# Self-Sovereign Identity for the Financial Sector: A Case Study of PayString Service

*Abstract*—**PayString is an initiative to make payment identifiers global and human-readable, facilitating the exchange of payment information. However, the reference implementation lacks privacy and security features, making it possible for anyone to access the payment information as long as the PayString identifier is known. Also this paper presents the first performance evaluation of PayString. Via a large-scale testbed our experimental results show an overhead which, given the privacy and security advantages offered, is acceptable in practice, thus making the proposed solution feasible.**

*Index Terms*—**PayString, DID, Verifiable Credentials, Self-sovereign Identities**

## I. Introduction

The latest technological advancements have revolutionized many aspects of everyday life: from mobility through health-care and to communications. However, when we take a look at the banking industry there is still much room for improvement and innovation, especially when we talk about international transactions. While for example, the email has slashed the time necessary to send and receive a letter from days to seconds, the same is not yet valid for international payments, which are still processed according to old, traditional approaches and technologies. International payment systems are siloed and disconnected; between some systems, payment is complicated or not even possible. Transfers often take days to be fulfilled, without direct feedback from the receiver to the sender, and oftentimes incurring high fees. Moreover, while the underlying communication technology is complex, to send an email one only needs the recipient's email address, which follows an easy-to-remember and easy-to-share format: *alice@example.com*. The same is not (yet) true for financial transactions, either in fiat or cryptocurrencies, where one needs to manage and share multiple accounts expressed as long and complex strings of alphanumeric characters.

Although several *Distributed Ledger Technologies* (DLT)[1] seek to address aspects of inter-connectivity, payment speed, fees and feedback, to our knowledge the same is not true concerning the ease and freedom of usage for the end-user, when dealing with complex payment end-point descriptors, and especially when the banking systems and the bank account formats are different, which is often the case for international payments. *PayString* aims to close this gap but there is still room for improvement concerning the security and privacy of user's data.

[1]Like, for example, *Ripple*, *Quorum* (JP Morgan), *Open Chain* (Ant Financial), *Corda*, *Stellar*, *Ethereum*, *Hyperledger Fabric* and projects like *Interledger* (Ripple), *Stella* (EU and Japan), *Jasper* (Canada), *Ubin* (Singapore) and more

Our contribution is threefold: (1) Enrich PayString with privacy and security features (2) Evaluate the reference implementation of PayString and the proposed solution via a comprehensive load testing (3) A working prototype that demonstrates the practical applicability of our solution.

The rest of the text is organized as follows: Section II describes the required background information. In Section III we explain our solution, *"PayString Secure"*, followed by a prototype implementation. Extensive evaluation experiments are presented and discussed in Section IV. Section V places our work in the context of literature work. Finally, we draw our conclusion and discuss the future work in Section VI.

## II. Background

### A. PayString

*PayString* is a web-based protocol designed to facilitate the exchange of payment information. PayString addresses look like an email address but with a dollar symbol "$" instead of the "@" symbol - for instance: *alice$example.com.* The aim is to replace complicated bank account numbers and cryptocurrency wallet addresses with an easy to memorize identifier. As such, users can use a single address linked to several bank accounts or wallet addresses. The resolution of a given PayString address occurs in a dedicated server, similar to how DNS resolves domain names to infer the corresponding IP addresses.

The design principles of PayString are simplicity, neutrality, decentralization, extensibility, sovereignty and composability with existing standards and namespaces. Fulfilling the first principle, the PayString protocol is built upon HTTP and DNS, comprising a request/response application-layer protocol [1]. Neutrality makes PayString a currency agnostic protocol, being able to work both with crypto and fiat currencies. The decentralization allows the protocol to work without a central authority over the web, dismissing the need to comply with different standards and jurisdictions. Extensibility is about making PayString open and upgradable. Sovereignty concerns PayString's ability to put the control in the user's hands, such that users own and control their identity and data. Being highly abstract, PayString can wrap existing namespaces and standards, allowing the protocol to be composable.

*1) Use Case:* The operation of PayString is quite simple. Figure 1 shows a use case in which Alice wishes to transfer money to Bob. Bob already registered his payment information on the PayString Server, so he only needs to send his easy-to-remember PayString address (*bob$example.com*) to Alice, that will request the payment information to the server. The server

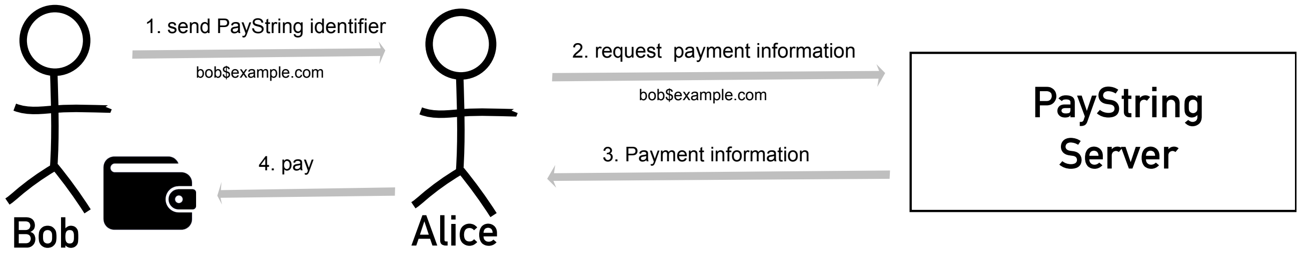


Fig. 1: **PayString Use Case.** Alice wishes to transfer money to Bob. Bob sends his PayString address to Alice, that requests Bob's payment information from the PayString Server

will then send the payment information: a pointer to Bob's wallet. Alice will use this pointer to transfer money from her wallet to Bob's wallet.

