Wrocław University of Science and Technology

M.Sc. Thesis

Scheme for self-expiring signatures

Rafał Pacut

supervised by
Dr inż. Przemysław Błaśkiewicz

June 2020

Contents

1. Introduction

1.1. Motivation

In recent decades the introduction of new technologies has presented many challenges to lawmakers. From digital copyright to privacy regulations, we struggle to impose the rules deemed fundamental and beneficial to our society. Six years after establishing the Article 17 of the GDPR [8] – the "Right to be forgotten"– it continues to pose substantial difficulty to enforce. The lack of the right tools becomes more and more apparent.

In today's increasingly digital society digital signatures are commonly used to provide credibility to the signed data. In the case of many types of data, like hospital onboarding data or loan data, it stops beeing a hearsay and becomes sensitive if signed by the right person or organization. On the other hand one could argue that when a signature is unverifiable, it is also meaningless and sensitivity of the signed data can be questioned.

This work proposes three transformation mechanisms that can be applied to most digital signature schemes resulting in a privacy enhanced digital signatures. The resulting digital signature schemes are non-repudiable up to a predefined date in the future, after which they cannot be verified.

1.2. Difference with existing mechanisms

An expiry mechanism already exists in digital signatures. Public keys are certified to ensure the identity of the signer. Certificates have an expiry feature, but it serves mostly as a warning, as the mathematical operations can be performed even if the expiry date was exceeded. An individual can choose to ignore the warning and nonetheless verify the signature, proving that before the expiry date the signer signed the accompanying message. Although non-expired certificates can be revoked by means of Certificate Revocation Lists[5] and Online Certificate Status Protocol[6] they still can be ignored. Existing system does leave a way to repudiate a digital signature, albeit a radical one: one can claim his secret key was stolen. An obvious disadvantage is that the key cannot be used afterwards and no signature signed with it can be trusted.

1.3. Problem Statement

The stated problem is a direct transfer of *retrospective privacy* problem (described in chapter 2.1) to digital signatures:

User signs a message at the time t_0 and publishes the signature to the Internet. Up to the time t_1 the signature should be verifiable. At time t_1 an adversary gains access to all of user's cryptographic keys. Adversary should not be able to verify the signature.

The user should not be required to take any action after t_0 for the signature to become unverifiable after t_1.

1.4. Contribution

The contribution of this work is an adaptation of techniques from the *retrospective privacy* field to digital signatures resulting in signature schemes that:

- self-expire,
- after expiry are repudiable (cannot be verified).

2. Related Work

2.1. Retrospective Privacy

Once data gets uploaded to services on the Internet, many times it gets distributed to data centers all over the world to become more available for access and for backup purposes.

The term *retrospective privacy* means that a user, after publishing his data to the Internet, can be certain that at some time in the future access to every copy of this data will be revoked. Most of the techniques used rely on encrypting data before publishing it to the Internet in such a way that at some point in the future, decryption of this data will become infeasible or even impossible.

Techniques from the field of *retrospective privacy* rely heavily on a concept of *self-destructive data*, which constitute the foundation of this work.

Self-destructive data is the key to fulfilling the *privacy* element. Usually one of two approaches are taken – the key used in data encryption either:

- comes from a reliable, ephemeral data source and is discarded right after encryption, [9]
- is generated by the user, published to an ephemeral data structure, then discarded. [7]

In both scenarios the key should be available for reconstruction up to some predefined point in the future to allow access. The data source or the data structure should have a reliable churn rate to allow predictions about a lifetime of the key. One of the challenges of such systems is to have well-defined, reliable key lifetimes.
The *retrospective privacy* model is described below:

The user runs an algorithm on his data at time t_0, then publishes the processed data to the Internet. Until the time t_1 the user has access to his data. After t_1 an adversary gains access to all of user's cryptographic keys. Adversary should not be able to read the published data.

As a consequence of an adversary owning user's cryptographic keys, the user should not be able to read published data after time t_1, as an adversary could impersonate the user. The user should not be required to take any action after t_0 for the data to become unreadable after t_1.

