Ring Signatures - Analysis and Implementation

Andrés Fábrega, Jonathan Esteban, Damian Barabonkov

{andresfg, jesteban, damianb}@mit.edu

1 Introduction

When signatures were first introduced in Diffie and Hellman's seminal paper, these were presented in a single-user context: Alice uses her secret key to sign a message, and Bob uses her public key to verify it. As the field grew, however, cryptographers started envisioning a "multi"-signer notion of digital signatures. In particular, group signatures became a well studied concept: a user signing a message on behalf of a group, while preserving total anonymity. The big caveat with this model is that it relies on the presence of a group manager, who adds people to the group, and can reveal the signer if needed. These scheme certainly have applications, but, in some contexts, this dependency on an organizing party is not possible or desired. As such, ring signatures schemes are a way of solving this problem, providing total anonymity without requiring additional setup nor manager. Note that neither group nor ring signatures are better than the other: the context of the problem at hand determines which model is more appropriate. In this project, we analyze ring signatures, their construction, their security, and go over a functioning Python implementation.

2 Overview

Ring signatures were formalized by by Rivest, Shamir, and Tauman[1]. In this project, we follow their definitions and proposed construction.
Essentially, ring signatures are a way for a single user to anonymously sign a message on behalf of a group of people. This is perhaps best explained with an example use case: let's imagine that MIT has decided that Fall 2020 will be in person (fingers crossed), but decides not to reveal this just yet, for whatever reason. Having said this, a rogue member of the administration decides that students need to know this information as soon as possible. As such, he/she use a ring signature scheme to sign this information on behalf of all members of the administration, and sends this to the students. Later on, students can verify that, indeed, someone inside the administration signed this message, but without knowing which specific member it was, and thus confirm that its legit and valid. Importantly, other administrative members can't know which of their

colleagues signed and leaked this message. Anonymity is totally preserved, even for members of the group.

More concretely, to sign, the signer takes all public keys of the people that he/she wants to include on the group (from personal websites, or wherever the specific people decide to publish their public keys), and uses these and his/her secret key to sign the message. Later, a signature, which contains all public keys inside, can be verified, confirming that the message was signed with some secret key corresponding to one of the public keys in the signature, and thus being a legit leak. Importantly, note that there is no setup nor coordination between members involved, and we don't assume anything about the relationship of the group members. Also group members do not have a say or not if they are included in the group, as this is up to the signer and which public keys he/she takes when computing.

3 Construction

Now, we will go over a specific construction from the original paper that uses RSA (although we could use other trapdoor functions with some tweaking). Note that the primitives and details presented here are oversimplified; for a full description, reference the original paper.
First, we will go over a set of primitives used in the construction:

1. An issue with dealing with so many public keys is the fact that evidently, all the public moduli are different. So, each of the trapdoor functions may have different domain sizes. This makes it hard to combine into one signature, so instead we will use an "extended" trapdoor permutation, which basically extends the domain of each one of the f to to $\{0,1\}^b$, where $2 * *b$ is greater than all public moduli. Essentially, for an input of b bits, g_i will run f_i on the lower order bits of m, up to the modulus, and leave the higher ordered ones unchanged. Concretely: $g_i(m) = q_i \cdot n_i + f_i(r_i)$, where $m = q_i \cdot n_i + r_i, 0 <= r_i <= n_i$.

2. We assume black box access to an ideal symmetric algorithm. Our only assumptions about E_k is that it's deterministic, and it's a permutation.

