# ALINE - Attestation that Online Content Existed

Charline Montial

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

Fall 2019

**Responsible**
Prof. Bryan Ford
EPFL / DEDIS

**Supervisor**
Noémien Kocher
EPFL / DEDIS

# Contents

# 1 Introduction

## 1.1 Motivation

The Web is a rich source of information but is not a stable one, unfortunately. Indeed, one might read a controversial blog post, make a case against it to finally see its original content updated or removed with no way to prove what has been originally posted. This is only one of the many cases where having a proof that some online content was once available at some point would be a great help for users. We can think of journalists who could then cite online articles, or online victims who could prove they have been harassed, even if the actual content has been removed. The aim of this project is to propose a browser plugin that allows users to certify the content of some web pages at anytime.

## 1.2 Context

My work is in line with the decentralized system using "cothority" (collective authority), a framework to develop, analyze and deploy decentralized distributed protocols. This system is developed by the DEDIS laboratory. The entities running the protocol in a distributed way are called "cothority servers" or "conodes".

### 1.2.1 Blockchain

A blockchain can be seen as a linked-list with specific properties. It can also be a double linked-list. In this second case, we are talking about skipchains. Blockchains are constituted of blocks that contain digital pieces of information and can be perceived as a "distributed" database of records of all events that happened and were shared among participating parties - each block being one of such event. All network participants, which are conodes in our case, must agree on events that are added to the chain. This is achieved through the use of consensus algorithms - in this case, proof-of-consensus.

### 1.2.2 Byzcoin

I worked with "Byzcoin", which implements the distributed ledger called OmniLedger [5]. OmniLedger is a skipchain whose purpose is to preserve security under permissionless operations. Moreover, it is also designed to scale-out without compromising the security.

### 1.2.3 Smart contracts

In my project, I also have to implement a new smart contract. A smart contract is a digital protocol that guarantees the automatic execution of a contract when the agreed conditions are met. They allow agreements to

be processed without the need for a third party to verify or enforce them. Moreover, they are traceable and irreversible. In the context of this project, the smart contract will certify that the content of some web page existed at some time t. When such a contract is created to certify the content of a website, we call it an instance, and its corresponding instance ID will uniquely identify it.

## 1.3 Goal

This project consisted of, first, implementing a smart contract that would certify the content of a web page with the aid of a decentralized, proof-of-consensus, and blockchain-based ledger. Then, I also had to implement a browser plugin that would allow users to use this smart contract in an intuitive and easy way. The plugin aims to provide a way for internet users to prove the existence of some content on a web page at some time. Figure 1 shows how it works. When Alice is consulting the web page she wants to certify, she can use the plugin ALINE on her laptop that will connects to the cothority and each node will get the content and if they reach a consensus, the smart contract will be spawned and its instance ID returned to Alice.

