1 Abstract

We continue our exploration of applications of Threshold Cryptography. We present the threshold variant on Schnorr’s signatures. We describe the Client-Server (2-party) model, and define Client-Server Schnorr signatures. We begin with the 4-flow variant of Client-Server Schnorr signatures and its security: to show security, one needs extractable and/or equivocable commitments, and can use random oracle and/or rewinding. We then move to the 2-flow variant and the one-more discrete log assumption.

2 Threshold Schnorr Signatures

Recall the setup of the general Schnorr protocol: it’s a discrete log-based cryptosystem. Let $p$ be a (large) prime, $q$ be a smaller prime; fix the secret key $x$ and the public key $g^x \in \mathbb{Z}_p$. To sign a message, do:

$$\text{Sig}(m) : \quad k \leftarrow \mathbb{Z}_q$$
$$r = g^k \mod p$$
$$e = H(m, r)$$
$$s = k + xe$$
$$\sigma = (r, s)$$

Notice that $H$ is a cryptographic hash function, usually modeled as random oracle.

Upon receiving $m$ and $(r, s)$, the verifier checks the equation $g^s \overset{?}{=} ry^e$ because

$$g^s = g^{k+xe}$$
$$= g^k g^{xe}$$
$$= g^k (g^x)^e$$
$$= r \cdot y^e.$$  

It’s known that this signature scheme is secure in the Random Oracle Model under the Discrete Log assumption.

Let’s describe a threshold variant of this scheme.

**Setup.** First, do distributed key generation (see previous lecture for details). We can do both static and adaptive attacker variants. The security property is that a simulator $S$ can forge any outcome.

**The Interactive Process.** Recall that in the VSS schemes we’ve seen, the servers didn’t need to talk to each other; in this case, we will need to allow communication between the servers\(^1\) in order to jointly generate keys.

\(^1\)“...which is a bit annoying.” -YD
Step 1 Do distributed generation of $r$; now each player $P_i$ has a Shamir share $k_i$ of $k$, and the values $g^{k_i}$ are public.

Step 2 Each server sends the value $s_i = k_i + x_i e$. (The servers compute $r$; after they compute $r$, they can jointly compute $e$, the challenge.\footnote{We’re using the letter $c$ for the client in this lecture, which is why we suddenly call a challenge by the name $e$.})

Step 3 Anybody who knows the values $r_i = g^{k_i}$ and $y_i = g^{x_i}$ (remember that $y_i$’s are also public) can verify that $g^{x_i} = r_i \cdot y_i^e$. The only trick is that $e$ is computed as $H(m, r)$.

When there are at least $t_f$ correct shares, we simply interpolate $s$ from the correct $S_i$’s. This implies that $g^s$ is indeed equal to $r \cdot y^e$.

We’re not going to detail the proof of security of this scheme. However, we observe that with pre-processing, we can adapt the scheme to make it somewhat less interactive: put Step 1 in the setup stage. That is: off-line, before knowing the message, the servers do the distribution.

3 Two-Party Protocols

So far, the schemes we’ve seen worked only for $n > 2$ because we’ve always required robustness. With a majority of bad guys, you can’t make a robust scheme, so we don’t have a chance at a robust Two-Party scheme: one bad server can block computation. But let’s look for a Two-Party variant with all the desirable properties but robustness.

Usually, Two-Party protocols are done in the ‘Client-Server’ model, where the ‘user’ is now called ‘Client’ and is one of the servers. (Remember that secret sharing is actually easier if the party recovering the message is one of the servers, which will work to our advantage here.) In this situation, security proofs will look a bit different because security against Client and Server are slightly asymmetric.


The attack game is natural:
- You can interact with whatever is corrupted.
- You win if you can forge a message without explicitly signing it with the other party.

Sharing is simple: it’s 2-out-of-2. Split $x$ into $x = x_c + x_s$; the client gets $x_c$, the server gets $x_s$. We assume that key generation is done honestly.

