Blockchain-Based Auditing of Transparent Log Servers

Hoang-Long Nguyen, Jean-Philippe Eisenbarth, Claudia-Lavinia Ignat, Olivier Perrin

Blockchain-Based Auditing of Transparent Log Servers

Hoang-Long Nguyen, Jean-Philippe Eisenbarth, Claudia-Lavinia Ignat, and Olivier Perrin

Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
hoang-long.nguyen@loria.fr

Abstract. Public key server is a simple yet effective way of key management in secure end-to-end communication. To ensure the trustworthiness of a public key server, CONIKS employs a tamper-evident data structure on the server and a gossiping protocol among clients in order to detect compromised servers. However, due to lack of incentive and vulnerability to malicious clients, a gossiping protocol is hard to implement in practice. Meanwhile, alternative solutions such as EthIKS are too costly. This paper presents *Trusternity*, an auditing scheme relying on Ethereum blockchain that is easy to implement, inexpensive to operate and resilient to malicious clients. We also conduct an empirical study of system behaviour in face of attacks and propose a lightweight anomaly detection algorithm to protect clients against such attacks.

Keywords: Authentication · Public key · Blockchain · Auditing · Ethereum

1 Introduction

In order to meet user demands regarding online privacy and prevent digital snooping, identity and data theft, we can see in the last years an increase in the number of end-to-end encryption (*E2EE*) messaging services. A major challenge in any E2EE system is to prevent man-in-the-middle attack (*MITM*) where an adversary impersonates a legitimate communication participant. Thus, some E2EE systems leverage Out-of-Band (OOB) channels for client *authentication* by means of manual comparison of public key fingerprints [27] or pre-known shared passwords [5]. However, secure and easy to use OOB channel is hard to achieve in practice. Password entropy is often overlooked by users while fingerprint comparison is error-prone and cumbersome [22]. Other client authentication solutions rely on trusted third parties, i.e. *key servers* to distribute and authenticate public keys among clients. Many popular E2EE services such as WhatsApp [21] adopt centralized key servers as they are easy to use and straightforward to implement. However, a centralized key server becomes a system single point of failure being vulnerable to attacks from adversaries or surveillance agencies. Therefore, secure and autonomous client authentication remains a major challenge for E2EE.

Rather than preemptively verify the exchanged keys, by using the *Key transparency* approach [13] [17] [26], clients can verify if the key server behaves correctly

during communication. The general idea is to turn the key server to a *transparent log server* using an *authenticated data structure* [18] that is append only and can be efficiently audited. The key server acts as a *prover* who returns public keys upon request along with compact proofs that can be verified by clients. Thus, clients do not worry about MITM attack as any attempt to modify client keys is recorded on the auditable server log.

Authenticated data structure ensures that the server cannot change user keys without being recorded. It is, however, possible for a compromised key server to *equivocate* by presenting different answers to different clients. Therefore, log clients need a way to *cross validate* the received information to ensure the key server *consistency* among clients. This process is called *auditing*. There are third-party clients (*auditors*) who *frequently* query the key server for proofs. Thus, whenever clients receive replies from the key server, they can cross check the proofs with these auditors. State of the art suggests the use of a *gossiping protocol* among log clients and auditors to exchange information and effectively blacklist any exposed compromised key server.

However, such gossiping mechanism is hard to implement in practice [3]. It is vulnerable to certain classes of failures when attackers are present in the network i.e. Sybil attack [10]. It is hard to incentivize clients to participate and bootstrap the gossiping network. Users' privacy may also be at risk [20]. So far, we are not aware of any complete gossiping protocol design in current Transparent log systems. A similar effort in Certificate Transparency [15] is being standardized though after several years, and it is still not finished. Rather than using a separate gossiping protocol, EthIKS [4] implements the transparent log server on Ethereum blockchain [23]. However, as EthIKS operation cost increases proportionally with the number of users and due to the significant increase in the price of ETH, the system does not scale to large key servers with millions of users.

