EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Providing trusted datafeeds to the blockchain

Grooteman, B.J.

Award date:
2019

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Providing Trusted Datafeeds to the Blockchain

Bram Grooteman, TU Eindhoven

***Abstract*—On the blockchain, smart contracts are specialized code that enforces logic on demand. To be able to provide complex services, these programs need information from outside the blockchain. Services that provide a general way of querying the internet from within a blockchain are called oracles. Oracles guarantee the authenticity of the provided data. Current oracle solutions provide these guarantees by combining information provided by multiple parties, dedicated hardware, or by providing cryptographic proofs that can be verified outside the blockchain. This thesis aims to examine public verifiable datafeeds on the blockchain and the gas-efficiency of possible solution. We proposes NAME, an oracle service that uses general cryptographic signatures to enable public on-chain verification of the provided data, and shows how NAME can be used in decentralized applications. NAME combines a blockchain interface with a software back-end to fetch Internet resources and serve the fetched data, along with a signature, to relying smart contracts. A formal specification of NAME using TLA+ is included to verify the design. Experiments conducted on the Ethereum platform to test the gas-efficiency of signature schemes show that Elliptic Curve signatures have the lowest gas cost and that BGLS signature verification on the EVM is too expensive to benefit from the reduced signature size.**

I. Introduction

BLOCKCHAINS are designed to provide a verifiable, trustless and immutable ledger among the contributing members. Any information about this ledger has to be added through transactions. Starting from an empty ledger we can rebuild the current state of the network by replaying all transactions in order. Smart contracts are user-defined functionality on this blockchain that can be triggered by sending transactions to it. The concept of a smart contract was introduced by Szabo [1] in 1994. He envisioned smart contracts as a way of automatically enforcing legal contracts. The introduction of technologies like Bitcoin [2] and Ethereum [3], which provide means to directly and fairly control assets based on predetermined terms and without a trusted third party, allows for the realization of this idea.

Smart contracts do not have networking capabilities, so to be able to act based on the real-world state, information has to be embedded in the blockchain. Oracle services provided by third parties allow smart contracts to request information from websites on the Internet. These services will retrieve the information and build a transaction containing the result along with publicly verifiable guarantees about the origin and integrity of the data. A desired interaction with an oracle service is shown in Figure 1.

Decentralized Applications (DApps) are applications that are designed to operate on the blockchain or other peer-to-peer networks and are implemented in one or multiple smart contracts. Oracle services can be used to provide necessary information to DApps. With the rise of DApps, the demand for cheap, easy-to-use oracles has grown.

Few oracle services exist, but they rely on trust in the provider, or dedicated hardware to get data to the contracts. The lack of cheap, easy-to-use and trustless solutions is cited as a critical obstacle to the evolution of Ethereum and decentralized smart contracts in general [4].

Oracle example

The Dutch Central Bank (DNB) is an institution that has two main roles in the financial sector: it caries out monetary policy and supervises Dutch financial institutions, like banks and insurance funds. Its main role is to maintain financial stability. The DNB also wants to stimulate new participants on the financial market and acts as a catalyst for emerging businesses. DNB regularly publishes financial information like exchange rates, interest rates, and supervised institutions on their website. As part of the aforementioned catalyst role, DNB is interested in allowing developers to use this information in their blockchain applications. One use-case would be to allow consumers to validate, on-chain, that a smart contract on the blockchain is owned by a supervised institution. An example being a contract that issues bonds on the blockchain. Here a DNB oracle service would allow DApps to validate that the contract they are interacting with is run by a supervised institution.

In this thesis, we provide an overview of the existing technologies that can be used to provide a trusted data feed on the blockchain, and introduce an HTTP-based oracle service called NAME. NAME acts as a relay, bridging the gap from the closed off blockchain ecosystem to the internet full of commonly trusted data feeds. The service retrieves data from websites, along with a cryptographic signature, and allows users to verify its integrity on the blockchain using these signatures.

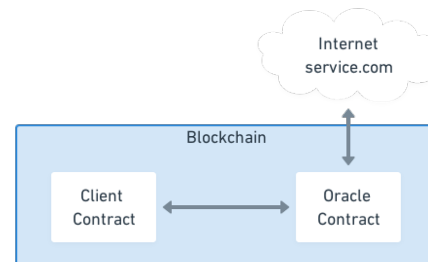

Fig. 1. Desired interaction with an oracle service. The cient smart contract will be able to request data from the oracle service and get information from the Internet back.

II. Problem Description

Data on the internet is not a static resource and may change over time. Since blockchains are designed to be fully deterministic, querying data from the internet is not supported. Different nodes validating the blockchain would execute the request at a different time from a different location, possibly retrieving different data. To maintain the deterministic nature of the blockchain the data has to be injected through transactions, creating one snapshot embedded in the chain which will be the same for all validator nodes. An oracle is a service that fetches internet resources and builds a transaction so that the data can be used on-chain.

On a smart contract platform, every operation which is executed has to be executed by all nodes in the network. In this thesis, Ethereum [3] was used due to conduct experiments due to being the biggest public smart contract platform, and providing a transaction systems that allows for quantitative comparison of solutions. On Ethereum, every operation has an associated gas cost, during a transaction the number of performed operations is counted and billed to the initiator of the transaction. It is therefore important for smart contracts to be gas-efficient.

An oracle service should never be able to manipulate the response from the requested website without the users noticing, and the relation between the user and the oracle should therefore be fully trustless. The response should be accompanied by a proof, providing non-repudiation. An oracle service should aim to have minimal overhead in terms of cost, be reliable, and be widely applicable. The main research question is:

'How can the authenticity of external data provided to a smart contract be verified in a cost effective way?'

This research question raises the following sub-questions:

1) Which signature schemes are suitable for implementing oracles for smart contracts executing on a gas-driven blockchain?
2) How should the components of an oracle system be organized to minimize the cost of deploying and operating the oracle?

In this thesis, we consider that blockchain applications want to make use of different data feeds that are provided on the internet. The oracle service acts as a bridge that allows these applications to request information and on return be able to verify that the received data is authentic.

Contributions

This thesis offers the following contributions:

- A library that can verify RSA PKCS v1.5, RSA PSS and BGLS signatures, which is published on the Ethereum blockchain platform.
- Design, implementation, and evaluation of NAME, a cryptographic signature based oracle service which allows for authentication of the provided data.

III. Preliminaries

A. *TLS and Digital Signatures*

We assume basic familiarity with TLS and HTTPS, as well as the idea behind cryptographic signatures like RSA and ECDSA. The RSA IETF standard [5] provides the main protocol which is used within NAME.
Transport Layer Security (TLS) is the successor of the SSL protocol. TLS is a suite of cryptographic protocols designed to provide secure communications over a computer network [6]. It was first introduced in 1994 and is currently on version 1.3. The protocol provides authentication, confidentiality, and integrity over any exchanged messages. TLS supports a range of different encryption schemes, including RSA. Although TLS relies on public-key signatures for authentication, the protocol protects integrity and confidentiality of the session via shared (symmetric) secret keys that are exchanged at the beginning of each session. After the initial TLS handshake, any messages sent in a TLS session are encrypted using a Message Authentication Code (MAC), which is generated using this shared secret. Thus, the TLS protocol on its own does not provide non-repudiation for the exchanged messages during a session. Since the Message Authentication Codes are being generated using a shared, symmetric key, the sender of a message can deny having ever sent the message.

RSA [7], which was named after the authors Rivest, Shamir, and Adleman, is an asymmetric encryption system which is based on the difficulty of factorizing the product of two large prime numbers. The algorithm works as follows: If n is the product of two large primes p and q, and $e = \frac{1 \bmod \phi(n)}{d}$, where we pick d such that $\gcd(d,(p-1)\cdot(q-1)) = 1$. We can now encrypt message with:

$$\begin{aligned} C \ = E(M) \ &= \quad M^e \pmod n \\ D(C) \ &= \quad C^d \pmod n \end{aligned}$$

where M is a message, C is a cipher text, E and D are the encrypting and decrypting function, respectively. e and d are called the private and the public keys, respectively.

The paper proposes the aforementioned encryption scheme but also includes a description of a signature scheme. In this signature scheme, message M is signed with a unique private key k_{priv} and using the resulting signature S can be verified by everyone that has the corresponding public key k_{pub}.

$$\begin{aligned} S \ = \ Sign(M, k_{priv}) \ &= \ M^e \ (mod\ n) \\ Verify(M, S, k_{pub}) \ &= \ (S^d \ (mod\ n) = \mathtt{M}) \end{aligned}$$

where $k_{priv} = (e, n)$ and $k_{pub} = (d, n)$ as in the encryption scheme.

BLS [8] is a signature scheme based on bilinear pairings. A bilinear pairing is a function $e : \mathbb{G}_1 \times \mathbb{G}_2$ that maps elements of two cryptographic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ to a third group $\mathbb{G}_T$. g_1 and g_2 are the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.