Let’s establish some notation from the get-go: let $y = g^x$, $y_c = g^{x_c}$, $y_s = g^{x_s}$. (Notice that $y = y_c \cdot y_s$.) Both the client and the server know all those values (because $y$ is public, so $y_c$ and $y_s$ are known to both parties). We need to compute signatures:

Protocol. At the outset, the client has $x_c$ and knows $y$, $y_c$ and $y_s$. The server has $x_s$ and knows $y$, $y_c$ and $y_s$. We present a four-flow protocol:

1. Client sends $m$ to the server.
2. Server generates $k_s \leftarrow Z_q$, computes $r_s = g^{k_s}$ and sends a commitment to $r_s$ to the Client.
3. Client generates $k_c \leftarrow Z_q$, computes $r_c = g^{k_c}$ and sends $r_c$ to the Server (in the clear).
4. Server computes $r = r_c \cdot r_s$ and $e = H(m, r)$. Then he computes his part of the signature: $s_s = k_s + x_s e$. Server sends opening of $r_s$ and his part of the signature, $s_s$, to the Client.
After the fourth flow, the Client computes \( e = H(m, r) \) and now checks that \( g^{s_e} = r_s = y^e \). He computes \( s_c = k_c + x_c \cdot e \), and finally sets \( s = s_c + s_e \). The client outputs \((r, s)\).

**Remark.** Later, we will reduce this from a four-flow protocol to a two-flow protocol.

**Remark.** Conceivably, you could run this protocol concurrently. Here, however, we will assume that the Client and Server interact with only one copy of each other at a time.

**Proof of Security.** We want to reduce the security of the threshold variant of Schnorr’s signature scheme to the security of the ordinary Schnorr signature (not directly to Discrete Log!). We will consider the (asymmetric) cases separately: the malicious Client \((C^*)\) and then the malicious Server \((S^*)\).

**The Malicious Client Case.** Assume there’s an attacker who controls the Client and thinks he interacts with the Server. (The server will be the simulator, who has a signing oracle; the simulator will be the one who breaks the original Schnorr scheme.) First, Client (the attacker \(A\)) extracts \( x_c \): just choose \( x_c \) at random and give it to him. Then \( C^* \) extracts for honest players.

At the beginning of the protocol, the Server (simulator) gets \((r, s)\) of \( m \) from his signing oracle, and he generates \( x_c \) at random. In the first flow, the Server (simulator) sends a commitment to garbage and the value \( x_c \) to the Client (attacker). The Client sends \( r_c \) to the Server. The Server pretends that \( r_s = r/r_c \) and in the third flow, sends the opening of the commitment to \( r_s \). (Assume for the proof of security that we have a trapdoor commitment and that the simulator knows the trapdoor key.)\(^3\) Now we need to cook up \( s_s \). The simulator knows \( s \) from his signing oracle and knows \( s_c \) (because the simulator is the one who gave the Client his secret key in the first place), so he can simply compute \( s_s = s - s_c \) and send \( s_s \) along to the Client in the third flow. Eventually, the Client computes the forgery, \((m^*, (r^*, s^*))\), which the simulator outputs as his forgery.

**The Malicious Server Case.** In this setup, the Client (simulator) talks to the signing oracle and the Server is the attacker. At the outset of the protocol, the simulator generates \( x_s \) at random (but he doesn’t know \( x_c \)) and for message \( m \), gets \( \text{Sig}(m) = (r, s) \) from his signing oracle. First, the Client sends \( x_s \) to the Server. In the second flow, the Server sends \( \text{Commit}(r_s) \) to the Client. Now the client’s only chance at success is to send \( r_c = r/r_s \), but he doesn’t know \( r_s \), only \( \text{Commit}(r_s) \). We’ve got two options to proceed:

- We can assume that the commitment is **extractable**: with secret parameters, we can break the hiding property. It’s the opposite of a trapdoor commitment, in which secret parameters allow you to break the binding property. Explicitly, we need \( \text{Setup}(1^k) \rightarrow (CK, TK) \) so there exists some \( \text{Extract} \) algorithm such that if \( A(CK) \rightarrow (c, d) \), then \( \text{Extract}(C, TK) = \text{Open}(c, d) \) with probability \( 1 - \text{negl}(k) \). (For example, any encryption is an extractable commitment: public key encryption, for instance.) This is a good option; if the attacker actually did commit to \( r_s \), then the simulator would be able to open it.