Auditing is a mandatory mechanism to secure a transparent log scheme. However, proposed auditing mechanisms using gossiping are vulnerable and difficult to implement. Meanwhile, blockchain based auditing is considered too expensive to operate as demonstrated in EthIKS. To tackle this problem, we present ***Trusternity***, a practical transparent log auditing scheme using blockchain that is secure, easy to implement, suitable for large scale key servers, as well as lightweight for clients. The contributions of this paper are the following:

- We design Trusternity, a secure, scalable auditing mechanism using a blockchain to ensure key server consistency. Our scheme is complete and more cost effective in comparison to state-of-the-art approaches.
- We implement a proof-of-concept for Trusternity using Ethereum and extending a state-of-the-art solution.
- We simulate an attack that deceives clients to accept a compromised blockchain and provide metrics to help detection of such attack.

2 Requirements

We now define several requirements for our auditing system.

R1 Trustless auditor: The system must be able to detect anomaly in auditing process even when clients connect to malicious auditors.

R2 Scalability: The system is scalable with unbounded number of servers and clients. For this, the system must satisfy the following sub requirements.

- **R2.1** *Incentive:* Auditors must be incited for their service of querying key servers and answer client requests.
- **R2.2** *Budget operation:* We want to reduce the operation cost of the key server when participating in the auditing process.
- **R2.3** *Thin client:* The auditing mechanism must not require extensive client resources so that it can be easily adopted in practice.

3 Background and related work

In this section, we shortly describe some background notions and related work.

3.1 Key Transparency

Key transparency brings autonomous key verification to end users in order to eliminate the need to fully trust a key server. Melara et al. introduced CONIKS [17], the first key transparency scheme which also preserves user privacy. Google Key Transparency [13] and Yahoo End-to-End [26] rely on this approach.

A CONIKS system includes three major components: (1) a CONIKS server managed by an *Identity Provider* (IP) that stores bindings between user identities and their public keys, (2) CONIKS clients which run on users' devices to manage cryptographic keys and (3) auditors who help clients to verify IPs consistency.

A CONIKS server uses a *Merkle radix tree* to map each user to his public key in a *binding*. The index path of each binding in the tree is randomized based on the user identity. At every fixed period of time (called an *epoch*), the CONIKS server signs the *root* of the Merkle tree to create a *Signed Tree Root* (STR) value. A STR_t at epoch t is also hashed together with STR_{t-1} to form a hash chain of the entire history of the key server. The server then *publishes* STR_t to all clients and auditors. When a client queries for a public key, the CONIKS server returns the chain of STR values, the binding at the leaf and an *authentication path* from the leaf to STR to prove that the binding exists in the tree. The client can cross validate STR value with any auditors to validate the CONIKS server answer.

CONIKS guarantees two security properties:

S1 No malicious keys: At every epoch t, a client looks up its own binding on the server by performing a *monitor* operation. Thus, an IP cannot insert malicious keys binding for users without being detected.

S2 Non-equivocation: After monitoring, the client queries STR_t from auditors via an *auditing* protocol. Thus, an IP cannot provide different answers to different user queries without accepting a high risk of being exposed. In case that an IP and auditors collude to equivocate a client, Melara et al. [17] shows that by choosing randomly 4 auditors, a CONIKS client can discover a malicious server with 99,7% probability.

As CONIKS data structure is very efficient and privacy-preserving, our solution extends CONIKS by replacing its auditing mechanism.

3.2 Gossiping

In CONIKS execution model, an IP and clients need to *disseminate* STR in every epoch to ensure that the key server does not equivocate different STR to different clients. CONIKS suggests a decentralized *gossiping* protocol for this purpose. In this protocol, all IPs act as auditors for each other. For example, $Alice@foo.com$ can perform audit with $bar.com$ while $Bob@bar.com$ can audit his key server with $foo.com$.

However, such gossiping network is hard to design in practice where there are potentially millions of IPs and clients. The protocol has to be decentralized so that the system does not depend on any single trust party. Each IP needs to broadcast his STR at every epoch to all other parties and has to answer random queries from any clients. There is no incentive for IPs to provide such extra overhead of communication bandwidth or for third party auditors to query an IP and to answer to random clients.

