ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Forward-secrecy on POP

Arthur Villard

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Master Project

September 2017

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linus Gasser
EPFL / DEDIS

Contents

1 Introduction

Many online services want to have accountability for their users while said users want to stay anonymous. For instance, an online voting service wants to be sure that each physical person participating only votes one time. The ideal situation would be to require the real identity of each user to detect the duplicates. On the other side, users of said service do not want their vote to be linked to their real identity or to another vote. The Proof Of Personhood framework (POP) solves the problem by allowing authentication of users in a trusted list in a anonymous but linkable way.

However, in the current version, if a private key of a user is broken or leaked, it is possible to break the anonymity of this user's authentication attempts. DAGA (Deniable Anonymous Group Authentication) offers a way to authenticate a user inside a group without revealing which user is authenticating and in a forward secure way.

This work consists of improving DAGA with more modern cryptographic primitives and implementing a complete version of DAGA in Go to be used with POP. The two previous implementation are available here[3] and here[9] as part of larger packages. The implementation done during this project is available on GitHub[13].

2 Definitions

Below are the definitions of some key concepts used later in this report.

Accountability means that it is possible to designate which user sent the message / used the service. It is often required either to be able to block a misbehaving user from the service or to limit how many times an action can be performed.

Anonymity is the situation in which any real information about a user is unknown. It can be essential in some services such as online voting systems. This concept is sometimes seen as opposed or incompatible with accountability.

Linkability is the ability to know if two messages come from the same user or not. It does not imply that said user is identifiable.

Zero-knowledge proof (or zero-knowledge protocol) "is a method by which one party (the prover) can prove to another party (the verifier) that a given statement is true, without conveying any information apart from the fact that the statement is indeed true" (source: Wikipedia [14]). Its main advantage is that someone listening to the exchanges between the prover and the verifier learns nothing about the secret whose knowledge is being proved.

3 Goals and Motivations

This project has three main goals detailed below. The first one is to solve two problems in the POP framework, the second to implement a complete version of DAGA in Go and the last one to improve DAGA with modern cryptographic primitives.

3.1 Current POP problems

POP's authentication method currently suffers from two main problems: no forward secrecy and cross-service de-anonymisation. The first one means that if a secret key is known, it is possible to reveal which tags where created from this private key (and which were not), thus breaking anonymity. The second occurs when the same user authenticates to multiple services. Because of the way linkage tags are created in POP, if two of these services talk together, they can cross-link the authentication attempts. It allows them to track the user when he moves from one service to the other but still without directly knowing which user he is.

Both of these problems can be solved through DAGA. Authentication requests and their linkage tags in DAGA can only be processed in a given context. This context is different from one service to another, preventing cross-linking. Additionally, a service can renew its context at will, erasing the secrets required to process requests in the previous context and, in doing so, providing forward secrecy. More precisely, if a given client authenticates in two different contexts (being two different services or the same service with a renewed context), he will receive a different linkage tag in each of the context.

3.2 Implementation of DAGA

As a result of this project, a complete implementation of DAGA in Go is available on GitHub[13]. Compared to previous DAGA implementations, this version offers distributed randomness and misbehaving client handling. See Section 6 for further details.

3.3 Improvements on DAGA

Before the implementation work, possible improvements were studied. The first result is the translation of DAGA from RSA-based cryptography to elliptic curve cryptography, which should improve message size and computation time. See Section 6 and Section 7 for further details. The second result is a study of accumulators and how to use them to reduce the size of the proof in DAGA. See Section 8.

4 Background on POP

4.1 Goal

The Proof Of Personhood framework offers a way to maintain both the accountability and the anonymity of the users. It aims at satisfying both the services and the users: services want to be able to determine if two actions were made by the same user (to prevent double voting or to block a misbehaving user) while users do not want to give any information about them to the services.

4.2 How it works

The framework relies on public/private key pairs for the users and a party transcript for the anonymous group authentication.

Multiple organizers create a ”POP party”. Users attending this party (called the attendees) generate their own public/private key pair and send their public key to every organizer. Once all the attendees have sent their public key to all the organizers, the organizers can generate the collective signature and form the party transcript. This transcript can then be used by services to authenticate users as part of this party without knowing which attendees they are.

Organizers are in charge of making sure that each attendee only registers one public key in the party and making sure that only the keys sent by the attendees are in the transcript (meaning that no organizer added unknown keys). As long as at least one of the organizer is honest (any-trust model), we are guaranteed that the transcript is valid.