*2) Security and Privacy:* An important consideration to make before discussing security and privacy matters is that, while the PayString protocol works between an entity and a server, for a payment to occur successfully, we need the participation of a payer, a payee, and the wallets of both entities. The entity needs to trust it's wallet since the PayString protocol cannot prevent byzantine behavior from the wallet's part [2]. Remembering that in this scenario a "wallet" is not necessarily a cryptocurrency wallet and can be, as an example, a bank account.

There is, however, some risk attached to the PayString Server, being it an always-online system: an attacker could hijack or impersonate the server, sending different payment information for a given pointer to deviate the funds to the attacker's wallet. The impersonation attack is considered to be of high risk when the keys used by the PayString Server to sign the payment information are compromised. There is less risk if only the keys used for establishing secure channels are endangered.

Besides the server, an attacker may try to impersonate an entity, registering different payment information in the name of an entity that still did not register its true information. The attacker may also change the information of an already existing entity's pointer if the keys got compromised.

As a privacy consideration, PayString allows the entity to choose to keep its payment information public or private, but without granularity. Also, there is not yet any implemented method to verify the requester's identity. In this context, there is no strong guarantee of privacy on the PayString protocol. We discuss the subject in Section III.

### B. Self-sovereign Identities

*Self-sovereign identities* (SSIs) are the last generation of identity proof methods on the Internet. Different from previous models, it gives users total control over their identity and data, while trusting a third-party solely to verify the authenticity of a given identity [3].

The Internet itself was never designed to provide means of managing and authenticating the identities of the users. Very soon the authentication of users became an issue, with the first straightforward solution being the institution of user credentials: username and password. Each service provider then needs a database to authenticate the credentials of the users and to store all the data necessary at the moment and possibly necessary in the future.

The disadvantages of this method are easily spotted: for each service the user needs an identity, being necessary to remember credentials for different services, and to replicate the same information across different providers. Besides that, it is also necessary to trust all these entities, even when they are susceptible to involuntary data leakages.

The next generation of identity authentication came to facilitate the managing of multiple credentials for a single user and is called *Federation*. A single entity is responsible for managing the user's credentials and data; the service providers then refer to this authority to authenticate and provide the data when needed. The user has partial control over which data is shared with whom. Although better than the previous one, this model has the obvious problem of trusting in a central authority that may, for any reason, stop providing service for unique or multiple users. In this way, the user may lose his identity and all of his data if the authority decides so or if the authority gets compromised.

Self-sovereign identities turn the identity providers into *trust providers*. The user is the sole provider and authenticator of their identity and the *trust providers* are the entities that provide trustfulness for these identities i.e. : they attest to the veracity of the identity.

This new model requires the support of decentralized infrastructure, providing something like a new layer to the Internet. Decentralized ledgers are the perfect technology for this matter. With this concept in mind, the Sovrin Foundation created a permissioned blockchain called Sovrin to act as infrastructure for the identity layer [3].

### C. Distributed Identifiers and Verifiable Credentials

*Decentralized Identifiers* (DIDs) and *Verifiable Credentials* (VCs) are technical concepts that also play huge roles in the domain of identity management on distributed ledgers. Both are being developed as standards by the World Wide Web Consortium (W3C).

DIDs [4] are, in a simple way, unique identifiers for decentralized verifiable identities. Being decentralized, DIDs don't need any central authority to store, manage and validate the identities they represent. These identities can represent real people, organizations and abstract entities between others.

Each DID is represented by a URI associated with a *DID subject* and a *DID document*, where the *DID subject* is the represented entity, i.e. the identity owner. The *DID document* describes the *subject*, holding public keys, biometrics, and any other mechanism used for authenticating the ownership of the *subject* over the DID.

The use of credentials is a way of proving ownership, permissions, concessions, accordances, and even identities. *Verifiable Credentials* [5] are those whose authenticity can be verified by trustable authorities. As an example, we can cite university degrees, which are credentials used to assert the level of education of an individual. It is usually hard to express these VCs digitally while preserving people's privacy, considering that for the validation of a credential, holder, issuer, subject and verifier must be known. In this context, the pseudonymity provided by the blockchain is a powerful asset to be taken advantage of.

DIDs and VCs can be used as building blocks for identity management systems based on self-sovereign identities [6]. The DIDs present an easy way to have unique identifiers linked to a real-life identity, while the VCs can benefit from blockchains to securely and privately express the credentials needed for authentication.

## III. PayString Secure

PayString is a solution for simplifying payments in nowadays scenarios, where users need to deal with multiple payment methods, currencies, and identifiers. However, it does not integrate privacy by design [7]. Privacy by design is the concept of embedding privacy to the design and operations of IT systems, networks, infrastructure, and business practices. With this purpose in mind we enriched the protocol with some privacy and security features.

The first issue we tackled was the fact that anyone could access the payment information of a given user by knowing their PayString. Let us consider a user called Bob that stores all his payment information on a PayString Server, including bank account and XRP wallet addresses. It is not of Bob's interest to publicly disclose the bank on which he holds an account and the fact that he owns an XRP wallet.

