^*$ of order q
$x_c, x_s \leftarrow$ random elements of $\mathbf{Z}_q$
$y = g^{(x_c+x_s)} \bmod p$

The two private key halves are x_c and x_s.

For distributed key generation between the client and server, the client chooses p, q, g, and x_c, computes $y_c = g^{x_c} \bmod p$, and sends the server $\{p, q, g, G(y_c)\}$. The server then picks x_s and sends the client $y_s = g^{x_s} \bmod p$. Finally, the client reveals y_c, both parties compute $y = y_c y_s \bmod p$, and the public key is $\langle p, q, g, y\rangle$. To prevent the client from (maliciously) choosing "bad" group parameters [7], the server may require the client to prove it has generated the values p, q, g according to some specific algorithm. In the implementation described in Section 4, we use the method proposed by the NIST in [16], for which such a proof can be easily provided.

Signature generation. To sign a message m, the client and the server each select a random element of $\mathbf{Z}_q$—k_c for the client, k_s for the server. The two parties then exchange the three messages shown in Figure 1. First the server picks at random an ephemeral private key k_s from $\mathbf{Z}_q^*$, computes the corresponding ephemeral public key $r_s = g^{k_s} \bmod p$, and sends $G(r_s)$ (message **1**). Similarly, the client computes its ephemeral key pair $\langle k_c, r_c\rangle$ and sends the second flow of the protocol, consisting of the value $G(r_s)$ it got in message **1**, its ephemeral public key r_c, and the message m it wishes to sign. Upon receiving message **2**, the server checks that r_c belongs to the group specified by p, q and g by verifying the equality $r_c^q \bmod p \stackrel{?}{=} 1$, then computes $r = r_c r_s \bmod p$, $e = H(m, r)$, $s_s = k_s + x_s e \bmod q$, and replies with message **3**, which reveals the value of r_s. The client computes $G(r_s)$ and verifies that it matches the value received in message **1**; if so, it verifies r_s and computes s_c in a way analogous to that described above for s_s. Finally, the client sets $s = s_c + s_s \bmod q$, and obtains the pair $\langle r, s\rangle$—an ordinary Schnorr signature.

The protocol in Figure 1 requires three messages. The first message can be precomputed and sent to the client in advance, reducing network latency to a single round trip from the time the client receives the message to be signed. As discussed later, however, a system with a constant bound on the number of concurrent signatures requested by the client can simply eliminate the first message of the protocol (and remove $G(r_s)$ from the second message). The resulting two-message protocol may be more convenient to implement.

Signature verification. Signature verification is identical to Schnorr. Given a public key $\langle p, q, g, y\rangle$, a message m, and a signature $\langle r, s\rangle$, the signature is valid if and only if $g^s \equiv r y^e \pmod p$, where $e = H(m, r)$. It can be easily checked that signatures obtained from an honest execution of the signing protocol do indeed verify correctly.

$k_c \stackrel{R}{\leftarrow} \mathbf{Z}_q$	$k_s \stackrel{R}{\leftarrow} \mathbf{Z}_q$
$r_c = g^{k_c} \bmod p$	$r_s = g^{k_s} \bmod p$
$r = r_c r_s \bmod p$	$e = H(m, r)$
$s_c = k_c + x_c e \bmod q$	$s_s = k_s + x_s e \bmod q$
$s = s_c + s_s \bmod q$	

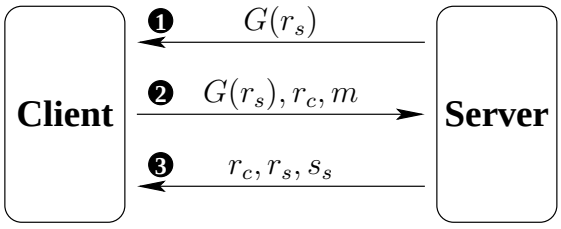

Figure 1. 2Schnorr signature protocol. m is the message being signed. k_c is chosen by the client. k_s is chosen by the server. The final signature is $\langle r, s\rangle$.

Key update. The aim of key updates is to tolerate multiple compromises of each party. Clearly, any adversary that compromises both parties between two successive key updates will learn the entire state of the system and completely break the security of the public key. However, the system should be able to withstand multiple break-ins as long as one *honest key update* happens in between.

Key updates are considerably simpler in the two-party case than in general proactive signatures. Since either party can already destroy the private key by erasing its own share, there is no need to preserve the private key when the client or server misbehaves. Further simplifying the problem, our update protocol assumes a secure channel between the client and server, because our system already requires such a channel for other purposes. (Otherwise, the client and server could use a 2Schnorr signature as part of negotiating a secure channel.)

To update the key halves x_c and x_s, the client picks a random $\delta \in \mathbf{Z}_q$ and sends it to the server over a secure channel. The new key halves x_c' and x_s' are simply computed as:

$x_c' = x_c - \delta \bmod q \qquad x_s' = x_s + \delta \bmod q$

This update algorithm is not only simple, but also quite effective in completely re-randomizing the state of the system, thus simplifying the arguments for the security proofs in the following subsections.

GQ signatures. The Schnorr signature scheme is often studied together with the GQ scheme, its "twin" based on the RSA assumption. The two can be thought of as variations of the same basic theme. Semantically, the difference between the two is that in Schnorr, all secrets are drawn from the additive group $\mathbf{Z}_q$ and their public counterparts are obtained by exponentiation on a fixed base; with GQ,

private data is instead taken from the multiplicative group $\mathbf{Z}_N^*$ (where N is the product of two big primes) and public quantities are obtained by exponentiating to a fixed exponent. Syntactically, this is equivalent to converting all additions in Schnorr into multiplications, and all multiplications (involving secrets) into exponentiations. A mechanical application of such conversion rules to our 2Schnorr protocol (described below) yields the 2GQ protocol, which enjoys analogous security guarantees based on the RSA assumption.

3.1. Security against Malicious Clients

Theorem 1 *If a malicious client can forge a signature without the approval of the server in probabilistic polynomial time (PPT) with non-negligible probability, then we can also compute discrete logs in PPT with non-negligible probability.*

Proof. Let p, q, and g be as in Schnorr public keys. Let y, in the group generated by g, be an element of which we wish to compute the discrete log. Let A be an adversary that behaves like a 2Schnorr client but then outputs a forged signature (which the server did not approve) in PPT (with non-negligible probability).

Choose a random element $x_c \in \mathbf{Z}_q$. Let $y_c = g^{x_c} \bmod p$ and $y_s = y y_c^{-1} \bmod p$. Give A public key $\langle p, q, g, y\rangle$ and private key x_c. Now ask A to forge a signature.

A can make four types of oracle query we must respond to. It can make random oracle queries to the hash functions H and G. It can make update queries to refresh the key halves. It can ask the server to start a signature (corresponding to message **1** in the protocol.) Finally, it can ask the server to endorse a signature (i.e., return the third flow in the protocol.)

Without loss of generality, we will assume throughout this paper that A does not ask for the same hash query to the same oracle twice (A could simply store previously computed values in a table). All oracle queries to G are answered with random values. As for oracle queries to H, we reply to them with random but consistent answers: however, during certain other queries (described below), we set the value of the random oracle H on certain inputs to specific, though uniformly distributed, values.