Following sections will describe two chosen *self-destructive data* schemes.

2.2. FADE

FADE [9] creates an encryption key from Google's search engine results in the *news* category. It's top 10 headlines (first presented page) for a query can be viewed as an ephemeral data. FADE encloses encrypted data and information needed to decrypt it within a data structure called FADE Data Object (FDO).
The procedure to construct the FDO is as follows:

1. Generate a random, four-digit number that will be a query
2. Query search engine in the *news* category
3. Apply combining function to the results
4. Construct the key
5. Encrypt the data with the key
6. Construct FDO as a tuple (*ciphertext*, *query*)

FDO can be then published freely to the Internet.
A natural approach to producing a key from top ten results headlines is to hash them together. There is a number of ways those headlines can combined, each resulting in a slightly different FDO lifetime. Hashing top ten results implies that if only one of them changes, the key is lost.

Proposed result combining functions are:

- FirstNHeadings(n) - returns the hash of the first n search results headings
- LastNHeadings(n) - same as above but on the last n
- FirstNHeadingsSorted(n) - returns the hash of the first n sorted search results headings

- LastNHeadingsSorted(n) - same as above but on the last n

To retrieve the plaintext:

1. Extract query from FDO
2. Query the search engine
3. Apply combining function to the results
4. Reconstruct the key
5. Decrypt the ciphertext

Users have no control over the results of search engine query. The results are in a sense volatile, because unexpected and unforeseen events can change the query results. To have some stability and repeatability of results, author devised an interesting *reverse* variant of Shamir Secret Sharing algorithm. There, instead of splitting a key into shares, the results of many queries are merged into a key. Because Shamir Secret Sharing is a threshold encryption scheme, reconstruction of a key requires only some of the results to remain unchanged. Moreover, this procedure is parametrized by the shares number and threshold amount of shares needed to reconstruct the key giving more room for *lifetime* control.

2.3. Neuralizer

Neuralizer [7] uses public DNS cache as an ephemeral data structure used to store an encryption key. This approach has a number of advantages, for example: every DNS cache entry has a set *time to live* parameter that defines it's lifetime accurately.

Unlike the search engine, here we have the power not only to read, but also to write data into cache. Neuralizer used this procedue to provide flexible time revocation based on access heuristics. A user, instead of picking an expiration date at random, can for example choose the data to expire on a drop of interest heuristic which is very natural.

Authors give an example of publishing a picture from some event. Early after the event, the interest might be substantial, but decreases with time. We would like the picture then to become unavailable.

Neuralizer scheme constructs an Ephemeral Data Object (EDO) which contains encrypted data along with the information needed to recreate the decryption key.

EDO construction:

1. Generate key, encrypt the data;
2. Encode key into DNS cache;
3. EDO is list of pairs (domain_name, DNS resolver's IP address) and ciphertext.

To encode the key, associate every bit with a DNS entry pair of the form (domain name, DNS IP address). Then, if the bit was 1, perform a recursive query for its domain name to its associated DNS resolver, placing the domain name in resolver's cache. If the bit was 0, do nothing.

To avoid the issues that arise if DNS resolver clears cache, every bit is associated with multiple DNS entry pairs, thus providing redundancy.

Authors advise to use a precompiled list of domain names that are used so rarely they are not likely to reside in DNS cache. They also propose a (much more time consuming) method to generate such a list: perform reverse DNS queries for a random range of IP addresses.

Data access from EDO:

1. Retrieve the encrypted data, and extract the DNS entry pairs for every key bit;
2. Perform non-recursive queries to DNS resolvers for corresponding domain names;
3. Prolong EDO lifetime.

In the step (2) the original key is recreated. Resolver generally will respond with either an IP address (meaning the bit was 1) or with a list of name servers it knows (in this case the domain was not in cache and bit was 0).

After the interest drops, step 3 of Access will be executed less and less often resulting in overwriting the data in DNS cache, thus destroying the key.

3. Self-Expiring Signature Schemes

This chapter describes the contribution of this work and contains various ways a self-expiring signature can be produced using a self-destructive data scheme.

