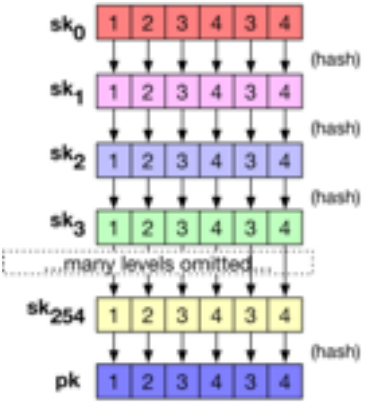
Hash-based Signatures: An illustrated Primer

Matthew Green 5 hours ago

Over the past several years I've been privileged to observe two contradictory and fascinating trends. The first is that we're *finally starting to use* the cryptography that researchers have spent the past forty years designing. We see this every day in examples ranging from encrypted messaging to phone security to cryptocurrencies.



The second trend is that cryptographers are getting ready for all these good times to end.

But before I get to all of that — much further below — let me stress that this is *not* a post about the quantum computing apocalypse, nor is it about the success of

cryptography in the 21st century. Instead I'm going to talk about something much more wonky. This post will be about one of the simplest (and coolest!) cryptographic technologies ever developed: hash-based signatures.

Hash-based signature schemes were first invented in the late 1970s by Leslie Lamport, and significantly improved by Ralph Merkle and others. For many years they were largely viewed as an interesting cryptographic backwater, mostly because they produce relatively large signatures (among other complications). However in recent years these constructions have enjoyed something of a renaissance, largely because — unlike signatures based on RSA or the discrete logarithm assumption — they're largely viewed as resistant to serious quantum attacks like Shor's algorithm.

First some background.

Background: Hash functions and signature schemes

In order to understand hash-based signatures, it's important that you have some familiarity with cryptographic hash functions. These functions take some input string (typically or an arbitrary length) and produce a fixed-size "digest" as output. Common cryptographic hash functions like SHA2, SHA3 or Blake2 produce digests ranging from 256 bits to 512 bits.

In order for a function

$$H(\cdot)$$

to be considered a 'cryptographic' hash, it must achieve some specific security requirements. There are a number of these, but here we'll just focus on three common ones:

1. Pre-image resistance (sometimes known as "one-wayness"): given some output

$$Y = H(X)$$

, it should be time-consuming to find an input

$$X$$

such that

$$H(X) = Y$$

. (There are *many* caveats to this, of course, but ideally the best such attack should require a time comparable to a brute-force search of whatever distribution

is drawn from.)

2. Second-preimage resistance: This is subtly different than pre-image resistance. Given some input

, it should be hard for an attacker to find a *different* input

such that

.

3. Collision resistance: It should be hard to find *any* two values

such that

. Note that this is a much stronger assumption than second-preimage resistance, since the attacker has complete freedom to find any two messages of its choice.

The example hash functions I mentioned above are *believed* to provide all of these properties. That is, nobody has articulated a meaningful (or even conceptual) attack that breaks any of them. That could always change, of course, in which case we'd almost certainly stop using them. (We'll discuss the special case of quantum attacks a bit further below.)

Since our goal is to use hash functions to construct signature schemes, it's also helpful to briefly review that primitive.

A digital signature scheme is a public key primitive in which a user (or "signer") generates a *pair* of keys, called the public key and private key. The user retains the private key, and can use this to "sign" arbitrary messages — producing a resulting digital signature. Anyone who has possession of the public key can verify the correctness of a message and its associated signature.

From a security perspective, the main

property we want from a signature scheme is *unforgeability*, or "*existential unforgeability*". This requirement means that an attacker (someone who does not possess the private key) should not be able to forge a valid signature on a message that you did not sign. For more on the formal definitions of signature security, see this page.

The Lamport One-Time Signature

The first hash-based signature schemes was invented in 1979 by a mathematician named Leslie Lamport. Lamport observed that given only simple hash function — or really, a one-way function — it was possible to build an extremely powerful signature scheme.

Powerful that is, provided that you only need to sign one message! More on this below.

For the purposes of this discussion, let's suppose we have the following ingredient: a hash function that takes in, say, 256-bit

inputs and produces 256-bit outputs. SHA256 would be a perfect example of such a function. We'll also need some way to generate random bits.