Three functions are defined in the BLS signature scheme:

- *KeyGen* selects a random integer $p_{priv} \in \mathbb{Z}_p$, which acts as the private key. The corresponding public key is $p_{pub} = g_1^{sk}$.

- *Sign* computes the signature of message M given private key p_{priv}: $S = H(m)^{p_{priv}}$
- *Verify* checks if the signature S matches the public key p_{pub} by checking equivalency between $e(g_1, S)$ and $e(p_{pub}, H(M))$

Within bilinear pairings we have the distinction into three types:

Type 1: Symmetric pairings where $\mathbb{G}_1 = \mathbb{G}_2$

Type 2: Asymmetric pairings where $\mathbb{G}_1 \neq \mathbb{G}_2$ and we have a homomorphism $\psi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$ which is easy to compute.

Type 3: Asymmetric pairings where $\mathbb{G}_1 \neq \mathbb{G}_2$ and we do not have have an homomorphism $\psi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$

The main advantage of BLS signatures is their length, whereas RSA signatures can be as long as 4096 bits, BLS signatures are 160 bits long while providing similar security.

B. Ethereum and Smart Contracts

Ethereum [3] is currently the most used Turing complete smart contract platform. On Ethereum smart contracts can be regarded as semi-autonomous agents on the blockchain: they have an address, a currency balance, a volatile memory to do calculation and persistent storage in the form of a key-value store. Smart contracts are not fully autonomous since they have to be triggered by a transaction. Smart contracts have to be deployed on the network, and can not be updated. On the Ethereum Virtual Machine (EVM) we have three different operations that transfer data:

- Transaction: A transaction is the only way to alter the global state of the network. By initiating a transaction, a user can trigger a smart contract or transfer Ether. Transactions can also be sent between smart contracts. For every operation that an transaction triggers, the initiator will have to pay.
- Call: Will invoke a 'constant' method and execute its smart contract method in the EVM without sending any transaction. Constant functions do not change the state of the network, and thus do not have to be paid for.
- Event: Smart contracts have the ability to emit events. These are included in blocks but can not be read by other contracts. The typical usage of an event is to communicate to a non-blockchain component that an operation was completed.

To prevent Denial of Service attacks, prevent non-terminating contracts, and generally control network resource expenditure, every operation done within smart contracts has a fixed gas cost [9]. The total transaction cost of a Ethereum transaction is: $transaction_Cost = gas_used * gas_price$. The gas price is determined by the market, allowing users to pay less for transactions when the network is not fully utilized and driving transaction cost up when there are lots of transactions happening. Gas can be bought with Ether and the price is usually denominated in Gwei, 10^{-9} Ether.

In this thesis we take the Ethereum price of $198.98 at 17-9-2019 as our baseline and use a Gas price of 20Gwei except when explicitly stated otherwise.

IV. Related Work

The need for external data on the blockchain and in smart contracts gave rise to several academic publications. Here we present the existing oracle solutions.

Software oracles:

Provable establishes trust by using TLSNotary [10]. TLSNotary is a service that can provide proof to an observer that network traffic happened between a client and a server. This is achieved using the fact that until TLS 1.1 a combination of two hashing functions are used to compute the premaster secret (PMS). This PMS is shared by the client to the server and allows the two parties to establish a symmetric key for encrypting further traffic. TLSNotary introduces a third party, the observer, which will only communicate with the client. This observer will act as an auditor, and will be convinced that the communication between the client and the server did really take place. The structure and proof generation in TLSNotary is shown in Figure 2. The steps of the algorithm are:

1) The TLS handshake is initiated by the client.
2) The server supplies the information that is needed by the client to construct the premaster secret.
3) The client as well as the observer generate half of the premaster key of the client
4) The observer will share enough information so that the client can construct all TLS keys, except for the server MAC key. The client can send requests but when the client receives data from the server it won't be able to decrypt it, since the server MAC key is needed.
5) The client shares a commitment of the message from the server with the observer.
6) The observer shares the data that the client needs to compute the server MAC key.
7) After decryption the message itself. The client will pass the encrypted page to the observer who can now verify that the commitment was valid. If this is the case, the observer will be convinced the data transfer between the client and the server happened, since there was no way of the client to fake the commitment due to lack of the required MAC key.

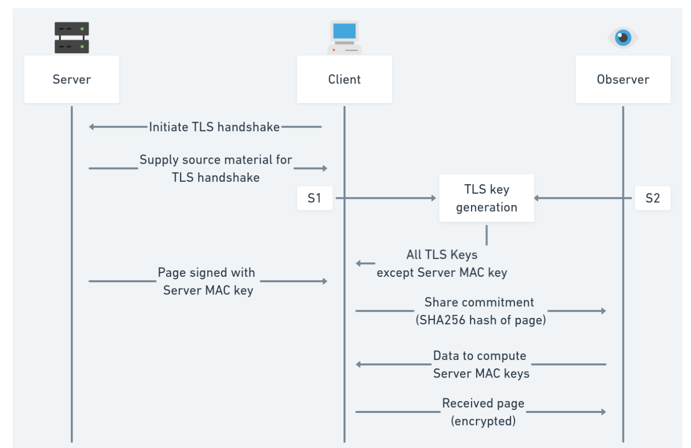

Fig. 2. The parties in the TLS Notary Service

For a party that is not directly involved in this process, the generated proof has no value since the observer could be colluding with the client. We call this an interactive proof, a validator has to be part of the proving process to be convinced the proof is valid. Provable uses a walled-off Amazon instance as the observer, circumventing this interactive property. The proofs that are produced by TLSNotary are too elaborate to store on the blockchain and are made available on the IPFS [11]. This allows for manual checking after the transaction. This means the proofs can not be verified in a smart contract and clients will have to rely on the reputation of the provider to not provide tampered data. Also, since TLS 1.2 the suite has been updated and has changed the used hashing function to one non-splitable hashing function. This update renders the TLSNotary proof impossible since the construction of the PMS can no longer be split among two parties (the client and the observer).

TLS-N [12] is a software oracle that allows for general non-repudiation of the validity of the session. Like TLSNotary, it makes use of the TLS protocol. TLS-N is a TLS extension, which is supported since TLS version 1.2, and requires both the client and the server to have the extension installed. TLS-N generates non-interactive proofs about the content of a TLS session that can be verified by third-parties. Non-interactive means here that the verifier does not have to be part of proof construction. In the TLS handshake, the TLS-N proof is initiated by agreeing on the initial session parameters. Once the TLS session starts both participants build evidence as shown in Figure 3: The server keeps track of a hash chain of all the messages in the session while the client stores the full plaintext log of the session. This choice of letting the client doing the bulk of the work was made to defend against Denial-of-Service attacks, making sure the server can not be attacked by opening a lot of TLS-N sessions, overloading it in the process.

The server builds a hash chain by using the TLS records in combination with a salt tree. This salt tree is build up from the TLS Nonce and the TLS Traffic Secret which is only known by the server. Every record that is being sent or received is first split into fixed-sized chunks before being committed by hashing it together with a salt from the salt tree. When the session is closed, the server sends the hash chain along with the salt tree to the client. The client is now able to select the parts of the record that will be included in the proof. By having the hash chain be built by the server (which has the TLS Traffic Secret) the client is unable to change any data, but can simply choose not to include chunks to protects its privacy. Non-Repudiation is maintained since the hash of the block that is not included will be published (and therefore the existence can be verified in the hash chain).
TLS-N provides non-repudiation on the whole TLS session, which allows users to not only verify that the data is authentic, but also the context in which this data is received. The downside of using TLS-N is that the verification on-chain is quite expensive compared to other services as can be seen in Table I. Two elliptic curves are shown because no elliptic curve is supported by both TLS and Ethereum (TLS supports secp256r1 and Ethereum uses secp256k1). Thus, the authors of the TLS-N paper had to implement verification for secp256r1 on top of Ethereum, which adds about 1.1 million gas. The secp256k1 curve is also shown to show how much the choice of platform influences the cost.

MUlti-SourCe oraCLE (MUSCLE) [13] aims to see how information can be made available on the blockchain more efficiently using aggregate signatures. The research of this thesis has many similarities with the MUSCLE paper but was done entirely independently. By using aggregate signatures signatures, where multiple signatures can be combined into one without loss of information, MUSCLE proposes that authenticity proofs can more efficiently be included and verified by smart contracts. Two aggregate signature schemes are proposed and their Gas usage compared with existing solutions where each data source has a single proof like RSA and the aforementioned TLS-N service. MUSCLE shows that the proposed signature schemes do not offer lower gas usage than using non-aggregate Elliptic curve signatures.