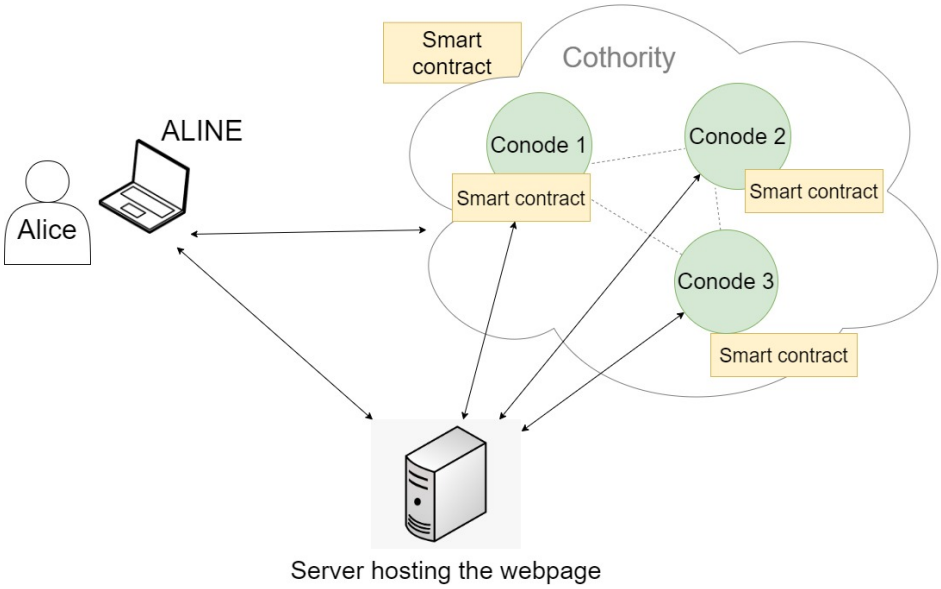


Figure 1

# 2 Implementation

## 2.1 Smart contract

The smart contract that I have implemented is written in Go. I coded it using the provided template of cothority, which is a generic smart contract that takes key-value pairs. My smart contract, that has been adapted to the situation, takes as inputs the URL of a web page and a CSS selector that can specify the page content that has to be certified (for example a specific paragraph of an article). The selected content must be deterministic, since each node will need to see the same content in order to reach a consensus. Thus, dynamic pages are excluded. The user can also specify if he wants the full HTML code or just the text of the chosen web page. Once the above specifications chosen, the smart contract saves the hash of the extracted content, the actual content, the URL, the selector, the current date and if the content consists of HTML code or if it is just the text on the skipchain.

The API I had to provide for this contract is constituted of three methods, namely `spawn`, `invoke` and `delete`. To spawn an instance of a smart contract means to create one. The method `invoke` is useful to update an already existing smart contract instance. I thought it was not relevant in this situation, as the goal of this project is to conserve the content of a web page at some point in time, and there is no need to further update the content of an instance. If a user wants to certify the same web page later, spawning another instance seems more logical to me, because even if the two instances refer to the same website, the content may not be the same. Thus, both of these instances are useful. This is why I have not implemented this method. Finally, a smart contract instance can also be deleted. However, even though this method is implemented, it is not present in the plugin because it not a feature that is particularly relevant.

In order to extract only some specific section of a web page, CSS selectors have been used to indicate what to save on the skipchain. To use the passed selector, the library `goquery`[1] has been used.

The hash function chosen is BLAKE2b [2]. For the smart contract written in Go, the package `blake2b` [4] has been used and for the plugin (that is described in the next section), the library used is named `blakejs` [6]. Both libraries implemented the BLAKE2b hash algorithm defined by RFC 7693 [7]. Also, to randomize the hashes, namely to avoid that a same content leads to the same hash, I concatenated the creation date to the content and then hashed this. This could be improved since a same content that is certified on the same day will still lead to the same hash value, but this is a start.

## 2.2 Browser plugin

The browser plugin has been developed as a Chrome extension because I am familiar with Chromium since it is the browser I use. This extension is written in Typescript because the types are assisted, unlike in JavaScript. Webpack has been used as the bundler, and Babel as the ES6 transpiler. The file `webpack.config.js` at the root of the plugin is telling Webpack to start from the file `src/popup.ts` and look for TypeScript and JavaScript files. Typescript files are transpiled to ES6, while ES6 files are transpiled to ES5 with Babel. After this, everything is bundled in the `dist/bundle.min.js`. The plugin allows the user to invoke the smart contract described above when he is consulting a web page and wants to certify it. Its features are made to be easily found and understood as it can be seen in Figure 2.

