Using ECDH and symmetric cryptography to build digital signatures

Ronald Landheer-Cieslak

October 15, 2023

ROUGH DRAFT

1 Introduction

1.1 Background

Digital signatures are used in every-day life to determine the authenticity of any number of messages of any type, from websites and APIs to E-mail and other messages. Whenever someone connects to a website with HTTPS, digital signatures are involved in the authentication of that website and the public key infrastructure (PKI) used to establish trust with the server. These digital signatures are based on a set of algorithms collectively called "digital signature algorithms" that include RSA, DSA, and ECDSA as well as some of the newer quantum-resistant algorithms such as "Dilithium" and "Sphincs+".

Diffie-Hellman (DH) key exchange algorithms enable two parties communicating over an insecure channel to establish a mutual secret without it being transmitted over that channel. These algorithms allow the two parties to agree on a shared symmetric secret key over the insecure channel. That shared symmetric secret key can then be used for any purpose, including authenticated encryption with additional data (AEAD) using algorithms such as AES-GCM, or authentication using symmetric keys employing hash-based message authentication codes (HMAC).

Diffie-Hellman-type algorithms are arguably the most valuable assymetric algorithms known to man: they are typically used for generating shared secret keys, but they *could* be used for signatures as well. In this paper, I will argue for this assertion by presenting a protocol for signing and verifying using a Diffie-Hellman-type construct in conjunction with quantum-safe symmetric cryptography. Grover's algorithm notwithstanding, symmetric cryptography *is safe* as soon as one doubles the key sizes vs. before quantum computers were a thing.

The issue for post-quantum cryptography is that with the advent of ever-larger quantum computers, we are likely to reach a point where an implement of Shor's algorithm capable of breaking current cryptography becomes viable. Shor's algorithm is the algorithm that breaks all "classic" asymmetric cryptography used today, including digital signature algorithms like RSA, DSA, ECDSA, and EdDSA; diffie-hellman key exchanges like Diffie-Hellman and Elliptic Curve Diffie Hellman (ECDH), and Key Encapsulation Mechanisms like RSA. Once a quantum computer large enough to implement Shor's algorithm with sufficient capacity to break a 2048-bit RSA key is viable, which will likely happen within the next two decades, we will have to rely on post-quantum cryptography to replace the classic algorithms we use today.

There are currently no standard post-quantum algorithms to replace the standard asymmetric cryptography algorithms we use in every-day life. NIST, the US National Institute for Standards and Technology, is managing an open process to find and standardize such algorithms.

Until recently, one of the most promising post-quantum asymmetric cryptography algorithms was an algorithm from the family of DH algorithms, and DSA and KEM algorithms seemed less promising both in their efficiency and their security. That was until July 2022, when an attack on Supersingular Isogeny Diffie-Hellman (SIDH) was found that can be performed on a classical computer and thoroughly breaks the security of SIDH. There is still hope, however, with lattice-based algorithms, and with algorithms based in learning-with-errors.

The four algorithms that made it to round four of NIST's search for post-quantum cryptography include three DSA-type algorithms and one KEM-type algorithm. None of the are DH-type algorithms.

I am obviously not presenting a post-quantum Diffie-Hellman-type algorithm: I'm not that good at math! I also don't know how much more difficult it would be to come up with a DH-type algorithm than it is to come up with a DSA-type or KEM-type algorithm, but I understand that all of these algorithms are based on the difficulty of reversing certain operations. RSA and DH are based on the difficulty of finding the prime factors of a large integer (which Shor makes relatively easy); ECC is based on the difficulty of dividing one point on an elliptic curve by another to find a very large integer (the one originally used for the multiplication); lattice-based algorithms are based on the difficulty of finding a point on a multi-dimensional lattice defined by a set of vectors that are almost aligned with each other (which is much easier when those vectors are almost perpendicular to each other); etc. These are called "trapdoor functions" and are what allow asymmetric cryptography to exist. What's needed is a trapdoor function that is difficult for both classical and quantum computers, and allows a DH-like protocol where a local private key and a remote public key can be used to generate a shared secret, without having to communicate anything other than the public key. Both KEM and DSA can be built out of that, the former of which I've already shown, and the latter of which I will show in this paper.