Then, each user can create its pop-token from the party transcript and use it to access services. Additionally, a service must register the party transcript to use it for authentication. After that, when a client wants to authenticate, he will use his pop-token to sign the context and the nonce received from the service. He will then send back to the service the signature and the tag he created for verification. The service can finally determine if the authentication request is valid or not.

Figure 1 illustrates the typical usage of POP.

For a precise usage guide and the implementation, see [5].

5 Background on DAGA

5.1 Goal

The DAGA framework (Deniable Anonymous Group Authentication framework) allows users to authenticate as part of a group in a linkable but anonymous way. The protocol is run by a group of servers where every server is involved to process a request. Compared to the POP framework,

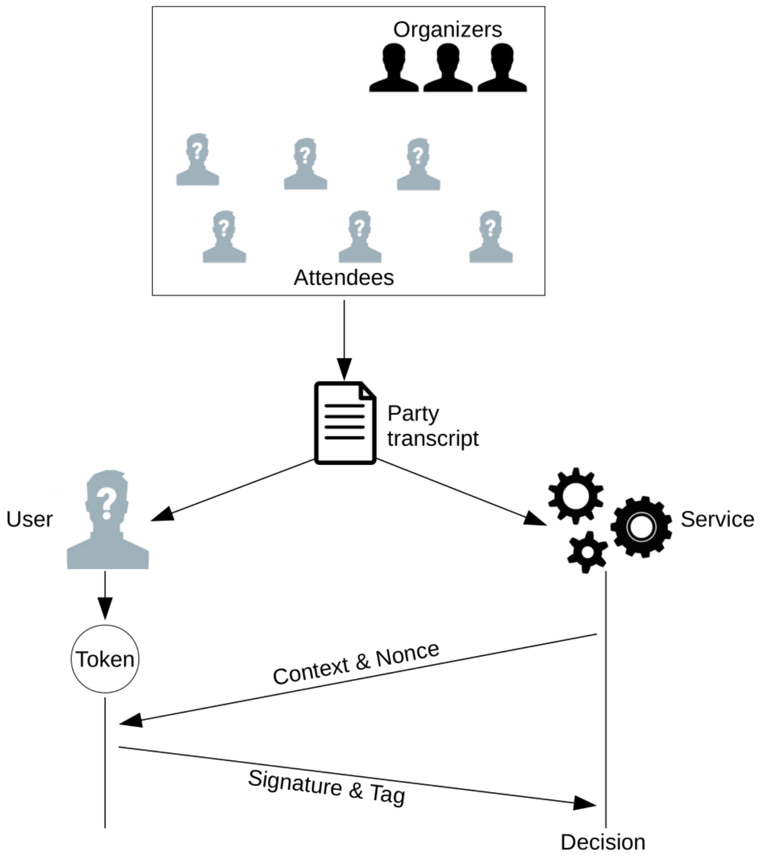


Figure 1: POP use case

it has the added property that the linkage tags are only valid in a specific context. Once this context expires, a user's linkage tag will be different (loss of linkability when the context is renewed) but the servers will also lose the ability to process requests made in the previous context.

As a concrete example, let us consider an online voting service. When a vote is opened, a context is associated to it. Each user can them participate in the vote and will receive its linkage tag. By keeping track of these linkage tags, the service can ensure that a user did not try to vote twice. If a user did try to vote twice, the service can identify the previous vote made by this user using his linkage tag and revoke it for instance. Once the vote ends, the context is expired, preventing the service from tracking a user between different votes and from accepting requests on closed votes.

5.2 How it works

This section will describe the protocol without going into the equations. The security model is the any-trust model where we consider that there exist at least one honest server, although we do not know which one. Additionally, we assume that secure communication channels already exist between all the parties involved.

Moreover, a misbehaving server in DAGA is supposed to be taken care of administratively. It means that, for the protocol to succeed, each server, whether honest or malicious, must follow the computations of the protocol. Otherwise, the processing of the request is stopped.

For a precise mathematical description of DAGA, see [11].

The context in which a user wants to authenticate contains:

- a description of the group made of the public keys $(X_0, \ldots, X_n)$ of all the users in the group and the public keys $(Y_0, \ldots, Y_m)$ of the servers running the protocol
- a list of commitments $(R_0, \ldots, R_m)$ where each server commits to a secret valid only for this context
- a list of client's generators $(H_0, \ldots, H_n)$, one for each user in the group.