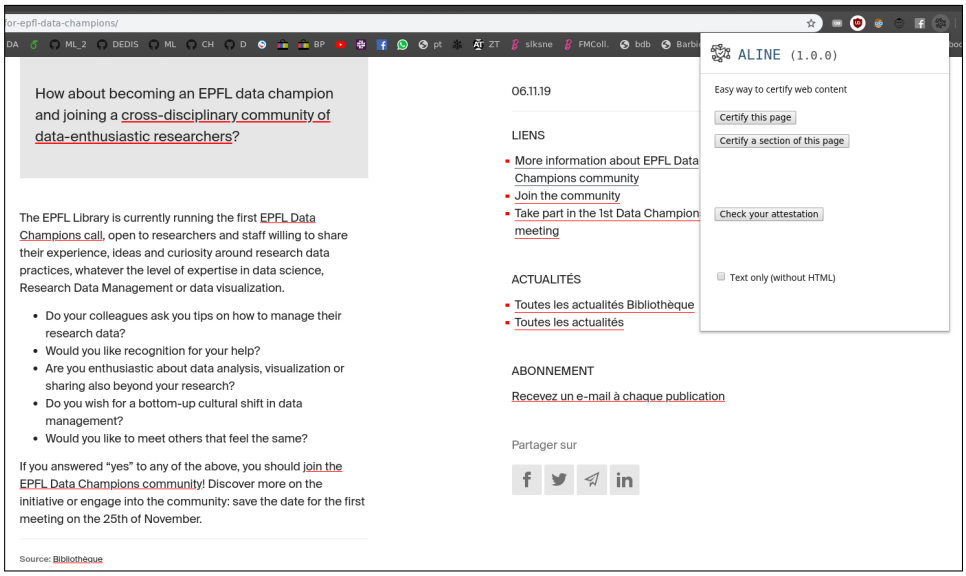


Figure 2: Aline Chrome extension

It connects to the cothority and sends the current URL with the CSS selector and the Boolean value of a check-box which indicates if the user wants the text only or the full HTML code. In Figure 3, we can see that the Chrome extension is currently creating the attestation.

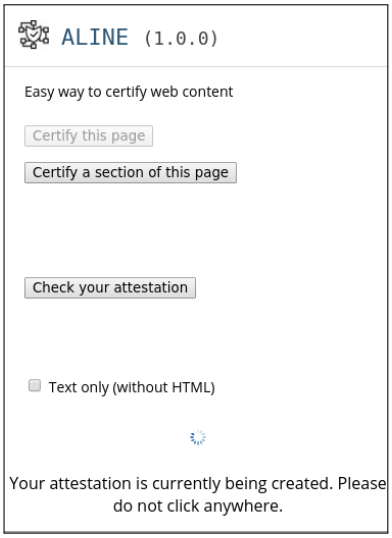


Figure 3: The attestation is currently being created

If the cothority reaches a consensus, the hash value and the other attributes are saved on the cothority ledger. If the user only wants to certify a section of the web page, he can interactively select it as it can be observable in Figure 4. In my opinion, this is an important feature to provide since the average type of user will not know anything about CSS selectors. Thus, allowing him to choose what to certify in an intuitive way seemed indispensable. To do so, I used a module called "Selector Gadget" [3] that can be called on the current web page we are consulting. To use it in the context of my Chrome extension, I had to code various scripts that would call Selector Gadget when it was relevant for the user. Once the page section chosen, the attestation can be generated. Figure 5 shows the screen the user sees when its attestation has been successfully created.