We keep a running total, Δ, of all the update requests A makes. Initially, $\Delta = 0$, but for each update request δ, we add δ to Δ (mod q).

When A makes a start signature query, we choose two random numbers $\alpha, \beta \in \mathbf{Z}_q$ and compute $r_s = g^\alpha y_s^\beta \bmod p$. We now fix some random output for the value of $G(r_s)$, and return this to A. Notice that A cannot learn anything about r_s from the random value $G(r_s)$, unless it has previously asked for the hash of r_s to the oracle G; however, the chance of A having already asked for $G(r_s)$ is negligible, since r_s is uniformly distributed in the group of order q generated by g.

When A asks us to endorse a signature with query $\{G(r_s), r_c, m\}$, we compute $r = r_c r_s \bmod p$, and fix the random oracle value of $e = H(m, r)$ such that $H(m, r) \equiv -\beta \pmod q$. Notice that the probability of having already set this value due to a previous hash query to H is negligible, since up to this point r_s is a random element unknown to A (A just saw $G(r_s)$, a random value that doesn't not leak any information about r_s) and so r is also uniformly distributed in the group generated by g. We set $s_s = \alpha + \Delta e \bmod q$, and return $\{r_s, s_s\}$. If A was talking to an ordinary server, the server would reply with $s_s = k_s + (x_s + \Delta)e \bmod q$, where k_s is the discrete log of r_s and x_s is the discrete log of y_s. Even though we cannot compute k_s and x_s, we are still computing the correct value of s_s by returning $\alpha + \Delta e$:

$$\begin{aligned} g^\alpha y_s^\beta &\equiv r_s \pmod p \\ g^\alpha &\equiv r_s y_s^{-\beta} \pmod p \\ g^\alpha &\equiv g^{k_s} (g^{x_s})^{-\beta} \pmod p \\ g^\alpha &\equiv g^{k_s} g^{x_s e} \pmod p \\ \alpha &\equiv k_s + x_s e \pmod q \\ \alpha + \Delta e &\equiv k_s + (x_s + \Delta)e \pmod q \end{aligned}$$

Because our responses to A's oracle queries are indistinguishable from those of a real server and random oracles, A will output with non-negligible probability some message and forged signature $m, \langle r, s\rangle$.

We can now rewind A's state to the first time it queried the random oracle for $e = H(m, r)$, and return some new, randomly chosen value $e' \neq e$. Such rewinding argument is quite common in analyzing the security of similar schemes [23, 3, 2]: intuitively, since A only makes polynomially many oracle queries and e' is statistically indistinguishable from e, there is a non-negligible probability that A will again forge a signature for the same m and r, yielding a second signature $\langle r, s'\rangle$ for m, with $s' \neq s$. The exact probabilistic analysis is based on the *forking lemma* of [30], and is quite similar to the one for the standard Schnorr signature scheme.

By the verification property, $\langle r, s, e\rangle$ and $\langle r, s', e'\rangle$ satisfy the following two congruences:

$$\begin{aligned} g^s &\equiv r g^{xe} \pmod p \\ g^{s'} &\equiv r g^{xe'} \pmod p \end{aligned}$$

By memberwise division of the above congruences, we can compute x, the discrete log of y, as follows:

$$\begin{aligned} g^{s-s'} &\equiv g^{x(e-e')} \pmod p \\ s - s' &\equiv x(e - e') \pmod q \\ x &= (s - s')(e - e')^{-1} \bmod q \end{aligned}$$

∎

3.2. Security against Malicious Servers

Theorem 2 *If a malicious server can forge a signature without the client's help in PPT with non-negligible probability, then we can also compute discrete logs in PPT with non-negligible probability.*

Proof. Let p, q, g, y be a challenge in which we need to find the discrete log of y. Let A be an adversary that acts as a 2Schnorr server and can forge signatures. We choose a random $x_s \in \mathbf{Z}_q$ and compute $y_s = g^{x_s} \bmod p$ and $y_c = yy_s^{-1} \bmod p$. Give A public key $\langle p, q, g, y\rangle$ and private key x_s, and ask it to forge a signature.

As before, A can make four types of oracle query: random oracle queries (to H or to G), update queries, asking the client to initiate the signature of a particular message, and asking the client to finish computing the signature of a message for which an initiate query was previously done.

The two random oracles are treated as before. For update queries, we are now even allowed to choose a random δ ourselves: since the client refreshes its key half by subtracting δ, here Δ is defined as the running total of the values $q - \delta$ for each update (mod q).

When A asks to initiate the protocol, we choose two random numbers $\alpha, \beta \in \mathbf{Z}_q$ and compute $r_c = g^\alpha y_s^\beta \bmod p$. Notice that we allow A to choose the message m it wants to sign, but still, A must provide its "commitment" $G(r_s)$. Since G behaves like a random oracle, A must have asked for the value of $G(r_s)$—otherwise it will have only a negligible chance of guessing the correct value needed to open, in the third flow of the protocol, the commitment sent in the first message. Hence, upon receiving an initiate query, we can perform a simple lookup for $G(r_s)$ in the table containing the pairs $\langle$*query, answer*$\rangle$ for all the oracle queries that A has done to G so far.[1] Then, we compute $r = r_c r_s \bmod p$ and fix the random oracle value of $e = H(m, r)$ such that $H(m, r) \equiv -\beta \pmod q$: e is still uniformly distributed in Z_q since we chose β at random.

The complete signature queries are handled virtually identically to the proof of Theorem 1, reversing the s and c subscripts in variables. The only difference is that when completing a signature, we must return $\langle r, s\rangle$ instead of $\langle r_c, s_c\rangle$. r and s are easy to compute given r_c, r_s, s_c, s_s, all of which we have.

Since the interaction of A with the oracles thus simulated is indistinguishable from a real attack on the 2Schnorr signing protocol, with non-negligible probability A will forge a signature $\langle r, s\rangle$ for a message m that the client didn't finish signing. Therefore, using the same technique as in Theorem 1, we can compute the discrete log of y with non-negligible probability. ∎

[1] As we mentioned, we can replace G by any "extractable" commitment which would also allow us to recover r_s in our simulation.

3.3. Security against Mobile Adversaries

In practice, an attacker may compromise both the client and the server, as long as there is an honest refresh between two successive compromises. Such a mobile adversary can be modeled in two different ways, depending on whether it sees any commitment to the values y_c and y_s. If, for example, message **3** of a signature is sent in cleartext over the network and the adversary subsequently compromises either the client or the server before an intervening update, then the adversary can verify that $g^{s_s} = r_s y_s^e$ or $g^{s_s} = r_s (y y_c^{-1})^e$.

In the model we consider here, the adversary is slightly weaker and cannot see the values of r_s and s_s communicated when neither party has been compromised. This model was inspired by our implementation, which always encrypts traffic between the client and server. However, there is a small subtlety—the adversary may capture encrypted copies of message **3**, and later obtain the session key for decrypting them when it compromises the client or server. Even if the client and server quickly erase any keys used to encrypt messages to each other, in reality there will be a small window of vulnerability after the message is transmitted and before the key is erased. For this reason, we believe that the stronger adversary is actually the more useful model. We will consider such adversaries in future work.