Let's imagine that our goal is to sign 256 bit messages. To generate our secret key, the first thing we need to do is generate a series of 512 separate *random* bitstrings, each of 256 bits in length. For convenience, we'll arrange those strings into two separate lists and refer to each one by an index as follows:

The lists

represent the *secret* key that we'll use for signing. To generate the public key, we now simply *hash* every one of those random strings using our function

. This produces a second pair of lists:

We can now hand out our public key

to the entire world. For example, we can send it to our friends, embed it into a certificate, or post it on Keybase.

Now let's say we want to sign a 256-bit message

using our secret key. The very first thing we do is break up and represent

as a sequence of 256 individual bits:

The rest of the signing algorithm is blindingly simple. We simply work through the message from the first bit to the last bit, and select a string from *one of* the two secret key list. The list we choose from depends on value of the message bit we're trying to sign.

Concretely, for i=1 to 256: if the

message bit

, we grab the

secret key string

and output that string as part of our signature. If the message bit

we copy the appropriate string (

from the second list. Having done this for each of the message bits, we concatenate all of the strings we selected. This forms our signature.

Here's a toy illustration of the process, where (for simplicity) the secret key and message are only eight bits long. Notice that each colored box below represents a *different* 256-bit random string:

When a user — who already has the public key

— receives a message

and a signature, she can verify the signature easily. Let

represent the i^{th} component of the signature: for each such string. She simply checks the corresponding message bit

and computes hash

. If

the result should match the corresponding element from

. If

the result should match the element in

.

The signature is valid if *every* single element of the signature, when hashed, matches the correct portion of the public key. Here's an (admittedly) sketchy illustration of the verification process, for at least one signature component:

If your initial impression of Lamport's scheme is that it's kind of insane, you're both a bit right and a bit wrong.

Let's start with the negative. First, it's easy to see that Lamport signatures and keys are be quite large: on the order of thousands of bits. Moreover — *and much more critically* — there is a serious security limitation on this scheme: each key can only be used to sign *one* message. This makes Lamport's scheme an example of what's called a "one time signature".

To understand why this restriction exists, recall that every Lamport signature reveals *exactly* one of the two possible secret key values at each position. If I only sign one message, the signature scheme works well. However, if I ever sign *two* messages that differ at any bit position

, then I'm going to end up handing out *both* secret key values for that position. This can be a problem.

Imagine that an attacker sees two valid signatures on different messages. She may be able to perform a simple "*mix and match*" forgery attack that allows her to sign a third message that I never actually signed. Here's how that might look in our toy example:

The degree to which this hurts you really depends on how different the messages are and how many of them you've given the attacker to play with. But it's rarely good news.

So to sum up our observations about the Lamport signature scheme. It's simple. It's fast. And yet for various *practical* reasons it kind of sucks. Maybe we can do a little better.

From one-time to *many*-time signatures: Merkle's tree-based signature

While the Lamport scheme is a good start, our inability to sign *many* messages with a single key is a huge drawback. Nobody was more inspired by this than Martin Hellman's student Ralph Merkle. He quickly came up with a clever way to address this problem.

While we can't exactly retrace Merkle's steps, let's see if we can recover some of the obvious ideas.

Let's say our goal is to use Lamport's signature to sign many messages — say

of them. The most obvious approach is to simply generate

different keypairs for the original Lamport scheme, then concatenate all the public keys together into one mega-key.

(Mega-key is a technical term I just invented).

If the signer holds on to all

secret key components, she can now sign

different messages by using exactly one

secret Lamport key per message. This seems to solve the problem without ever requiring her to re-use a secret key. The verifier has all the public keys, and can verify all the received messages. No Lamport keys are ever used to sign twice.

Obviously this approach sucks big time.

Specifically, in this naive approach, signing

times requires the signer to distribute a public key that is $latex N$ *times* as large as a normal Lamport public key. (She'll also need to hang on to a similar pile of secret keys.) At some point people will get fed up with this, and probably

won't every get to be very large. Enter Merkle.

What Merkle proposed was a way to retain the ability to sign

different messages, but *without* the linear-cost blowup of public keys. Merkle's idea worked like this:

1. First, generate

separate Lamport keypairs. We can call those

.