Using this context, a client chooses an ephemeral key z_i and generates:

- its initial linkage tag T_0^i
- a set of commitments $(S_0, \ldots, S_m)$ to a secret shared with each server of the group (one secret for each server)
- an interactive OR-proof (see Section 8 for details on the proof and its improvements).

Then the client sends this message to an arbitrary server. One after the other, each server will then:

- Check the client's proof and the previous server's proofs.
- Update the linkage tag using the secret it committed to if the client did the correct computations or expose the miscomputation otherwise.
- Generate a proof that it correctly computed its part.
- Sign the message and send it to the next server (or broadcast it to all the other servers and to the client if it is the last server of the round-robin).

When the client receives the answer to its request, it knows whether it was accepted or not as well as its final linkage tag. This final linkage tag is independent from the ephemeral key chosen at the beginning of the protocol and will stay the same as long as the context is valid.

Figure 2 summarizes the exchanges between the client and the servers.

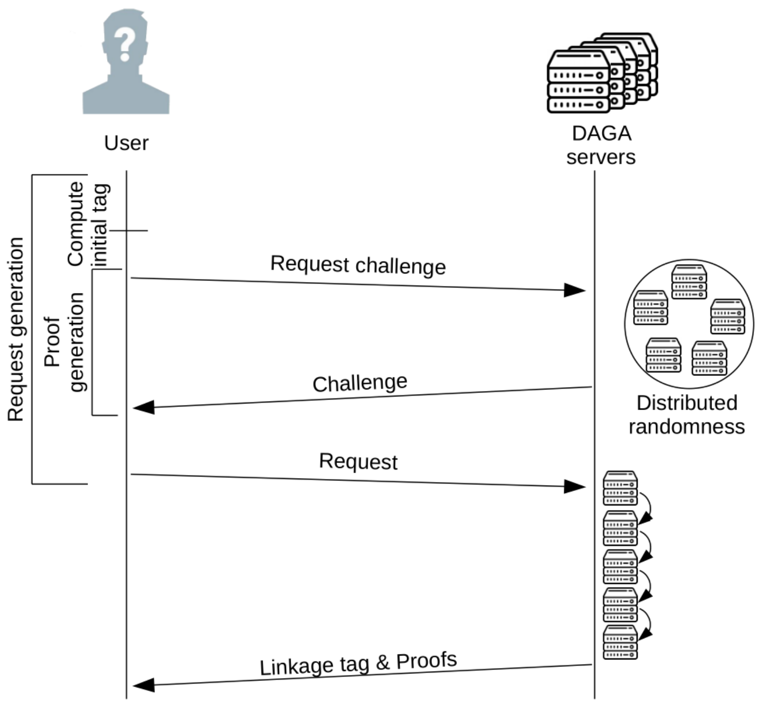


Figure 2: Interactions between the user and the DAGA servers

5.3 Why it solves the POP weaknesses

The linkage tag created by the client in DAGA is defined as follows $T_0^i = \left(\prod_{k=0}^{m} s_k\right) * H_i$ where H_i is the generator for this user defined in the context and s_k are the secrets shared with the servers defined by $s_k = \mathcal{H}_1(z_i * Y_j)$ with Y_j the public key of the server j, z_i the ephemeral key chosen by the client and $\mathcal{H}_1$ a hash function.

As explained above, the initial linkage tag in DAGA does not depend on the long-term secret key x_i of the client. So, a leakage of a client's private key does not allow an attacker who has intercepted a message to know which client it comes from or is about.

Additionally, both the shared secrets and the initial linkage tag depend on elements from the context. So the same client authenticating to two different services or to the same service but at two different times with two different contexts will not have the same linkage tag.

When a server processes a client's request, it updates the intermediate linkage tag in the following way: $T_j = (T_{j-1})^{(r_j)(s_j^{-1})}$ where r_j is the per-round secret of the server and s_j is its shared secret with the client (calculated from the client's request). Here again, the update depends on a context-dependent element (r_j) and a request-dependent element (s_j) but not a long-term secret key. Therefore it is protected from a leak of a secret key x_i.

5.4 Integration with POP

Inside the POP framework, DAGA will be used to replace the authentication process but not the group creation process.

The user's public keys in the party transcript are used in the description of the group. The service delegates the processing of the authentication request to a group of servers dedicated to it. See Figure 3 for an illustration.