Bob needs to receive some payment from his friend, Alice. Bob then sends his PayString to Alice via unencrypted e-mail. Somehow an anonymous and malicious third party intercepts this e-mail and gets access to Bob's payment information. This third party now knows the bank where Bob holds his account and sends several phishing e-mails to capture Bob's bank account credentials.

The immediate answer to this problem is to implement an *Access Control List* (ACL), allowing Bob to grant and deny access to his payment information. Now, considering that Bob wishes to receive only payments on fiat currency from Alice, there is no need to give her access to his XRP wallet address. Looking through this perspective, this ACL must permit granular control to the payment information.

With the ACL integration to the PayString Server, we solve the primary privacy issue encountered on the protocol. But how do we verify the identities of the users that request access to another user's payment information? In other words, how can we assure that the requester is indeed Alice and not the third malicious user?

The answer to these questions is pretty straightforward: we can verify identities using asymmetric cryptography. So Alice would need to send her public key to Bob, which would include this key in his ACL. When requesting the payment information, Alice would need to sign the request using her private key.

This solution works well but makes us fall again into the issue we are trying to avoid: the non-human-readable identifiers. To address this problem we decided to use SSIs in the form of DIDs and VCs taking advantage of HyperLedger Indy tools and capabilities.

By using DIDs and VCs, the Blockchain serves as a single source of truth, enabling transparency, security and trust, which is the base of any identity system. So Alice is now able to prove her identity to the PayString Server using a DID Verifiable Credential instead of a long sequence of random characters, facilitating the entire process of payment.

### A. Hyperledger Indy

*Hyperledger Indy* is a distributed ledger that provides tools and libraries for supporting the development and integration of solutions based on distributed identities. The purpose of Hyperledger Indy is to provide a platform for the creation of identity solutions across different domains. The Sovrin Foundation contributed initially to the project by open-sourcing the code used to build the Sovrin Network [8].

Over the distributed ledger, there are built-in capabilities that implement Decentralized Identifiers (DID) without the need for a central authority. Any two entities can have a 1:1 secure relation. Hyperledger Indy also uses zero-knowledge proofs to authenticate claims without disclosing any additional information about the identity owner.

Indy makes use of *Hyperledger Aries*, a toolkit that provides secure management for creating, transmitting, and storing verifiable digital credentials. Aries consumes the shared cryptographic library *Hyperledger Ursa*. In this way, Indy, Aries, and Ursa work together to provide a secure fabric on which solutions based on self-sovereign identities can be built.

### B. Architecture

Hyperledger Indy serves as the foundation layer on which PayString Secure is built, as shown in Figure 2. On top of that, we have an extended PayString Server with two new modules, the ACL - explained above - and the *Credential Manager*, which stores and verifies the credentials.

We also built the *PayString Digital Notary*, which is the entity that issues the VCs. When Alice needs to prove her identity, she must request a credential to the Notary. The Notary will issue the VC, which will be registered on the Blockchain and saved into Alice's wallet. Having this credential, Alice

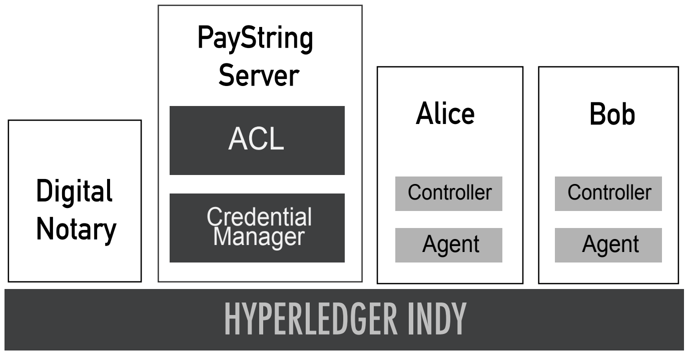


Fig. 2: **PayString Secure Architecture.** We built a Digital Notary to issue credentials and extended the PayString server with two modules: the ACL and the Credential Manager

becomes able to prove to the PayString Server that she is indeed who she says she is.

To connect to the Server and the Notary, both Alice and Bob need *agents* which will manage the connections between them and the entities, and will keep a record of the credentials issued. On top of these agents, we have the so-called *controllers*, that provide the business logic to the agents, facilitating the entire process from the user's perspective.

### C. Solution Workflow

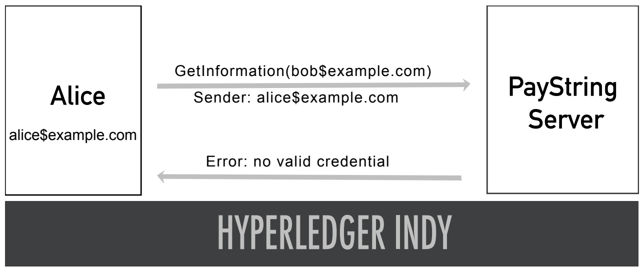

Alice
alice$example.com
GetInformation(bob$example.com)
Sender: alice$example.com
Error: no valid credential
PayString Server
HYPERLEDGER INDY