2. Next, place each public key at one leaf of a Merkle hash tree (see below), and compute the root of the tree. This root will become the "master" public key of the new Merkle signature scheme.
3. The signer retains all of the Lamport public and secret keys for use in signing.

Merkle trees are described here. Roughly speaking, what they provide is a way to collect many different values such that they can be represented by a single "root" hash (of length 256 bits, using the hash function in our example). Given this hash, it's possible to produce a simple "proof" that an element is in a given hash tree. Moreover, this proof has size that is *logarithmic* in the number of leaves in the tree.

Merkle tree, illustration from Wikipedia. Lamport public keys go in the leaves of this tree, and the root becomes the master public key.

To sign the

message, the signer simply selects the

public key from the tree, and signs the message using the corresponding Lamport secret key. Next, she concatenates the resulting signature to the Lamport public key and tacks on a "Merkle proof" that shows that this specific Lamport public key is contained within the tree identified by the root (*i.e.*, the public key of the entire scheme). She then transmits this whole collection as the signature of the message.

(To verify a signature of this form, the verifier simply unpacks this "signature" as a Lamport signature, Lamport public key, and Merkle Proof. She verifies the Lamport

signature against the given Lamport public key, and uses the Merkle Proof to verify that the Lamport public key is really in the tree. With these three objectives achieved, she can trust the signature as valid.)

This approach has the disadvantage of increasing the "signature" size by more than a factor of two. However, the master public key for the scheme is now just a single hash value, which makes this approach scale much more cleanly than the naive solution above.

As a final optimization, the secret key data can itself be "compressed" by generating all of the various secret keys using the output of a cryptographic pseudorandom number generator, which allows for the generation of a huge number of (apparently random) bits from a single short 'seed'.

Whew.

Making signatures and keys (a little bit) more efficient

Merkle's approach allows any one-time signature to be converted into an

-time signature. However, his construction still requires us to use some underlying one-time signature like Lamport's scheme. Unfortunately the (bandwidth) costs of Lamport's scheme are still relatively high.

There are two major optimizations that can help to bring down these costs. The first was also proposed by Merkle. We'll cover this simple technique first, mainly because it helps to explain the more powerful approach.

If you recall Lamport's scheme, in order sign a 256-bit message we required a vector consisting of *512 separate secret key* (and public key) bitstrings. The signature itself was a collection of 256 of the secret bitstrings. (These numbers were motivated by the fact that each bit of the message to be signed could be either a "0" or a "1", and thus the appropriate secret

key element would need to be drawn from one of two different secret key lists.)

But here's a thought: what if we *don't sign all of the message bits*?

Let's be a bit more clear. In Lamport's scheme we sign every bit of the message — regardless of its value — by outputting one secret string. What if, instead of signing both *zero* values and *one* values in the message, we signed only the message bits were equal to *one*? This would cut the public and secret key sizes in half, since we could get rid of the

list entirely.

We would now have only a *single* list of bitstrings

in our secret key. For each bit position of the message where

we would output a string

. For every position where

we would output... *zilch*. (This would also

tend to reduce the size of signatures, since many messages contain a bunch of zero bits, and those would now 'cost' us nothing!)

An obvious problem with this approach is that it's *horrendously* insecure. Please do not implement this scheme!

As an example, let's say an attacker observes a (signed) message that begins with "1111...", and she want to edit the message so it reads "0000..." — without breaking the signature. All she has to do to accomplish this is to *delete* several components of the signature! In short, while it's very difficult to "flip" a *zero* bit into a *one* bit, it's catastrophically easy to do the reverse.

But it turns out there's a fix, and it's quite elegant.

You see, while we can't prevent an attacker from editing our message by turning *one* bits into *zero* bits, we can *catch them*. To do this, we tack on a simple "checksum" to

the message, then sign *the combination of the original message and the checksum.* The signature verifier must verify the entire signature over both values, and *also* ensure that the received checksum is correct.

The checksum we use is trivial: it consists of a simple binary integer that represents the *total number* of zero bits in the original message.

If the attacker tries to modify the content of the message (excluding the checksum) in order to turn some one bit into a zero bit, the signature scheme won't stop her. But this attack have the effect of increasing the *number of zero bits in the message*. This will immediately make the checksum invalid, and the verifier will reject the signature.