3. We will need something called a combining function. This is a function that takes a pair (k, v) as a key value, a series of b-bit strings, and produces some output z as a function of these. We want this function to satisfy three properties. First, it needs to be one-to-one (if we have all b-bit strings except for one). Secondly, we need to be able to efficiently solve for a single missing y value. That is, if we know k, v, the output z, and say y_2 to y_r, we can efficiently find the y_1 value that makes this equation true. Note that there is one and only one y_1 value that satisfies this (one-to-one, as mentioned above). Thirdly, if all y_i are constructed from g_i, we can't find none of the underlying x_i without knowing the secret key for f_i. A construction that satisfies this, and the one we will use, is

$C_{k,v}(y_1, ..., y_r) = E_k(y_r \oplus E_k(y_{r-1} \oplus E_k(...\oplus E_k(y_1 \oplus v)...)))$. This is clearly one-to-one since E and XOR are permutations. Also, we can efficiently compute the output value by performing all nested encryptions straight through. Lastly, note that, if we are missing a specific y_s value, we can solve for it, if we know k, v: we can solve backwards from Z, by XOR ing nested decryptions up to to the missing value, and XORing this with nested encryptions from the missing value onwards.

With these primitives at hand, we can now build the sign and verify operations.

3.1 Sign

As stated in the overview, the ring sign operation takes as inputs the public keys for all ring members, the message, and the secret key of the signer. In addition, the signer specifies (via an index), which of the public keys corresponds to his/her private key. We then proceed as follows:

1. Compute the key for the symmetric encryption algorithm, by hashing the message (note that we assume we are in the ROM): $k = h(m)$.
   This may seem a bit concerning, naturally, but the security of the scheme does not rely on the choice of key for E, as we will explain later.

2. Pick, at random, the second half of the key for the combining function: $v \leftarrow \{0,1\}^b$.

3. Pick x_i for members of the group, except for the signer, and compute the y_i strings, using the public keys: $y_i = g_i(x_i)$.
   Note that here we are using the extended domain alternative, instead of f_i directly.

4. "Invert" the combining function to find y_s such that $C_{k,v}(y_1, ..., y_s, ..., y_r) = v$
   A few interesting things to highlight: first, note that here we are using two of our assumptions about C. As stated earlier, we assume that there is only one such y_s value that satisfies this ring equation, and we assume that, if we have all other y_i (in addition to k, v), we can efficiently find the missing string (in this case y_s). In addition, note that we are using the same value, v, as both the (half) key value and the output value. This creates a "cyclic", ring-shaped (and hence, the name "ring signature") structure of computation, where the first value (the inner XOR) is the same as the last value (the output). This ring structure is crucial for the unforgeability of the scheme.

5. The signer computes his/her x_i value, by using the secret key: $x_s = g_s^{-1}(y_s)$.

6. The signature, to be outputted, is then $(P_1, ..., P_r, v, x_1, ..., x_r)$.

This is the entire sign algorithm. Essentially, the signer samples a bunch of random x_i strings, and takes advantage of the combining function to find the single x_s such that the equation is satisfied. In particular, note that we need knowledge of a secret key: the cyclical dependency on v, and our use of the symmetric encryption algorithm, makes it such that randomly guessing all y_i values (i.e., randomly guessing x_i values and computing $g_i(x_i)$) that make the equation true is negligible. So, the only reasonably way to compute all values is by solving for one of them. Another interesting observation is that we output all the x_i values instead of all the y_i values. It may be confusing at first why this is required: if, given the x_i, anyone can compute the y_i, why can't we just simply output the y_i directly? The reason for this is that this forces the signer to hold a private key. Without this, forging a signature would be trivial: compute k, v and all y_i except for one, and efficiently solve for the missing y_s value.

3.2 Verify

As stated in the overview, the ring verify operation takes as inputs the signature, and a message. Essentially, verification is applying all steps of the signature algorithm, but in reverse. We proceed as follows:

1. Use the public keys to get all the y_i from x_i: $y_i = g_i(x_i)$.
   As stated earlier, note that outputting all x_i and having the verifier compute the y_i, instead of sending the y_i directly, is crucial. Also, note that the public keys (as well as all x_i) are found inside the signature.

2. Find the first half of the key: $k = h(m)$.
   The value v is already inside the signature, so these two values uniquely specify which combining function, inside this family, we are using.