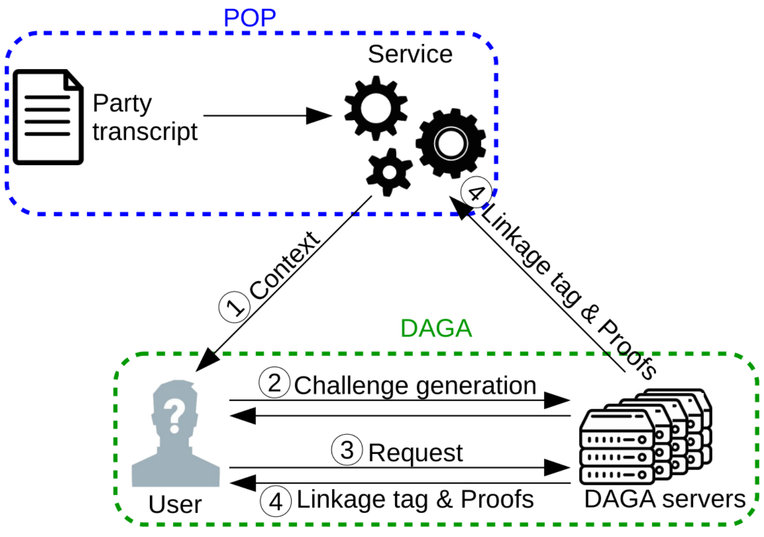


Figure 3: Integration of DAGA with POP

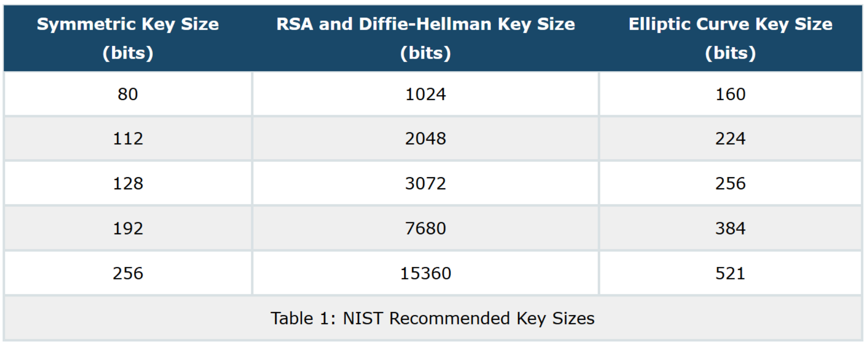
6 Implementing DAGA in Go

6.1 RSA and Elliptic Curve cryptography

RSA cryptography is based on integer arithmetic modulo a number n which is the product of two prime numbers. Its security relies on the difficulty of factoring n, so its length determines the security level when using RSA.

Elliptic curve cryptography is built on finite field arithmetic. Its security relies on the hardness of the discrete logarithm in some finite fields. The best algorithms to solve this problem (e.g. baby-step giant-step, Pollard's rho) run in $\mathcal{O}(\sqrt{n})$ with 2^n being the size of the finite field on which the curve is built.

However, the key length relationship with the security level is not directly comparable between the two systems, as detailed in the following table (taken from the NSA website [8]):



Symmetric Key Size (bits)	RSA and Diffie-Hellman Key Size (bits)	Elliptic Curve Key Size (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Table 1: NIST Recommended Key Sizes

The previous DAGA implementations use 2048-bit RSA keys. The implementation done during this project uses the Curve Ed25519 with 256-bit keys. This change should reduces the size of messages while improving the security level and was also necessary to make DAGA work with POP. Because elliptic curve operations are less CPU and memory intensive, it should also reduce the time required to authenticate. See Section 7 for the practical results and their interpretations.

6.2 Equation translation

As DAGA was designed using only RSA cryptography in [11], the first step was to translate the equations from the RSA cryptosystem (based on multiplicative groups) to the Elliptic Curve cryptosystem (based on additive groups). Generally speaking, an equation of the form $g^a * h^b$ in the RSA cryptosystem translates to $a * G + b * H$ in the Elliptic Curve cryptosystem with a and b being scalars and G and H points on the curve.