Of course, a clever attacker might *also* try to mess with the checksum (which is also signed along with the message) in order to "fix it up" by increasing the integer value of the checksum. However — and this is

critical — since the checksum is a binary integer, in order to *increase* the value of the checksum, she would always need to turn some zero bit of the *checksum* into a one bit. But since the checksum is *also* signed, and the signature scheme prevents this kind of change, the attacker has nowhere to go.

(If you're keeping track at home, this does somewhat increase the size of the 'message' to be signed. In our 256-bit message example, the checksum will require an additional eight bits and corresponding signature cost. However, if the message has many zero bits, the reduced signature size will typically still be a win.)

Winternitz: Trading space for time

The trick above reduces the public key size by half, and reduces the size of (some) signatures by a similar amount. That's nice, but not really revolutionary. It still gives keys and signatures that are thousands of bits long.

It would be nice if we could make a bigger dent in those numbers.

The final optimization we'll talk about was proposed by Robert Winternitz as a further optimization of Merkle's technique above. In practical use it gives a 4-8x reduction in the size of signatures and public keys — at a cost of increasing both signing and verification time.

Winternitz's idea is an example of a technique called a "time-space tradeoff". This term refers to a class of solutions in which space is reduced at the cost of adding more computation time (or vice versa). To explain Winternitz's approach, it helps to ask the following question:

> *What if, instead of signing messages composed of bits (0 or 1), we treated our messages as though they were encoded using larger symbol alphabets? For example, what if we signed four-bit 'nibbles'? Or eight-bit bytes?*

In Lamport's original scheme, we had *two* lists of bitstrings as part of the signing (and public) key. One was for signing *zero* message bits, and the other was for *one* bits.

Now let's say we want to sign *bytes* rather than bits. An obvious idea would be to increase the number of secret key lists (and public key) from two such list to *256 such lists* — one list for each possible value of a message byte. The signer could work through the message one *byte* at a time, and pick from the much larger menu of key values.

Unfortunately, this solution really stinks. It reduces the size of the signature by a factor of *eight*, at a cost of increasing the public and secret key size *by a factor of 256.* Even

this might be fine if the public keys could be used for many signatures, but they can't — when it comes to key re-use, this "byte signing version of Lamport" suffers from the same limitations as the original Lamport signature.

All of which brings us to Winternitz's idea.

Since it's too expensive to store and distribute 256 truly random lists, what if we generated those
lists *programatically* only when we needed them?

Winternitz's idea was to generate single list of random seeds

for our initial secret key. Rather than generating additional lists randomly, he proposed to use the hash function

on each element of that initial secret key, in order to derive the *next* such list for the secret key:

. And similarly, one can *use the hash function again* on that list to get the next list

. And so on for many possible lists.

This is helpful in that we now only need to store a single list of secret key values

, and we can derive all the others lists on-demand just by applying the hash function.

But what about the public key? This is where Winternitz gets clever.

Specifically, Winternitz proposed that the public key could be derived by applying the hash function one more time to the final secret key list. This would produce a single public key list

. (In practice we only need 255 secret key lists, since we can treat the final secret key list as the public key.) The elegance of this approach is that *given any one of the possible secret key values* it's always possible to check it against the public key, simply by hashing forward multiple times.

The whole process of key generation is illustrated below:

To sign the first byte of a message, we would pick a value from the appropriate list. For example, if the message byte was “0”, we would output a value from

in our signature. If the message byte was “20”, we would output a value from

. For bytes with the maximal value “255”, we can output nothing, or we can output the appropriate element of

.

Note as well that in practice we don't really need to store each of these secret key lists. We can derive any secret key value *on demand* given only the original list

. The verifier only holds the public key vector and (as mentioned above) simply hashes *forward* an appropriate number of times — depending on the message byte — to see whether the result is equal to the appropriate component of the public key.

Like the Merkle optimization discussion in the previous section, the scheme as presented so far has a glaring vulnerability. Since the secret keys are related (*i.e.*,

), anyone who sees a message that signs the message "0" can easily change the corresponding byte of the message to a "1", and update the signature to match. In fact, an attacker can *increment* the value of any byte(s) in the message. Without some check on this capability, this would allow very powerful forgery attacks.