Let us define two procedures: $DataCreate$ and $DataFetch$ that will respectively create and recreate data following any self-destructive data scheme described in chapter 4. $DataCreate$ will be parametrized by a desirable expiry date d_e and should produce a random bitstring d and that bitstring's public *portrayal* p, that is, a 'recipe' for recreating that bitstring. $DataFetch$ will take as a parameter p and return the corresponding bitstring.

3.1. Transient key pair scheme

The idea is to create a transient key pair from self-destructive data d and sign the message with the resulting transient secret key a_t. Then publish a function f that, given self-destructive data d and a long-term, certified public key A, will return the transient public key A_t. Verifier will request the long-term public key A, obtain self-destructive data d and recreate A_t with which the signature can be verified. A high-level sketch of the scheme is decribed below:

ExpSign(A, m, d_e):
$d, p = DataCreate(d_e)$
$a_t = g(d)$
$A_t = f(A, d)$
$\sigma = Sign(m, a_t)$
return $\{\sigma, m, p, f\}$

ExpVerify(σ, A, m, p, f):
$d = DataFetch(p)$
$A_t = f(A, d)$
$Verify(\sigma, m, A_t)$

Where:
A is long-term public key,
m is message,
d_e is desirable expiry date,
d and p are the self-destructive data and its portrayal,
g is key generating pseudo-random function,
f is public key generating function,
(a_t, A_t) is transient key pair,
p is portrayal,
σ is signature;

Sign and $Verify$ correspond to to signing and verification procedures of an underlying signature scheme.
Expiration:
As soon as d becomes unavailable, the f function lacks an argument, A_t cannot be recovered and the signature cannot be verified. Moreover, it is then repudiable as it can be argued that the value of d was different and the transient key pair with which the message is signed did not belong to the signer.

3.2. Binding schemes

One way to obtain a signature that self-expires is to bind a message to self-destructing data. The following ideas are modifications of exisiting signature schemes and will be presented on the Shnorr Signature Scheme [2].

Recall that the Shnorr Signature Scheme is:

Sign(a,m):
$x \leftarrow_{\$} \mathbb{Z}^*$
$X = g^x$
$c = \mathcal{H}(m, X)$
$s = x + a \cdot c$
$\sigma = (s, X)$

Verify(A,m,σ):
$(s, X) = \sigma$
$c = \mathcal{H}(m, X)$
$g^s =^{?}= XA^c$

Where:
a is a private key
A is a public key
m is a message
g is a group generator
$\mathcal{H}$ is a secure hash function
σ is the signature
$\leftarrow_{\$}$ stands for 'take at random from'

3.2.1. Hash binding scheme

Because of its properties and ubiquity in digital signature schemes, a hash function is a natural place to bind a message with self-destructing data.

The following is a self-expiring scheme based on the Shnorr Signature Scheme using the hash binding mechanism:

ExpSign(a,m,d_e):
$x \leftarrow_{\$} \mathbb{Z}^*$
$X = g^x$
$p, d = DataCreate(d_e)$
$c = \mathcal{H}(m, X, d)$
$s = x + a \cdot c$
$\sigma = (s, X, p)$

ExpVerify(A,m,σ):
$(s, X, p) = \sigma$
$d = DataFetch(p)$
$c = \mathcal{H}(m, X, d)$
$g^s =^{?}= XA^c$

Expiry:

When d becomes unrecoverable, then it is infeasible to brute-force search for d (if it has reasonable length) or search for some d' s.t. $\mathcal{H}(m, X, d') = \mathcal{H}(m, X, d)$ because of hash function properties. Original hash value is unrecoverable and the signature is not verifiable.

Repudiability:

After expiration (without d) it can be argued that the hash c was produced with a combination of different message and d.

Correctness:

If we assume that the verifier and the signer fetch the same value d, then $c_{sign} = \mathcal{H}(m, X, d) = c_{ver}$ and $g^s = g^{x+ac} = XA^c$.

3.2.2. Randomness binding scheme

To protect from various attacks, digital signature schemes involve randomness. It is a perfect element to mix with *self-destructing data*, as it prevents any reasoning about the resulting value.