Fig. 3: Workflow for the case where Alice doesn't have a credential

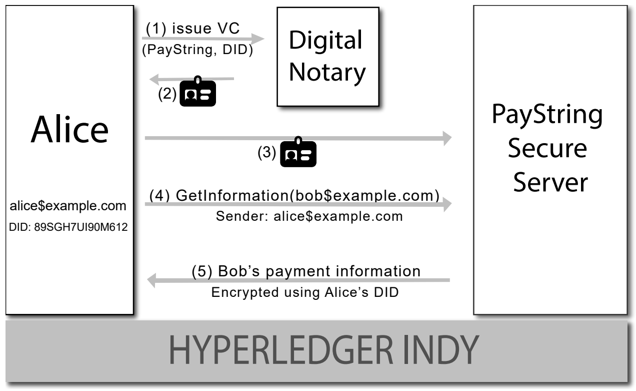


Fig. 4: Complete workflow involving the Notary, PayString server and Alice

Figures 3 and 4 show an example of the interaction between the modules and entities using PayString Secure, to safely exchange payment information between two users: Alice and Bob. Alice knows Bob's PayString identifier, (*bob$example.com*), and wishes to make a payment to Bob using fiat currency. She sends a request to the PayString Server, identifying herself using her PayString identifier (*alice$example.com*) and asking for Bob's payment information. The Credential Manager of the Server looks for an existing credential for Alice's PayString identifier; if none is found, the Server will reject the request (Figure 3). To fix this, Alice needs to ask the Notary for a new credential (Figure 4); for that, she needs to provide her PayString and DID identifiers. The Server will issue the credential and send it to Alice's wallet, where the credential will be stored.

Possessing the credentials, Alice can now make a valid request. But first, she needs to present her credentials to the PayString Server, which will use the Credential Manager to store them. Afterwards, she will send a new request for Bob's payment information, the same way she did before. But this time the Server will be able to validate the request and will then check the ACL to see if Alice is authorized to see the information. If she is, the Server will encrypt Bob's payment information using Alice's DID (provided by the credential) and send her the encrypted message that she shall be able to decrypt using her private key.

### D. Prototype Implementation

A prototype of PayString Secure was built for the PayString Block-Sprint Hackathon. The prototype consists of four entities: Alice, Bob, the PayString Secure server and the Digital Notary. Even with Bob's role in the use case, he has no controller and agent on the prototype. However, his payment information is already stored on the PayString Server, which has also a rule in its ACL authorizing Alice to retrieve Bob's payment information.

Alice, the PayString Server and the Digital Notary have agents and controllers, so the user can interact with any of them. The PayString Server has also the Credential Manager and the ACL. The Digital Notary issues credentials to Alice; these credentials need to be sent to the PayString Server for releasing Bob's payment information. This way, the prototype implements the use case described in Section III, the only extra step added being that Alice needs to establish a secure connection with the Digital Notary before asking for her credential. This step is necessary to guarantee a secure connection and is implemented by Hyperledger Indy itself. The same secure connection is also used between Alice and the PayString server.

## IV. Performance Evaluation

PayString is a quite recent technology; based on the best of our knowledge the performance of the PayString server has not yet been evaluated. That means we don't have a baseline to assess the overhead of the novel added features. This section first evaluates the reference implementation of PayString[2], and then evaluates the performance of our enhanced version.

[2]https://github.com/PayString/paystring

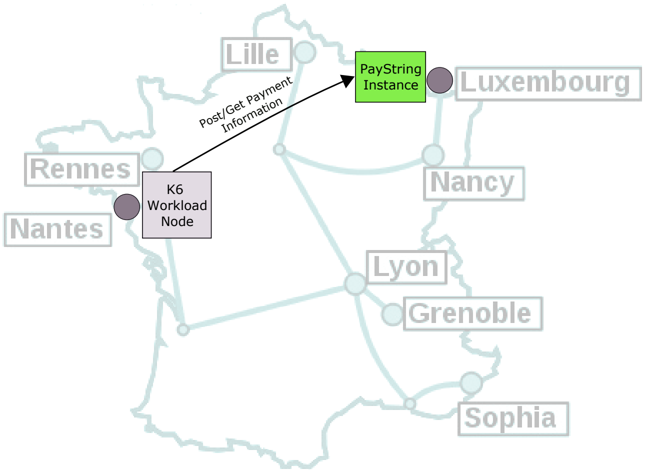


Fig. 5: Grid'5000 Evaluation Testbed

### A. *Experimental Environment*

In this work, we use the Grid'5000[3](g5k) platform for testing. The g5k is a large-scale testbed environment for experiment-driven research. A large number of resources are scattered over many geographic locations in France and Luxembourg. We selected two locations that are geographically separated but they still well-connected thanks to g5k infrastructure. As shown in Figure 5 the first node is located in Luxembourg and runs an instance of PayString server, while the second node, located in Nantes (France), runs the workload generator script. Nodes specifications are described in Table I.

### B. *Testing Tool and Applied Tests*

Load testing is the standard approach for assessing how software systems behave under load. We aim to discover load-related problems such as performance (e.g., memory leak) or functional problems (e.g., buffer overflow) that could be discovered only when the PayString server is under load [9].

In this work, the k6[4] API testing tool is used to automate, scale and reproduce the load tests. The k6 tests are based on the Virtual Users (VUs) concept to create workloads for targeted applications. The expected behavior of application users is encoded in JavaScript script files. During the load test, VUs execute the test script in parallel. The k6 test script has two main functions emulating user behavior for registering and retrieving payment information.