1.2 Notation

In most of this paper, I will use a Haskell-like notation, in which functions are represented as uppercase letters and values are lowercase letters. So the notation

$$a = F\ b\ c$$

means the function F takes parameters b and c and returns a. Tuples are represented as $\langle a, b, c \rangle$ (in this case a tuple of a, b, and c).

1.3 Definition of a digital signature algorithm

With that notation in mind, a digital signature algorithm (DSA) consists of three functions: a function to create a keypair, a function to sign, and a function to verify. The function to generate a keypair generates two keys, which we will call sk for the private key and pk for the public key

$$\langle sk, pk \rangle = G$$

The signing function S is defined as

$$sig = S\ sk\ m$$

where sig is the signature and m is the message. The verification function V is defined as

$$r = V\ pk\ m\ sig$$

where r is a boolean result.

If Alice wants to send a message $m_{a \to b}$ to Bob, she can use a DSA to sign that message, and anyone with her public key pk_a can verify that signature. A DSA can be used to sign any message, including a public key or a certificate containing one.

1.4 Definition of a Diffie-Hellman-type key exchange

This paper presents a protocol in which Diffie-Hellman can be used for digital signatures. Diffie-Hellman consists of two functions: a keypair generation function

$$\langle sk, pk \rangle = G$$

and a shared secret generation that takes the local private key and the remote public key as parameters:

$$s = X\ sk_a\ pk_b = X\ sk_b\ pk_a$$

1.5 Proofs of concept

Two proofs of concept are presented in this paper, both written in Python. While Haskell combines the power of abstract mathematics with the intuitive expressiveness of abstract mathematics, Python concentrates on intuitive expressiveness of software engineering. This is more of an engineering problem than it is a math problem, but also, there is no Jupyter kernel for Haskell.

The first proof of concept is presented in the paper as in-line Python code and is meant to show a simplified implementation of the protocol as transparently as possible. The second proof of concept is implemented as a Python module and is built using Python modules that are meant to be used in production and are intended to be secure. While I make no representations about the security of the implementation, it is my intent for this second proof of concept to be something one would be able to build a secure implementation on.

1.6 Layout of the remainder of this paper

The remainder of this paper is laid out as follows: the next section will describe the protocol in abstract terms including the principles of its operation and the basic functions required. The Proof of concept section will lay out how the basic functions, signing, and verification, can be implemented using Python. The Using the module section explains how the module that implements this protocol in Python using production-level code as its underlying implementation can be used to implement the use cases of signing and verifying messages. The Security analysis section provides an overview of the protocol's security, and the Conclusion section concludes this paper.

2 The protocol

The protocol is entirely built on a combination of asymmetric and symmetric cryptography, where the only asymmetric cryptography used is a Diffie-Hellman-type algorithm. In the proof of concept an Elliptic Curve Diffie-Hellman is used, but any Diffie-Hellman-type algorithm that implements the two functions G and X described above can be used.

2.1 Principles of operation

The protocol is based on the idea of using public data to deterministically generate the second key pair used in the DH key exchange $s = X\ sk_a\ pk_b = X\ sk_b\ pk_a$. That public data is derived from the message being signed itself, using secure hash algorithms, and then used to generate a symmetric key which, in turn, is used to sign the message using an HMAC algorithm. The idea here is that, in a scenario where Alice signs a message for Bob to verify, both Alice and Bob can generate the second key pair, but only Alice knows the private key of the first key pair. The security of the scheme is based on the difficulty of finding a private key for a given public key, and the security of the hashing algorithms used.

