

1 Administrative Notes

For those keeping track: the instructor of “Advanced Cryptography” is Victor Shoup (found at victor@shoup.net or <http://www.shoup.net/>) and the course will meet on Tuesdays from 5:00pm to 6:50pm in WWH 513.

By a somewhat democratic process, it was established that the graded work for the course will be two-fold: scribe notes (such that the participants and the instructor emerge with a relatively complete set of lecture notes) and homework exercises (probably three sets of challenging-but-not-life-threatening problems, such that everybody gets some practice solving problems and the instructor may omit some tedious details in lecture).

Suggested prerequisites include basic algebra and number theory (groups, rings, fields, cyclic groups...), basic probability theory (an undergraduate course would suffice) and the ground work of cryptography. This course will be a continuation of last semester’s Intro to Cryptography course in the sense that it will build on the topics presented there and will cover more ground, but will not be inextricably linked to that course.

The instructor politely requested that everybody register.

2 Overview of the Course

1. We’ll start with Identification Schemes. (An identification scheme, which we’ll discuss today, is a 2-party protocol that allows Alice to convince a skeptical Bob that she is who she says she is.) This will lead to Σ -protocols.
2. Next, Signature Schemes.
3. Then Public Key Encryption: we’ll give constructions (based on elementary number theory) in the random oracle model and the standard model.
4. The above topics are based on standard tools that have been used in cryptography since the mid ’80s. The fourth course topic, Pairing-Based Cryptography, will make use of new techniques that have evolved in the last five years based on elliptic curves. (Note that we will use elliptic curves as a ‘black box’.)

We may add topics as we go along.

3 Identification Schemes

Identification Schemes are of interest for a number of reasons, possibly the least of which is actually identifying yourself. But we'll use exactly that as the motivation for developing the scheme.

Let's say we have a prover, P , and a verifier, V . A reasonable illustration of such a scheme is logging into a computer account: you're the prover, the computer is the verifier; you send your password to V , and V decides whether or not it's the correct password; V either accepts or rejects your login attempt.

Two questions spring to mind immediately. How does V verify your password? And how do you transmit the password safely in the first place?

One way for V to verify your password is simply to keep a list of passwords and then check whether or not your password appears in the correct place. This makes V rather open to attacks, so instead, perhaps V will keep a hash of each password or a salted (randomized) hash. When P sends the candidate password, V performs the hash and checks the result against its list. Since V stores something that looks random, it's less vulnerable to attacks.

As for the second question: how do we address the eavesdropping problem? Once upon a time, we accessed our accounts via Telnet, and to log in, we simply sent our password strings unencrypted across a (possibly insecure) channel. An eavesdropper could look at a transcript of the conversation to get a valid username and password. These days, we adopt a more secure solution like **symmetric key cryptography**.

3.1 Symmetric Key Identification Schemes

Definition A **pseudo random function** (PRF) is a function $y = F(k, x)$ which generates an output y when given a key k and an input x . The security condition for a PRF, informally, is that if we perform an experiment over and over again in which you give me a value x and I give you its evaluation y under F , then at the end of the day, at the end of the day, you cannot tell the difference between $F(k, \cdot)$ and a truly random function.

Challenge Response Protocol Suppose the prover P and verifier Q both start out with the (secret) key k . The verifier will send a challenge to the prover; the prover will respond, and the verifier will use the response to either accept or reject the attempted proof.

- V randomly generates x from the space X and V sends x to P (this is the challenge).
- P computes $y \leftarrow F(k, x)$ and P sends y to V (this is the response).
- V checks whether or not $F(x, k) \stackrel{?}{=} y$. If it passes the equality test, the door opens.

There are some drawbacks to this protocol. First, V has to store the value k , leaving the verifier vulnerable to attacks. Second, establishing the secret keys in the first place is an issue. We'll develop a public key alternative to address both issues.

3.2 Levels of Attacks

The adversary may attack our identification scheme with varying levels of strength. First, we will consider the “barebones” attack: without any previous information, the adversary walks up to V and attempts a proof. (This is called a “public key only” attack.)