One convenient property of the model with weaker adversaries is that a simulator can always assume a key is refreshed right before a party is compromised—the adversary will have no way to notice that y_c and y_s have just changed. That, in turn, means that one can assume without loss of generality that the adversary always compromises either the client or the server, which reduces the number of cases to consider. (If there is a period when neither is compromised, we can pretend the adversary has asked to compromise, say, the client; if the adversary then wants to compromise the server, we pretend it also asks for a key refresh.)

Theorem 3 *The three-message 2Schnorr protocol is secure against an adversary that selectively compromises both the client and the server, provided an honest share update has occurred before the other party has been compromised.*

Proof. To prove the theorem, we might assume that if a mobile PPT adversary A could forge a signature, then we could construct a new algorithm A' that could compute discrete logs with non-negligible probability. In this case, our construction of A' would involve answering A's random oracle queries, update queries, server initiation queries, and queries corresponding to three flows of the protocol. In other words, this new A' would combine the techniques in Theorems 1 and 2. To avoid repetition of previous arguments, we instead use a higher-level approach and prove that the existence of a PPT mobile forger contradicts at least one of the previous two theorems.

Suppose that by repeatedly compromising the client and server (while always permitting an honest key update in between), some PPT adversary A is able to forge signatures with non-negligible probability ε. If A outputs forgery $m, \langle r, s\rangle$, it must with all but negligible probability have queried the random oracle for $e = H(m, r)$. Call this the *crucial* query. Since the client and server can never be simultaneously compromised, one (or both) of them will have probability $\geq 1/2 - \mathsf{negl}$ of being uncompromised during crucial queries. We distinguish the two cases.

Case 1. Suppose that at least half of the time, A makes crucial queries while *not in control of the server*. In other words, with probability $\varepsilon/2$, after several (non-simultaneous) compromises of both parties, A successfully forges a signature $\langle r, s\rangle$ for some message m after querying the oracle for $e = H(m, r)$ at a time when the server was not compromised.

We now show how to construct a malicious client A' that, using A as a black box, forges signatures without the server's approval with (non-negligible) probability $\varepsilon/2$.

A' is given the public key PK $= \langle p, q, g, y\rangle$, the client's share x_c, and access to hash oracles H and G and to a server oracle willing to engage in as many protocol runs as A' wishes. In order to forge a signature, A' runs A, giving it public key PK. Each time A asks for access to a compromised client, A' gives A its own secret share x_c (which must have been refreshed since the last server compromise). A' then responds to any oracle queries from A by simply forwarding them to the real server, H and G oracles.

When A asks to compromise the server, A' must give it some share x_s and then emulate the client in a manner that will be indistinguishable from a real client. Since A' cannot compromise the server to learn the real value of x_s, it instead employs the technique from Theorem 2: A' generates a random $x_s \in \mathbf{Z}_q$ and gives this x_s to A. A' then replies to A's H-oracle queries with carefully prepared (though still uniformly distributed) values, instead of returning values from the real hash oracle H. Because our coin flipping protocol ensures r is random, there is only negligible probability of A using a value of $H(m, r)$ it requested before compromising the server.

Since we are assuming an honest key update will occur between any two consecutive break-ins, and since each time the sharing of the secret key x as $x_c + x_s$ is completely re-randomized, these simulations will proceed exactly as A expects. Hence, with probability $\varepsilon/2 - \mathsf{negl}$, A will produce a forged signature $\langle r, s\rangle$ for some message m. By our assumption, A asked for $H(m, r)$ while not in control of the server, and the construction of A' guarantees that $e = H(m, r)$ is a *real* random oracle response; A' only "fakes" H when the server is compromised. Thus, $\langle r, s\rangle$ is a *real* signature for m, one that satisfies the verification property. A' will therefore output $\langle r, s\rangle$ as its own forgery.

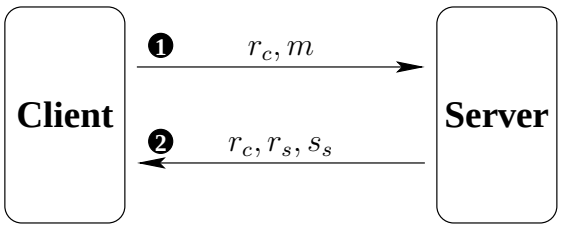

Figure 2. A two-message signature protocol, for applications with a bounded number of concurrent signature requests from the client. The constants are computed as in Figure 1.

But the fact that A' forges signatures with non-negligible probability while compromising only the client contradicts Theorem 1.

Case 2. Suppose now that A is most successful in its forgery when it asks for the relevant hash value $e = H(m, r)$ while *not in control of the client*. We can then construct a malicious server A'' that, using A as a black box, again forges a signature with probability $\varepsilon/2 - \mathsf{negl}$. The construction is analogous to Case 1, where we "swap" the techniques used to handle A's compromise requests. A'' uses the simulation of Theorem 1 when A breaks into the client, while it gives A x_s and returns real values of H when A compromises the server. This leads to a contradiction of Theorem 2.

Since at least one of the two cases above must occur at least half of the time, it follows that 2Schnorr is secure against mobile adversaries. ∎

3.4. Two-Message Protocol

In certain cases, it may be desirable to compute signatures with only two messages. If there is a constant bound on the number of concurrent signatures a client requests, the first message can be eliminated (and the second flow consequently modified removing the hash value $G(r_s)$), yielding the simpler two-message protocol in Figure 2.

From Theorem 1, we see that the two-message protocol is secure against a malicious client. Indeed, in this case the malicious client is just less powerful than in the previous scenario (since it does not have access to the initiate signature oracle): If an adversary A could forge signatures by acting as a two-message client, one could trivially build a three-message adversary A' in terms of A.

To prove the two-message protocol secure against a malicious server, however, we must rely on a stronger assumption than the difficulty of computing discrete logs, and assume a constant bound t on the number of concurrent runs of the protocol[2]. The intuitive reason behind the strengthen-

[2]More precisely, we assume that the client is willing to initiate an un-

ing of the assumptions is that in the two-message protocol the server can control the value of r by (maliciously) choosing $r_s = rr_c^{-1} \bmod p$. In this way, the server can be able to compute $H(m, r)$ before sending the value r_s to the client. This would break the simulation described in the Subsection 3.2, so that the proof does not go through. Still, it is not clear how this attack could help the server in forging a signature $\langle r, s\rangle$, since the choice of $r_s = rr_c^{-1}$ leaves the server with the problem of computing its the discrete log, k_s; hence, the server will not be able to compute the correct $s_s = k_s + (x_s + \Delta)e \bmod q$ needed to complete the signature $\langle r, s\rangle$.