2.2 Basic functions needed

Several functions are needed to implement this scheme:

- a function G that generates a public and private key suitable for use in the following functions
- a function X that takes a private key sk_a and a public key pk_b, and outputs a shared secret k.
- a function P that will transform a single large integer into a private key suitable for use with the function X.
- a function U that computes a public key from a given private key.
- a secure hash function H.
- a secure HMAC function M.
- an HKDF function using the underlying HMAC M, K_M

2.3 Signing

Formally, the protocol is defined as follows:

1. Given a message from Alice to Bob $m_{a\to b}$.
2. Given Alice's keypair $sk_a, pk_a = G$.
3. Compute $h_1 = H\ m_{a\to b}$.
4. Compute $h_2 = H\ h_1|m_{a\to b}$ where $|$ is a concatenation operator.
5. Compute $sk' = P\ h_1'$.
6. Compute $pk' = U\ sk'$
7. Compute $k = X\ sk_a\ pk'$
8. Compute $s = K_M\ h_2\ k$
9. Compute $m' = M\ s\ m_{a\to b}$

The value m' is the signature for the message $m_{a\to b}$. It is calculated using a keyed HMAC. This HMAC uses a key that is derived using an HKDF that uses a value derived from the message as its salt, and a value derived from the DH-type exchange function as its IKM. The public key used in that exchange is derived from a private key which, itself, is derived from the message as well. This means that to produce this signature, the only value that is employed that is not known to Bob is Alice's private key. Alice can therefore send $\langle m_{a\to b}, m'\rangle$ to Bob and does not need to provide any additional data beyond her public key.

2.4 Verifying

Given the same functions X, P, H, M and K_M (U is not needed in this context), Alice's public key pk_a, and Alice's message $\langle m_{a\to b}, m'\rangle$, verification is formally defined as follows:

1. Compute $h'_1 = H\ m_{a\to b}$.
2. Compute $h'_2 = H\ h'_1|m_{a\to b}$ where | is a concatenation operator.
3. Compute $sk'' = P\ h'_1$.
4. Compute $k' = D\ sk'\ pk_a$
5. Compute $s' = K_M\ h'_2\ k'$
6. Compute $m'' = M\ s'\ m_{a\to b}$
7. Verify that $m' == m''$. If so, the message is verified as having been signed with sk_a.

Note that during this verification, the value h'_1 calculated by Bob is the same value as h_1 previously calculated by Alice; the value h'_2 is the same value as h_2 previously calculated by Alice, the value sk'' is the same value as the value sk' previously calculated by Alice; Bob does not need to calculate the equivalent of pk'; the value k' is the same as the value k previously calculated by Alice, by virtue of the DH-type function X; the value s' is the same value as s previously calculated by Alice; and therefore $m' == m''$.

3 Proof of concept

With the formal definition now given, I will present a proof of concept written in Python. In this proof of concept, the message "sent" by Alice will be a random text of three "Lorem ipsum" type paragraphs generated using the `lipsum` module. The `hashlib`, `hmac`, and `hkdf` modules are all used as they are regular modules for these types of cryptographic applications, but for the ECDH implementation I will use the `tinyec` library to make the proof of concept as explicit as possible. For the Python module, these modules are all replaced with the `cryptography` module from the Python Cryptographic Authority (PyCA). This has the added benefit of providing a second, indepent (in the sense of not using the same code) implementation of the same proof of concept.

[2]:
```
import tinyec.ec as ec
import tinyec.registry as reg
import os
import lipsum
import hashlib
import hmac
from hkdf import *
```

In the context of this paper, we will use Elliptic Curve Diffie-Hellman as an example of a Diffie-Hellman-type key negotiation scheme, but the only requirements on the scheme, represented here by the function X, are that:

1. the function X takes a private key and a public key as parameters and
2. a private key suitable as a parameter to the function X can be deterministically generated from a single large integer value

When using ECDH for the function X, we can choose an elliptic curve $C = \langle p, a, b, g, n, h\rangle$ or $C = \langle m, f, a, b, g, n, h\rangle$ where the former option is a prime curve and the second option is a binary curve; p or m, f determine the size of the curve a, b are the defining parameters of the elliptic curve, g is the generator, n is the order of the point at infinity, and h is the cofactor. These parameters are all public.

For the proof of concept, I have arbitrarily chosen SECP521r1 as the curve.

[3]:
```
curve = reg.get_curve('secp521r1')
```

We do, of course, need the function to generate a keypair, G, as well.

> Given a function G that generates a public and private key suitable for use in the following functions

In ECDH, the secret key is a large integer that is strictly smaller than n. To obtain the public key, this value is multiplied by the curve's generator g. This makes a very simple implementation of our function G:

[4]:
```
def G():
    sk = (int.from_bytes(os.urandom(66), 'big') & (2 ** 521 - 1)) % curve.field.
 ↪n
    pk = sk * curve.g
    return sk, pk
```

Note that I do not claim this implementation to be secure.