We applied four types of load tests to give different insights about the PayString server. The selected tests are as follows:

- *Smoke Test* is configured with one VU to get the payment information over 60 seconds. This minimal load for the PayString server is used to assess the readiness of the server for the next tests.
- *Load test* evaluates PayString server under typical and peak load. The test gradually increases the number of VU requests to 100 in 5 minutes; this rate will last for 10 minutes and then ramp-down to 0 requests.
- *Stress Load* is used to evaluate the stability of PayString server under heavy load, so the workload will be sent over multiple stages: below normal load, normal load, around the breaking point, beyond the breaking point and, finally, scale down.
- *Spike Test* is a kind of stress test, but it does not gradually increase the load, instead, it spikes to extreme load over a very short window of time. Therefore we can determine how the PayString server will perform under a sudden surge of VU requests. So, we start the test with a normal load (i.e 100 VU requests over 1 minute) then we spike to 1,400 VU requests for 3 minutes.

[3]www.grid5000.fr

[4]https://k6.io

To account for the problems that may arise due to network connections, we run the tests 5 times. For comparison between the two instances, we record the request-response time, which is defined as time spent waiting for a response from the remote host. The average, median, maximum, 90th percentile and 95th percentile are calculated. We take into account that during the tests the PayString server may respond slowly or very fast. So if we used only the average or median metrics, these fast and slow responses may be lost as they were observed less frequently. To have a closer view of the behavior of the PayString server response times in different percentiles we analyze *p(90)* and *p(95)*. *p(95)* is the maximum response time to respond to 95% of all requests [10].

In the next sections, we present the evaluation results of the basic and the proposed PayString Secure version.

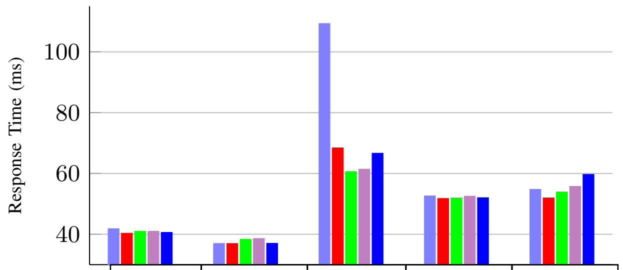
### C. *Performance of PayString Reference Implementation*

For this experiment, a dockerized version of PayString has been configured in a node and the k6 tests scripts in another node. Figure 6 shows the results of the smoke test, where 1 VU keeps sending *get payment information* requests for a one-minute time window. The results show that over 56 requests the average response time is 40ms with a maximum of 73ms. Looking for the *p(95)* we found that 95% of *get payment information* requests stay in the range of 54ms-59ms. Figure 7 presents the load test results. The test increases gradually the VUs to 100 over 5 minutes, then fix VUs for another 10 minutes. In total, we observed 111,100 requests sent to the server with a 100% success ratio. The observed response time results show that, on average, the request takes 39ms and 95% of the requests need 55ms.

The stress test is presented in Figure 8. In the first 7 minutes, 100 VUs are created then doubled in the next 7 minutes. We assume that the breaking point of the server is 400 VUs, so we keep this ratio for another 7 minutes. In total, we send 523,590 requests in a 38 minutes time window. The average time response is 825ms, while the maximum is 2,959ms. As 95% response times are calculated, it has been noticed that the response time varies between 1,334ms-2,730ms during the 5 trials. Therefore, we conclude that the PayString server is stable under heavy load.

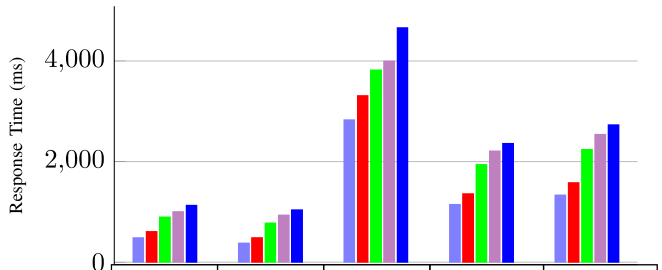
| Location | CPU | Cores | RAM | Network | OS | Disk |
|---|---|---|---|---|---|---|
| Nantes | 2x Intel Xeon ES5-2660 | 8 cores/CPU | 64 GiB | 10 Gbps | Ubuntu 18.04 | 2.0 TB HDD |
| Luxembourg | 2x Intel Xeon E5-2630L | 6 cores/CPU | 32 GiB | 2x 10 Gbps | Ubuntu 18.04 | 250 GB HDD |

TABLE I: G5k testing nodes specifications



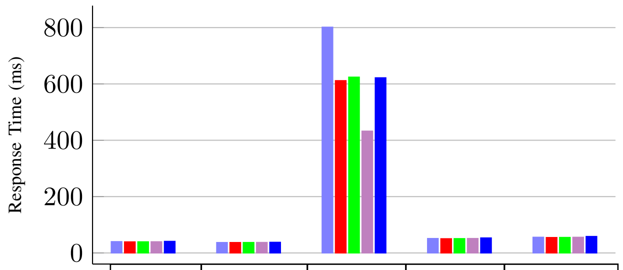
| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 41.78 | 36.91 | 109.25 | 52.56 | 54.71 |
| 🟥 Test2 | 40.27 | 36.89 | 68.37 | 51.71 | 51.90 |
| 🟩 Test3 | 40.92 | 38.27 | 60.56 | 51.89 | 53.86 |
| 🟪 Test4 | 40.94 | 38.54 | 61.35 | 52.46 | 55.69 |
| 🟦 Test5 | 40.56 | 36.96 | 66.59 | 51.96 | 59.62 |
| **Average** | **40** ms | **37** ms | **73** ms | **52** ms | **55** ms |