Specialized Hardware oracles:
Town Crier [14] is an oracle that leverages the Intel SGX platform [15] to execute and look-up queries in a Trusted Execution Environment (TEE). Due to the fact that the oracle runs in an environment which can not be tampered with, the code that runs on the TEE is published and the Intel Attestation Service allows anyone to verify that this is the code

Oracle Service	TLS-N		Provable	Chainlink	Town Crier	NAME
Curve[1]	secp256r1	secp256k1				
Total GAS	1,284,723	131,286	110,000	0.1n LINK [4]	217,500[2]	153,723
Ether used	0.0256945	0.0026257	0.0022		0.00435	0.0030745
Cost in USD	$4.96	$0.51	$0.48	$0.17n	$0.84	$0.59

[1] The Curve secp256r1 is the standard Ethereum curve. The secp256k1 is included for TLS-N to show the difference the chosen curve can have on the total price.
[2] 215,000 gas fixed cost and 2500 gas per request parameter. In this comparison we take one parameter to denote the data source.
[4] Chainlink request cost 1 LINK (which is their token) per request per validator. We denote the number of validators as n.

.

TABLE I
GAS COST ON ETHEREUM FOR THE DIFFERENT ORACLES SERVICES WHEN REQUESTING 1KB OF DATA

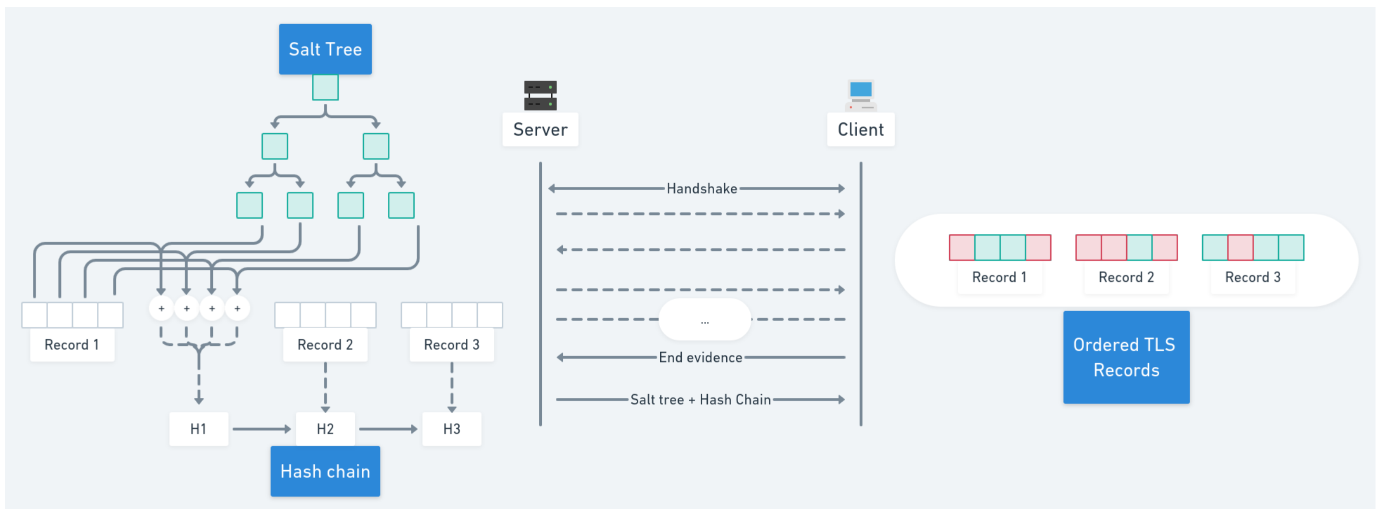

Fig. 3. The structure of TLS-N Evidence Generation. The server computes a hash chain and keeps track of the salt tree. The client stores the plaintext messages of the session. Once the session ends, the server shares his part of the evidence and the client will construct the proof.

running on the TEE, the Town Crier oracle service can operate completely transparently. This oracle service does not rely on signatures from the data provider, and thus it can do specific operations on the requested data, aggregate responses or strip away additional data that was included in the request, without requiring trust. However, SGX has been shown to be prone to side channel attacks [16] as well as malware [17] which might compromise the integrity of such an oracle. Provable also makes use of a TEE to handle requests in a trustless way. Android phones are used as trusted computing units and by making use of Googles SafetyNet and hardware attestation they provide proofs that show the integrity of the data [18]. While these services provide a way of operating transparently and are cheap to use, they have the downside of requiring an investment in specialized hardware from the service provider.

Proof of stake oracles:
Proof of stake oracles (Chainlink [19], Witnet [20]) are services where participants stake money on their truth. Participants run nodes, these nodes make data requests and return the result, without providing proof. By querying multiple participating nodes and combining the results, the ‘truth’ can be determined by majority. The nodes that deviate too much from the majority of reported ‘truth‘ can be recognized and punished. Truthful nodes will receive reputation and will be able to accept higher paying requests. This creates a financial incentive for the node-providers to behave truthfully and stays true to the decentralized nature of the blockchain. Deviation has to be specified in a data-specific manner. Boolean responses will be subject to other rules than floating point information. One big caveat with this approach is that running a node has no associated cost, and therefore Sybil attacks are an unsolved problem in these systems. This same mechanism of incentivizing honesty is also used in so-called prediction markets, where human participants provide truthful information or get punished financially (Augur [21] and Gnosis). These platforms allow users to bet on the outcome of real-life events. It allows participants to report on events, and stake money on their truth, reported truths can be disputed, in the end the truth with the highest stake will be accepted by the system. Dishonest actors get punished financially and lose their stake, which is then distributed among the honest nodes that disputed the claim. Because of human involvement, these prediction-market oracles are typically more expensive and constrained in the possible data types that can be requested.

Comparison of different solutions:
To be able to get qualitative comparison of each of the different solutions we need to specify the different criteria on which the solutions will be judged. In Table II the mentioned solutions are scored based on the these criteria using a three level scale: $-$, $\bigcirc$ and $+$, which represent negative, neutral and positive, respectively. The criteria are:

- Security: How secure is the service?
- Cost for user: How much did the user have to pay for receiving and verifying the information?
- Cost of Operation: How much does it cost to setup and run the service for the service provider?
- Server Agnostic: Is the functionality of the service dependent on the requested website implementing or supporting specific TLS extensions?
- Trustless: Can the user verify that this is the information he requested without trusting the oracle?
- Cherry-pick data: Can we extract specific information from the response, making the amount of data that needs to be included in the transaction smaller, therefore cheaper.
- Protocol usage: Which internet protocol is supported by this service?

By comparing existing solutions on these criteria in Table II, it can be seen that there is are currently no services that have a low use- and operation cost, and operate fully trustless. NAME aims to fill this gap.

Solutions	Criteria						
	Security	User Cost	Operation Cost	Server Agnostic	Trustless	Cherry-pick data	Works on Protocol
TLS-Notary	○	○	+	+	−	−	HTTPS (until TLS 1.1)
TLS-N	+	−	+	−	+	+	HTTPS (from TLS 1.2)
Town Crier	○	+	−	+	+	+	HTTPS/HTTP
Chainlink	−	○	+	+	○	+	HTTPS/HTTP
NAME	○	+	+	+[1]	+	−	HTTPS/HTTP

[1] When the IETF Draft [22] is accepted and implemented, NAME will be fully server agnostic.

TABLE II
OVERVIEW OF ORACLE SOLUTIONS
CLASSIFICATION IN −, ○, +. WHERE − IS NEGATIVE. ○ IS NEUTRAL AND + IS POSITIVE.

V. DESIGN

A. *Goals*

In designing an oracle service we aim to satisfy the following requirements:

- Decentralization: The blockchain is designed as a decentralized system. An oracle service should also fit into this philosophy by either decentralizing the service, or making sure a service does not have to be trusted to act honestly. In the case of centralized service, the integrity of the service should be publicly verifiable in a smart contract.
- Cost-effectiveness: Requesting, receiving and verifying data in a smart contract should be relatively cheap. The cost for the user should be minimal, but be enough to reimburse the provider.
- Ease of adoption: An oracle service should aim to rely on existing infrastructure to ease adoption. For DApps, the interface for requesting and receiving data should be intuitive and easy to implement.

B. *General structure*

With these goals in mind, we can sketch a general structure of how the different components of our software oracle NAME. NAME makes use of the existing public key infrastructure to be able to verify cryptographic signatures on data that is fetched from the Internet. This signature will provide evidence on the origin and authenticity of the data that is provided and can be verified on chain.