Indeed, we show that the modified scheme *is secure* under the assumption that the *known-target discrete log problem* (DL-KT) [1] is hard. The DL-KT problem consists of getting some number n of discrete log challenges in the same group, then computing their discrete logs while making at most $n - 1$ queries to a discrete log oracle. Although the assumption that the DL-KT problem is hard for any polynomially-bounded n is relatively new, it has already been used quite a bit in proving the security of various schemes [1, 3, 2, 9]. However, in all these other cases the claimed result follows almost trivially from such assumption. On the contrary, our application is considerably more involved: it requires an additional "twist" in the reduction argument, namely the assumption on the bounded concurrency, and the development of some novel ideas in the probabilistic analysis.

Theorem 4 *If a malicious server, interacting with a client with bounded concurrency, can forge a signature without the client's help in PPT with non-negligible probability, then we can also solve the DL-KT problem in PPT with non-negligible probability.*

Proof. Let p, q, g define a group as usual. Let $z_0, z_1, z_2, \ldots$ be challenges of which we want to compute the discrete logs. Let t be the concurrency bound of the client. Let A be an adversary that acts as a 2Schnorr server and forges signatures in PPT. We will compute the discrete logs of n challenges for some number $n > 0$, while making only $n - 1$ queries to a discrete log oracle.

We first choose a random $x_s \in \mathbf{Z}_q$. Let $y = z_0$ (i.e., the first challenge), $y_s = g^{x_s} \bmod p$, and $y_c = yy_s^{-1} \bmod p$. We give A public key $\langle p, q, g, y\rangle$ and private key x_s. Finally we ask A to forge a signature.

Again, A can make four kinds of oracle queries. It can query the random oracle, ask for a key update, ask the client to initiate the protocol on some message, or ask the client to complete and output a signature. We emulate the random oracle H in a completely honest manner, by replying to each query with a random value. More specifically, we choose a random (and sufficiently long) list of values for the hash oracle H *before* we even start executing A: we will use these values one after the other to answer A's queries, no matter what specific value it's asking for. Also, we randomly choose and *fix* the entire random tape of A. Finally, we also choose update values δ randomly, and keep a running sum $\Delta = \sum_\delta (q - \delta) \bmod q$. To put it differently, all the randomness we need for the entire simulation is chosen and fixed.

When A asks us to initiate the signature on a message m, we set $r_c = z_j$ for some challenge z_j we have not yet used, and we send A the message $\{r_c, m\}$.

When A wishes us to complete the signature of some m, it will send us $\{r_c = z_j, r_s, s_s\}$. Let $r = r_c r_s \bmod p$ and $e = H(m, r)$. We query our own discrete log oracle to find the discrete log of $r_c(y_c g^\Delta)^e \bmod p$. Let s_c be this logarithm. Let k_c be the log of r_c and x_c be the log of y_c. Both k_c and x_c are unknown to us. However:

$$\begin{aligned} g^{s_c} &\equiv r_c(y_c g^\Delta)^e \pmod p \\ g^{s_c} &\equiv g^{k_c}\left(g^{x_c+\Delta}\right)^e \pmod p \\ s_c &\equiv k_c + (x_c + \Delta)e \pmod q \end{aligned}$$

Thus, s_c is exactly as it should have been. We compute $s = s_c + s_s \bmod q$, and output the signature $\langle r, s\rangle$.

Since we are exactly simulating the attack scenario, A will eventually output, with non-negligible probability, a message m and forged signature $\langle r, s\rangle$.

We then rewind A's state to the time it queried the random oracle for the value of $H(m, r)$ (say this was the i^{th} A did to the oracle H.) Call this query *crucial*. This time, instead of using the value in the "list of randomness" we prepared before beginning executing A, we discard this value, and answer such *crucial* query with a new, random value. We then continue the simulation as in the first execution: in particular, we will continue using our "list" to reply to random oracle queries, but will use brand new (already prepared) challenges z_j.

As we will shortly prove in Lemma 5, there is a non-negligible probability that A will forge again the same message m with randomness r but different values for $H(m, r)$ (plus some additional property will hold; see below). Once we come up with two signatures for the same message m, the same randomness r but *different* hash values $e \neq e'$ (and hence $s \neq s'$), it is just a matter of modular arithmetic to recover x (see Theorem 1.) Now, given x, we can answer all the challenges z_j as follows.

For each initiate query that the adversary decided not to complete (i.e. he initiated a run of the protocol but then it decided *not* to complete it), we have used one of the challenge z_j without "consuming" any query to our discrete log oracle. Therefore, we can use it now to find the discrete log of z_j. So we concentrate on the initiate queries which were completed by A.

Next, since $y = g^x \bmod p = z_0$, x itself is the answer to

bounded number of concurrent executions of the protocol, but it will only complete one of the t most recent pending runs.

the challenge z_0 (and we did not consume any discrete log queries for getting x). Moreover, once we know both x and x_s, we can also compute $x_c = x - x_s \bmod q$. Given x_c, we can recover the value k_c—the discrete log of each z_j—from each of the $s_c = k_c + (x_c + \Delta)e \bmod q$ values we asked our discrete log oracle to compute.

There is only one subtle problem: we have to ensure that we consume at most one discrete log query per each value z_j (for $j \geq 1$), i.e. per each (completed) initiation query of the adversary. Such queries can be divided into three parts: the queries initiated and completed before the crucial hash query i, the (at most t) queries initiated before but completed after the critical query, and the queries initiated and completed after the critical query. There is no problem with the first and the last kind of queries. Indeed, they both consumed exactly one z_j and utilized exactly one call to the discrete log oracle (recall, we use fresh z_j's in the second run). However, the second category could present problems: each query utilizes one z_j, but can potentially call the discrete log oracle *twice*, i.e. once in each run of A. Here is where we will use that t is bounded (by a constant). For each such "semi-completed" signature query m_j, let r_j be the corresponding randomness in the first run, and call the t hash queries $H(m_j, r_j)$ *important* (in the first run). Notice, important queries could appear both before and after the critical query i, since the server can control the randomness by choosing a "cheating" value of $r_{s,j} r_j r_{c,j}^{-1} \bmod p$ for each m_j, and query $H(m_j, r_j)$ way before the initiation query for m_j. Similarly, we can define these t important hash queries in the second run (corresponding to the same m_j, $r_{c,j}$ but possibly different values r'_j, $r'_{s,j}$). However, since we chose all the randomness for our hash query answers at the beginning and reused this randomness in the second run, we do not have to make the extra t calls to the discrete log oracle provided *the important queries in the first and the second run have the same indices*. Indeed, in this case we would return the same value e_j for the important query j in both runs, and therefore will need to compute the discrete log of two very similar values $r_{c,j}(y_c g^{\Delta_j})^{e_j} \bmod p$ and $r_{c,j}(y_c g^{\Delta'_j})^{e_j} \bmod p$. It is clear that these discrete logs differ by $(\Delta_j - \Delta'_j)e_j \bmod q$, which is a known value, so we can compute the discrete log $s'_{c,j}$ by returning $(s_{c,j} - (\Delta_j - \Delta'_j)e_j) \bmod q$. We will argue in Lemma 5 that (for a bounded t) the important queries will indeed be the same with non-negligible probability.