As a practical example, the shared secrets between the client and each server are originally computed with $s_j = \mathcal{H}_1\left(Y_j^{z_i}\right)$ where $\mathcal{H}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_q^*$ is a hash function, Y_j the public key of server j and z_i the ephemeral key chosen by the client. When using elliptic curves, they are now computed with $s_j = \mathcal{H}_1(z_i * Y_j)$ where $\mathcal{H}_1 : \{0,1\}^* \rightarrow \mathbb{Z}_l^*$ is the new hash function with l the order of the curve.

6.3 Distributed randomness

This implementation of DAGA makes use of the distributed randomness between the servers to generate the challenges used by the clients. The implemented solution is very close to the one described in [11].

Step 1: Upon receiving a client's request, server j (chosen arbitrarily by the client) assumes the role of a leader and requests that the other servers generate a new challenge c_s for the client by publishing its signed commitment C_j.

Step 2: Each server i chooses $c_i \in \mathbb{Z}_l^*$ and then calculates a commitment $C_i = c_i * G$ with G the generator of the curve. Server i signs and sends C_i to the leader.

Step 3: Upon receiving C_i from every other server, server j verifies if all C_i are of valid form (meaning that they are on the curve) and properly signed. If yes, server j publishes an opening c_j of its commitment C_j and requests other servers to open their commitments.

Step 4: Upon receiving an opening c_i from every other server, server j verifies if every c_i is indeed a valid opening of C_j. If yes, server j calculates $c_s = c_1 + \cdots + c_m$. Then server j signs c_s and forwards it to the next server along with the collected signed commitments and their openings. Each server can then verify the commitments, that c_s was correctly computed, add its signature of c_s and forward it to the next server.

Step 5: Upon receiving c_s signed by every other server, server j forwards c_s to the client along with the signatures from every server.

Under the any-trust assumption, there is at least one honest server h who will randomly choose his c_j. With the assumptions that the commitment is binding (there is only one opening per commitment) and hiding (no information on the opening can be learned from the commitment), it is guaranteed that the collective random challenge c_s is properly generated.

Figure 4 summarizes this protocol.

7 Results

7.1 Test environment

The tests were performed as a single process simulation, meaning that both the client and the servers were running in the same single threaded program. As expected, only one logical core was used by the benchmarks when running.

The network communication impact was simulated by having the participants use byte arrays to communicate. No data was exchanged without these intermediate marshaling and unmarshaling steps.

The timing was estimated with the time package from the Go standard library and the sizing by sequentially adding the lengths of the byte arrays as the entities communicate. The number of servers goes from 1 to 32 by powers of 2 except for the Server to Server Traffic as it is empty with only 1 server. The size of the group varied from 2 to 32768 by powers of 2. These values were chosen to reproduce the setups from Section 4.8.2 of [11].

The setup used was go1.9 on a 64-bit Windows 10 system running on a i7-7700K at stock speed (@4.2 GHz, turbo @4.5 GHz) with 2x8 GB of DDR4 RAM @3200 MHz.

Refer to Annex 11.2 for the guide on how to install and run the tests.