Below is a scheme depicting a self-expiring scheme based on the Shnorr Signature Scheme using the randomness binding mechanism:

ExpSign(a,m,d_e):
$p, d = DataCreate(d_e)$
$x \leftarrow_{\$} \mathbb{Z}^*$
$X = g^{x+d}$
$c = \mathcal{H}(m, X)$
$s = x + a \cdot c$
$\sigma = (s, c, p)$

ExpVerify(A,m,σ):
$(s, c, p) = \sigma$
$d = DataFetch(p)$
$X' = g^s A^{-c} g^d$
$\mathcal{H}(m, X') =^{?} = c$

Expiry and repudiation occurs is a similiar manner to the hash binding scheme. Correctness:

We will assume that the verifier and the signer fetch the same value d, then

$$X' = g^s A^{-c} g^d = g^{x+ac} g^{-ac} g^d = g^{x+d} = X$$

Hence:

$$\mathcal{H}(m, X') = \mathcal{H}(m, X) = c$$

Security proof:
The following is a proof of existential unforgeability under chosen-message attacks in the random oracle model assuming that the discrete-logarithm problem (DLP) is hard.

Sketch of the proof:
Assume that a probabilistic polynomial-time forger $\mathcal{F}$ for the above scheme exists. We will embed a DLP instance into the public key and then use the oracle-replay attack to obtain two different forgeries that share the same randomness. This will allow us to break the given DLP instance, proving that $\mathcal{F}$ cannot exist.

Proof:

I. Init stage:
   Let:
   -λ be the security parameter,
   -p, q be two large primes s.t. $q \mid (p-1)$ and $2^{\lambda-1} \leq q \leq 2^{\lambda}$,

-g be an element of $\mathbb{Z}_p^*$ of order q,
-g, g^α be the given DLP instance;
Set the public key to g^α.

II. Simulating oracles for $\mathcal{F}$:
For $\mathcal{F}$ to produce forgeries under chosen-message attacks we will need to simulate three oracles: hash oracle $\mathcal{O}_{\mathcal{H}}(\cdot)$, signature oracle $\mathcal{O}_{Sign}(\cdot)$ and self-destructing data oracle $\mathcal{O}_{SDD}(\cdot)$. This will be accomplished using two separate tables: the hash table (for $\mathcal{O}_{\mathcal{H}}$ and $\mathcal{O}_{Sign}$) and sdd table for $\mathcal{O}_{SDD}$. Tables will be populated in a lazy manner.

Signature oracle will handle requests $\mathcal{O}_{Sign}(m)$ in the following way:

1. Take p, d at random from $\mathbb{Z}_q^*$ and create a record in the sdd table:
   input: p, result: d,
2. Take at random s, c, x from $\mathbb{Z}_q^*$. Create a record in the hash table:
   input: $m, (g^s g^{\alpha - c} g^d)$, result: c,
3. Return $m, (s, c, p)$ as the signature;

$\mathcal{F}$ has to be able to verify the above signature. He will query $\mathcal{O}_{SDD}(p)$ for which we return d. Then $\mathcal{F}$ calculates $X' = g^s g^{\alpha - c} g^d$ and queries $\mathcal{O}_{\mathcal{H}}(m, X')$, which returns c. Then $L = \mathcal{H}(m, X') = \mathcal{H}(m, g^s g^{\alpha - c} g^d) = c = R$ and verification holds.

Hash oracle $\mathcal{O}_{\mathcal{H}}$ for request (m, X):

1. if entry for (m, X) exists in the hash table, return corresponding c,
2. if not, then take c at random from $\mathbb{Z}_q^*$,
3. create a record in the hash table:
   input: m, X, output: c;

SDD oracle $\mathcal{O}_{\mathcal{SDD}}$ for request d_e:

1. Take p, d at random from $\mathbb{Z}_q^*$ and create a record in the sdd table:
   input: p, result: d,
2. return p, d;

III. Forgery stage:

1. Pass the public key g^α to $\mathcal{F}$,
2. Handle queries to oracles in the manner described above,
3. $\mathcal{F}$ returns a forgery $(m, (s, c, p))$;