3. Check that the y_i (and, hence, the x_i) are valid: $C_{k,v}(y_1, ..., y_r) = v$.

The check above tells the verifier if the signature is valid. As mentioned earlier, a list of valid x_i can only be found with knowledge of at least one secret key. As such, checking the combining function guarantees that, w.h.p., the signature was indeed crafted by a ring member.

4 Security Analysis

The main novel feature of ring signatures is the anonymity of the signer. To reiterate this point, when a signature is published with all of the potential actors (in the form of their public keys), an adversary cannot tell, w.h.p., which of the public key bearers was the perpetrator of the signature. This anonymity comes from the property that the outputs of any given trapdoor function and its inverse is random and unpredictable. Recall that in constructing the signature, the signer picks values of x_i at random for all members except themselves (x_s). Then using these x_i's, the trapdoor function and combining function, a valid y_s is solved for which validates and completes the ring signature. Then this y_s

goes through the inverse to retrieve x_s. Notices that at this step, x_s is entirely random. And so where the other x_i's. The resulting $x_1 \dots x_r$ including x_s are completely random. And thus the adversary has no information to use to guess which x_i came from the real signer.

Additionally, this signature scheme must also be secure in the traditional sense of signature security – that is an adversary cannot create any valid signature in the name of another party (unforgeability). This proof is reduced to the fact that a trapdoor cannot be easily inverted without the complementing private key. In other words, the adversary could complete all steps of the signing process up until when they would need to invert y_s to get x_s which is by definition hard without the signer's private key. Moreover, even with a signing oracle that can be applied on all messages except the once in question, that message is the key to the pseudo-random permutation function. So the resulting signature would be useless in helping compute the signature of another message since the E_k is entirely different and scrambles entirely differently in a completely random and unpredictable manner.

5 Implementation

Our application was split into two parts: the backend where the cryptographic functions would be computed, and the frontend which would be a user's gateway to these functions. The entire application runs entirely on the localhost network to avoid sending sensitive information over the wire.

5.1 Backend

For the backend, we used the Flask framework that serves the cryptographic Python code to the frontend. Flask exposes port 5000 to communicate with the frontend. To generate RSA keys and support AES functions, we used the "cryptography" python library. Our cryptographic primitives were public and private keys generated through RSA. We used AES with CBC initialized with a random IV as our pseudorandom permutation. However, per signature, the same IV is used to both sign and verify. The trapdoor functions follow the textbook (e, d, n) modular exponentiation of RSA. All randomness is generated from the os.random library which is cryptographically secure.

The library code, found inside the "crypto" subdirectory of our project repository, is loosely organized as follows:

- crypto_utils.py Contains our implementation of E_k. As mentioned above, the way we went about this is by using AES-CBC, but reusing the same IV inside the signature. That is, we sample a fresh IV every time we start a signature, but reuse it every time we call E_k inside the signature. We then include the IV as part of the signature, so that the verifier can use the same permutation. Another small "hack" we had to do was to make

the value b a multiple of 128 (AES's block length). Without this, the AES subroutine would try to pad the message, and it wouldn't be a true permutation. With these two tricks in place, our algorithm behaved like a true pseudorandom permutation.

- ring.py Is an interface representing a ring of users. This interface is later on extended by sign.py and verify.py. Essentially, this class contains information about the public keys and the value b of the extended domain.

- signer.py, Which extends ring.py, represents a ring of users for which a message is being signed. In addition to the aformentioned attributes, it holds the signers secret key, and provides utility functions to sign a message. This essentially implement the sign algorithm described on section 3.1.