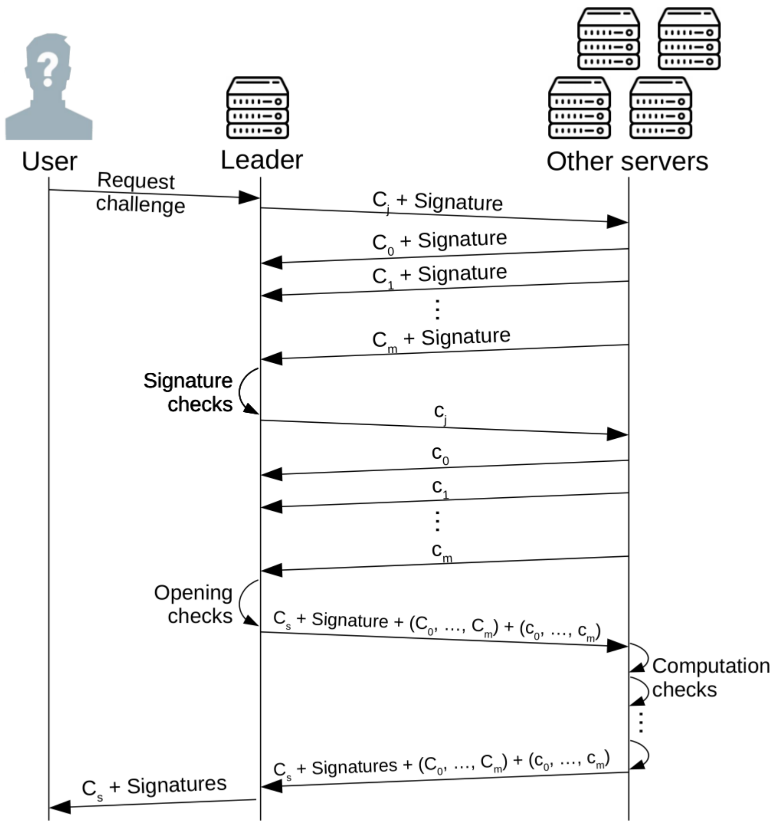


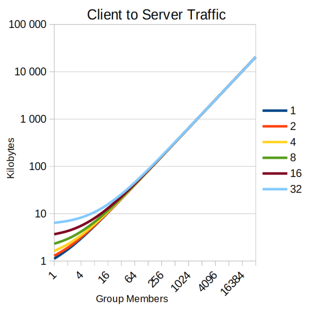
Figure 4: Distributed random challenge generation

7.2 Data and interpretations

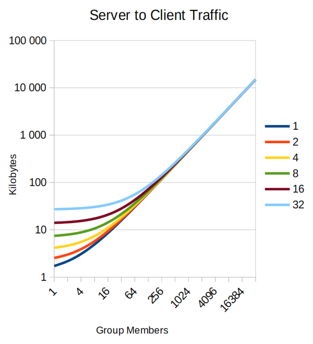
The results are presented in Figure 5. They will be compared to the results from the previous benchmarked implementation of DAGA available in Section 4.8.2 of [11] as the benchmarks of the previous implementation could not be reproduced on this machine.

Opposed from what was described in Section 6.1, the sizes are only slightly lower in all the communication channels. One explanation is the overhead added by the marshaling operation with an intermediary JSON-compatible representation. As en example, an Ed25519 point uses 32 bytes when stored but uses 46 bytes when converted using the methods implemented. It is still lower than the 256 bytes used by an RSA public key but it adds nearly 50% of overhead. However, even for more complex structures, it does not explain why the improvement is so small. Further investigations are required to really benefit from using elliptic curves.

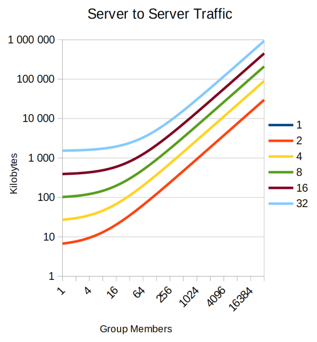
When it comes to timing however, there is an improvement of more than 10x for all the possible setups. Care should be taken as the latency and transfer rate of a real network were not included in this benchmark and



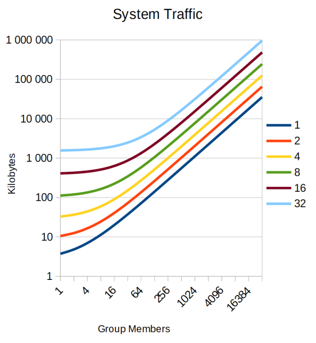
(a)



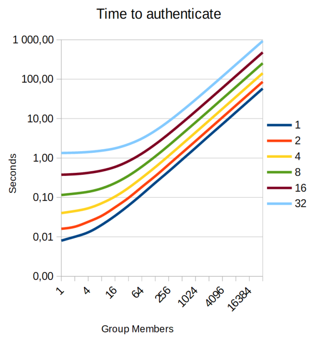
(b)



(c)



(d)



(e)

Figure 5: Results of the benchmarks

processing speed has increased since 2015. But we can still conclude that using Elliptic Curve cryptography greatly improved the time required to authenticate. This improvement brings DAGA close to the RSA Linkable Ring Signature (LRS) even with the most complex setup tested (32 servers, 32768 members). It can therefore be an interesting alternative to an RSA-based LRS authentication mechanism.

8 Improving the proof

8.1 OR-proof cost

The client's OR-proof of knowledge is described in Section 4.3.7 of [11].

To stay anonymous while authenticating, the client has to generate the following proof of knowledge:

$$PK\{(x_i, s) : \vee_{k=1}^{n} (X_k = x_k * G \wedge S_m = s * G \wedge T_0 = h_k^s)\}$$

This proof is understood as "I know the private key corresponding to the first public key and I did my computations correctly OR I know the private key corresponding to the second public key and I did my computations correctly OR ...” for each client’s public key in the group description.