IV. Forking:
Apply the forking lemma [4] to rewind $\mathcal{F}$ to the point where he commited to the randomness x and d. For all the subsequent queries to $\mathcal{O}_{\mathcal{H}}$ respond with different values than previously. The forger produces another forgery $(m', (s', c', p))$. Because the forger performed at most q queries to the random oracles $\mathcal{O}_{\mathcal{H}}$ and $\mathcal{O}_{Sign}$, he will have to forge on one of those queries, so with probability at least $\frac{1}{q}$, $m = m'$. Then $s = x + \alpha c$ and $s' = x + \alpha c'$. From this we can calculate

$$\alpha = \frac{s' - s}{c - c'}$$

which is the solution to the given DLP instance.

V. Conclusion:
This happens with probability at least $\frac{1}{q}$. Since $\mathcal{F}$ is a probabilistic polynomial-time adversary, the number of performed queries q is polynomially-bounded, so $\frac{1}{q}$ is non-negligible. Because we assumed that DLP is hard, then $\mathcal{F}$ cannot exist. □

4. Implemented self-destructing data schemes

In times of mass data collection for machine learning algorithms and very cheap storage it seems like no publicly available data is ephemeral. For the data source to be useful in the self-expiring signature context it also has to also satisfy other conditions. Ephemeral beacons [12] [11] fulfill the ephemeral part, but they are susceptible to scraping. The sources that are hard to scrape have to further allow reliable predictions about the data lifetime. Sources satisfying all those conditions are few and far between.

This section describes three *self-destructing data* schemes implemented in the prototype. Two are improved versions of the Neuralizer and FADE schemes described in 2.2 and 2.3. One is a locally hosted service.

4.1. DNS cache scheme

4.1.1. Overview

The DNS Cache self-destructing data scheme is an enhanced version of the Neuralizer scheme. One disadvantage of the Neuralizer scheme is that it requires a list of rarely used domains. Dictated mostly by the ease of implementation the implemented scheme makes use of the DNS negative cache [3].

Negative cache implemented in DNS servers allows caching of requests for non-existing domain names. DNS requests for mistyped domain names or expired links to

non-existing domains generate a lot of internet traffic. Every such request is a recursive one and always traverses the full path to the domain's root nameserver.

In the past the implementation of the DNS negative cache was optional, but today it is not. A recent study [13] shows that even though its adoption is slow, by now most of resolvers have this feature implemented.

This not only simplifies the process of generating a list of domains, but also impacts the execution time. In Neuralizer scheme authors advise to check if the domain is located in the cache before using it to portray a key's bit. If resolver's cache contains a record with a domain that corresponds to a key's bit with value 0, then for the record's remaning *time-to-live* the key is only recoverable if we chose to use redundancy.

With negative cache in mind, we can generate a random string and use it as a subdomain. For example, if the random string is '513411b2', then we will ask a resolver for the IP address of '513411b2.example.com'. Here, we can safely assume that with a reasonably long random string, the subdomain is not likely to reside in resolver's cache, which eliminates the need to issue a confirming request.

Other difference is the expiry time mechanism. Whereas Neuralizer provided flexible data expiry time, here the data is continually placed in the DNS cache by the signer, providing precise control – up to the max *time-to-live* value – of the expiration date.

4.1.2. Scheme

Below are described DataCreate and DataFetch procedures for DNS cache scheme. Implemented versions of those procedures additionally contain redundancy, that is, in DataCreate every key bit is placed in many DNS resolver's cache and DataFetch assumes the key bit value only after it reaches a predefined threshold value. Here, for clarity, those details were omitted.

input : expiry date d_e, DNS resolver IP *ip*
output: data d and portrayal p

```
d ← random();
p ← [ ];
for every bit b of d do
    randomStr ← genRandomString();
    randomSubdomain ← "randomStr.example.com";
    if b == 1 then
        //place in the cache
        requestAddress(ip, randomSubdomain);
    end
    p.append(ip, randomSubdomain);
end
RefreshData(p, d_e);
return d, p;
```

