CPSC 467: Cryptography and Computer Security

Michael J. Fischer

Lecture 19
November 6, 2013

Full Feige-Fiat-Shamir Authentication Protocol

Full FFS overview

The full Feige-Fiat-Shamir Authentication Protocol combines ideas of serial and parallel execution to get a protocol that exhibits some of the properties of both.

A *Blum prime* is a prime p such that $p \equiv 3 \pmod 4$.

A *Blum integer* is a number $n = pq$, where p and q are Blum primes.

If p is a Blum prime, then $-1 \in \mathrm{QNR}_p$, so $\left(\frac{-1}{p}\right) = -1$. This follows from the Euler criterion, since $\frac{p-1}{2}$ is odd, so

$$(-1)^{\frac{p-1}{2}} = \left(\frac{-1}{p}\right) = -1.$$

If n is a Blum integer, then $-1 \in \mathrm{QNR}_n$ but $\left(\frac{-1}{n}\right) = 1$.

Square roots of Blum integers

Let $n = pq$ be a Blum integer and $a \in \mathrm{QR}_n$. Exactly one of a's four square roots modulo n is a quadratic residue.

Consider $\mathbf{Z}_p^*$ and $\mathbf{Z}_q^*$. $a \in \mathrm{QR}_p$ and $a \in \mathrm{QR}_q$.

Let $\{b, -b\} = \sqrt{a} \pmod{p}$ and apply the Euler Criterion to both. Since

$$(-1)^{(p-1)/2} = -1 \quad \text{and} \quad b^{(p-1)/2} \in \{\pm 1\},$$

then either $b^{(p-1)/2} = 1$ or $(-b)^{(p-1)/2} = 1$.
Hence, either $b \in \mathrm{QR}_p$ or $-b \in \mathrm{QR}_p$. Call that number b_p.

Similarly, one of the square roots of $a \pmod{q}$ is in QR_q, say b_q.

Applying the Chinese Remainder Theorem, it follows that exactly one of a's four square roots modulo n is a quadratic residue.

Full FFS key generation

Here's how Alice generates the public and private keys of the full FFS protocol.

- She chooses a Blum integer n.
- She chooses random numbers $s_1, \ldots, s_k \in \mathbf{Z}_n^*$ and random bits $c_1, \ldots, c_k \in \{0, 1\}$.
- She computes $v_i = (-1)^{c_i} s_i^{-2} \bmod n$, for $i = 1, \ldots, k$.
- She makes $(n, v_1, \ldots, v_k)$ public and keeps $(n, s_1, \ldots, s_k)$ private.

Notice that every v_i is either a quadratic residue or the negation of a quadratic residue.

It is easily shown that all of the v_i have Jacobi symbol 1 modulo n.

One round of the full FFS authentication protocol.

A round of the protocol itself is shown below. The protocol is repeated for a total of t rounds.

	Alice		Bob
1.	Choose random $r \in \mathbf{Z}_n - \{0\}$, $c \in \{0, 1\}$. $x = (-1)^c r^2 \bmod n$	$\xrightarrow{x}$.	
2.		$\xleftarrow{b_1, \ldots, b_k}$	Choose random $b_1, \ldots, b_k \in \{0, 1\}$.
3.	$y = r s_1^{b_1} \cdots s_k^{b_k} \bmod n$.	$\xrightarrow{y}$	
4.			$z = y^2 v_1^{b_1} \cdots v_k^{b_k} \bmod n$. Check $z \equiv \pm x \pmod{n}$ and $z \neq 0$.

Correctness of full FFS authentication protocol

When both Alice and Bob are honest, Bob computes

$$z = r^2(s_1^{2b_1} \cdots s_k^{2b_k})(v_1^{b_1} \cdots v_k^{b_k}) \bmod n.$$

Since $v_i = (-1)^{c_i} s_k^{-2}$, it follows that $s_i^2 v_i = (-1)^{c_i}$. Hence,

$$\begin{aligned} z &\equiv r^2(s_1^2 v_1)^{b_1} \cdots (s_k^2 v_k)^{b_k} \\ &\equiv x(-1)^c(-1)^{c_1 b_1} \cdots (-1)^{c_k b_k} \equiv \pm x \pmod{n}. \end{aligned}$$

Moreover, since $x \neq 0$, then also $z \neq 0$. Hence, Bob's checks succeed.

The chance that a bad Alice can fool Bob is only $1/2^{kt}$. The authors recommend $k = 5$ and $t = 4$ for a failure probability of $1/2^{20}$.