More precisely, for each client i in the group (including himself), the requester generates three commitments $t_{i.0}$, $t_{i.10}$, $t_{i.11}$, a coefficient c_i and two responses $r_{i.0}$ and $r_{i.1}$. So the space requirement of this proof grow in $\mathcal{O}(6 * n)$ with n the number of clients in the group.

Moving from 2048-bit RSA to 256-bit elliptic curve should reduce the size of the proof to roughly an eighth of the previous size (although in this implementation it did not happen, as seen in Section 7) but did not change the growth rate. A part of this project was to look into accumulators and related cryptographic schemes to determine if they could be used as constant-size proofs. This work resulted in guidelines for further investigations and was not part of the resulting implementation.

8.2 Cryptographic tools

This section is based on exchanges with Kasra Edalatnejadkhamene, PhD student at the DEDIS lab.

The first tool is an accumulator. Accumulators (formally defined in [4] for instance) are schemes where multiple values are accumulated into a single one such that there is a proof that each value was correctly accumulated. Many of the available schemes are defined for RSA-based cryptography. The original definition has since be improved with dynamic abilities [2] (the ability to adds and remove elements without recomputing the accumulator from scratch) or even an universal dynamic scheme [6] where it is possible to have proof of non-membership (proving that a value was not accumulated).

The most promising scheme among them is the Nguyen scheme defined in ”Accumulators from Bilinear Pairings and Applications”[7] because it has the best compatibility with elliptic curves. However, some of its features need tweaking to be usable in DAGA:

- It needs a trusted setup $\rightarrow$ The setup can be distributed between the servers of DAGA.
- The accumulator is initialized with a bound to the maximum number of values it can accumulate. $\rightarrow$ We can ask the servers to increase the bound if needed.
- Its efficiency can be improved with a trusted authority. $\rightarrow$ This improvement should be assessed and, if significant, the trusted authority can be distributed on the servers.

The second tool is a signature of knowledge. A signature of knowledge is used to prove the knowledge of a private key corresponding to one public key among multiple ones. The simplest form is a ring signature defined in

[10]. Various shorter versions have been proposed [12] [1] and may be an interesting path to follow.

The main difficulty of a signature of knowledge is the choice of the message signed. It can either be given by the servers, in which case it must be selected using a process close to the choice of the random challenge for the client proof (see Section 6.3 for details), or be constructed from the client's request using a hash function as a random oracle. Either wayss, its randomness should be studied to prevent replay or impersonation attacks.

8.3 How it can help

In DAGA, the OR-proof can be decomposed in two distinct zero-knowledge proofs:

- the client knows a private key for a public key,
- this public key is a member of the group,

where the public key cannot be identified (or else it breaks the properties of DAGA). The combination of a signature of knowledge for the first part and an accumulator for the second one can result in a constant-size zero-knowledge client's proof.

The use of accumulators in the context to replace the list of client's public keys may also look interesting. However, the signature of knowledge with a ring signature still requires the explicit list of public keys to be computed.

9 Future work

The implementation done during this work is currently only available as a standalone library (see Annex 11.1 for installation instructions). The next step would be to integrate it with POP and Cothority, the distributed authority created by and used at DEDIS. The work would mainly consist into adapting the current implementation to satisfy the integration discussed in Section 5.4.

As seen in Section 7, the space improvements we expected from using elliptic curve did not occur. A first step would be to study more precisely which data consume space and how to reduce their size.

The work done to improve the proof has resulted into the list of requirements necessary to replace the current costly proof with shorter constant-size and constant-time cryptographic primitives. However, no schemes satisfying all these requirements were found before the end of the project. The follow-up would be either to keep looking and wait until such a scheme is proposed or to design such a scheme. In both cases, the scheme should be implemented in Go and integrated with DAGA.

Lastly, the current implementation is not resistant to side-channel attacks. The computations are not done in constant-time and the secret keys are not securely managed in memory nor securely erased from it. However, the Go tools and libraries currently available do not provide satisfying results to achieve these goals. The language may need to mature more before being used for cryptographic-based public applications.

10 Conclusion

This project shows that DAGA can be used to bring forward secrecy to the POP framework. The DAGA framework is now implemented in Go with all its core features and with the use of elliptic curve cryptography. There is still investigations to perform to improve the framework with the implementation of modern primitives such as accumulators and to make the implementation robust enough to be used in public applications.