Algorithm 1: DNS Cache DataCreate (without redundancy)

input : portrayal p
output: data d

```
d ← [ ];
for ip,subDomain of p do
    res ← requestNonRecursiveAddress(ip, subDomain);
    if res == NXDOMAIN then // wasCacheHit?
        b ← 1;
        else
            if res == NOERROR then // wasCacheMiss?
                b ← 0;
            end
        end
    end
    d.append(b);
end
return d
```

Algorithm 2: DNS Cache DataFetch (without redundancy)

4.1.3. Malicious lifetime prolonging

Since the DNS cache is a publicly accessible mechanism with read and write permissions, it is possible for a malicious attacker to prolong the validity of the signature indefinitely if he obtains the key before its expiration. Unfortunately, an elegant mitigation does not seem to exist. The key can be made unrecoverable by placing the domains in the cache for every key bit, including 0 bits. This in effect turns the key to 1^n bitstring, but those records have to be continually refreshed in the cache locking the signer into a marathon race with the adversary.

4.2. News data scheme

The news data scheme is based on the FADE scheme with a slight improvement. In contrast to other implemented schemes, in this one the data is not generated by the user, but fetched from a search engine. As a consequence, a precise date after which the data becomes unavailable can only be approximated.

4.2.1. Revese Shamir Secret Sharing

To understand how the news data scheme works it is instrumental to understand the Reverse Shamir Secret Sharing (RSSS) algorithm proposed in [9].

RSSS will create a secret key from many search engine query results, then will split the key into shares. The key can be recreated only with the threshold amount of query result and share pairs. RSSS builds up on the Shamir Secret Sharing [1] (SSS) with two more procedures: RSSS-Create and RSSS-Combine. In the procedures below, the SSS-Split and the SSS-Combine are standard SSS splitting and combining procedures. The threshold is the minimal amount of shares needed to recreate the key.

input : queries result list q, shares amount n, shares threshold m
output: key, encrypted shares

key ← random();
shares ← SSS-Split(n, m, key);
for *i from* 1 *to* n **do**
  $shares_i \leftarrow encrypt(shares_i, q_i)$;
end
return *key, shares*;

Algorithm 3: RSSS-Create [9]

input : queries result list q, encrypted shares
output: key

for *i from* 1 *to* n **do**
  $shares_i \leftarrow decrypt(shares_i, q_i)$;
end
key ← SSS-Combine(shares);
return *key*;

Algorithm 4: RSSS-Combine [9]

4.2.2. Data fetching

RSSS parameters n,m have a direct effect on the signature expiry time. The signature is dependent on m keys out of n, instead of one. To better match the desired expiry time an empirical analysis of expiration times was conducted for different m,n combinations. Because the news search results are unpredictable, we can only capture the probability of expiration at some time t, not the real expiration time. For a given desirable expiry date d_e, we choose m,n combination that resulted in 100% probability of expiration for signatures at d_e.

d_e up to:	m	n
three hours	6	10
five hours	4	10
twelve hours	2	10

The author of the FADE scheme suggested using a four-digit number as a query, but it runs the risk of colliding with historic dates and dates near present times. Especially because of the multitude of annualy celebrated national holidays and anniversaries worldwide. It is generally safer to place a lower bound on the interval from which numbers are chosen at 3000 to avoid collisions with historic dates and dates near present times. If the number comes from a range 3000-9999, then every such number has more or less the same likelihood to be used in tomorrow's top news article.

The DataCreate and DataFetch procedures look as follows:

$DataCreate(d_e)$:

1. Based on d_e choose RSSS parameters: the number of shares s and the threshold value
2. Generate s random numbers
3. Query the search engine for each of those numbers

4. Create a key using RSSS-Create

5. Return the key and portrayal: (encrypted shares, queries)

$DataFetch(p)$:

1. Extract encrypted shares and queries from p

2. Query the search engine for the queries

3. Use obtained results and encrypted shares to recover the key using RSSS-Combine

Signatures created with *self-destructing* data coming from the news data source have maximum expire times comparable with the data from FADE scheme.