> Given a function X that takes a private key sk_a and a public key pk_b, and outputs a shared secret k.

Generating the shared secret depends on the way the public key is generated. In this case, because $sk_a * g = pk_a$ and $sk_b * g = pk_b$ we can see we have a common factor g of both pk_a and pk_b. We can therefore rely on the fact that $sk_a * pk_b = k = sk_b * pk_a$ because \$pk_b = $sk_b * g$ and $pk_a = sk_a * g$, which means, by substitution: $sk_a * (sk_b * g) = sk_b * (sk_a * g)$ which, without the brackets, means $sk_a * sk_b * g = sk_a * sk_b * g$. We should note, though that this multiplication of a secret integer value with the generator point of the curve gives us a point on the curve, not a large integer value. To obtain a large integer value, we "compress" the point we obtained by taking only the X coordinate of the value, and the final bit of the Y coordinate. To make things easier to align and manipulate, we also insert three zero bits between the X and the Y coordinate in the code below.

[5]:
```
def X(sk, pk):
    k = sk * pk
    return (k.x * 16 + k.y % 2).to_bytes(525+7//8, 'big')
```

Again, no representation is made for this code being secure: it is only intended to show the mechanism by which the scheme operates.

> Given a function P that will transform a single large integer into a private key suitable for use with the function X.

In much the same way as described above, in ECDH, the secret value of the private key is a large integer that is strictly smaller than the curve's n value. Our "transformation" then is only a mod function.

[6]:
```
def P(h):
    '''$$P_C(h) = h \mod n$$'''
    return int.from_bytes(h, 'big') % curve.field.n
```

> Given a function U that computes a public key from a given private key.

Again, a public key is obtained by multiplying the private key by the generator point:

[7]:
```
def U(pk):
    return pk * curve.g
```

Note that it is generally extremely unwise to "roll your own cryptography". The code for this part of the proof of concept is for ***illustrative purposes only*** and should not be taken as a valid, cryptographically secure, implementation of ECDH. It is intended to illustrate the underlying math and to show the workings of the protocol, but the true cryptgraphic proof of concept is contained in the second implementation of the proof of concept which, while it is more opaque due to the use of the PyCA `cryptography` module, is also both accurate and secure.

Given a secure hash function H.

We will use SHA-256 for our hash function in this proof of concept, but any cryptographically secure hash function may be used.

[8]:
```
def H(data):
    sha = hashlib.sha256(data)
    return sha.digest()
```

Given a secure HMAC function M.

[9]:
```
def M(key, data):
    if isinstance(data, str):
        data = data.encode('utf-8')
    h = hmac.new(key, msg=data, digestmod='sha256')
    return h.digest()
```

Note that this version of M in Python tries to be nice with its user by allowing both `str` and byte strings.

Also note that while some secure hashes don't need the HMAC construct for secure keyed message authentication, we will simply always use it because it's a nice composable construct and the additional cost for the HMAC construct is negligeable.

Finally, the RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is used to generate shared secrets.

Given an HKDF function using the underlying HMAC M, K_M

[10]:
```
def K(salt, ikm, length=32):
    prk = hkdf_extract(salt, ikm, hash=hashlib.sha256)
    return hkdf_expand(prk, length=length)
```

3.1 Signing

Now that our functions are all defined, we can generate a message and sign it.

Given a message $m_{A \to B}$

For this proof of concept, we will use three randomly-generated parahraphs of pseudo-random "lipsum" text. The Python `lipsum` module uses a text by Cicero as its source text.

[11]:
```
m = lipsum.generate_paragraphs(3)
```

Given Alice's keypair $sk_a, pk_a = G$.

For the purposes of this proof of concept, Alice will have previously shared her public key with Bob, and Bob already trusts it. How that is shared is outside the scope of this POC. Typically, this would be done using an X.509 certificate which would, in turn, by signed by a Certificate Authority, perhaps using the same algorithm.