Another limitation for the gossiping network is the epoch time. CONIKS suggests an epoch time of one hour. This period depends not only on the computational power of each key server, but also the efficiency of the gossip protocol. The longer the epoch time is, the longer it takes for a client to register or revoke its key to the system, hence the longer the vulnerability window for an attack is. Meanwhile, shorter epoch time will increase the communication traffic in the gossiping network.

Finally, a decentralized gossip protocol is still vulnerable to network partition attack. An attacker can isolate a client from honest gossiping nodes to trick the client to accept compromised results from the server. Similarly, an attacker can plague the network with a great number of malicious nodes to increase the possibility that the client will connect to his nodes (*Sybil attack*).

A good example for the challenging aspect of this situation is the standardization process for gossiping protocol in Certificate Transparency [20] where browsers, auditors and certificate authorities gossip about the root hash of Certificate Transparency log. The standard has been on discussion for several years but it is still not finalized.

To sum up, while the proposed gossiping network protocol in CONIKS and similar systems is necessary for auditing CONIKS key server, such solution is hard to implement in an efficient, scalable, Sybil resilient and incentive manner.

3.3 Blockchain

Blockchain [19] is an append-only list of *blocks* where each block is linked directly to the previous one with cryptographic hashes. A blockchain system operates using a *peer-to-peer* (P2P) architecture where peers create and exchange *transactions* to modify the *state* of the system. Those transactions can hold different data types from financial records [6] to arbitrary code execution instructions [23].

Wust et al. [25] show a methodology to determine how a blockchain system solves various technological problems. Indeed, we can simplify a set of requirements for an auditing method as follows. First, a CONIKS server needs to disseminate a *state* (which is encapsulated in a STR) every epoch. There should be multiple servers that can disseminate information asynchronously since a user might have different accounts at different IPs. There is also no trusted third party in the network. Wust methodology shows that blockchain – either permissioned or permissionless, depending on whether only authorized set of entities or any entities can read or write the blockchain respectively – is the suitable solution for those requirements. In this paper, we consider a generic auditing mechanism that allows any untrusted CONIKS server to participate. Therefore, we choose permissionless blockchain, in particular, Ethereum as the underlying platform.

Ethereum is one of the major permissionless blockchain systems in the world besides Bitcoin. Ethereum uses blockchain as a ledger of transactions where a sender deposits coins (money) to a receiver. The sender signs the transaction with his private key and gives the ownership of the coin to the receiver so that later the receiver can redeem the received coin for subsequent transactions. A main issue addressed by the system is how to avoid sender generating invalid transactions of double spending a coin to two different receivers. Due to the decentralized nature of the system, the two receivers might not know each other, thus blindly accept the invalid transaction.

Similar to Bitcoin and some other blockchain solutions, Ethereum resolves this issue using *Proof-of-work (POW)*. A group of *miners* participating in the system uses their computing power to solve a puzzle in a form of exhaustive search at a given *difficulty*. The first miner who solves the puzzle can create a block consisting of a set of pre-selected transactions and broadcast the solution. Other miners validate the block and move on to solve next puzzles. In the case that a sender attempts to double-spend, only transactions chosen by the winning miner will be considered valid.

In case there are multiple forks of the chain, a miner always chooses the *longest chain*, i.e. the chain with the highest accumulated difficulty, to work on the next block. Thus, after some time, the whole network will abandon shorter forks. According to this consensus rule, the longer a block stays in the blockchain, the harder it is to be discarded by other miners. An alternative fork would have to solve all puzzles starting from the mentioned block to the end of the chain. Unless there is a party who owns more than 50% of the computing power of the whole Ethereum network, it is impossible for somebody to always produce a longer chain.