  The flows of the protocol would look like this:

  - For each query, \( S^* \) sends \( \text{Commit}(r_s) \) to \( C \).
  - \( C \) sends \( r_c \) to \( S^* \).
  - \( S^* \) sends \( r_s \) and \( s_s \) to \( C \).
  - \( C \) sends \((r, s)\) to \( S^* \).

- Alternatively, use **rewinding.** The simulator cooks up some \( r_s^* \) in the third flow. The attacker sends \( r_s \) and \( s_s^* \) in the fourth flow. From here, we rewind back to the third line, compute \( r_c = r/r_s \) instead, and send that. The rest goes as before:
  - \( C \) sends \( r_c \) to \( S^* \).

\(^3\) This is a valid assumption, but there are some subtleties related to deniability.
– \( S^* \) sends \( r_s \) and \( s_s \) to \( C \).
– \( C \) sends \((r, s)\) to \( S \).

Check (as an exercise) that this rewinding trick works (albeit with worse exact security than the first option – it’s \( \varepsilon^2 \)) and doesn’t require extraction. (Note that we especially forbid concurrent executions of the scheme with rewinding, but this allows us to safely rewind a single scheme many times.)

Thus, to get overall security: either \textbf{Commit} should be both extractable and equivocable, or we relax the requirements and permit \textbf{Commit} that is only equivocable, but the scheme has worse security. (The former setup gives concurrent security, by the way.)

Let’s look at the first idea. How easy is it to construct a commitment scheme which is both extractable and equivocable? This is essentially what is known as a universally composable (UC) commitment. There are inefficient constructions from general schemes; there are also more efficient constructions which use some heavy-duty number theory. [3]

In the random oracle model, the extractable commitment scheme is trivial: simply use \textbf{Commit}(\( m \)) = \( G(\alpha) \). (This doesn’t give true hiding, but for this application, we didn’t need it. To get the hiding, use \textbf{Commit}(\( m, r \)).) The protocol then looks like:

– \( C \) sends \( x_s \) to \( S^* \).
– \( S^* \) sends \textbf{Commit}(\( r_s \)) to \( C \).
– \( C \) sends \( r_c \) to \( S^* \).
– \( S^* \) sends \( r_s \) and \( s_s \) to \( C \).
– \( C \) sends \((r, s)\) to \( S^* \).

It turns out that there’s a different option for the whole analysis: so far, we’ve been reducing to the security of Schnorr’s original signature scheme in our security analysis. Perhaps in the Random Oracle Model, we can consider a direct reduction, as follows.

• Reduce directly to \textbf{discrete log} (assuming that \( H \) is a random oracle) and use an extractable commitment. Then the argument for security against malicious client looks like this: the simulator controls oracle access for the attacker to \( H \). In the preamble, the simulator chooses \( x_c \) at random. At the beginning of the signing phase, the simulator sets \( r_s = g^\alpha \cdot y^\beta_s \) where \( \alpha, \beta \) are chosen at random. The simulator sends \( x_c \) once, and for each signing query, sends \textbf{Commit}(\( r_s \)) (honestly) to the attacker. The attacker replies with \( r_c \). The simulator computes \( r = r_s \cdot r_c \), and he pretends that \( H(m, r) = -\beta \) (remember, we’re reducing directly to DLog). Notice that

\[
g^\alpha = r_s \cdot y_s^{-\beta} = r_s \cdot y_s^{H(m, r)} = r_s \cdot y_s^{\varepsilon}.
\]

Then (in the third flow) the simulator sends \( s_s = \alpha \cdot r_s \) to the attacker.