Zero knowledge property

Theorem

The full FFS protocol is a zero knowledge proof of knowledge of the s_j for $k = O(\log \log n)$ and $t = O(\log n)$.

Proof.

See Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988. □

Non-interactive Interactive Proofs

Eliminating interaction from interactive proofs

Going from serial composition to parallel composition reduces communication overhead but may sacrifice of zero knowledge.

Rather surprisingly, one can go a step further and eliminate the interaction from interactive proofs altogether.

The idea is that Alice will provide Bob with a trace of a pretend execution of an interactive proof of herself interacting with Bob.

Bob will check that the trace is a valid execution of the protocol.

Of course, that isn't enough to convince Bob that Alice isn't cheating, for how does he ensure that Alice simulates random query bits b_i for him, and how does he ensure that Alice chooses her x_i's before knowing the b_i's?

Keeping Alice from cheating

The solution is to make the b_i's depend in an unpredictable way on the x_i's.

We base the b_i's on the value of a "random-looking" hash function H applied to the concatenation of the x_i's.

A non-interacting version of FFS

Here's how it works in, say, the parallel composition of t copies of the simplified FFS protocol.

- The honest Alice chooses $x_1, \ldots, x_t$ according to the protocol.
- Next she chooses $b_1 \ldots b_t$ to be the first t bits of $H(x_1 \cdots x_t)$.
- Finally, she computes $y_1, \ldots, y_t$, again according to the protocol.
- She sends Bob a single message consisting of $x_1, \ldots, x_t, y_1, \ldots, y_t$.
- Bob computes $b_1 \ldots b_t$ to be the first t bits of $H(x_1 \cdots x_t)$ and then performs each of the t checks of the FFS protocol, accepting Alice's proof only if all checks succeed.

Why can't Alice cheat?

A cheating Alice can choose y_i arbitrarily and then compute a valid x_i for a given b_i.

If she chooses the b_i's first, the x_i's she computes are unlikely to hash to a string that begins with $b_1 \ldots b_t$.[1]

[1] This assumes that the hash function "looks like" a random function. We have already seen artificial examples of hash functions that do not have this property.

Why can't Alice cheat?

If some b_i does not agree with the corresponding bit of the hash function, she can either change b_i and try to find a new y_i that works with the given x_i, or she can change x_i to try to get the i^{th} bit of the hash value to change.

However, neither of these approaches works. The former may require knowledge of Alice's secret; the latter will cause the bits of the hash function to change "randomly".

Brute force cheating

One way Alice can attempt to cheat is to use a brute-force attack.

For example, she could generate all of the x_i's to be squares of the y_i with the hopes that the hash of the x_i's will make all $b_i = 0$.

But that is likely to require 2^{t-1} attempts on average.

If t is chosen large enough (say $t = 80$), the number of trials Alice would have to do in order to have a significant probability of success is prohibitive.

Of course, these observations are not a proof that Alice can't cheat; only that the obvious strategies don't work.

Nevertheless, it is plausible that a cheating Alice not knowing Alice's secret, really wouldn't be able to find a valid such "non-interactive interactive proof".

Contrast with true interactive proofs

With a true zero-knowledge interactive proof, Bob does not learn anything about Alice's secret, nor can Bob impersonate Alice to Carol after Alice has authenticated herself to Bob.

On the other hand, if Alice sends Bob a valid non-interactive proof, then Bob can in turn send it on to Carol.

Even though Bob couldn't have produced it on his own, it is still valid.

So here we have the curious situation that Alice needs her secret in order to produce the non-interactive proof string π, and Bob can't learn Alice's secret from π, but now Bob can use π itself in an attempt to impersonate Alice to Carol.

Feige-Fiat-Shamir Signatures

Similarity between signature scheme and non-interactive IP

A signature scheme has a lot in common with the “non-interactive interactive” proofs.

In both cases, there is only a one-way communication from Alice to Bob.

- Alice signs a message and sends it to Bob.
- Bob verifies it without further interaction with Alice.
- If Bob hands the message to Carol, then Carol can also verify that it was signed by Alice.

Not surprisingly, the “non-interactive interactive proof” ideas can be used to turn the Feige-Fiat-Shamir authentication protocol into a signature scheme.

Signature scheme from non-interactive IP

We present a signature scheme based on a slightly simplified version of the full FFS authentication protocol in which all of the v_i's in the public key are quadratic residues, and n is not required to be a Blum integer, only a product of two distinct odd primes.

The public verification key is $(n, v_1, \ldots, v_k)$, and the private signing key is $(n, s_1, \ldots, s_k)$, where $v_j = s_j^{-2} \bmod n \, (1 \leq j \leq k)$.