Second, we can give the adversary more ammunition: she may listen to several conversations between an honest prover and a verifier.

Third, the most active attack: rather than just watching a conversation take place, the adversary may choose to actively probe the verifier. (This corresponds to a “chosen message attack” on a signature scheme.)

Remark This is not the only sort of attack: another popular flavor is the “man in the middle” attack. We won't discuss this one at length, but develop it as a point of interest by means of two examples. Consider the “friend or foe” problem: an airplane (probably of some military stripe, for dramatic flair) is flying into an airport and ground control wants to verify that it is friendly (that is, not about to blow up the base). A bad guy can get in between a good guy flying in and ground control like this: the ground control sends a query to the bad guy, the bad guy pretends to be ground control and sends the same query to the good guy, the good guy responds properly to the bad guy, and the bad guy passes on that response to ground control, thereby fooling the base into accepting his proof.

In a similar (but less sinister) vein, consider the “chess grand master” problem: Marisa, knowing next to nothing about chess, orchestrates a game with two grand masters. She arranges things such that one grand master will play black and the other will play white, and everybody agrees to make one move per day. She simply uses the moves that each grand master makes against the other, thereby convincing both players that she is herself a worthy opponent.

This is the “man in the middle” attack, which –like I said– we won't study at all right now. Instead, let's consider public key schemes and how they stand up to the first three attacks.

3.3 Public Key Identification Schemes

In a public key identification scheme, the verifier needn't store any secret information and no key will be agreed upon in advance, so we immediately cure both weaknesses identified in the symmetric key scheme. First, we establish in advance some public key infrastructure (PKI), so that the verifier has a table of corresponding public key / identity pairs (Imagine the scenario of logging into a computer, and think of “identity” as “user ID”)., and the prover generates a public

key / secret key pair. The prover will walk up to the verifier and say: “I am Victor, and my public key is asdfjkl. I know you have a table in which you may look up the fact that asdfjkl is Victor’s PK. Now I will prove to you that I also have the correct SK.” Then there is an interactive proof between P and V .

Our goal in designing this scheme will be to make it hard for an adversary to forge Victor’s identity. We will consider our scheme under several kinds of attacks.

Before we define our protocol, we need a tiny bit of algebraic machinery so that we may set our system parameters in advance. Let q be a big prime - roughly 200 bits long, say, so that $q \approx 2^{200}$. Let G be a cyclic group of order q (that is, $|G| = q$) and let g be a generator of G (that is, $g \in G$, $g \neq e$, $\langle g \rangle = G$).

In practice, what you do is pick a really big prime, p (say 1000 bits long) and choose q to be a divisor of $p - 1$. Consider \mathbb{Z}_p^* , the multiplicative group of units modulo p : it is a cyclic group of order $p - 1$ and its subgroups are in one-to-one correspondance with the divisors of $p - 1$, so it will contain a subgroup G of order q . Although we will not discuss concrete implementation here, we briefly note that the arithmetic will take place modulo p and the data types and operations should be adjusted accordingly.

In addition to G and g , we will need a (probabilistic) key generation algorithm that picks x at random from \mathbb{Z}_q and computes $h = g^x$. The secret key (SK) will be x and the public key (PK) will be h .

Now we have the the setup of the scheme: the prover, P , gets the secret key x and the system parameters, G and g ; the verifier, V , gets the public key h and the system parameters, G and g .

The idea of the interactive proof is for P to convince V that she “knows” x , the discrete log of h (that is, $x = \log_g h$). The security of the protocol will rely on the assumption that $\log_g h$ is hard to compute.

Remark To be explicit, the Discrete Log problem is the following: given a generator g of G , and given an element $h = g^x$ where x is chosen at random, it’s hard to compute $\log_g h$. Cryptosystems are always built on assumptions of this form; we start out supposing that $P \neq NP$, and then make some further assumptions on top of that (e.g., Discrete Log is hard, or factoring is hard). Proving the security of a cryptosystem often boils down a proof by contrapositive: assume that it may be broken in polynomial time; show that such an algorithm would break Discrete Log.