NAME wants to provide an interface that allows clients to use any information from the Internet. This involves three parties: The client, the oracle provider and the website. The oracle provider acts as a relay for the request from the client to the website and builds a transaction to the client with the response from the website. The overall structure of the oracle follows these steps:

1) The initiator of the communication will be a *Client Contract*. This is a DApp or smart contract that needs a resource provided by the website. A request is sent to the *Oracle Contract*. The *Client Contract* needs to include a payment along with the request to reimburse the oracle for the transaction containing the response in Step 5.
2) The *Oracle Contract* is a smart contract that accepts requests, stores the payment and emits an event so that the *Oracle Server* can receive the request.
3) The *Oracle Server* listens to events emitted by the *Oracle Contract* and fetches the data from the internet.
4) The *Oracle Server* builds a transaction from the data, the signature and sends it back to the callback function in the *Client Contract*. The *Oracle Server* looks up the amount paid to the *Oracle Contract* to determine the amount of Gas it can use for this transaction.
5) The client contract can verify the signature and process the data.

This structure, which uses a blockchain interface to interact with a non-blockchain component, is similar to existing services like Town Crier and Provable. NAME distinguishes itself from these services by including a publicly verifiable proof of authenticity along with the data.

Example: Let's say there exists a DApp, called SWAPPER, that allows users to exchange two different tokens. To do this at a rate which is fair to both SWAPPER and the user, the smart contract needs the exchange rate between these two tokens, it uses an oracle service to retrieve this exchange rate. When a user initiates a swap by sending his tokens to the SWAPPER Contract, the contract will send a transaction to the *Oracle Contract* with the request for the exchange rate. This request is then broadcast using an event and received by the *Oracle Server* who will carry out the request. When the server receives the result, it will create a transaction to the SWAPPER contract which will contain the data, as well as the signature. On receiving this transaction the SWAPPER contract will verify the signature, and when the authenticity is verified, use the data to release the right amount of tokens to the user. In case the signature fails, the contract ignores the data and waits for a matching response to be received.

C. *Formal specification*

To get better understanding of how such a service would operate, the possible security vulnerabilities and whether there are faults in the design, a formal specification of the system was modeled using TLA+. TLA+ is a high level modeling language designed by Leslie Lamport [23]. It is used for designing, modeling and verifying concurrent, distributed systems. The language allows systems to be specified in well-defined mathematical notation, creating an abstraction that lends itself for proving invariants, or verifying correctness of the system. The TLA+ specification of the general layout of NAME can be found in the Appendix B.

A model is defined by specifying a finite state machine. Each state transition is 'guarded' by conditions and can only be applied to the state when these conditions are met. There is no fixed order of steps in a TLA+ specification, the model checker looks at which steps are enabled at the current state and explores them all, checking if the specified invariants hold. It repeats this process until no new states are found. This exploration of the possible states makes it a tool for reasoning about distributed algorithms which might run into out-of-order execution or sporadic complex interactions between components.

We specify the state machine for each of the two components that can be seen in Figure 4. The communication is modeled by two channels, a public set, which all servers monitor and which is broadcast to by the client and an ordered sequence to which every server sends the messages containing the responses. By using a set and an ordered sequence we model the behavior of EVM events and transactions respectively. The client has five distinct states:

1) Idle: When this state is triggered, the client composes and broadcasts a request for new data on the public channel.
2) Waiting: The client waits for a valid response on the request.
3) Verifying: When a message is received, the client will verify the signature and move either to the Accepted or Rejected state accordingly.
4) Accepted: In this state the Client will process the message by saving the value of the message and move back to the Idle state where another request can be made.
5) Rejected: When the message is found to not be valid, the client will not save the message and move back to the Waiting state to accept a valid response to the request.

There are three server states:

1) Idle: The server is waiting for requests to be broadcasted.
2) Fetching: Once a message is observed on the public channel. The server will fetch the data.
3) Sent: The fetched message is added to the directional channel and the server moves back to the Idle state.

The assumptions we made when specifying the model:

- Messages that are sent are received eventually. When the platform does not provide lossless communication, a Reliable Data Transfer protocol [24] can be implemented that makes sure messages are received.
- There is at least one honest server. If no server is honest, the system would stall and never make progress since no message will contain a valid signature.

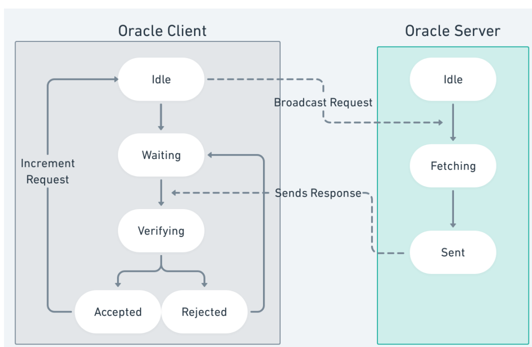

Fig. 4. The different states of the TLS+ Specification and their relations. We model multiple servers that all respond to one client

We specify invariants that hold for our model, and prove that they hold, using the TLC model checker. These invariants show that NAME can deal with messages that arrive out of order and multiple responses to the same request.

- The first valid response that is received will be accepted by the client.
- Once a response has been accepted, all subsequent responses to the same request will be rejected.
- No invalid request is ever accepted.
- The client accepts only responses that match the requested data.

In the code listing the transition from Waiting to Verifying for the client is shown. The transition is guarded by three conditions: There is a message in the toClient channel, the state of client is Waiting and the data in the message is corresponding to the current request. If the transition is taken, the messaged will be checked for validity. This state transition shows how the client makes sure it only accepts responses to messages it requested.

```
ClientReceive ==
    /\ Len(toClient) /= 0
    /\ clientState.state = "Waiting"
    /\ IF clientState.requested = Head(toClient).data
       THEN   /\ clientState' = [state |-> "Verifying",
                                 valid |-> Head(toClient).valid,
                                 requested |-> clientState.requested,
                                 data |-> clientState.data,
                                 received_data |-> Head(toClient).data]
              /\ toClient' = Tail(toClient)
       ELSE   /\ clientState' = clientState
              /\ toClient' = Tail(toClient)
    /\ UNCHANGED<<serverStates, honestServer, msgs>>
```

The full model can be found in Appendix B. The output of running the model can be found in Appendix C.

D. *Platform*

There are several smart contract platforms that could be used to publish NAME on. We specify the criteria that a platform needs to fulfill to be able to support NAME:

- Transaction system: The system that is used to pay for transactions, or to make use of the network.
- Curve: The cryptographic curve that is used within the platform to sign transactions. Platforms have built-in functions to verify signatures on the used curve (to verify the sender of a transaction) which makes verification of signatures on the same curve easier/cheaper.
- Smart Contract programming language: The language in which the smart contracts are written on this platform. Some languages have constructs that allow for easy implementation of signature verification (e.g. C# and Java supply BigInteger type in their language).
- Big Integer Exponentiation: For the verification of RSA signatures a function for exponentiation of large integers should be available.

Criteria	**Hyperledger Fabric**	**EOS**	**Ethereum**	**NEO**	**Tron**	**Libra**
Transaction system	No transaction costs	Stake tokens	Gas system	Gas system	Gas system	Gas system
Smart Contract Language	Go	C++	Solidity	C#	Solidity	Move
Curve	prime256v1[1]	secp256k1	secp256k1	secp256r1	secp256k1	ed25519
Big Integer Exponentiation	Via library	Outdated	Built-in	Built-in	Built-in	Unknown
Pairing Support	Official library	Not available	Built-in	Planned for NEO3	Not available	Unknown

[1] Multiple curves are supported.

TABLE III
DIFFERENT SMART CONTRACT PLATFORMS THAT WERE CONSIDERED

- Pairing support: To verify BGLS signatures, the platform will need to support certain operations on the elliptic curve (addition and scalar multiplication) as well as implement a pairing function.

We compare the considered platforms in Table III. In this thesis we use the Ethereum platform to reason about efficiency and implementation details since the gas system gives us a quantitative way of reasoning about these properties. The compared platforms were selected based on popularity.

The main advantage of NAME comes from the fact that verification can be done on-chain. While there is no direct datatype that can hold large integers on Ethereum, the bytes datatype can be used to represent big integers. Since the Constantinople hard fork in October 2018, the Ethereum Virtual Machine allows for computing large integer modular exponentiation [25]. RSA requires the computation of big exponents, thus the implementation of this function allows for a contract that does verification of RSA signatures on-chain.

E. *Implementation*

In the implementation of NAME we encountered several options that influence the way the service is used, as well as the cost of using the oracle. In this section we highlight these options and design an experiment to distinguish which approach is the most efficient. We focus on the two main components that influence the cost: the dataflow and the signature scheme that is used.

Dataflow:
The components discussed in subsection V-B can be arranged in multiple ways, all leading to oracle services with slightly different properties and use-cases. In Figures 5, 6 and 7 these arrangements are shown. We distinguish three ways the code can be structured within NAME.

Thick Interface does the verification of the signature in the *Oracle Contract*, as shown in Fig. 6. When a request is made, the *Oracle Contract* will verify the signature and will only return data to the *Client Contract* that is accompanied by a matching signature.