The solution to this problem is similar to the we discussed just above. To prevent an attacker from modifying the signature, the signer calculates and also signs a *checksum* of the original message bytes. The structure of this checksum is designed to prevent the attacker from incrementing any of the bytes, without invalidating the checksum. I won't go into the gory details right now, but you can find them here.

It goes without saying that *getting this checksum right* is critical. Screw it up, even a little bit, and some very bad things can happen to you. This would be particularly unpleasant if you deployed these signatures in a production system.

Illustrated in one terrible picture, a 4-byte toy example of the Winternitz scheme looks like this:

Note that the Message in this case consists of *bytes*, not bits. I'm pretty sure I calculated the checksum correctly, but I did it by hand and that doesn't always go so well.

What are hash-based signatures good for?

Throughout this entire discussion, we've mainly been talking about the *how* of hash-based signatures rather than the *why* of them. It's time we addressed this. What's the point of these strange constructions?

One early argument in favor of hash-based signatures is that they're remarkably fast and simple. Since they only require only the evaluation of a hash function and some data copying, *from a purely computational cost perspective* they're highly competitive with schemes like ECDSA and RSA. This could hypothetically be important for

lightweight devices. Of course, this efficiency comes at a huge tradeoff in bandwidth efficiency.

However, there is more complicated reason for the (recent) uptick in attention to hash-based signature constructions. This stems from the fact that all of our public-key crypto is about to be broken.

More concretely: the imminent arrival of quantum computers is going to have a huge impact on the security of nearly all of our practical signature schemes, ranging from RSA to ECDSA and so on. This is due to the fact that Shor's algorithm (and its many variants) provides us with a *polynomial-time* algorithm for solving the discrete logarithm and factoring problems, which is likely to render most of these schemes insecure.

Most implementations of hash-based signatures are not vulnerable to Shor's algorithm. That doesn't mean they're completely immune to quantum computers, of course. The best general

quantum attacks on hash functions are based on a search technique called Grover's algorithm, which reduces the effective security of a hash function. However, the reduction in effective security is nowhere near as severe as Shor's algorithm (it ranges between the square root and cube root), and so security can be retained by simply increasing the internal capacity and output size of the hash function. Hash functions like SHA3 were explicitly developed with large digest sizes to provide resilience against such attacks.

So at least in theory, hash-based signatures are interesting because they provide us with a line of defense against future quantum computers — for the moment, anyway.

Epilogue: the boring security details

If you recall a bit earlier in this article, I spent some time describing the security properties of hash functions. This wasn't just for show. You see, the security of a

hash-based signature depends strongly on which properties a hash function is able to provide.

(And by implication, the *insecurity* of a hash-based signature depends on which properties of a hash function an attacker has managed to defeat.)

Most original papers discussing hash-based signatures generally hang their security arguments on the *preimage-resistance* of the hash function. Intuitively, this seems pretty straightforward. Let's take the Lamport signature as an example. Given a public key element

, an attacker who is able to compute hash preimages can easily recover a valid secret key for that component of the signature. This attack obviously renders the scheme insecure.

However, this argument considers only the case where an attacker has the *public key* but has not yet seen a valid signature. In this case the attacker has a bit more

information. They now have (for example) both the public key and a portion of the secret key:

and $latex sk^{0}_1$. If such an attacker can find a *second pre-image* for the public key

she can't sign a different message. But she has produced a new *signature.* In the strong definition of signature security (SUF-CMA) this is actually considered a valid attack. So SUF-CMA requires the slightly stronger property of second-preimage resistance.

Of course, there's a final issue that crops up in most practical uses of hash-based signature schemes. You'll notice that the description above assumes that we're signing 256-bit messages. The problem with this is that in real applications, many messages are longer than 256 bits. As a consequence, most people use the hash function

to first hash the message as

and then the sign the resulting value

instead of the message.

This leads to a final attack on the resulting signature scheme, since the existential unforgeability of the scheme now depends on the *collision-resistance* of the hash function. An attacker who can find two different messages

such that

has now found a valid signature on two different messages. This leads to a trivial break of EUF-CMA security.

Categories: Uncategorized

Leave a Comment