To summarize, if we are lucky that the following three conditions hold during the "double" run of A, we utilize one less discrete log query than the number of discrete log challenges we are computing, thus breaking the DL-KT assumption:

1. A succeeded the first time on some critical query i.
2. A succeeded the second time on the same critical query i, but the i^{th} response we gave was different.
3. A used the *same* (up to) t important queries in the first and second run.

We argue that the above three conditions hold (with non-negligible probability) in the following final lemma, which completes the proof. ∎

Lemma 5 *Let ε be the probability of A successfully producing a forgery in one run, and δ be the success probability of satisfying the above three conditions in the "double" run of A. Assume also that A makes at most q_{hash} random oracle queries in one run, and t is the maximum number of concurrent signature queries the client initiates. Then $\delta \geq \varepsilon^2 / q_{\mathsf{hash}}^{t+1} - \mathsf{negl}(k)$. (here k is the security parameter and $\mathsf{negl}(k)$ is a negligible function in k).*

Proof. Without loss of generality, we will assume throughout the proof that A always calls the random oracle H to validate all issued signatures and its final forgery—otherwise A has at most a negligible success of probability, which is consumed in the $\mathsf{negl}(k)$ term. Let $i \in [1, q_{\mathsf{hash}}]$ be an index and $J \subset [1, q_{\mathsf{hash}}]$ be a subset of at most t indices. Let $\varepsilon_{i,J}$ be the probability that an execution of the adversary will succeed in forging a signature, and that during this execution the crucial query corresponding to the forgery will be query number i, and that the set of (at most t) important queries will be exactly J. It follows that: $\sum_{i,J} \varepsilon_{i,J} = \varepsilon$.

Next, recall δ is the probability of success of the above experiment. For any i and J defined as above, let $\delta_{i,J}$ be the probability that both executions of the adversary will output forgeries, and that in both executions the critical query will be i and the important queries will be exactly J. Again, it follows that $\sum_{i,J} \delta_{i,J} = \delta$.

We claim that if the size of the universe of hash responses has size $L \geq 2^\ell$, we have:

$$\delta_{i,J} \geq \varepsilon_{i,J}^2 - \varepsilon_{i,J} 2^{-\ell} \tag{1}$$

Assuming Equation (1) is true, by using Cauchy-Schwartz inequality[3] we get:

$$\begin{aligned}
\delta &= \sum_{i,J} \delta_{i,J} \geq \sum_{i,J} (\varepsilon_{i,J}^2 - \varepsilon_{i,J} 2^{-\ell}) \\
&= \sum_{i,J} \varepsilon_{i,J}^2 - \varepsilon 2^{-\ell} \geq \frac{1}{q_{\mathsf{hash}}^{t+1}} \Big(\sum_{i,J} \varepsilon_{i,J} \Big)^2 - \frac{\varepsilon}{2^\ell} \\
&= \frac{\varepsilon^2}{q_{\mathsf{hash}}^{t+1}} - \frac{\varepsilon}{2^\ell} = \frac{\varepsilon^2}{q_{\mathsf{hash}}^{t+1}} - \mathsf{negl}(k)
\end{aligned}$$

It remains to show the validity of Equation (1). For any fixed i and J, consider all the random input to the system,

[3] Cauchy-Schwartz inequality states that:
$(\forall N \in \mathbf{N})(\forall a_1, \ldots, a_N \geq 0).[\sum_i a_i^2 \geq 1/N (\sum_i a_i)^2]$

including the random tape of the adversary, the q_{hash} oracle responses for values of H, and any other randomness needed by the simulator. Split the random inputs into the two parts: the value returned in response to the ith request for a value of H (call it e), and everything else, which we will call R. Now consider a matrix with one column for every possible value of e, and one row for every possible value of R. For each cell of the matrix, put a dot in the cell if on the corresponding input the adversary outputs a successful forgery with i the crucial query and J the set of important queries. Let $\varepsilon_{i,J,R}$ be the probability of success conditioned on R, i.e. the density of marks in row number R of our matrix. We have that:

$$\varepsilon_{i,J} = \frac{1}{|R|}\sum_{R} \varepsilon_{i,J,R} \tag{2}$$

Next, let $\delta_{i,J,R}$ be the probability of success of the whole experiment (with rewinding), conditioned on the fact that the row was R, i.e. conditioned on the fact that the randomness used in the entire experiment (except for the answer to query number i) is described by R. We want to estimate the probability that, when we selected at random the answer to the i^{th} query *twice*, we got a dot both times and the answers chosen—e_1 and e_2—were different. Indeed, since i and J are fixed, in this case A succeeds twice and uses the same important/critical queries. Since these two runs are now *independent*, the needed probability is simply computed as:

$$\delta_{i,J,R} = \varepsilon_{i,J,R}(\varepsilon_{i,J,R} - 2^{-\ell}) = \varepsilon^2_{i,J,R} - \varepsilon_{i,J,R}2^{-\ell} \tag{3}$$

Finally, by conditioning $\delta_{i,J}$ on the value of R, using Equation (2), (3), and Cauchy-Schwartz again, we get:

$$\begin{aligned}\delta_{i,J} &= \frac{1}{|R|}\sum_{R}\delta_{i,J,R} = \frac{1}{|R|}\Big(\sum_{R}\varepsilon^2_{i,J,R} - \sum_{R}\varepsilon_{i,J,R}2^{-\ell}\Big)\\ &= \Big(\frac{1}{|R|}\sum_{R}\varepsilon^2_{i,J,R}\Big) - \frac{\varepsilon_{i,J}}{2^{\ell}} \geq \Big(\frac{1}{|R|}\sum_{R}\varepsilon_{i,J,R}\Big)^2 - \frac{\varepsilon_{i,J}}{2^{\ell}}\\ &= \varepsilon^2_{i,J} - \varepsilon_{i,J}2^{-\ell}\end{aligned}$$

Equation (1), and hence the overall bound, follows. ∎

3.5. Security Against Mobile Adversaries in the Two-Message Protocol

We now show the equivalent of Theorem 3 for the Two-Message protocol. Unfortunately, our previous analysis no longer holds. In that earlier proof, we argued that A' could effectively simulate a client without knowing its key. Indeed, A' could output phony values of $H(m,r)$ because coin-flipping assured that r is truly random. In the Two-Message protocol, we have no such assurances. Rather, we need to consider the lower-level details of the previous reduction. Using the same definition of a "mobile adversary" as above:

Theorem 6 *If a mobile bounded-concurrency PPT adversary can forge a 2Schnorr signature with non-negligible probability, then we can also solve the DL-KT problem in PPT with non-negligible probability.*

Sketch of Proof. Consider the same experimental setup as in Theorem 4. z_i are the discrete log challenges, t is the concurrency bound of the client, and $y = z_0$ is the first discrete log challenge. Next, assume a PPT mobile adversary A that can forge 2Schnorr signatures with non-negligible probability. The goal is now to construct an algorithm A' that calls upon A to solve the DL-KT problem. When it runs A, A' must respond to six different oracle queries. As usual, A' must respond to random oracle queries to H. It does so by outputting a random value except for when it is preprogrammed to output $-\beta = H(m,r)$ as seen later. The other five queries are specific to the two states that A can be in: *server-compromise state* (SCS) or *client-compromise state* (CCS).