- verifier.py, Which extends ring.py, represents a ring of users for which a message is being verified. It provides utitlity functions to verify a message, and essentially implements the algorithm described in 3.2

- sing_main.py Is the main interface used to sign messages. This can be run by clients directly, as a CLI. In addition, this is what our frontend calls to interact with our crypto code. As mentioned in the high-level description, signer.py and verifier.py work with RSA keys from the "cryptography" package, which are RSAPrivateKey and RSAPublicKey objects. As such, this interface serves as the bridge between the "real world", and the working functions which use these objects. More specifically, sign_main.py takes, via command line arguments, the message to be signed (as a string), a .PEM file containing all public keys, the index corresponding to the signer's public key, a .PEM file containing an (encrypted) secret key, and the name out the output file where the signature should be saved. We then parse the keys, in the universal PEM format for keys, into the appropriate objects, create a Signer object (from signer.py), and do all the work. Then, we save a .txt file (as specified by the last command line argument), containing all public keys, the x_i, the value v, and the iv.

- verify_main.py Is the main interface used to verify messages. It works very similar to sing_main.py, and serves a similar purpose, so we will not go into much depth. Essentially, users pass in a message (as a string), and a signature file, following the format from sign_main.py, and verifies if this is a valid signature.

Overall, the neat thing about our code (besides being complete and secure, of course) is that it is very implementation-independent, and works as an "application" instead of simply a "library". In particular, users just need to hold standard RSA keys, and don't have to hold these as specific Python objects in order to interact with the library.

5.2 Frontend

The frontend comprises of a simple user interface built using Node.js and the Vue.js framework. The Vuetify component library was used to achieve a minimal look. The frontend server exposes port 8080 in order to communicate with the backend server.

Before using the application, users need to, unsurprisingly, start it, by running "./run.sh" inside the main directory. This application runs locally in the machine of the user, for security reasons, instead of being an external web server.

To sign a message, a user must first input a .PEM file that contains the public keys of the members of the ring. Then they must add the index of their public key within the ring. Afterwards they must input a file that contains the encrypted private key along with the password used for the encryption. Finally the user adds the message to be signed into the form. On submission, a file "ring-signature.txt" is created on the local directory.

To verify a message a user must simply attach the "ring-signature.txt" file into the verification form and add the message to be verified. The application will then verify the message and will inform the user accordingly.

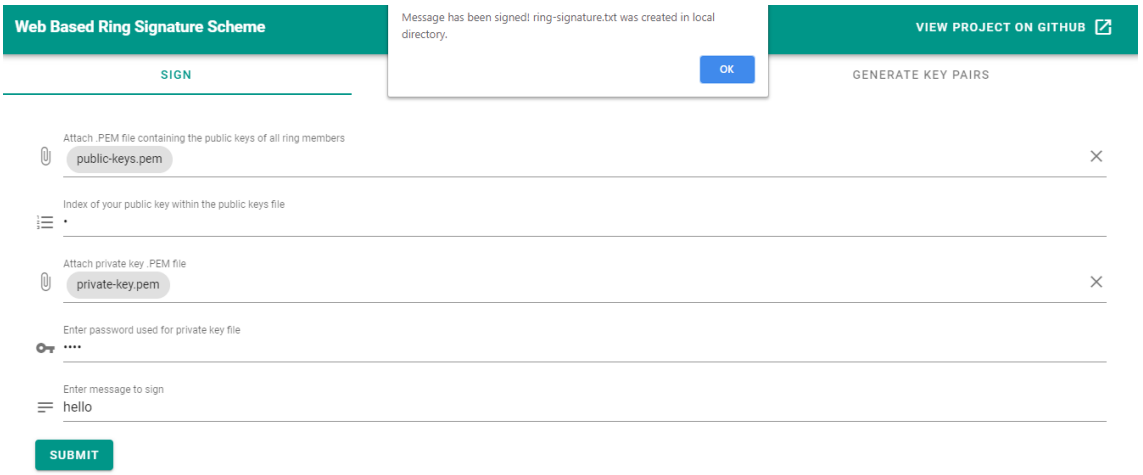
6 Results

Our application works amazingly well: as of this moment, we don't know of a single edge case where it doesn't behave as expected. We tested a large amount of different scenarios (which you can test for yourself by using the application), and the output was always the expected one. To name a few:

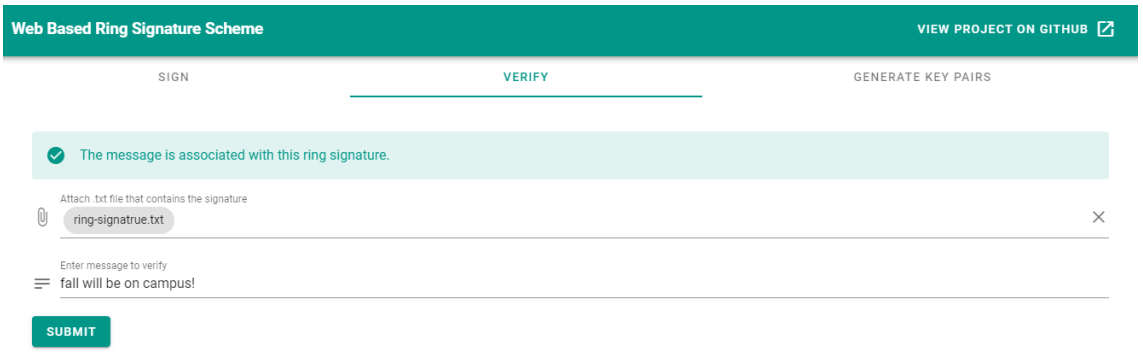
- Valid inputs produce a valid signature.
- A valid signature is correctly verified.
- Attempting to sign a signature with the wrong public key index throws an error.
- Attempting to verify a signature for x, when this is actually a signature for $y \neq x$, is correctly verified to be false.
- Attempting to change the signature in any way (including, but not limited to, deleting public keys not corresponding to the signer, changing the order of any two keys on the signature, etc) is correctly verified to be false.

These are just a few of the things we tried. To our knowledge, it behaves exactly as specified (albeit our evidence is merely empirical).

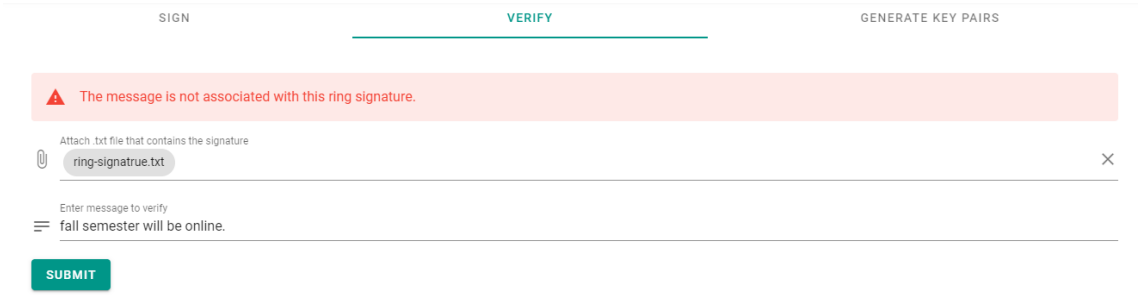
Signing a message



Verifying a correct message



Verifying a wrong message



7 Conclusion

Ring signatures are a fascinating concept. This project gave us a great opportunity to explore them in great depth, and get a very solid grasp of how/why they work. In addition, writing our own crypto library was a very rewarding and rich experience, as we got a chance to see all the nuances that this involves. Overall, we are very proud of our end product, and the fact that this is a functioning application which anyone can use.

8 Acknowledgments

We want to thank the authors, Rivest, Shamir, and Tauman, for their paper *How to Leak a Secret*, which motivated this project and served as the guide. Also, we want to thank our professors Ron Rivest and Yael Kalai, and the rest of the 6.857 staff, for all the knowledge they shared with us.