+ Ease of use. The client can request data and be sure that the data that is returned is authentic.
- Expensive. To be able to verify the signature of a response, the interface needs to store the public key into storage for each request. Storing the key in memory is not sufficient since the data does not persist in memory between requesting and receiving the response. Due to the length of the RSA public key this is an expensive operation.

Thick Client does the verification of the signature in the *Client Contract*, as shown in Fig. 5. This means every *Client Contract* will need to include the verification code. There is minimal redirection of the data, only one transaction from the *Oracle Server*.

+ Minimal passing of the data. Only one transaction which contains the data, as well as the signature.
- Code duplication, every Client contract needs to include the same signature verification code.

Client Library deploys the verification code as a library, which is a contract with no internal state, as shown in Fig. 7. This allows anyone to use the functions in their smart contract without having to include the code itself.

+ No duplicate code deployment. The verification code lives in the library and only has to be deployed once.
- There has to be an extra redirection step, the data and signature need to be send to the Library, increasing the cost of usage.
- Hard to verify. When the code is in a deployed contract, only the bytecode is public. Since bytecode is not designed to be humanly readable, it is non-trivial for users to verify that this contract does the verification correctly.

Signature schemes:
We propose the use of RSA signatures to verify the authenticity of data. In a 2018 Internet Engineering Task Force (IETF) proposal a standard for adding an RSA signature to all HTTP response headers was introduced [22]. Including a signature with each HTTP response would allow a client be able to verify the sender as well as the authenticity of the received data. The implementation of this draft allows for an oracle service that can provide authenticity proofs, and which does not rely on extensions to be implemented on the server side. When the draft is accepted and implemented, the oracle can act as a bridge, requesting on behalf of users and providing the data together with the signature to the client. Such an oracle will abstain from trust, since clients can verify the signature and therefore the integrity of the oracle. By supporting this

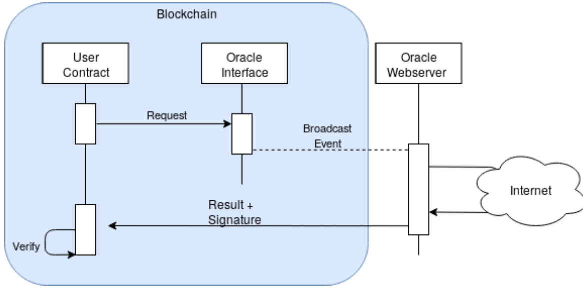

Fig. 5. Thick Client. The *Client Contract* has the verification code and is responsible for making sure it only uses data that is accompanied by a matching signature.

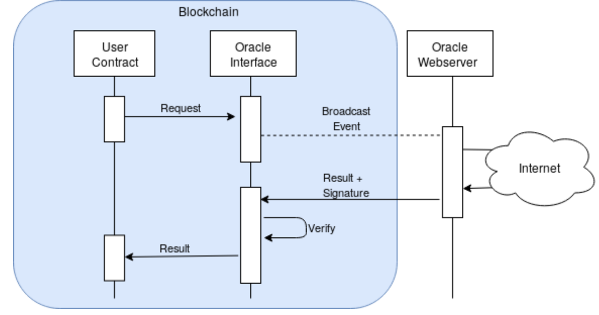

Fig. 6. Thick Interface. The *Oracle Contract* receives the data with the signature. It will verify the signature and only then return the data to the *Client Contract*

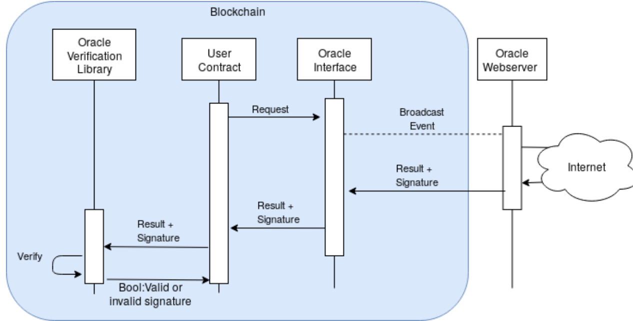

Fig. 7. Client Library. The *Client Contract* is linked with the verification library, which holds the verification code.

specification for the HTTP protocol, HTTPS request can also be served. Both HTTP and HTTPS the two protocols make use of the same base protocol (HTTP), but HTTPS adds TLS to ensure extra security/authentication at the transport layer.

Along with RSA consider two additional signature algorithms that are currently not supported in the TLS specification. These might be added in future versions of the TLS protocol and have properties that make them suitable to be used as replacements for RSA.

Elliptic curve signatures: Elliptic curve encryption systems are based on elliptic curves in a finite field, requiring smaller keys while providing equivalent security compared to the RSA encryption scheme. Ethereum uses the secp256k1 curve to generate addresses and sign transactions. The operations to verify signatures from these curves are available on the EVM. The cost of these verification operations is greatly reduced when compared to operations needed for RSA signature verification. At the same time, Elliptic curve signatures are less computationally expensive to compute, which would make the data providers less prone to Denial-of-Service attacks.

BGLS signatures: Introduced by Boneh, Gentry, Lynn and Shacham [26], allow a third party to compress an arbitrary group of BLS signatures $(\sigma_1, ..., \sigma_n)$ that verify a collection of public key and message pairs $((pk_1, m_1), ..., (pk_n, m_n)$ and produce an aggregated signature that verifies the same collection. We call these signatures BGLS signatures, after the authors. An oracle could make use of BGLS signature by allowing its users to request data from multiple data sources, and aggregate the different signatures into one general signature so the user can verify the whole set of requests by verifying one signature. Less information will have to be included in the transaction, saving of gas.

In the aforementioned Constantinople hardfork that introduced the modulus for big integers built-in contract which made RSA verification possible, there were also three functions added allows for computing a pairing [27] as well as basic operations on a curve [28]. These additions make it possible to implement verification of BGLS signatures on the EVM. We have to make a small adjustment, Ethereum currently only supports Type 3 pairings (as discussed in Section III-A) on a specific Elliptic curve called `alt bn128`. Since we need a Type 2 pairing to verify BGLS, we include a modification, proposed by Chatterjee [29], to the Sign algorithm that makes BGLS compatible with Type 3 pairings.

RSA padding schemes:

In this thesis we use RSA as discussed in Section III. To increase the security of the RSA signature, a padding scheme is used in the TLS standard. Such a scheme makes it so that every signature has the same length, and is therefore leaking minimal information about the private key that is used to sign the message. In this survey [30] is shown that with the information from unpadded signatures, an attacker could be able to forge signatures. The TLS standard allows for two padding schemes: PKCS v1.5 and PSS as defined in the RFC [31]. NAME implements the verification of both padding schemes.

RSA-PKCS v1.5: This is a deterministic padding scheme that pads each message to a fixed length before applying the RSA signature algorithm. The padding that is added to the message M is:

$$EM = 0x0001ff...ff00 \mid\mid DER(Hash(M))$$

Where $||$ is the concatenation operation and the dots will be filled with $0xff$ bytes until the total length of EM is equal to the RSA key length (which is 128 or 256 bytes in the setting of this thesis). Hash is a function that computes the SHA-256 hash of the message. DER is a function that encapsulates the hash of the message in an encoding scheme, denoting the hash function used. The resulting EM will be used as the message in the rest of the RSA algorithm as discussed in Section III-A.

RSA-Probabilistic Signature Scheme (PSS): In a review of the security of the RSA signature scheme [32] it was shown

that PKCS v1.5 signatures are vulnerable to broadcasting attacks, in which an attacker collects enough signatures to be able to forge valid signatures himself. That's why the latest version of the TLS 1.3 protocol no longer includes the PKCS v1.5 padding scheme for RSA but solely the PSS variant. PSS uses a random salt, which means that two signatures of the same message, signed with the same private key, will not be equal. The signature construction consists of 12 steps, thus too extensive to discuss here fully, and can be found in the RFC [31].

Cost Analysis:
With all the possibilities highlighted, an analysis of the cost is included to give more insight in the efficiency of each of the approaches. The gas usage of a transaction can be broken down into $total_gas = 21000 + 68 * lenth(data) + contract_execution$ with the length in bytes. During the experiments we are interested in this $total_gas$, since this is what has to be paid by the user, thus the best measure of gas-efficiency.

We first look at the cost of the built-in functions that are used to verify the signature schemes that were proposed in the previous subsection. The main operations that verify the signature have a constant cost, and are not dependent on the size of the message. A line-by-line analysis of each of the implementations could be done to derive a formula that calculates the total cost of the verification based on the size of the input. However, for the BGLS and PSS schemes this is a very labor intensive and tedious exercise. By singling out the biggest constant-cost operations that are are needed for verification, we get an idea of the base cost. The gas-cost for the RSA functions are calculated in Appendix A. From Table IV we can see that the pairing operation a great deal more expensive. There is a proposal to reduce the cost of this built-in function in EIP 1108 [33]. This would reduce the cost of the pairing operation to 45 000 + 34 000 per point. This change is included in the Istanbul hardfork which is scheduled for December 4, 2019.