Fig. 6: Response time (k6 Smoke Test) for the basic PayString



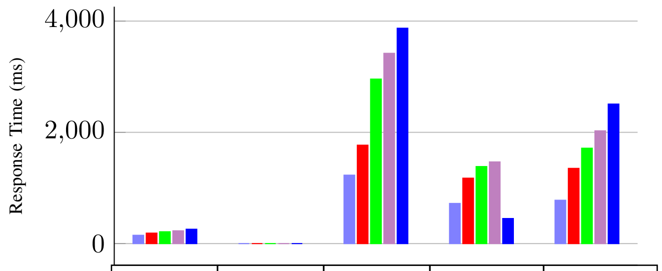
| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 486.13 | 379.62 | 2,829.63 | 1,147.63 | 1,334.47 |
| 🟥 Test2 | 608.7 | 486.74 | 3,310 | 1,360 | 1,580 |
| 🟩 Test3 | 896.78 | 777.83 | 3,820 | 1,940 | 2,240 |
| 🟪 Test4 | 1,004 | 935.66 | 4,000 | 2,210 | 2,540 |
| 🟦 Test5 | 1,130 | 1,040 | 4,660 | 2,360 | 2,730 |
| **Average** | **825** ms | **723** ms | **2,959** ms | **1,803** ms | **2,084** ms |

Fig. 8: Response time (k6 Stress Test) for the basic PayString



| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 40.16 | 37.06 | 801.41 | 51.28 | 55.76 |
| 🟥 Test2 | 39.33 | 37.03 | 611.62 | 50.48 | 54.77 |
| 🟩 Test3 | 39.55 | 37.16 | 624.09 | 50.84 | 55.09 |
| 🟪 Test4 | 39.7 | 37.31 | 432.41 | 51.3 | 55.67 |
| 🟦 Test5 | 41.14 | 37.82 | 621.78 | 53.31 | 58.53 |
| **Average** | **39** ms | **37** ms | **618** ms | **51** ms | **55** ms |

Fig. 7: Response time (k6 Load Test) for the basic PayString



| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 148.97 | 0 | 1,229.95 | 721.39 | 778.79 |
| 🟥 Test2 | 187.82 | 0 | 1768.76 | 1175.76 | 1351.93 |
| 🟩 Test3 | 212.38 | 0 | 2957.14 | 1384.60 | 1714.52 |
| 🟪 Test4 | 229.33 | 0 | 3420.47 | 1467.90 | 2026.78 |
| 🟦 Test5 | 257.62 | 0 | 3871.72 | 450.36 | 2508.02 |
| **Average** | **207** ms | **0** ms | **2,649** ms | **1,040** ms | **1,676** ms |

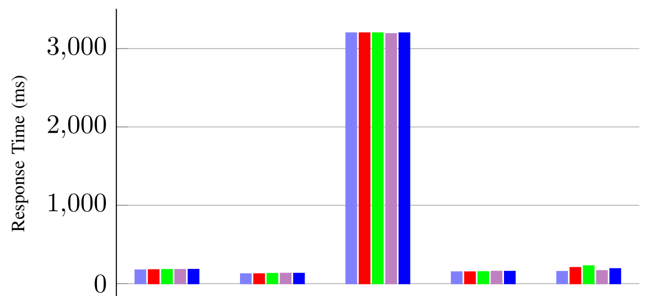
Fig. 9: Response time (k6 Spike Test) for the basic PayString

The spike test depicted in Figure 9 is the toughest one since it boosts massively the number of requests in a very short period. The k6 tool sent 387,220 requests in 7 minutes (compare to 38 minutes in the stress test). As a result, we have only a 26.18% success ratio, explaining the zero value of the median. The successful requests required 207ms on average and 2,649ms is the maximum observed response time.

### *D. Performance of PayString Secure*

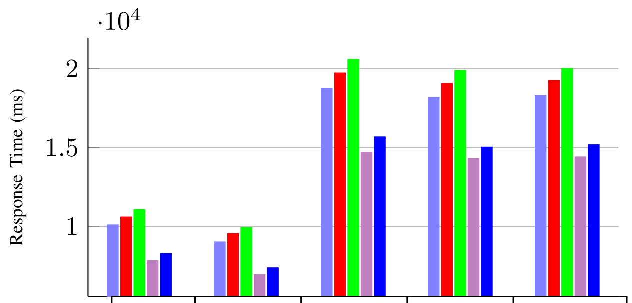
As mentioned before, PayString Secure comes with ACL and privacy features based on Hyperledger Indy and DID technology. The k6 tool scripts were extended with a setup stage where we first establish the secure connections between different components, issue the ownership credential and then present the credential to the PayString server. Thus in these experiments, we measure the response request time to cross-check the ACL and validate the claimed digital credential.