If A starts the experiment in SCS, then A' chooses $x_s \in \mathbf{Z}_q$ at random, computes $y_c = y y_s^{-1} \bmod p$, and gives x_s to A. If A starts in CCS, then A' chooses $x_c \in \mathbf{Z}_q$ at random, computes y_s, and sends x_c. We now consider both cases. A can ask three oracle queries specific to SCS: client signature initiation, client signature completion, and refresh/transition to CCS. In the first two types of queries, A' follows the simulation used in Theorem 4. It chooses $r_c = z_j$ for some challenge z_j not yet used and responds to a completion request with a call to its discrete log oracle. A can also request a key-refresh, thereby transitioning to CCS. When A requests a refresh, A' picks a random $x'_c \in \mathbf{Z}_q$, computing $y'_s = y g^{-x'_c} \bmod p$, and sending x'_c to A.

In CCS, A only asks two queries: server signature endorsement, and refresh/transition to SCS. A' endorses signatures using the y'_s it computed in the transition, following the same simulation used in Theorem 1; it generates at random $\alpha, \beta \in \mathbf{Z}_q$ and sets $r_s = g^{\alpha} y'^{\beta}_s \bmod p$. When it computes $r = r_s r_c \bmod p$, A' stores $H(m,r) \equiv -\beta \pmod q$ in a table for future output. When A requests a refresh A' will generate a new x''_s and corresponding y''_c as usual, returning x''_s to A.

The analysis of A' proceeds as in Theorem 1 and Theorem 4. Whether in SCS or CCS, A cannot distinguish between running with A''s oracle values and running in a real system. This is guaranteed by A''s access to the discrete log oracle in SCS, the uniform random distribution of β in CCS, and the random refresh of the system between the two states. If A produces a forgery for message m, it must have at some point computed a *crucial* query, $H(m,r)$. If this query occurred when A was in CCS, we use the rewinding argument given in Theorem 1 to compute x. If in SCS, the rewinding argument given in Theorem 4 applies, and A' can compute x_c knowing x_s, thereby computing x. In either case, A' recovers the discrete logs of

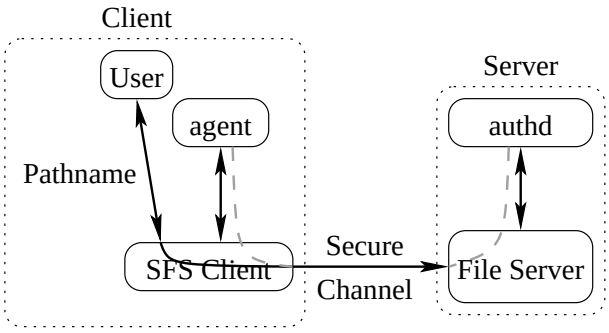

Figure 3. SFS user-authentication architecture.

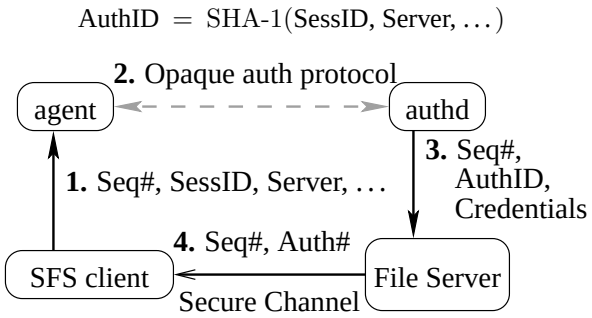

Figure 4. Messages exchanged during the user-authentication process. The authentication protocol between the agent and authd is opaque to the core file system software.

the other z_i after it recovers x. A variation of Theorem 4's bounded concurrency argument applies to guarantee that A' did not exhaust its quota of queries to the discrete log oracle. Thus, A' is shown to be a PPT algorithm that can compute DL-KT with non-negligible probability. ∎

4. Implementation

Figure 3 illustrates the major components of SFS involved in user authentication. The file system client and file server communicate over a TCP connection, encrypting and MACing all traffic to obtain a secure channel. User authentication itself is actually performed by processes external to the file system. On the client, every user runs an *agent* program responsible for authenticating it to remote servers. On the server side, a program *authd* is responsible for validating authentication requests and translating them into credentials meaningful to the file server.

When a user accesses a file server for the first time, the file system client delays the access and asks the user's agent to authenticate her to the server. The agent then communicates with the server's authd to obtain appropriate privileges for the user. The agent and authd communicate through the file system's secure channel, but the file system views their messages as opaque byte arrays. Thus, new authentication protocols can be implemented without modifying the file system software.

Figure 4 shows the interface between the file system, agent, and authd. Every secure channel between a client and server is identified by a unique session ID, *SessID*. SessID, when hashed together with the server's name, public key, and certain other information, produces a value called *AuthID*. When the SFS client asks a user's agent to authenticate her to a server, it sends the agent the SessID of the session with that server, a sequence number, *Seq#*, identifying the authentication request within that session, and several other pieces of information including the name of the server. The agent computes AuthID and then communicates with the server's authd. If the authentication protocol succeeds, the authd informs the file server of the user's Seq#, AuthID, and credentials. The server then returns a short handle *Auth#* to the client, which the client subsequently uses to tag all file system requests on behalf of that user.[4]

In the original SFS authentication system, authd keeps a mapping of users' public keys to credentials, while the agent keeps one or more private keys in memory. The authentication protocol consists of the user digitally signing {Seq#, AuthID}. The original protocol used Rabin-Williams [34] digital signatures.

In addition to validating file server users, SFS's authd plays a separate role as a repository of users' encrypted private keys. SFS users can store encrypted copies of their private keys with the authd of their "primary" SFS server. After logging into a client machine, users typically connect to their primary server's authd over the network, authenticate themselves through the SRP [35] secure password protocol, and then retrieve their encrypted private keys. (Users also end up securely downloading server public keys this way; see [28] for details.)

4.1. Implementing 2Schnorr in SFS

Integrating 2Schnorr in SFS was relatively straightforward, as the original user authentication protocol already consisted of a simple digital signature on {Seq#, AuthID}.

On the server side, we made several modifications to authd. We extended it to support both Schnorr and Rabin public keys. We modified the server's encrypted private-key repository functionality, so that it now optionally holds both an encrypted half of a user's private key and an unencrypted one. We added an option to the RPC by which users update their login information so as to update the two key halves whenever users change their passwords. Finally, we added

[4] SFS could equally well have chosen to tag requests with Seq#, but Auth# is a shorter and therefore slightly more convenient value.