Function	Signature	Gas cost	
`ecrecover`	ECDSA	3 000	
`big int mod` - 1024 bit	RSA	11 315	
`big int mod` - 2048 bit	RSA	37 888	
`pairing`	BGLS	100 000	+ 80 000 per point
`ecmul`	BGLS	40 000	

TABLE IV
GAS COST OF THE DIFFERENT BUILT-IN FUNCTIONS THAT ARE USED IN THE PROPOSED SIGNATURE SCHEMES. PKCS AND PSS ARE BOTH RSA BASED AND THUS MAKE USE OF THE `BIG INT MOD` FUNCTION

We design two experiments to investigate the gas cost of the proposed setups. An implementation of each dataflow options was deployed on the testnet. These implementations contained the 1024bit RSA signature verification code. By sending each implementation messages with an increasing number of bytes, we can see how each of them behaves when the size of the input gets bigger. This shows how much the client would have to pay for using the oracle if a certain dataflow was chosen. We also include the deployment cost of the different implementations.

To compare the different signature schemes, we implement each discussed signature scheme and deploy it on the testnet. The transactions that are sent contain the data and a signature. The public key is set in a separate transaction, and this cost is also measured. In this experiment we test the cost for verifying 1,2,5 and 10 messages of 32 bytes, which are all signed with the same public key. The aim of this experiment is twofold. Firstly, it allows us to compare the cost of verifying a single message, which will give a good baseline performance for the signature schemes. The goal with increasing the number of messages is getting an idea of how many messages have to be aggregated for BGLS to be a cheaper alternative than the non-aggregating signature schemes. The results of these experiments can be found in the Results section.

F. Security Considerations

It is important that information supplied by the oracle is authentic and can be publicly verified. This also means that the service should be resistant to malicious actors.

Man in the middle attack (MitM) / Impersonation: In a MitM attack, the attacker impersonates the server, in hopes of retrieving information from the user or providing the user with faulty data. When data is requested, an event is emitted for an external actor to respond to. Since events and transactions are public, anyone can monitor these requests and forge a response. Now this malicious actor can send false information to the contract. However, NAME checks the public key and the signature, which gives sufficient insurance that the origin of the data is the possessor of the matching private key. Messages with non matching signatures are discarded.

Denial of Service (DoS): In a Denial of Service attack the attacker floods the service with requests, slowing it down or denying other users from using the service. The NAME oracle service has three components which could be targeted in a DoS attack:

- *Oracle Contract*: This component exists on the blockchain and the attacker would need to do a DoS attack on the whole network. The concept of gas was introduced to make this an expensive endeavor.
- *Oracle Server*: The server is not part of the blockchain but acts only when it sees an event emitted from the *Oracle Contract*. We can follow the same reasoning as before, and conclude that the only way to deny anyone from using the Oracle Server is by congesting the whole network.
- *Website*: The website that was requested could be attacked. If the Oracle Server can not reach the site that is requested, it won't be able to relay the information on the site back to the requester. This problem is not specific to NAME and is an issue that all oracle services suffer from.

TLS Attacks: When providing non-repudiation on a TLS session, there are possible attacks that can lead to wrong interpretations of the session. We highlight the attacks that are discussed in the TLS-N [12] paper:

Time Shifting Attack: Reponses to old requests can be rebroadcast (while still holding a valid signature). This is

especially a problem for time-sensitive data like stockprices. These attacks can be circumvented by including the time of signing in the response data, the problem being that smart contracts do not have access to the current date and thus verifying that the included date is valid would be a manual task.

Content Omission Attack: Messages are being left out of the conversation. This leads to a wrong interpretation of the conversation. An oracle could pretend to provide the Bitcoin price, but in reality supply the Ether price from the same provider. Since they come from the same information provider, the signature is valid and the DApp will accept the data. To circumvent this data providers should be encouraged to include some kind of context in their responses. This increases the response size, but allows any client to validate the context of the data without needing to have proof of the request. If no context is included, an oracle can not do anything other than prove the whole session (TLS-N) or make use of multiple different providers (Chainlink). Other solutions need to rely on the users to verify the data.

VI. Results

The conducted experiments resulted in multiple tables with the gas cost of of each approach. Figure 9 shows how the cost grows when more data is sent, and Table V shows the deployment cost of each approach. Figure 8 shows the cost of each signature scheme when the number of messages increases.

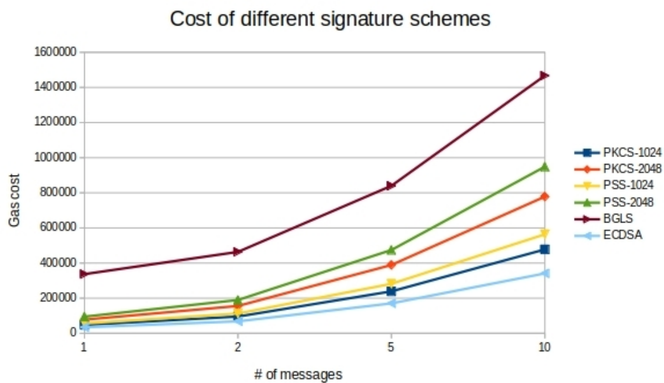

Fig. 8. Plot of how the gas cost grows when more messages are sent and verified using different signature schemes. We include RSA-PKCS and RSA-PSS with keylengths of 1024 and 2048 bit.

Option	Gas Cost	Cost in USD
Thick Interface	1460189	$5.81
Thick Client	1050677	$4.18
Client Library	1217528	$4.84

TABLE V
Deployment cost of the different implementations

VII. Discussion

From Figure 9 we see that the verifying interface has a high associated cost of usage for the Oracle client. By giving up some usability, this cost can be reduced by moving the verification code to the client. Placing the verification code in a library has practically no influence on the cost for the oracle client and only differs in the deployment phase, which can be seen in Table V.

In Figure 8 we see that the BGLS signature is the most expensive signature scheme. While BGLS has the smallest signature, the cost of verification makes it the most expensive option. Elliptic curve signatures are the most gas-efficient, for any number of messages we tested. These observations are in line with the numbers in Table IV. For RSA we see that the PKCS padding scheme is more gas-efficient than the PSS scheme for 1024 bit keys, for 10 messages the difference with PSS is 80 000 gas. When the length of the public key is doubled to 2048 bits, we see that due to the number of operations that need to be done with the key the cost for verification of the PSS increases faster and the difference for 10 messages is 140 000 gas. In Table VI we see that elliptic curve signatures are the most gas-efficient to store. BGLS public keys have comparable cost to 1024 bit RSA keys. The 2048 bit RSA keys are long and thus are the most costly to send and store. Combining the cost from Table VI and Figure 8 we can conclude the Elliptic curve signatures are the most gas-efficient.

Signature Scheme	Gas Cost
PKCS-1024	131174
PKCS-2048	220032
PSS-1024	131174
PSS-2048	220032
BGLS	111913
ECDSA	28164

TABLE VI
The cost of sending and storing a public key into the memory of a smart contract

A. *Limitations*

1) Central Certificate Authority: The setting which is discussed in this thesis assumes that there is a way of retrieving the public keys of public data sources. These public keys need to be available for data consuming contracts, to validate the signatures. Currently, there does not exists such a repository on the web, or on the blockchain. A solution that allows data consumers to retrieve and use public keys of common data sources on the Internet needs to be in place before NAME can be efficiently integrated in blockchain applications.

2) Cost: The Gas cost that are discussed were measured on Ethereum. While the concept of using signatures to achieve non-repudiation, the way this was achieved in NAME might not be suited/possible on other platforms. One prime example of this is TLS-N, which suffered from high gas cost verifying proofs due to the fact that there was no cheap way of verifying TLS signatures on Ethereum. In Table I we show that the cost of the TLS-N service would be reduced by 90%, from five dollars to half a dollar if TLS supported the sec256k1 curve. The cost discussed in this thesis are platform dependent, and therefore the conclusions might not hold when NAME is implemented on another platform.