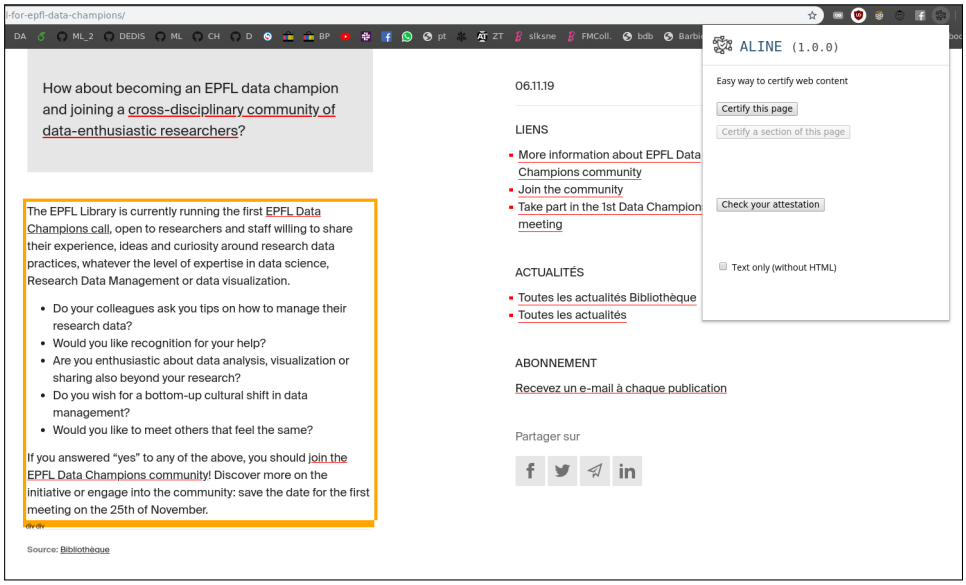


Figure 4: The user can interactively select only a section of the page if needed

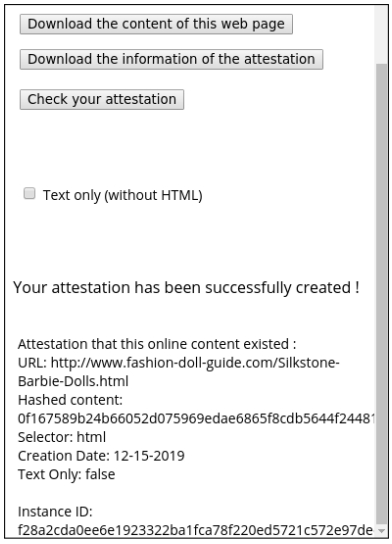


Figure 5: The attestation has been successfully created

Once the attestation has been created by spawning a new instance of the web page smart contract, the user can download the information of the attestation as well as the content of the certified web page.

The user can then provide the instance ID and the content to anyone and prove the existence of this content. The extension indeed allows users to check the content of an attestation as we can notice in Figure 7. By providing

the smart contract's instance ID of the certificate and its content, the plugin will hash the provided content and compare it with the one stored on the skipchain at the given instance ID. We can see that the plugin will output a green tick if the check was successful (please refer to Figure 8), or a red cross otherwise. This is useful because the user can send the content of the web page along with the ID and the person that needs to be convinced can use the plugin to check that it indeed holds. If the ID is not valid, ALINE let the user know it as shown in Figure 9. This procedure is illustrated in Figure 6.

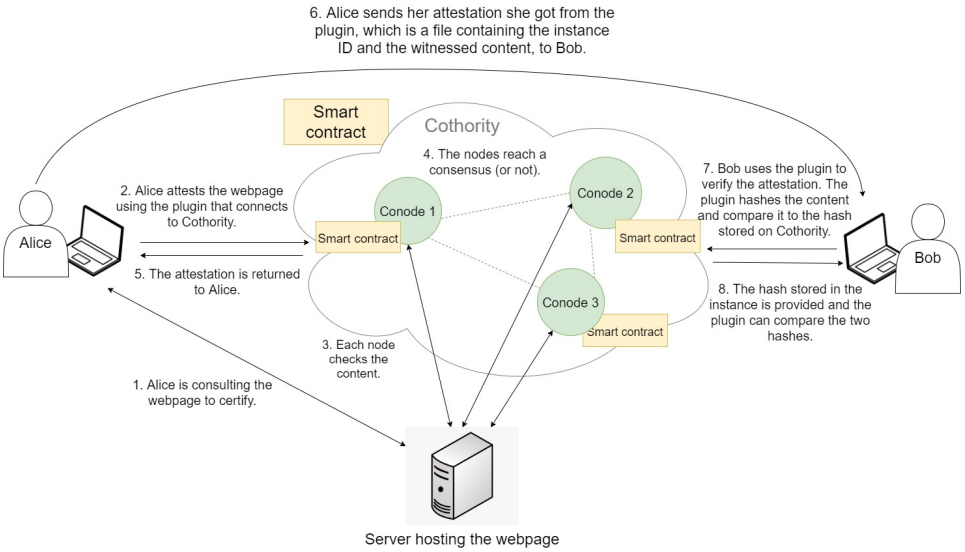


Figure 6


Figure 7: The content can be attested at any time after its creation

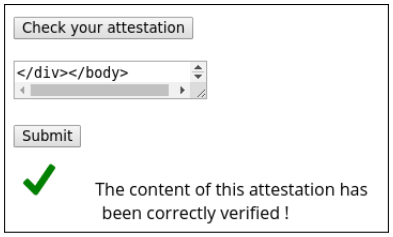


Figure 8: Here is an example of a content that has been verified


Figure 9: The user has not a valid ID and is informed of it

### 2.3 Problems encountered

Initially, the content of the web page was not supposed to be stored in the contract since the hash is already stored, as it is sufficient to check the certified content. However, to allow the user to save the content that has been certified, I had to extract the content again but from the plugin. It turned out that the text was not encoded and formatted in the same way as I extracted it in my smart contract written in Go. This is a problem since the user can upload the downloaded content to verify its attestation: this content will then be hashed and must exactly match the hash that is on the blockchain. This issue could have been resolved but due to time constraints and because this issue was not related to the core of the project, I decided to simply store the content of the web page in the attestation. I still compare the two hashes when a content is verified despite having it in the contract.