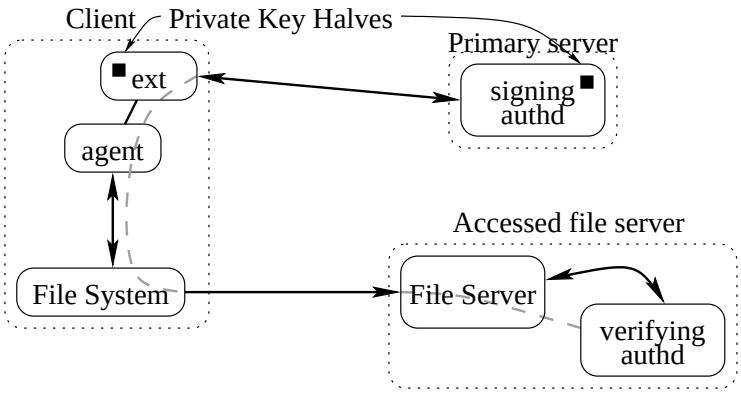

Figure 5. Implementation of proactive signatures in SFS

a SIGN RPC that implements the server side of the two-message 2Schnorr protocol.

In order to access the new SIGN RPC, a user must first authenticate himself to the server. The simplest way is through SRP. When a client downloads a user's encrypted private key half, it is permitted to keep the connection open to the server for issuing SIGN requests. The server is currently willing to endorse two types of message—login requests, and requests to change the user's public key on a particular server. Both types of messages include an AuthID, which authd computes and verifies. Computing AuthID involves hashing, among other things, the name and public key of the server being accessed and the type of service being requested (remote login, file server, etc.). Authd logs this information, leaving a complete audit trail in case an attacker steals a user's password.

On the client side, rather than hard-code 2Schnorr into the agent, we instead implemented an extension facility by which arbitrary external programs can plug into the agent and offer to attempt user-authentication. Figure 5 illustrates the complete system. Upon loading the 2Schnorr private key half, an external authentication process *ext* plugs into the agent, keeping open a connection to the user's primary authd, which we call the *signing authd*. When the user accesses a new file server, the agent queries the ext process, which executes 2Schnorr with the signing authd to produce an ordinary Schnorr signature. The *verifying authd* on the server that the user is accessing then verifies the Schnorr signature to authenticate the user.

Several other implementation details are worth mentioning. The new authd can actually store two private keys for a user. This is important so that a user who changes her public key can access both the old and new private keys for a time. On the client side, while ext is waiting for the server to endorse a signature, it precomputes $g^{k_c} \bmod p$ for the next signature, to reduce latency. Also, in order to compute the value $s_c = k_c + x_c e \bmod q$, the client actually computes $s_c = \left(k_c \left(e^{-1} \bmod q\right) + x_c\right) e \bmod q$ to thwart any timing attacks based on non-constant time of the modular reduction. s_s is computed similarly.

5. Performance

This section evaluates the performance of 2Schnorr and its impact on SFS. The two most important effects of 2Schnorr are on the responsiveness of the client and on consumption of server CPU time. In both cases, we compare the new 2Schnorr authentication protocol to the original SFS authentication protocol, which is based on an optimized, non-interactive Rabin signature scheme. While 2Schnorr itself is noticeably slower than Rabin, user-authentication is not on the critical path for file system performance. Thus, we show that 2Schnorr has an acceptable impact on client responsiveness. Furthermore, while 2Schnorr is considerably more expensive than Rabin on the server-side, it is still cheap in absolute terms, given the the relative infrequency of user authentication requests.

We measured the 2Schnorr and Rabin algorithms both in isolation and as part of a file system access that required user authentication. We used three separate machines in our experiments: a *verifying-authd* server, a *signing-authd* server, and a client. The *verifying-authd* machine served the file system we used in the file access benchmark, while the *signing-authd* performed the 2Schnorr server-side protocol (and hence was not used in the Rabin experiments). All three machines had 1.75 GHz Athlon processesors and sufficient memory so that no paging activity was detected during any of the trials. The three machines were connected by switched 100 Mbit ethernet, with round trip latencies below 0.2 ms between every pair of machines. The *verifying-authd* was running FreeBSD 4.6.2, while the *signing-authd* and client were running OpenBSD 3.1. All machines used GMP version 3.1.1 for large integer arithmetic.

The experiments were conducted using Rabin keys with a 1024-bit modulus, and Schnorr keys with 1024-bit ps and

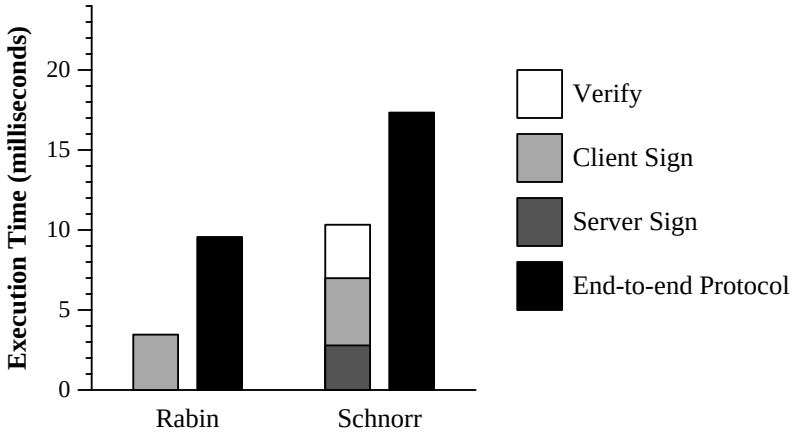

Figure 6. Benchmarks for signing and verifying in the Rabin and Schnorr signature schemes. End-to-end protocol shows user wait time for complete authentication.

160-bit qs. There are no known efficient reductions from the discrete log problem to factorization or vice-versa. However, given today's fastest algorithms, taking discrete logs over the group in the Schnorr algorithm should be roughly comparable to factoring the Rabin modulus.

To measure system responsiveness, we timed a **cd** command on the client to a directory (on the *verifying-authd*) that triggered an authentication. Under normal circumstances, a user is authenticated to a remote SFS server as long as that server is mounted and the user does not add or remove keys from his agent. In our experimental setup, however, we reset the user's agent after every successful authentication. The results of this experiment are shown by the black bars in Figure 6. Without network latency, the 2Schnorr protocol is 68% slower. However, in absolute terms, 2Schnorr is only 7 msec slower, which is a barely noticeable delay for the first access to a file system. In fact, when the file system client is not already connected to the server, there is additional time to connect and negotiate a session key, which further reduces the relative difference of Rabin and 2Schnorr. On the other hand, had there been greater latency between the client and *signing-authd*, the 2Schnorr authentication time would increase by the network round trip time.

Figure 6 also shows the CPU times required to compute and verify digital signatures. Note that verifying in Rabin is negligible, as no modular exponentiation is required. The *verifying-authd* can verify a Rabin signature in well under 0.1 msec. By contrast, Schnorr signature verification takes approximately 3 msec—a significant increase. For this reason, Schnorr might be a bad candidate for a *verifying-authd* server that supported huge numbers of users with high turnover. However, since every client connection also requires the server to engage in the key negotiation protocol, SFS servers cannot scale to 1,000s of new connections per second anyway.