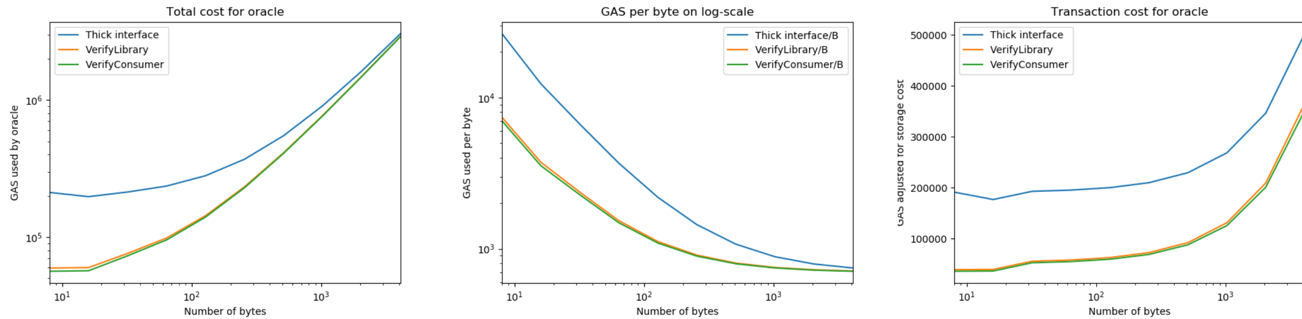

Fig. 9. The different GAS cost for the different implementation possibilities, combining expended GAS by the client and server. This plot is excluding the deployment cost for each approach. The figures, from left to right: The total Gas cost of the interaction. The GAS cost per byte. The total cost adjusted for writing to storage in the consumer.

B. *Further work*

Based on the aforementioned limitations we can identify topics that require further work:

- Decentralized Key Management Systems (DKMS) : The system that was proposed in this thesis relies on having access to the public keys of data providers. Currently, there is no way to retrieve these public keys from a smart contract and all keys have to be hardcoded. A DKMS would allow for the lookup of public keys, increasing the flexibility of the system. Further research has to be conducted to be able to ensure such a key management system is not susceptible to fraud and how to ensure parties can rely on the identities in such a system.
- Multisignatures: A multisignature allows a group of S signers to sign one message together. This differs from the BGLS scheme, which allows someone to aggregate the signatures of different messages. Multisignature are virtually the same length as an individual signature. An oracle could make use of this construct by querying different providers for the same data, supplying the resulting multisignature to the blockchain. Such a multisignature would give stronger guarantees on the data, making the DApp harder to manipulate/influence by a dishonest data provider.

VIII. Conclusion

Smart contracts only have access to the data that is within the network to ensure the deterministic execution on each node. To still be able to use information from the internet, oracle services provide a general interface for fetching data from the Internet. By providing a proof, these oracle services can show that the data is authentic.

In this thesis, we examined different options of providing verifiable information to smart contracts and compared these options based on their gas-efficiency. We found that existing solutions either fail to guarantee authenticity, only support specific websites or are expensive to use or run. We introduce NAME, an HTTP-based oracle service for providing trusted data feeds, and compared different implementations of this service. A formal specification was given to formalize the properties of NAME. We found that Elliptic curve signature are the most gas-efficient and that aggregate signature verification too expensive to benefit from the reduced signature size.

In comparison with existing solutions, the NAME service has low operating cost while still guaranteeing authenticity of the provided data. The main caveat are the assumptions made, both the acceptation and adaptation of the HTTP signature draft as well as a Certificate Authority being built on the blockchain.

Future research has to be conducted into implementing a Decentralized Key Management System on the blockchain, to allow smart contracts to access public keys. Future research and adoption of multisignatures could provide a way of introducing robustness to dishonest information providers to the NAME service.

Acknowledgment

The author would like to thank M. Holenderski for the supervision and great feedback. As well as, T. Aerts and the DNB, specifically the MIB/BVM department, for the opportunity.

References

[1] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.
[3] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.
[4] Coindesk Gideon Greenspan. Why many smart contract use cases are simply impossible. https://www.coindesk.com/three-smart-contract-misconceptions, 2016.
[5] B Kaliski. Pkcs #1: Rsa encryption, Mar 1998.
[6] E. Rescorla and Mozilla. Transport layer security 1.3, August 2018.
[7] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.
[8] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.
[9] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.
[10] Adam Gibson Dan Smith, oakpacific. Tlsnotary - a mechanism for independently audited https sessions. https://tlsnotary.org/TLSNotary.pdf, 2014.

[11] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[12] Hubert Ritzdorf, Karl Wst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. Tls-n: Non-repudiation over tls enabling - ubiquitous content signing for disintermediation. Cryptology ePrint Archive, Report 2017/578, 2017. https://eprint.iacr.org/2017/578.

[13] Bjorn van der Laan. Publicly verifiable authenticity of data from multiple external sources for smart contracts using aggregate signatures. Master's thesis, TU Delft, 2018.

[14] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 270–282, New York, NY, USA, 2016. ACM.

[15] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[16] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. Interface-based side channel attack against intel SGX. *CoRR*, abs/1811.05378, 2018.

[17] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel SGX. *CoRR*, abs/1902.03256, 2019.

[18] Provable. Android proof: Authenticated data gathering.

[19] Decentralized oracles — chainlink.

[20] The oracle network protocol — witnet.

[21] Decentralized prediction markets — augur.

[22] M. Cavage and M. Sporny. Signing http messages, April 2019.

[23] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[24] Steven L Scott, Dennis C Abts, Robert Alverson, and Edwin Froese. Reliable message transport network, July 29 2014. US Patent 8,792,512.

[25] Vitalik Buterin. Big integer modular exponentiation.

[26] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 416–432. Springer, 2003.

[27] Vitalik Buterin. Precompiled contracts for optimal ate pairing check.

[28] Christian Reitwiessner. Addition and scalar multiplication on the elliptic curve alt bn128.

[29] Sanjit Chatterjee, Darrel Hankerson, Edward Knapp, and Alfred Menezes. Comparing two pairing-based aggregate signature schemes. *Designs, Codes and Cryptography*, 55(2-3):141–167, 2010.

[30] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[31] B Kaliski and J Jonsson. Pkcs #1: Rsa cryptography specifications version 2.2, Nov 2016.

[32] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of tls 1.3 and quic against weaknesses in pkcs#1 v1.5 encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1185–1196, New York, NY, USA, 2015. ACM.

[33] Antonio Cardozo. Reduce alt bn128 precompile gas costs.

APPENDIX A
COST ANALYSIS CALCULATIONS

Calculate the cost of executing an big integer operation with a 1024 bit and 2048 bit long number. This is the key operation that is used in the RSA signature verification.

1024 Bit: In this operation we use the supplied public key, the hash of the data and the signature. All these have a fixed size, and thus, this operation has a fixed cost. Following the formula in the EIP results in:

$$consumes = \texttt{floor}(\texttt{mult_complexity}(\texttt{max}(128, 128)) * \frac{\texttt{max}(17, 1)}{20})$$
$$= \texttt{floor}(\texttt{mult_complexity}(128) * \frac{17}{20}) = 11315$$

where mult_complexity is function defined in the EIP [25] and is a function intended to approximate the difficulty of Karatsuba multiplication.

2048 Bit: When we double the length of the signature, the calculation on step three changes, since now both the base and modulo are 256 bytes long. The cost of this step for a 2048 bit input is:

$$consumes = \texttt{floor}(\texttt{mult_complexity}(\texttt{max}(256, 256)) * \frac{\texttt{max}(17, 1)}{20})$$
$$= \texttt{floor}(\texttt{mult_complexity}(256) * \frac{17}{20}) = 37888$$

using the aformentioned mult_complexity function defined in the EIP.

APPENDIX B
TLA SPECIFICATION

MODULE *NAME2*

Specification of the NAME *DApp* We abstracted the Oracle Interface away, since this object does not contribute any noteworthy functionality, and is just there as a convenience/implementation measure. The main goal of this spec is to formalize NAME and be able to prove that specific invariants hold during execution

EXTENDS *Integers*, *Naturals*, *Sequences*, *TLC*, *FiniteSets*, *Randomization*

CONSTANTS *PROVIDERS*

Here a strong assumption is made that assumes that when a message is sent, it will arrive. This is not so much a property of the protocol as it is a property of the platform that the protocol runs on

$Validity \triangleq \{0, 1\}$ Simple replacement for *RSA* signature
$Data \triangleq 0 \, .. \, 10$ Simple replacement for requested *Data*

$Messages \triangleq$
$[type : \{\text{"Request"}\}, data : Data]$
$\cup$
$[type : \{\text{"Response"}\}, valid : Validity, data : Data]$

VARIABLES
clientState,
serverStates,
msgs,
toClient,
honestServer

$vars \triangleq \langle clientState, serverStates, msgs, toClient, honestServer \rangle$

$NAMETypeOk \triangleq$

The type-correctness invariant

$\land\ clientState \in [state : \{\text{"Idle"}, \text{"Waiting"}, \text{"Verifying"}, \text{"Accepted"}, \text{"Rejected"}\},$
$valid : Validity, requested : Data, data : Data, received_data : Data]$
$\land\ serverStates \in [PROVIDERS \rightarrow [state : \{\text{"Idle"}, \text{"Fetching"}\}, data : Data]]$
$\land\ msgs \in \text{SUBSET } Messages$
$\land\ honestServer \in PROVIDERS$