Signing algorithm

To sign a message m, Alice simulates t parallel rounds of FFS.

- She first chooses random $r_1, \ldots, r_t \in \mathbf{Z}_n - \{0\}$ and computes
$$x_i = r_i^2 \bmod n \, (1 \le i \le t).$$
- She computes $u = H(mx_1 \cdots x_t)$, where H is a suitable cryptographic hash function.
- She chooses $b_{1,1}, \ldots, b_{t,k}$ according to the first tk bits of u:
$$b_{i,j} = u_{(i-1)*k+j} \, (1 \le i \le t, \; 1 \le j \le k).$$
- Finally, she computes
$$y_i = r_i s_1^{b_{i,1}} \cdots s_k^{b_{i,k}} \bmod n \, (1 \le i \le t).$$

The signature is
$$s = (b_{1,1}, \ldots, b_{t,k}, y_1, \ldots, y_t).$$

Verification algorithm

To verify the signed message (m, s), Bob computes

$$z_i = y_i^2 v_1^{b_{i,1}} \cdots v_k^{b_{i,k}} \bmod n \, (1 \le i \le t).$$

Bob checks that each $z_i \neq 0$ and that $b_{1,1}, \ldots, b_{t,k}$ are equal to the first tk bits of $H(mz_1 \cdots z_t)$.

When both Alice and Bob are honest, it is easily verified that $z_i = x_i \, (1 \le i \le t)$. In that case, Bob's checks all succeed since $x_i \neq 0$ and $H(mz_1 \cdots z_t) = H(mx_1 \cdots x_t)$.

Forgery

To forge Alice's signature, an impostor must find $b_{i,j}$'s and y_i's that satisfy the equation

$$\begin{aligned} b_{1,1} \ldots b_{t,k} \quad \preceq \quad & H(m(y_1^2 v_1^{b_{1,1}} \cdots v_k^{b_{1,k}} \bmod n) \\ & \ldots (y_t^2 v_1^{b_{t,1}} \cdots v_k^{b_{t,k}} \bmod n)). \end{aligned}$$

where "$\preceq$" means string prefix. It is not obvious how to solve such an equation without knowing a square root of each of the v_i^{-1}'s and following essentially Alice's procedure.

Pseudorandom Sequence Generation

Pseudorandom sequence generators revisited

Cryptographically strong pseudorandom sequence generators were introduced in Lecture 7 in connection with stream ciphers.

We next define carefully what it means for a pseudorandom sequence generator (PRSG) to be *cryptographically strong*.

We then show how to build one that is provably secure. It is based on the quadratic residuosity assumption (Lecture 16) on which the Goldwasser-Micali probabilistic cryptosystem is based.

Desired properties of a PRSG

A pseudorandom sequence generator (PRSG) maps a "short" random seed to a "long" pseudorandom bit string.

We want a PRSG to be *cryptographically strong*, that is, it must be difficult to correctly predict any generated bit, even knowing all of the other bits of the output sequence.

In particular, it must also be difficult to find the seed given the output sequence, since otherwise the whole sequence is easily generated.

Thus, a PRSG is a one-way function and more.

Note: While a hash function might generate hash values of the form yy and still be strongly collision-free, such a function could not be a PRSG since it would be possible to predict the second half of the output knowing the first half.

Expansion amount

I am being intentionally vague about how much expansion we expect from a PRSG that maps a "short" seed to a "long" pseudorandom sequence.

Intuitively, "short" is a length like we use for cryptographic keys—long enough to prevent brute-force attacks, but generally much shorter than the data we want to deal with. Typical seed lengths might range from 128 to 2048.

By "long", we mean much larger sizes, perhaps thousands or even millions of bits, but polynomially related to the seed length.

Incremental generators

In practice, the output length is usually variable. We can request as many output bits from the generator as we like (within limits), and it will deliver them.

In this case, “long” refers to the maximum number of bits that can be delivered while still maintaining security.

Also, in practice, the bits are generally delivered a few at a time rather than all at once, so we don’t need to announce in advance how many bits we want but can go back as needed to get more.

Notation for PRSG's

In a little more detail, a *pseudorandom sequence generator* G is a function from a domain of *seeds* $\mathcal{S}$ to a domain of strings $\mathcal{X}$.

We will generally assume that all of the seeds in $\mathcal{S}$ have the same length n and that $\mathcal{X}$ is the set of all binary strings of length $\ell = \ell(n)$, where $\ell(\cdot)$ is a polynomial and $n \ll \ell(n)$.