Remark To be fair: there do, in fact, exist sub-exponential time algorithms to compute discrete logs mod p . However, if the prime p is big enough, these computations are infeasible. Remember, though, that for the efficiency of the good guys, we want to work in a reasonably small subgroup G .

Remark We qualify “knows” for semantic reasons (c.f. Zero Knowledge Proofs).

We are now in a position to present a 3-flow public key protocol. Notice that it is designed to withstand eavesdropping (the second form of attack discussed above), since if we didn't care about eavesdropping, we could have stopped after the Challenge Response protocol.

Schnorr's Identification Scheme Protocol Suppose the prover P and verifier Q both start out with the system parameters, but the prover has SK and the verifier only has PK. The prover will send a (randomly generated) message to the prover; the verifier sends a (randomly generated) message to the prover; the prover will respond, and the verifier will use the response to either accept or reject the attempted proof.

- P generates r at random from \mathbb{Z}_p and P sends $a = g^r$ to V .
- V generates c at random from \mathbb{Z}_q and V sends c to P .
- P computes $z = r + xc$ and V sends z to P .
- V checks whether or not $a \cdot h^c \stackrel{?}{=} g^z$. If it passes the equality test, the door opens.

Examining the verification step in detail: if it is indeed the case that $z = r + xc$, then we have

$$g^z = g^{r+xc} = g^r \cdot (g^x)^c = a \cdot h^c.$$

We note briefly that the interactive proof itself can be done efficiently; the only time-consuming step is exponentiation mod q – say, computing $a = g^r$ – which can be done by repeated squaring in time $O(\text{len}(r) \text{len}(q)^2)$.

3.4 Forgery is Hard

We wish to prove, using the assumption that computing Discrete Logs is hard, that forging P 's identity in the above protocol is hard under the barebones (PK only) attack model. Let's call the adversary \tilde{P} . He walks up to V , with no a priori information (aside from PK, which V also shares), and V runs the protocol with \tilde{P} as normal. \tilde{P} has a bogus algorithm to compute a and z (which at least fit the desired syntax). Then the conversation looks like this:

- \tilde{P} generates a using some black box and \tilde{P} sends a to V .
- V generates c at random from \mathbb{Z}_q and V sends c to \tilde{P} .
- \tilde{P} generates z using some black box and \tilde{P} sends z to P .
- V checks whether or not $a \cdot h^c \stackrel{?}{=} g^z$. If it passes the equality test, the door opens.

Define \tilde{P} 's **advantage** to be the probability that V accepts at the end of the above protocol. Call the advantage ε ; we wish to show that ε is **negligible** if the Discrete Log problem is **hard**.

Remark We are sweeping a number of important things under the rug here. For example, the adversary \tilde{P} is assumed to be computationally bounded – it is a probabilistic polynomial time (PPT) algorithm. We have not defined important technical terms like “negligible” and “hard”; you may consult Dodis’s notes from Intro to Cryptography for details.

Now: suppose that there were an efficient adversary \tilde{P} whose advantage ε was non-negligible. Using the algorithm \tilde{P} as a subroutine, we will construct two efficient algorithms for the Discrete Log problem that succeed with non-negligible probability.

Remark Notice that we don’t need an algorithm that succeeds with probability 1, or even close to it. Why not? First, suppose $\langle g \rangle = G$ is fixed; then we have three independent random variables that describe the probability space: x , c and \tilde{P} ’s coins, which we will call y .¹ If there exists an algorithm that breaks Discrete Log with probability, say, 1/1000, then we can use that algorithm to make another (slower, but still efficient) algorithm that breaks Discrete Log with probability arbitrarily close to 1.

On the other hand, if we view $\langle g \rangle = G$ as random, the technique we will use to increase the probability of success will no longer be valid. We’ll examine this question in more detail shortly.