$Send(m) \triangleq msgs' = msgs \cup \{m\}$
$Send_toClient(m) \triangleq toClient' = Append(toClient, m)$
$Max(S) \triangleq \text{CHOOSE } i \in S : \forall j \in S : i \geq j$
$Abs(x) \triangleq Max(\{x, -x\})$

Client Actions

$ClientRequest \triangleq$ send a request
$\land\ clientState.state = \text{"Idle"}$
$\land\ Send([type \mapsto \text{"Request"}, data \mapsto clientState.data + 1])$

```
∧ clientState′ = [state ↦ "Waiting",
                  valid ↦ clientState.valid,
                  requested ↦ clientState.data + 1,
                  data ↦ clientState.data,
                  received_data ↦ clientState.received_data]
∧ UNCHANGED ⟨serverStates, toClient, honestServer⟩

ClientReceive ≜  receive a response
   ∧ Len(toClient) ≠ 0
   ∧ clientState.state = "Waiting"
   ∧ IF clientState.requested  = Head(toClient).data
       THEN  ∧ clientState′ = [state ↦ "Verifying",
                               valid ↦ Head(toClient).valid,
                               requested ↦ clientState.requested,
                               data ↦ clientState.data,
                               received_data ↦ Head(toClient).data]
             ∧ toClient′ = Tail(toClient)
       ELSE  ∧ clientState′ = clientState
             ∧ toClient′ = Tail(toClient)
   ∧ UNCHANGED ⟨serverStates, honestServer, msgs⟩

ClientVerify ≜  verify the response
   ∧ clientState.state = "Verifying"
   ∧ IF clientState.valid = 1  The only difference is whether data is updated or not
         THEN clientState′ = [state ↦ "Accepted",
                              valid ↦ 1,
                              requested ↦ clientState.requested,
                              data ↦ clientState.received_data,
                              received_data ↦ clientState.received_data]
         ELSE clientState′ = [state ↦ "Rejected",
                              valid ↦ 1,
                              requested ↦ clientState.requested,
                              data ↦ clientState.data,
                              received_data ↦ clientState.received_data]
   ∧ UNCHANGED ⟨serverStates, toClient, msgs, honestServer⟩

ClientAccept ≜  Accept the update
   ∧ clientState.state = "Accepted"
   ∧ IF ((Len(toClient)  ≠ 0) ∧ (Head(toClient).data < clientState.data))
       THEN ∧ toClient′ = Tail(toClient)
            ∧ clientState′ = clientState
       ELSE ∧ clientState′ = [state ↦ "Idle",
                              valid ↦ 1,
                              requested ↦ clientState.requested,
```

```
                                data  ↦ clientState.data,
                                received_data ↦ clientState.received_data]
                  ∧ toClient′ = toClient
     ∧ UNCHANGED ⟨msgs, serverStates, honestServer⟩

ClientReject ≜   Reject the update
     ∧ clientState.state = "Rejected"
     ∧ clientState′ = [state ↦ "Waiting",
                       valid ↦ 1,
                       requested ↦ clientState.requested,
                       data  ↦ clientState.data,
                       received_data ↦ clientState.received_data]
     ∧ UNCHANGED ⟨serverStates, toClient, msgs, honestServer⟩
```

```
Server actions

ReceiveRequest(p) ≜   Server p receives a request
     ∧ serverStates[p].state = "Idle"
     ∧ [type ↦ "Request", data ↦ serverStates[p].data + 1] ∈ msgs
     ∧ serverStates′ = [serverStates EXCEPT ![p] = [state ↦ "Fetching",
                                                    data ↦ serverStates[p].data]]
     ∧ UNCHANGED ⟨clientState, honestServer, msgs, toClient⟩

 Server p sends a response
SendResponse(p) ≜
     ∧ serverStates[p].state = "Fetching"
     ∧ IF (p = honestServer)
          THEN Send_toClient([type ↦ "Response", valid ↦ 1,
                              data ↦ serverStates[p].data + 1])
          ELSE Send_toClient([type ↦ "Response", valid ↦ 0,
                              data ↦ serverStates[p].data + 1])
     ∧ serverStates′ = [serverStates EXCEPT ![p] = [state ↦ "Idle",
                                                    data ↦ serverStates[p].data + 1]]
     ∧ UNCHANGED ⟨clientState, msgs, honestServer⟩
```

```
The steps of the specification

Init ≜ ∧ clientState = [state ↦ "Idle", valid ↦ 0, data ↦ 0,
                        requested ↦ 0, received_data ↦ 0]
       ∧ serverStates = [r ∈ PROVIDERS ↦ [state ↦ "Idle", data ↦ 0]]
       ∧ msgs = {}
       ∧ honestServer = CHOOSE o ∈ PROVIDERS : TRUE
       ∧ toClient = ⟨⟩

Next ≜ ∨ ClientRequest
       ∨ ClientReceive
```

$\lor$ *ClientVerify*
$\lor$ *ClientAccept*
$\lor$ *ClientReject*
$\lor \exists\, p \in PROVIDERS : (serverStates[p].state = \text{"Idle"} \land ReceiveRequest(p))$
$\lor \exists\, p \in PROVIDERS : (serverStates[p].state = \text{"Fetching"} \land SendResponse(p))$

Invariants and of the specification

$$
\begin{aligned}
Fairness \triangleq\ & \land \mathrm{SF}_{vars}(Next) \\
& \land \mathrm{SF}_{vars}(ClientRequest) \\
& \land \mathrm{SF}_{vars}(ClientReceive) \\
& \land \mathrm{SF}_{vars}(ClientVerify) \\
& \land \mathrm{SF}_{vars}(ClientAccept) \\
& \land \mathrm{SF}_{vars}(ClientReject) \\
& \land \exists\, p \in PROVIDERS : \mathrm{SF}_{vars}(ReceiveRequest(p)) \\
& \land \exists\, p \in PROVIDERS : \mathrm{SF}_{vars}(SendResponse(p))
\end{aligned}
$$

$NAMESpec \triangleq Init \land \Box[Next]_{vars} \land Fairness$
THEOREM $NAMESpec \Rightarrow \Box NAMETypeOk$

Invariants
If the client state has received honest data, it will accept it
$data_increasing \triangleq (clientState.received_data = 1 \land clientState.valid = 1) \leadsto (clientState.data = 1)$

If the honest server has sent data, the client will accept it
$server_sends \triangleq ((serverStates[honestServer].data = 1 \land serverStates[honestServer].state = \text{"Idle"}) \leadsto (clientState.data = 1))$

Only the first honest response will be accepted.
$$
\begin{aligned}
first_accepted \triangleq\ & (toClient \neq \langle\rangle \land \\
& clientState.state = \text{"Waiting"} \land \\
& clientState.requested = Head(toClient).data \land \\
& Head(toClient).valid = 1) \leadsto (clientState.data = 2)
\end{aligned}
$$

We can not have variables in invariants. But $clientState.data = i \leadsto i + 1$
$continues \triangleq ((clientState.data = 1 \land clientState.state = \text{"Waiting"}) \leadsto (clientState.data = 2))$

Only accepts data that was requested
$accept_requested \triangleq ((clientState.requested = 1) \leadsto (clientState.state = \text{"Accepted"} \land clientState.data = 1))$

$$
\begin{aligned}
accept_invalid \triangleq\ & (toClient \neq \langle\rangle \land \\
& Head(toClient).valid = 0 \land \\
& clientState.state = \text{"Verifying"}) \leadsto (clientState.state = \text{"Rejected"})
\end{aligned}
$$

APPENDIX C
TLA+ MODEL CHECKER OUTPUT

General

Start: 13:12:40 (Sep 25) End: 13:15:21 (Sep 25) **Not running**

Fingerprint collision probability: calculated: 1.8E-11

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:02:41	53	36,790	14,523	0
00:00:24	31	10,226	4,297	547
00:00:02	0	1	1	1

Actions at 00:02:41

Module	Action	Location	States Found	Distinct States
NAME2	Init	line 146, col 1 to line 146, col 4	1	1
NAME2	ClientRequest	line 56, col 1 to line 56, col 13	1,514	20
NAME2	ClientReject	line 110, col 1 to line 110, col 12	362	76
NAME2	ClientVerify	line 82, col 1 to line 82, col 12	1,867	156
NAME2	ClientAccept	line 97, col 1 to line 97, col 12	1,505	179
NAME2	ClientReceive	line 67, col 1 to line 67, col 13	8,984	384
NAME2	Next	line 158, col 33 to line 158, col 83	11,359	4,207
NAME2	Next	line 159, col 33 to line 159, col 85	11,301	9,500

Evaluate Constant Expression

User Output

TLC output generated by evaluating Print and PrintT expressions.

```
No output is available
```

Progress Output

Spec Status : parsed