[12]:
```
sk_a, pk_a = G()
```

Once all the functions are established, the message is created, and the private key is held secret by Alice but the public key is shared, we can start the protocol-proper to sign the messsage. As explained above, two hashes are generated: h_1, which will serve to generate the verifier's private key, and h_2, which will serve as a nonce for the HKDF, K_M.

compute $h_1 = H(m_{A \to B})$

[13]:
```
h_1 = H(m.encode('utf-8'))
```

As h_1 is generated directly from the message text, it is, in essence, public. That does mean that the verifier's private key is public as well, but *that* just means that anyone can verify the signature, which is part of the goals of DSAs.

The value h_2 is also generated from the message. We need a value that is distinct from h_1 but is also known to the verifier and is suitable as a nonce for K_M. To obtain this, the most obvious candidate is to use the same hashing function H with h_1 as an additional input, concatenated with the message text.

Compute $h_2 = H\ h_1 | m_{a \to b}$ where | is a concatenation operator.

[14]:
```
h_2 = H(b''.join([h_1, m.encode('utf-8')]))
```

The verifier's private key sk' may now be derived from the value h_1. This is one of the constraints we place on the DH-type algorithm: there must be a way for a private key to be derived from an arbitrary large integer value. With ECDH, this is a fairly straight-forward proposition as ECDH private keys are essentially arbitarary large integers with the constraint of being a value between 0 and the curve's n value minus one. In classic Diffie-Hellman, the private key is also an arbitrary large integer which, in that case, is used as an exponent rather than as a multiplier. While it is not clear what a new DH-type algorithm might look like, some function will be needed that can transform an arbitraty large integer into a private key. The details of that function are obviously outside the scope of this paper, but using h_1 as the arbitrary large integer, we will employ it.

Compute $sk' = P\ h_1$

[15]:
```
sk_prime = P(h_1)
```

Note that h_1 and h_2 are functionally interchangeable: both sides (the signer and the verifier) need to agree on which value to use for which purpose but there is no counter-indication that I know of that would prevent h_2 from being used to seed the verifier private key and h_1 from being used as the nonce in K_M in stead of vice-versa. As both sides need to agree in which value is used for which

purpose, however, this protocol will use h_1 as the seed for the private key and h_2 as the nonce for K_M.

Compute $pk' = U\ sk'$

[16]:
```
pk_prime = U(sk_prime)
```

Compute $k = X\ sk_a\ pk'$

[17]:
```
k = X(sk_a, pk_prime)
```

Compute $s = K_M\ h_2\ k$

[18]:
```
s = K(h_2, k)
```

Compute $m' = M\ s\ m_{a \to b}$

[19]:
```
m_prime = M(s, m)
```

Send $\langle m_{a \to b}, m' \rangle$ to Bob.

[20]:
```
to_bob = m, m_prime
```

3.2 Bob: verifying

Given the same functions D, P, H, M and K_M (U is not needed in this context).

[21]:
```
# inherited from cells above
```

Given Alice's public key pk_a.

[22]:
```
# inherited from cells above
```

Given Alice's message $\langle m_{a \to b}, m' \rangle$.

[23]:
```
m, m_prime = to_bob
```

Compute $h'_1 = H\ m_{a \to b}$.

[24]:
```
h_1_prime = H(m.encode('utf-8'))
```

Compute $h'_2 = H\ h'_1 | m_{a \to b}$ where $|$ is a concatenation operator.

[25]:
```
h_2_prime = H(b''.join([h_1_prime, m.encode('utf-8')]))
```

Compute $sk'' = P\ h'_1$.

[26]:
```
sk_double_prime = P(h_1_prime)
```

Compute $k' = D\ sk''\ pk_a$

[27]:
```
k_prime = X(sk_double_prime, pk_a)
```

Compute $s' = K_M\ h'_2\ k'$

[28]:
```
s_prime = K(h_2_prime, k_prime)
```

Compute $m'' = M\ s'\ m_{a \to b}$

[29]:
```
m_double_prime = M(s_prime, m)
```

Verify that $m' == m''$. If so, the message is verified as having been signed with sk_a.

[30]:
```
m_prime == m_double_prime
```

[30]:
```
True
```

4 Using the module

5 Security analysis

6 Conclusion