To break the Discrete Log, do the usual rewinding. When the attacker outputs \((m^*, (r^*, s^*))\), rewind \( A \) once, changing \( H(m^*, r^*) \) from \( e \) to a random \( e' \). Then hope that \( A \) outputs \((m^*, (r^*, s))\), which gives \( x \).


To reduce from a Four-Flow to a Two-Flow Protocol, we elide the commitment to \( r_s \). As before, in the setup, the Client knows \( x_c \) and the server knows \( x_s \). The protocol is as follows:

• Client sends \( m \) and \( r_c \) to Server.
• Server sends \( r_s \) and \( s_s \) to Client.

• Client computes signature.

Is this scheme still secure?

The party that gains power in this variant is the Server, so we need only re-prove security against malicious servers. The question, then is “Can a bad guy forge signatures?” Imagine that the Client is the simulator (and has access to a signing oracle) and the Server is the attacker. The simulator sends \( r_c \) to the attacker, and the attacker sends back \( r_s \).

The natural reduction doesn’t seem to work. No matter what \( r_c \) the client chooses, if the server chooses a bad \( r_s \), then you won’t get the right \( r \). Say the simulator knows \((r, s)\) (from his signing oracle) and he sends some \( r_c \) to the attacker. The attacker can choose \( r \) adversarily, then compute \( e = H(m, r) \). He wants to compute \( r_s = r/r_c \), but either he doesn’t know \( r_c \), or he gets \( r_c \) and follows the protocol but can’t forge the signature. The attacker needs \( S_s \) such that \( g^{s_e} = \frac{r}{r_c} \cdot y_e^c \). To find this, he needs to know DLog(\( r/r_c \)), but \( r_c \) was random, so it’s unclear how he can satisfy this.

It seems that it could be secure, but under what assumption? We claim that under an assumption called **1-more-DL**, we can show the security of the 2-round variant. 1-more-DL supposes that for any \( n \) (which is polynomial in \( k \)) and for any PPT adversary \( A \) (with DL oracle),

\[
Pr \left[ A^{DL}(g, p, Z_0, \ldots, z_n) = \text{DLog}_g(z_0), \ldots, \text{DLog}_g(z_n) \right] \leq \text{negl}(k)
\]

(where \( z_0, \ldots, z_n \) are \( n + 1 \) random numbers) provided that \( A \) made at most \( n \) calls to its DL oracle. This is a stronger assumption than DL.

The proof of security now goes as follows. The Client (simulator) has \((z_0, \ldots, z_n)\) where \( n = q \) is the number of signing queries. He sets \( y = z_0 \). To sign a message \( m_i \), he sends \( r_c = z_i \) to the Server (attacker). The Server responds by sending \( r_s \) and \( S_s \). Now the Client checks that \( g^{s_e} = \frac{r}{r_c} \cdot y_e^c \). What he needs is to compute the final signature; it suffices to find \( S_e \) such that \( g^{S_e} = z_i \cdot y_e^c \) (where \( z_i = r_c \)). Here, he uses a call to his DLog oracle – one query for each signature. He gets DLog(\( z_i \) + \( e \cdot \text{DLog}(z_0) \)). When \( A \) forges \( \text{Sig}(m) \), you rewind him. Change \( H(m, r) \) to some \( e' \); use fresh \( z_j \)'s in repeated signing queries. When you’ve extracted \( x_0 = \text{DLog}(z_0) \), get all the DLog(\( z_j \))’s from \( \text{DLog}(z_j) + e \cdot d \cdot \text{log}(z_0) \). The probability of success is \( \varepsilon^2 \).

\[
\square
\]

References