We chose Ethereum as our underlying platform as, in contrast to Bitcoin, it features a Turing-complete virtual machine that can execute scripts defined by users in various *smart contracts* and submitted inside a transaction. The script, along with the transaction, is permanently included within the blockchain unless it is scripted to self destruct at some point. Users can execute functions in the script by sending other transactions to the contract with appropriate *transaction fee* and parameters. These transactions are validated and executed by

all Ethereum clients. The transaction fee is calculated by an internal unit called *gas* and then paid by the sender in *ETH*. In this paper, we use an exchange rate of €500 for 1 ETH (as in January 1st 2018).

3.4 EthIKS

EthiKS is the first contribution that proposes using a blockchain to enhance CONIKS. EthIKS implements a CONIKS server in an Ethereum smart contract. In particular, the smart contract stores the Merkle tree in the persistent storage of Ethereum. The server can update the tree by executing the smart contract, while a client can query public keys by extracting data from the blockchain storage. As EthIKS clients have the same view of the key server within the blockchain, no separate gossip protocol is needed for key server consistency.

However, EthIKS introduces several inconveniences to the original CONIKS scheme. In order to fully trust the blockchain, EthIKS clients must download and validate every single transaction from the genesis block to the most recent block which is around 100GB. This is in contradiction with our **R2.3** requirement. Although Ethereum *light client* can significantly reduce the bandwidth amount, EthIKS must trust a third party to deliver the lightweight block header (see **R1**).

Moreover, EthIKS server operates entirely on Ethereum smart contracts. Every operation affecting the key server database has to be recorded in a transaction. EthIKS claimed that those transactions are relatively cheap (e.g. approximately €0.0004 for an insertion, not including the mandatory transaction fee). However, for a large size key server (million users) with high key change frequency, it will introduce great additional cost for the Identity Provider. Thus, **R2.2** is not satisfied.

4 Architecture

We present the architecture design and implementation of our proposed auditing scheme. As discussed above, we choose CONIKS data structure for our transparent-log server so we focus our proposal on the auditing scheme. Similar to EthIKS, we consider blockchain as an effective piggyback channel for such purpose. We also optimize the system so that server operation cost is kept at minimum. We develop ***Trusternity*** [9] and depict the general architecture in figure 1. The architecture contains four modules: *Storage*, *Smart Contract*, *Server* and *Client*. We explain each module in detail in the following subsections.

4.1 Storage

We consider Ethereum as an immutable distributed database. Thus, we can use Ethereum to store and distribute STR to all clients. We consider *transaction log* for this purpose. A transaction Log L is a collection of *Log entry* l which is the result of the code execution in Ethereum virtual machine (EVM) and can be recomputed at anytime by re-executing the code stored in the blockchain.

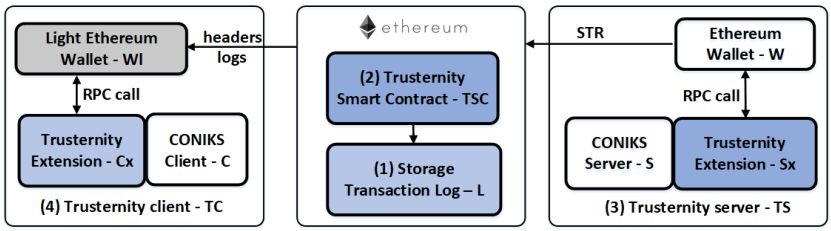

Fig. 1. Trusternity architecture

Therefore, storing data in log costs only 8 *gas* per byte, 80 times less than storing data inside the smart contract as in EthIKS [23]. The downside of this method is that we cannot directly access log data from smart contracts. Yet, we designed our own smart contract to address this problem.

We also consider the hybrid approach where actual data is stored in an immutable data structure provided by third party services such as IPFS [2] while the smart contract only holds a reference pointer to data location. Although this method significantly reduces the blockchain storage cost, clients have to rely on third party services. Thus, we do not use this approach.