$\ell(\cdot)$ is called the *expansion factor* of G.

What does it mean for a string to look random?

Intuitively, we want the strings $G(s)$ to "look random".
But what does it mean to "look random"?

Chaitin and Kolmogorov defined a string to be "random" if its shortest description is almost as long as the string itself.

By this definition, most strings are random by a simple counting argument.

For example, 011011011011011011011011011 is easily described as the pattern 011 repeated 9 times. On the other hand, 101110100010100101001000001 has no obvious short description.

While philosophically very interesting, these notions are somewhat different than the statistical notions that most people mean by randomness and do not seem to be useful for cryptography.

Randomness based on probability theory

We take a different tack.

We assume that the seeds are chosen uniformly at random from $\mathcal{S}$.

Let S be a uniformly distributed random variable over $\mathcal{S}$.

Then $X \in \mathcal{X}$ is a derived random variable defined by $X = G(S)$.

For $x \in \mathcal{X}$,

$$\mathrm{Pr}[\mathrm{X} = \mathrm{x}] = \frac{|\{\mathrm{s} \in \mathcal{S} \mid \mathrm{G}(\mathrm{s}) = \mathrm{x}\}|}{|\mathcal{S}|}.$$

Thus, $\mathrm{Pr}[\mathrm{X} = \mathrm{x}]$ is the probability of obtaining x as the output of the PRSG for a randomly chosen seed.

Randomness amplifier

We think of $G(\cdot)$ as a *randomness amplifier*.

We start with a short truly random seed and obtain a long string that "looks like" a random string, even though we know it's not uniformly distributed.

In fact, the distribution $G(S)$ is very much non-uniform.

Because $|\mathcal{S}| \leq 2^n$, $|\mathcal{X}| = 2^\ell$, and $n \ll \ell$, most strings in $\mathcal{X}$ are not in the range of G and hence have probability 0.

For the uniform distribution U over $\mathcal{X}$, all strings have the same non-zero probability $1/2^\ell$.

U is what we usually mean by a *truly random* variable on ℓ-bit strings.

Computational indistinguishability

We have just seen that the probability distributions of $X = G(S)$ and U are quite different.

Nevertheless, it may be the case that all feasible probabilistic algorithms behave essentially the same whether given a sample chosen according to X or a sample chosen according to U.

If that is the case, we say that X and U are *computationally indistinguishable* and that G is a *cryptographically strong* pseudorandom sequence generator.

Some implications of computational indistinguishability

Before going further, let me describe some functions G for which $G(S)$ is readily distinguished from U.

Suppose every string $x = G(s)$ has the form $b_1 b_1 b_2 b_2 b_3 b_3 \ldots$, for example 0011111100001100110000. . . .

Algorithm $A(x)$ outputs "G" if x is of the special form above, and it outputs "U"otherwise.

A will always output "G" for inputs from $G(S)$. For inputs from U, A will output "G" with probability only

$$\frac{2^{\ell/2}}{2^{\ell}} = \frac{1}{2^{\ell/2}}.$$

How many strings of length ℓ have the special form above?

Judges

Formally, a *judge* is a probabilistic polynomial-time algorithm J that takes an ℓ-bit input string x and outputs a single bit b.

Thus, it defines a *random function* from $\mathcal{X}$ to $\{0,1\}$.

This means that for every input x, the output is 1 with some probability p_x, and the output is 0 with probability $1 - p_x$.

If the input string is a random variable X, then the probability that the output is 1 is the weighted sum of p_x over all possible inputs x, where the weight is the probability $\mathrm{Pr}[X = x]$ of input x occurring.

Thus, the output value is itself a random variable $J(X)$, where

$$\mathrm{Pr}[J(X) = 1] = \sum_{x \in \mathcal{X}} \mathrm{Pr}[X = x] \cdot p_x.$$

Formal definition of indistinguishability

Two random variables X and Y are *ϵ-indistinguishable by judge* J if

$$|\mathrm{Pr}[\mathrm{J}(\mathrm{X}) = 1] - \mathrm{Pr}[\mathrm{J}(\mathrm{Y}) = 1]| < \epsilon.$$

Intuitively, we say that G is *cryptographically strong* if $G(S)$ and U are ϵ-indistinguishable for suitably small ϵ by all judges that do not run for too long.

A careful mathematical treatment of the concept of indistinguishability must relate the length parameters n and ℓ, the error parameter ϵ, and the allowed running time of the judges

Further formal details may be found in Goldwasser and Bellare and in handout 11.