Another problem comes from the way of how Chrome Extensions are designed. When the user looses focus, the pop-up closes, which led to two problems: when the user clicks on a web page section he wants to certify, the pop-up closes. I found a workaround that informs the user that the pop-up is going to close and that he simply needs to click on it again after having clicked on the desired page section. When reopened, in these cases, ALINE knows that the user is waiting for the attestation to be created and acts consequently: when the extension is opened, a background script checks if the selector has been stored in a `<div>` on the website made for this purpose. If it is found, the script informs the pop-up that the user reloaded the app to certify a section of the current web page and spawns the contract. The other situation where this automatic pop-up close was problematic was when the user wants to verify a content he had attested; at first, I wanted to have a field for the instance ID, and another one for the content. Since theses are typically values you copy and paste, when the user had to go back to its downloaded files to copy the second field, the pop-up would close automatically. The solution was simply to have one single field with both the instance ID of the attestation and the content, and parse it afterwards.

More generally, developing a Chrome extension was more complicated than expected. Indeed, for security reasons, the development environment is very restricted for such plugins. To provide another example, allowing the external module "Selector Gadget" to be called and executed from the plugin needed a few permissions and other options to be set in the correct way to the `manifest.json file`, which defines the permissions and the side-scripts used.

# 3 Potential improvements

Due to the problem described above, the content had to be saved on the skip chain along with its hash. What could be implemented in case it would be relevant to keep the certified content confidential would be to save it encrypted with a key only known from the user that would be randomly generated when the attestation is created.

At the moment, ALINE works using a local skipchain that I have to launch when I want to use the extension. A feature could allow the user to choose which remote blockchain or skipchain to use.

I also wish I had the time to add a historical of the attestations the user generated to have them on hand without needing to retrieve the files that may have been inadvertently deleted or lost. This historical would show the URLs and the ID of the attestation and would let the user download the content and its information whenever he needs it.

To be able to run, "Selector Gadget", that lets the user select a section of the page interactively, modifies the HTML code (it adds some HTML code to interact the page and temporarily save the CSS selector). But some web pages do not allow it for security reasons (I presume to avoid attacks such as cross-site scripting, injection attacks, and many others). Thus, for example, ALINE does not work on Twitter, which I find regrettable. I wish I had time to find a way to fix this. The user can however still use ALINE for such pages by certifying the whole content since, in this case, "Selector Gadget" is not used.

Also, since the nodes much reach a consensus, the content must be the same everywhere, which excludes dynamic pages. Being able to attest their content, by either removing the dynamic elements if they are not relevant or by some other measure would be a useful improvement.

Finally, working on the portability of ALINE to be able to use it outside the Chrome environment could be interesting.

# 4 Conclusion

This project, consisting of two parts, namely implementing the smart contract and the Chrome extension, has been completed in the required time. This project was very enriching because it was my first application using

blockchains and smart contracts which I knew, but only in theory. Moreover, I had never coded in Go nor in Javascript before and to be able to implement the smart contract, I had to understand and familiarize with the cothority template provided, which was also a challenge. It was also my first plugin. Being a daily Chromium user, for me, it was a natural choice to code a Chrome extension, but now I would have thought it over and compared the available API with other browsers such as Firefox for example to make the best possible choice. Overall, I am satisfied with how this project turned out because in the end, even though some aspects could be improved, I ended up with a usable and working Chrome extension and with a practical and concrete experience of blockchains.

## References


[1] Martin Angers. *GoDoc: package goquery*. URL: https://godoc.org/github.com/PuerkitoBio/goquery#readme.

[2] Jean-Philipe Aumasson et al. *BLAKE2: simpler, smaller, fast as MD5*. URL: https://eprint.iacr.org/2013/322.pdf.

[3] Andrew Cantino. *SelectorGadget: point and click CSS selectors*. URL: https://selectorgadget.com/.

[4] *GoDoc: package blake2b*. URL: https://godoc.org/golang.org/x/crypto/blake2b.

[5] Eleftherios Kokoris-Kogias et al. *OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding*. URL: https://eprint.iacr.org/2017/406.pdf.

[6] Daniel Clemens Posch. *SelectorGadget: point and click CSS selectors*. URL: https://github.com/dcposch/blakejs#readme.

[7] Markku-Juhani Saarinen. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. URL: https://tools.ietf.org/html/rfc7693.