4.3. Randomness beacon

In some cases the data source is not suited for the general, worldwide use case. However it still can be very useful in a local, specific setting, particularly if the processed data is especially sensitive. An example might include medical facilities or banks.

Implemented source simulates a doctor's office setting. A patient might have to get a blood test, X-ray examination and blood pressure before seeing the doctor. If examination results are transmitted digitally to the doctor then they should be digitally signed to prevent manipulation.

Implemented source acts as a randomness beacon, where data changes with a daily interval. In this case the data might be a list of today's patients or exact times of patient check-ins. While not exactly ephemeral, it can be argued that from a one-year perspective it cannot be, at the very least, easily accessed by an unauthorised person.

A randomness beacon with a fixed, long data change interval like the one used is vulnerable to scraping, but here the system is assumed to not allow public access. In the doctor's office the system users would be nurses, paramedics and doctors.

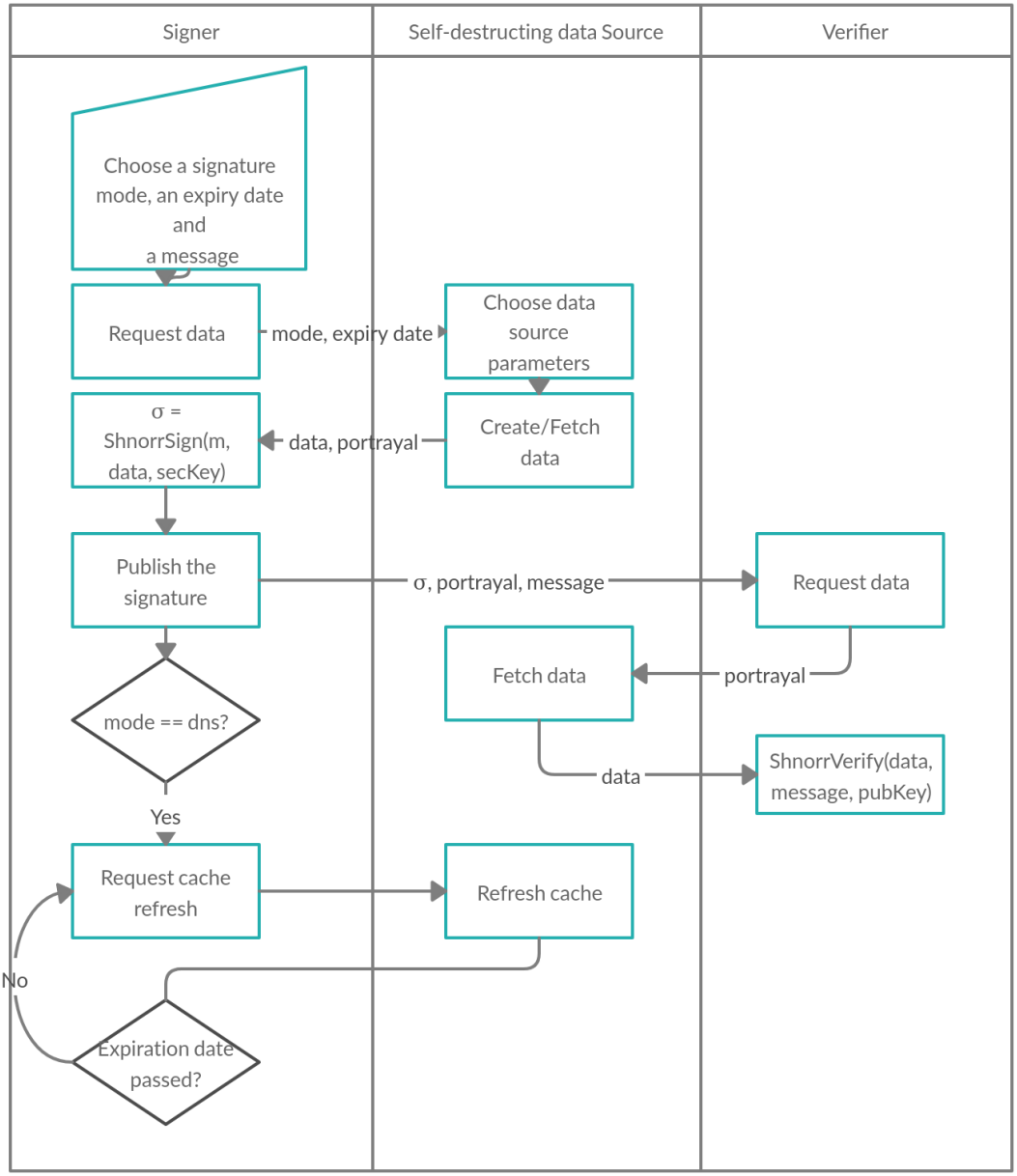
To demontrate this data source variant a simple server was hosted on the Amazon Web Service. Server handles a request for random data in the current interval. In this variant the expiry date d_e has fixed length of one day, and the portrayal p is not needed.