Figure 10 shows that for over 54 requests, on average, the response time is 178ms, while 95% of the requests consumes 188ms. Compared to the basic version of PayString, we noticed that the new features add an extra delay of 138ms and 133ms, respectively. The result of the load test is presented in Figure 11. With a total of 24,423 requests, the average time is 2,990ms, and 95% of the requests need 4,138ms. With a 100% success ratio, PayString Secure passes the stress test,



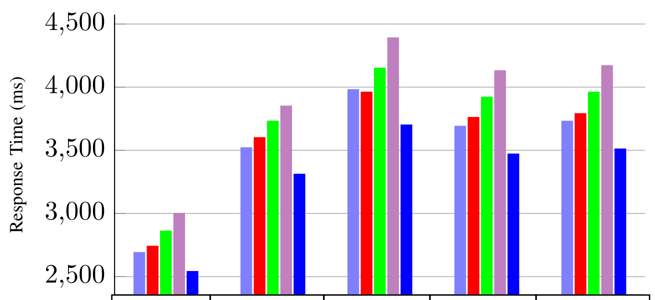
| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 173.8 | 125.37 | 3,200 | 150.32 | 154.65 |
| 🟥 Test2 | 176.96 | 124.78 | 3,200 | 149.18 | 204.9 |
| 🟩 Test3 | 179.7 | 129.52 | 3,200 | 152.09 | 227.19 |
| 🟪 Test4 | 179.66 | 132.02 | 3,190 | 157.37 | 165.38 |
| 🟦 Test5 | 180.89 | 131.08 | 3,200 | 156.22 | 190.28 |
| **Average** | **178** ms | **128** ms | **3,198** ms | **153** ms | **188** ms |

Fig. 10: Response time (k6 Smoke Test) for PayStringSecure



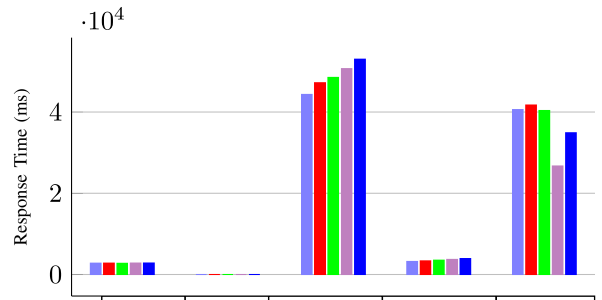
| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 10,090 | 9,010 | 18,750 | 18,160 | 18,290 |
| 🟥 Test2 | 10,590 | 9,540 | 19,720 | 19,060 | 19,240 |
| 🟩 Test3 | 11,060 | 9,920 | 20,580 | 19,880 | 20,000 |
| 🟪 Test4 | 7,820 | 6930 | 14,690 | 14,300 | 14.400 |
| 🟦 Test5 | 8,270 | 7370 | 15,670 | 150,20 | 15,170 |
| **Average** | **9,566** ms | **8,554** ms | **17,882** ms | **17,284** ms | **17,420** ms |

Fig. 12: Response time (k6 Stress Test) for PayStringSecure



| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 3,810 | 4,890 | 5,940 | 5,200 | 5,260 |
| 🟥 Test2 | 2,740 | 3,600 | 3,960 | 3,760 | 3,790 |
| 🟩 Test3 | 2,860 | 3,730 | 4,150 | 3,920 | 3,960 |
| 🟪 Test4 | 3,000 | 3,850 | 4,390 | 4,130 | 4,170 |
| 🟦 Test5 | 2,540 | 3,310 | 3,700 | 3,470 | 3,510 |
| **Average** | **2,990** ms | **3,876** ms | **4,428** ms | **4,096** ms | **4,138** ms |

Fig. 11: Response time (k6 Load Test) for PayStringSecure



| | avg | med | max | p(90) | p(95) |
|---|---|---|---|---|---|
| 🟦 Test1 | 2,796 | 0 | 44,333 | 3,215 | 40,605 |
| 🟥 Test2 | 2,805 | 0 | 47,214 | 3,359 | 41,725 |
| 🟩 Test3 | 2,759 | 0 | 48,523 | 3,531 | 40,377 |
| 🟪 Test4 | 2,820 | 0 | 50,690 | 3,720 | 26,720 |
| 🟦 Test5 | 2,830 | 0 | 53,010 | 3,930 | 34,900 |
| **Average** | **2,802** ms | **0** ms | **48,754** ms | **2,843** ms | **36,865** ms |

Fig. 13: Response time (k6 Spike Test) for PayStringSecure

where the average result is 9,566ms and the *p(95)* is 17,420ms. This means that under heavy load PayString Secure can still survive. Figure 13 shows the result of the spike test, where our solution hits only 11.14% success rate, compared to 26.18% in the basic version.

In general, the additional features add a few seconds in the case of the stress load and 100ms in the case of the smoke test to the *get payment information* request. However, this overhead is negligible compared to the privacy and security features given to the PayString users and service providers.

## V. Related Work

Although identity management and authentication have been an issue since the beginning of the Internet, only recently the idea of giving the user ownership of its digital identity aired with strength; the surge of DLTs, such as blockchains, gave this movement an engine on which to work over. It is important to note, however, that it is possible to build solutions based on the SSI paradigm without the use of blockchains, as stated by [11], that dissects some SSI solutions present on the market until 2019. This study also points to the primary disadvantage of blockchain-based solutions: if one does lose its private keys, one can't prove its identity anymore and consequently lose access to the system.

CanDID [12], although not using blockchains, tackles this problem and tries to solve it by using oracles. The idea is that the user can prove its identity by showing a previously successful login to a pre-selected account without revealing the account information. CanDID also proposes the use of real-life institutions, such as governments, to de-duplicate, attest veracity and exclude malicious users.

Another solution that, similarly to CanDID proposes the use of governments as a source of truth is UniqueID [13]. However, to prove its identity, the entity needs to state some tokens, similar to the process of adding data to a Token Curated Registry [14]. If there is consensus between the validators - here called *A-judges* - the entity can take its tokens back; if the validators don't agree on the authenticity of the identity, the entity loses the staked tokens.