11 Annex

11.1 Installation guide as a library

To use the library implemented during this project, simply import the package as follows:

```
import "github.com/dedis/student_17_pop_fs/daga"
```

and Go will download the package and its dependencies automatically at build time.

However, if the automatic installation is not available, you can manually download the package and its dependencies as follows:

```
go get "github.com/dedis/student_17_pop_fs/daga"
go get "gopkg.in/dedis/crypto.v0"
```

The package is then available using, for instance:

```
client, err := daga.CreateClient(index, s)
```

11.2 Installation guide to run benchmarks

The package daga comes with a second package called dagabenchmark. This second package provides a main function which runs the benchmark for configurable numbers of clients and servers.

If Go can automatically download the dependencies at build time, it can be installed and used with the following commands for Windows (similar commands exists for the other operating systems):

```
go get "github.com/dedis/student_17_pop_fs/dagabenchmark"
go build "./src/github.com/dedis/student_17_pop_fs/dagabenchmark"
./bin/dagabenchamrk.exe > benchmark.txt
```

However, if the automatic installation is not available, you can manually download the package and its dependencies and run it as follows on Windows (similar commands exists for the other operating systems):

```
go get "github.com/dedis/student_17_pop_fs/dagabenchmark"
go get "github.com/dedis/student_17_pop_fs/daga"
go get "gopkg.in/dedis/crypto.v0"
go build "./src/github.com/dedis/student_17_pop_fs/dagabenchmark"
./bin/dagabenchamrk.exe > benchmark.txt
```

It is highly recommended to pipe the output of the program to a file as done above to guarantee that no data is truncated when being displayed. In the default configuration, the benchmarks take a little more than one hour to run on the setup described in Section 7.

References

[1] Man Ho Au et al. "Short Linkable Ring Signatures Revisited". In: *Public Key Infrastructure*. Springer, 2006, pp. 101–115.

[2] Jan Camenisch and Anna Lysyanskaya. "Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials". In: *Advances in Cryptology — CRYPTO 2002*. Springer, 2002, pp. 61–76.

[3] *Deniable Anonymous Group Authentication as a service*. 2014. URL: `https://github.com/davidiw/dagas`.

[4] Nelly Fazio and Antonio Nicolosi. *Cryptographic Accumulators: Definitions, Constructions and Applications*. New York University. URL: `https://www.cs.stevens.edu/~nicolosi/tech-reports/FaNi03.pdf`.

[5] DEDIS Lab. *Proof of Personhood*. Oct. 16, 2017. URL: `https://github.com/dedis/cothority/blob/master/pop/README.md`.

[6] Atefeh Mashatan and Serge Vaudenay. *A Fully Dynamic Universal Accumulator*. Lasec, EPFL, Lausanne, Switzerland. URL: `https://infoscience.epfl.ch/record/188657`.

[7] Lan Nguyen. "Accumulators from Bilinear Pairings and Applications". In: *Topics in Cryptology – CT-RSA 2005*. Springer, 2005, pp. 275–292.

[8] NSA. *The Case for Elliptic Curve Cryptography*. Jan. 15, 2009. URL: `https://web.archive.org/web/20150627183730/https://www.nsa.gov/business/programs/elliptic_curve.shtml`.

[9] *PriFi: Low-Latency Tracking-Resistant Mobile Computing*. 2016. URL: `https://github.com/lbarman/prifi/tree/daga/auth/daga`.

[10] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret". In: *Advances in Cryptology — ASIACRYPT 2001*. Springer, 2001, pp. 552–565.

[11] Ewa Syta. "Identity Management through Privacy-Preserving Authentication". PhD thesis. Yale University, 2015. Chap. 4. URL: `http://ewa.syta.us/dissertation-final.pdf`.

[12] Patrick P. Tsang and Victor K. Wei. "Short Linkable Ring Signatures for E-Voting, E-Cash and Attestation". In: *Information Security Practice and Experience*. Springer, 2005, pp. 48–60.

[13] Arthur Villard. *Deniable Anonymous Group Authentication*. 2017. URL: `https://github.com/dedis/student_17_pop_fs`.

[14] Wikipedia. *Zero-knowledge proof*. Jan. 14, 2018. URL: `https://en.wikipedia.org/w/index.php?title=Zero-knowledge_proof&oldid=820448715`.