If we compare the cost of signing, the sum of the halves of the 2Schnorr signature protocol is 102% slower than Rabin. As 2Schnorr overlaps its calculation of $g^{k_c} \bmod p$ with network latency and server computation, the cost of this computation, about 1.7 msec, is not reflected in the CPU times shown in the graphs. Note that even given this overlap, the client requires more computation than the signing server. The reason is that the client must check the server has been honest before outputing a signature.

Finally, we should note that key generation with 2Schnorr is significantly slower. We generated keys on the *signing-authd* and found that Rabin keys can be generated in about 0.2 seconds, while 2Schnorr keysets require about 0.55 seconds to generate a new p, q, g, and $y = g^x \bmod p$. Since users rarely need to regenerate keys, this slowdown is acceptable. If an application needs to generate many keys (perhaps to create a large batch of user accounts at once), 2Schnorr can actually be made faster than Rabin by reusing the same p, q, and g parameters for different keys.

Though 2Schnorr is clearly more expensive than the original Rabin-based user authentication scheme, the performance is still perfectly acceptable for a procedure that only needs to be invoked when a user first accesses a new file server. Moreover, we believe the performance impact is more than offset by the 2Schnorr's added security.

6. Summary

We study proactive, two-party signature schemes (P2SS) as an effective tool to address the challenges of user-authentication in settings with many administrative realms. We present a three-message protocol, 2Schnorr, which is provably secure in the random oracle model assuming only

the difficulty of the computational discrete log problem. For systems with a constant bound on the number of concurrent signature requests, we also give a two-message version of 2Schnorr, which we prove secure using the stronger one-more-discrete-log assumption. We argue that similar techniques can be used for a P2SS version of the GQ signature scheme.

To demonstrate the utility of P2SS, we integrated 2Schnorr into SFS, a secure network file system. Using 2Schnorr, a user whose password is compromised can recover by simply changing his password on his primary server. This will immediately block attackers from accessing his accounts in all other administrative realms where he has registered the same public key. Moreover, the user can also obtain from his primary server a log of all servers accessed by the attacker—possibly including accounts the user has forgotten about. While 2Schnorr is slower than SFS's original Rabin signature algorithm, we show that the performance impact is quite acceptable, particularly given the added security.

Acknowledgments

We thank the anonymous reviewers of this paper, the members of the NYU cryptography reading group, our shepherd Dawn Song, and Mike Reiter for their helpful suggestions on this paper.

References

[1] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. Available as *IACR eprint archive Report 2001/002*, `http://eprint.iacr.org/2001/002/`, January 2001.

[2] M. Bellare and G. Neven. Transitive signatures based on factoring and RSA. In *Advances in Cryptology—AsiaCrypt'02*, 2002. To appear.

[3] M. Bellare and A. Palacio. GQ and Schnorr identification schemes: Proofs of security against impersonation under active and concurrent attacks. In *Advances in Cryptology—Crypto'02*, volume 2442 of *Lecture Notes in Computer Science*, pages 162–177, Berlin, 2002. Springer-Verlag.

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 62–73, Fairfax, VA, 1993.

[5] M. Bellare and P. Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology—Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 1996.

[6] M. Bellare and R. Sandhu. The security of practical two-party RSA signature schemes. Available as *IACR eprint archive Report 2001/060*, `http://eprint.iacr.org/2001/060/`, July 2001.

[7] D. Bleichenbacher. Generating ElGamal signatures without knowing the secret key. In *Advances in Cryptology—EuroCrypt'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 10–18, Berlin, 1996. Springer-Verlag.

[8] M. Blum. Coin flipping by telephone. In *IEEE Spring COMPCOM*, pages 133–137, 1982.

[9] A. Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-diffie-hellman-group signature scheme. Available as *IACR eprint archive Report 2002/118*, `http://eprint.iacr.org/2002/118/`, 2002.

[10] C. Boyd. Digital multisignatures. In *IMA Conference on Cryptography and Coding*, pages 241–246. Oxford University Press, 1989.

[11] J.-S. Coron. Optimal security proofs for PSS and other signature schmes. In *Advances in Cryptology—EuroCrypt'02*, volume 2332 of *Lecture Notes in Computer Science*, pages 272–287, Berlin, 2002. Springer-Verlag.

[12] I. Damgård and J. Nielsen. Perfect hiding and perfect binding commitment schemes with constant expansion factor. In *Advances in Cryptology—Crypto'02*, volume 2442 of *Lecture Notes in Computer Science*, pages 581–596, Berlin, 2002. Springer-Verlag.

[13] Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. Unpublished Manuscript, 2002.

[14] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology—Crypto'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Berlin, 1987. Springer-Verlag.

[15] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[16] FIPS 186. *Digital Signature Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, 1994.

[17] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. Proactive RSA. In *Advances in Cryptology—Crypto'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 440–454, Berlin, 1997. Springer-Verlag.

[18] R. Ganesan. Yaksha: Augmenting kerberos with public-key cryptography. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium*, pages 132–143, 1995.

[19] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, 1988.

[20] L. Guillou and J.-J. Quisquater. A "paradoxical" identity-based signature scheme resulting from zero-knowledge. In *Advances in Cryptology—Crypto'88*, volume 403 of *Lecture Notes in Computer Science*, pages 216–231, Berlin, 1988. Springer-Verlag.

[21] A. Herzberg, M. Jacobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature schemes. In *Fourth ACM Conference on Computer and Communication Security*, pages 100–110. ACM, 1997.

[22] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[23] G. Itkis and L. Reyzin. Forward-secure signature with optimal signing and verifying. In *Advances in Cryptology—Crypto'01*, volume 2139 of *Lecture Notes in Computer Science*, pages 332–354, Berlin, 2001. Springer-Verlag.

[24] G. Itkis and L. Reyzin. SiBIR: Signer-base intrusion-resilient signatures. In *Advances in Cryptology—Crypto'02*, volume 2442 of *Lecture Notes in Computer Science*, pages 499–514, Berlin, 2002. Springer-Verlag.

[25] P. MacKenzie and M. Reiter. Delegation of cryptographic servers for capture-resilient devices. In *Eight ACM Conference on Computer and Communication Security*, pages 10–19. ACM, 2001. Full version available at: `ftp://dimacs.rutgers.edu/pub/dimacs/TechnicalReports/TechReports/2001/2001-37.ps.gz`.

[26] P. MacKenzie and M. Reiter. Networked cryptographic devices resilient to capture. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[27] P. MacKenzie and M. Reiter. Two-party generation of DSA signature. In *Advances in Cryptology—Crypto'01*, volume 2139 of *Lecture Notes in Computer Science*, pages 137–154, Berlin, 2001. Springer-Verlag.

[28] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999. ACM.

[29] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 51–59, 1991.

[30] D. Pointcheval and J. Stern. Security Arguments for Digital Signatures and Blind Signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[31] C. Schnorr. Efficient identification and signature for smart cards. In *Advances in Cryptology—Crypto'89*, volume 435 of *Lecture Notes in Computer Science*, pages 235–251, Berlin, 1990. Springer-Verlag.

[32] C. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[33] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX*, pages 191–202, Dallas, TX, February 1988. USENIX.

[34] H. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.

[35] T. Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.

[36] A. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[37] T. Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.