UniqueID uses biometric data like private keys, much like the IDToken solution [15], that hashes the biometric data to create a private key. IDToken is a non-fungible blockchain token that represents a VC, used to prove ownership over a given identity. It is native to a public permissioned blockchain called IDChain, described as an improved version of Hyperledger Indy. The primary goal of the IDChain is to provide an infrastructure that allows the credentials to be stored on-chain, differently from what happens on the original implementation of Hyperledger Indy, which stores the credential off-chain, on the user's wallet. The latter is also the approach used by PayString Secure, although we provide mechanisms to register the VC on the blockchain.

In a more practical sphere, we can take a look at the work of [16], an initiative between the Delft University of Technology and the Dutch Government that aims to create a system based on SSI and biometric data for the issuance of paperless passports. The system is reported as not ready for large scale use yet, but dutch citizens can apply for an experimental digital passport since 2018 [17].

Also important to mention is uPort [18], a platform that provides tools, libraries and protocols for developers aiming to create user-centric solutions based on SSI. uPort also uses the concepts of DIDs and VCs; but unlikely HyperLedger Indy, it does not have its blockchain, being built over Ethereum. In this work, we opted to use Indy, since our goal is not to develop an Ethereum-based solution but to add a security layer to PayString, which is blockchain agnostic. In this context, Indy gives a more open solution.

## VI. Conclusion and Future Work

Being the first to address the performance of PayString, we also propose a security and privacy extension for it. Our security and privacy extension concerns the access control mechanism of PayString, so we propose both a formal model and a way to integrate it with DID to use VCs.

We evaluate the performance using a real-life testbed deployed on G5k, and our experimental results show an overhead which, given the privacy and security advantages offered, can be acceptable in practice, thus making the actual implementation feasible.

Future work concerns reducing the overhead, investigating the usage of *Distributed Hash Tables* (DHT) to improve the service reliability; *integrity of information* is also a potential challenge for those who will run a server because while the data will be accessible to clients, the service providers will know nothing about what is happening inside the server.

## References

[1] A. Malhotra, A. King, D. Schwartz, and M. Zochowski, "Paystring protocol," https://paystring.org/whitepaper.pdf accessed on 14/12/2020, Ripple, Tech. Rep., 2020.

[2] A. Malhotra and D. Schwartz, "Verifiable payid protocol internet draft," https://github.com/PayString/rfcs/blob/master/dist/spec/verifiable-payid-protocol.txt accessed on 14/12/2020, Ripple, Tech. Rep., 2020.

[3] A. Tobin and D. Reed, "The inevitable rise of self-sovereign identity," https://sovrin.org/wp-content/uploads/2018/03/The-Inevitable-Rise-of-Self-Sovereign-Identity.pdf accessed on 27/10/2020, Sovrin Foundation, Tech. Rep., 2016.

[4] D. Reed, M. Sporny, D. Longley, C. Allen, R. Grant, M. Sabadello, and J. Holt, "Decentralized identifiers (dids) v1.0," https://www.w3.org/TR/2020/WD-did-core-20201108/ accessed on 20/11/2020, W3C, Tech. Rep., 2020.

[5] M. Sporny, D. Longley, and D. Chadwick, "Verifiable credentials data model 1.0," https://www.w3.org/TR/vc-data-model/ accessed on 20/11/2020, W3C, Tech. Rep., 2020.

[6] M. Kubach, C. H. Schunck, R. Sellung, and H. Roßnagel, "Self-sovereign and decentralized identity as the future of identity management?" *Open Identity Summit 2020*, 2020.

[7] F. Scheidt de Cristo, W. Shbair, L. Trestioreanu, A. Malhotra, and R. State, "Privacy preserving paystring service," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021.

[8] "What is hyperledger indy?" https://sovrin.org/faq/what-is-hyperledger-indy/ accessed on 17/11/2020, The Sovrin Foundation, 2018.

[9] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 243–252.

[10] E. Kemer and R. Samli, "Performance comparison of scalable rest application programming interfaces in different platforms," *Computer Standards & Interfaces*, vol. 66, p. 103355, 2019.

[11] D. van Bokkem, R. Hageman, G. Koning, L. Nguyen, and N. Zarin, "Self-sovereign identity solutions: The necessity of blockchain technology," *arXiv preprint arXiv:1904.12816*, 2019.

[12] D. Maram, H. Malvai, F. Zhang, N. Jean-Louis, A. Frolov, T. Kell, T. Lobban, C. Moy, A. Juels, and A. Miller, "Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability," *IACR Cryptol ePrint Arch*, 2020.

[13] M. Hajialikhani and M. Jahanara, "Uniqueid: decentralized proof-of-unique-human," *arXiv preprint arXiv:1806.07583*, 2018.

[14] M. Goldin, "Token-curated registries 1.0," *Medium (accessed 2 April 2019) https://medium: com/@ ilovebagels/token-curated-registries-1-0-61a232f8dac7*, 2017.

[15] E. Talamo and A. Pennacchi, "Idtoken: a new decentralized approach to digital identi-ty," *Open Identity Summit 2020*, 2020.

[16] Q. Stokkink and J. Pouwelse, "Deployment of a blockchain-based self-sovereign identity," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 1336–1342.

[17] "Trustworthy identity on your phone," https://www.blockchain-lab.org/trust/#publications accessed on 24/11/2020, Delft University - Blockchain Lab, 2020.

[18] "Helping you build user centric apps on blockchains," https://developer.uport.me/overview/index accessed on 24/11/2020, uPort, 2020.