The key concept, when using \tilde{P} to break Discrete Log, is this: say \tilde{P} cooks up an a . If \tilde{P} has any hope of answering one challenge, c , correctly, then she should be able to correctly answer a second challenge, c' , where $c' \neq c$, for the same a . If she can do that successfully twice, then we will have two triples, (a, c, z) and (a, c', z') , such that:

$$\begin{aligned} g^z &= a \cdot h^c, \\ g^{z'} &= a \cdot h^{c'}. \end{aligned}$$

Dividing the first equation by the second, we have

$$g^{z-z'} = h^{c-c'}. \tag{1}$$

Since we assume $c \neq c'$ (that is to say $c-c' \neq 0$), it follows that $c-c'$ has an inverse modulo q . (Such an inverse can be computed efficiently with the Extended Euclidean Algorithm.) Now multiply both sides of Equation (1) by the newly minted inverse and we have

$$g^{(z-z') \cdot (c-c')^{-1}} = h,$$

which implies that $(z - z') \cdot (c - c')^{-1}$ is the discrete log of h .

Using this approach, we will construct two (similar but distinct) algorithms to compute discrete logs and we’ll analyze the running time and probability of success for each one.

As above, the probability space is given by three independent variables: x (which defines h), y (that is, \tilde{P} ’s coins – view it a bit string of some length) and c (that is, V ’s coins). Let t denote the pair (x, y) (so t is uniformly distributed over some set). Picture the joint probability distribution

¹“They need a name. I’ll give them a name now. y . I might be unhappy with that name later.” –VS

of t and c as a matrix: it's a matrix of 0's and 1's in which the rows are indexed by t -values and the columns are indexed by c 's. The entry in row i , column j is 1 if V accepts the corresponding conversation and 0 if V rejects. The density of 1's in the matrix is ε .

Let's establish some notation: let R denote the number of rows and q denote the number of columns (recall that $c \xleftarrow{R} \mathbb{Z}_q$). Let n_i denote the number of 1's in row i , and let $N = \sum_{i=1}^R n_i$ be the number of 1's in the matrix. Finally, let $\varepsilon_i = n_i/q$ denote the success in each row. Note that the density of 1's in the matrix is $\varepsilon = N/Rq$.

First Discrete Log Algorithm We present an algorithm with probability of success $\geq \varepsilon^2 - \varepsilon/q$.

- Input g, h . Treat h as PK.
- Choose coins y at random. These are \tilde{P} 's coins.
- Choose $c, c' \in \mathbb{Z}_q$ at random.
- If both (a, c, z) and (a, c', z') are accepting conversations, and if $c \neq c'$, then output

$$x = (z - z') \cdot (c - c')^{-1}.$$

Note that \tilde{P} is a PPT algorithm, and that it concocts a before it sees c .

Let α be the probability of getting two good c 's (possibly equal). The probability of landing in a particular row is $1/R$, and the probability of choosing two good c 's in row i is $(\varepsilon_i)^2$, so by conditional probabilities,

$$\alpha = \sum_{i=1}^R \varepsilon_i^2 \frac{1}{R}.$$

It follows that $\alpha \geq \varepsilon^2$ by Jensen's Inequality.²

(Why? Let X be a real-valued random variable. A special case of Jensen's Inequality indicates that $E[X^2] \geq (E[X])^2$, where E is expected value. View ε as a random variable and view its probability distribution as the uniform distribution over the rows of our matrix. This gives $\alpha \geq \varepsilon^2$.)

Now, the probability of getting two c 's that are good and equal is ε/q (the probability that the first c is good is ε , and the probability that the second c is equal to the first is $1/q$. We subtract this from ε^2 to get a bound on the probability that we get two good c 's that are distinct. ■

$\varepsilon^2 - \varepsilon/q$ is (technically) non-negligible, but not a tight reduction. Just for fun, can we get a tighter reduction? It might be a more meaningful theorem.

²Jensen's Inequality can be stated for finite probability distributions as follows. Let X be a real-valued random variable with $E(X)$ finite and let g be a convex function; then $E[g(X)] \geq g(E[X])$.