5. Implemented Prototype Overview

A prototype of a self-expiring signature was implemented. The prototype uses the mechanism of binding the self-destructive data to a message using a hash function (described in chapter 3.2.1) in the Shnorr Signature Scheme.

The signature prototype consists of three entities: the signer, the verifier and the *self-destructing* data source. The *self-destructing* data source represents any scheme described in chapter 4.

The following flow diagram provides a brief overview of the system:



6. Conclusion

In this paper several methods of transforming a digital signature scheme to its self-expiring version were proposed. The resulting signature schemes are non-repudiable up to a previously chosen date and repudiable afterwards, which is a novel signature property.

Implemented self-destructive data schemes, on which self-expiring signatures are built, were improved.

6.1. Future Work

The self-destructing data source is a central part of the self-expiring signature. In the case of the DNS cache data source the portrayal, that is, the recipe for recreating the key distributed as a part of the signature can get large. For every key's bit the portrayal contains a random bitstring of a parametrized length. If the bitstring is 8 bits long, then 256bit key requires at least 2048 bits of the portrayal without redundancy. Hence it is not well suited for signatures based on the RSA cryptosystem where the recommended key length is 2048 bits [10], but 4096 bit long keys are commonly used.

References

[1] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (Nov. 1979), 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[2] C. P. Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4 (1991), "161–174".

[3] M. Andrews. *Negative Caching of DNS Queries (DNS NCACHE)*. RFC 2308. RFC Editor, Mar. 1998. URL: https://www.rfc-editor.org/rfc/rfc2308.txt.

[4] David Pointcheval and Jacques Stern. "Security Arguments for Digital Signatures and Blind Signatures". In: *J. Cryptol.* 13.3 (Jan. 2000), 361–396. ISSN: 0933-2790. DOI: 10.1007/s001450010003. URL: https://doi.org/10.1007/s001450010003.

[5] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. RFC Editor, May 2008. URL: https://www.rfc-editor.org/rfc/rfc5280.txt.

[6] S. Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. RFC Editor, June 2013. URL: https://www.rfc-editor.org/rfc/rfc6960.txt.

[7] Apostolis Zarrasand Katharina Kohlsand Markus Dürmuthand Christina Pöpper. "Neuralyzer: Flexible Expiration Times for the Revocation of Online Data". In: *CODASPY '16 Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. 2016.

[8] *Regulation (EU) 2016/679 of the European Parliament and of the Council (GDPR)*. European Commission, Apr. 2016.

[9] Sam Wood. *FADE: Self Destructing Data using Search Engine Results*. 2018.

[10] Elaine Barker and Quynh Dang. "Special Publication 800-57 Part 1 Rev. 5". In: *NIST Special Publication 800-57* (2020).

[11] Cloudflare et al. *Distrubuted randomness beacon*. https://github.com/drand/drand.

[12] NIST. *Interoperable randomness beacons*. https://csrc.nist.gov/projects/interoperable-randomness-beacons.

[13] Lior Shafirand Yehuda Afekand Anat Bremler-Barrand Neta Pelegand Matan Sabag. "DNS Negative Caching in the Wild". In: *SIGCOMM '19* ().