The setup for the second Discrete Log algorithm will be slightly different from the first in that we will view x as a fixed quantity, so h is fixed. Under this assumption, we'll get an algorithm whose expected running time is proportional to polylog factors times $1/\varepsilon$, and it will always succeed in finding the discrete log. Let $\hat{\varepsilon}$ denote \tilde{P} 's advantage³ conditioned on x . For this fixed x , the algorithm runs in time $O(1/\hat{\varepsilon})$ and with probability of success at least $1/2$. Later, we'll describe how the algorithm for a fixed x implies an algorithm for an arbitrary x .

Second Discrete Log Algorithm The algorithm will happen in two phases.

```

fix x
Phase 1:
  repeat
    choose y, c at random
  until (a,c,z) is valid
Phase 2:
  fix y (this gives a row in the matrix)
  repeat
    choose c' at random
  until (a,c',z') is valid

```

Let's analyze this algorithm: we will show the probability of success is ~ 1 and the running time is $\sim 1/\varepsilon$. As above, fix x . Notice that the expected number of iterations of Phase 1 is $O(\frac{1}{\hat{\varepsilon}})$ (as it's simply the expected value of a geometric distribution). Phase 2 is harder to analyze: we might get unlucky in Phase 1 and get stuck in a row with very few 1's. Such rows are rare, though, and we're more likely to be in a good row than a bad row; in the end, they'll add up to $\hat{\varepsilon}$.

Let's compute the expected number of *loop* iterations. During Phase 1, you throw darts at the matrix until you hit a 1. After Phase 1, the probability that y lies in some fixed row i is the number of 1's in that row divided by the number of 1's in the matrix: that is, n_i/N . Conditioning on landing in row i , the probability of a second hit in that row (imagine throwing darts at row i until you hit a 1) is n_i/q . Therefore, the expected number of iterations is $q/n_i = 1/\varepsilon_i$. This implies that the total expected number of loop iterations is

$$\sum_{i=1}^R \frac{q}{n_i} \cdot \frac{n_i}{N} = \sum_{i=1}^R \frac{q}{N} = \frac{Rq}{N} = \frac{1}{\hat{\varepsilon}}.$$

That's the expected running time. At the end of the algorithm, we need to ask whether or not $c = c'$, and subtract off that probability. Assuming ε is sufficiently large (say $\varepsilon \geq 2/q$), the failure probability (where 'failure' means $c = c'$) is at most $1/2$. (That bound is far from tight.) So simply repeat the whole algorithm until you get c and c' with $c \neq c'$. ■

Finally, we need to describe how to use the algorithm that computes discrete log for a fixed x to design an algorithm to compute discrete log for any x . The solution is **random self-reducibility**:

³"Then we'll have hats on everything. I don't want to have x 's decorating everything, that's too much." -VS

the ability to solve all instances of a problem by solving a large fraction of the instances. Discrete log turns out to be self-reducible, and here's why. Say we have an algorithm that works for a particular h (remember that specifying x and specifying h are equivalent) and we want that algorithm to work for a random h' . Let $h' = h \cdot g^s$. Then:

$$\begin{aligned}\log_g h' &= \log_g(h \cdot g^s) \\ &= \log_g h + s\end{aligned}$$

Therefore, given $\log_g h'$ and s , we easily compute $\log_g h = \log_g h' - s$. This lets us randomize h in Phase 1. For each iteration of the loop in Phase 1, pick a new h' , so we iterate over h' , y and c . In Phase 2, keep both h' and y fixed. We no longer condition on x , and the algorithm computes the discrete log, as desired. The probability of success is ~ 1 and the running time is $\sim 1/\epsilon$.

Finally, we note that we can generalize the first Discrete Log algorithm by randomizing the system parameters G and g . We *cannot* extend the second algorithm in the same way because we don't know how to move from the discrete log mod p to the discrete log mod p' . This feature makes the first algorithm more robust.

4 References and Upcoming Topics

Google "Schnorr's Identification Scheme" for more information on today's topic.

Next week, we'll look at the Guillou-Quisquater Identification Scheme.