4.2 Trusternity Smart Contract

We develop a Trusternity smart contract TSC on Ethereum [1]. Each IP is mapped by its Ethereum wallet address in a map data structure *ProviderList*. We assume that each IP only uses one address to create and sign transactions. The smart contract exposes two main functions: *Register* and *Publish*. *Register* accepts server name and related meta-data to insert into *ProviderList* then set it's *lastepoch* as 0. At each epoch, a registered server calls the *Publish* function by sending an *epoch* number and a 32 bytes STR. A key server must not be able to publish different STR for the same epoch or modify the previous ones. It also must not be able to publish STR in different sequence order to limit client difficulty in tracking and ordering those values. To use transaction log storage, we defined event *Published* which is fired at every *Publish* function call. The event is indexed by two *topics*, i.e. the sender address and epoch number while STR is kept in data field.

4.3 Trusternity server

A Trusternity server TS is a transparent key server that enables auditing via Ethereum. TS consists of three components: a *CONIKS Server* S, a *Trusternity extension* for server Sx and an Ethereum wallet W. S is the original CONIKS server. A CONIKS server handles registration, look-up and monitoring keys operations. At every epoch, the server automatically recalculates its Merkle Tree

[1] The full source can be found at `https://github.com/coast-team/trusternity-contract/blob/master/src/trusternity_log.sol`

database. We then developed Sx as a plugin for S. The extension allows S to communicate with W, the official Go implementation of the Ethereum protocol [1], via a RPC API. In every *epoch*, TS sends an Ethereum *transaction*, embedded with STR, to a smart contract on the blockchain network.

1. **Register:** Sx calls smart contract *Register* function.
2. **Calculate STR:** As defined in CONIKS.
3. **Get $lastEpoch$:** Sx checks last epoch from TSC. Though this step is optional, it helps Sx to avoid sending duplicate transactions blindly to TSC. Sx then performs a check to make sure that the server is at the correct epoch e where $e = lastepoch + 1$.
4. **Publish:** Sx calls *Publish* function and sends the new STR_e to TSC.
5. **Canonical chain confirm:** It is required to wait for a certain number of blocks γ to avoid chain reorganization [24]. Currently, we set $\gamma = 5$. After γ block, Sx checks again if the transaction is correctly included in the chain.

4.4 Trusternity client

A Trusternity client TC is a key management software that a user runs on his computer. TC has three components: a *CONIKS client* C, a *Light Ethereum Wallet* Wl and a *Trusternity Extension* for client Cx. C performs public key registration and looks up other public keys by sending HTTP requests to S as designed in CONIKS.

We add an extension module Cx to C that handles public key auditing using Ethereum. The extension is configured to synchronize epoch time with the server and then it regularly performs look up and audits registered public keys. Unlike TS, TC uses a light Ethereum wallet that can significantly reduce local storage and network bandwidth concerning the blockchain. We also found that a light wallet for client is enough to secure Trusternity scheme. The auditing process involves three steps as follows.

1. **Register:** As defined in CONIKS.
2. **Lookup:** When TC enables auditing with Trusternity, TC periodically performs public key lookup operation with its own identity (i.e. email) and validates the authentication path.
3. **Light chain lookup:** After validating the authentication path and the public key, TC follows light wallet look up protocol to find the corresponding STR' of the server in that epoch. This value is then compared to the received STR from step 2.

4.5 Light Ethereum Wallet

In a cryptocurrency scheme, in order to fully trust the blockchain, a client must download and validate all transactions starting from the genesis block. Currently, an Ethereum wallet must download around 100 GB data. This type of client is called a full client/wallet which we run on Trusternity server. However, for a client

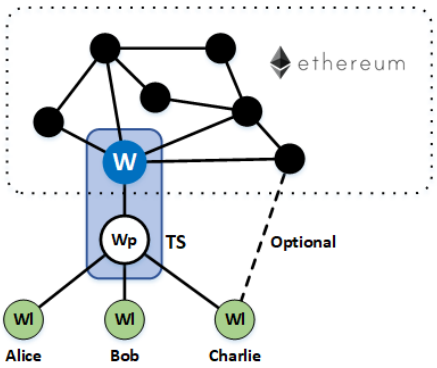

Fig. 2. Trusternity deployment with a proxy wallet

who is not interested in cryptocurrency, it is hard to force him to download and store all transactions just to extract the log from some particular transactions published by TS.

A *light client*, on the contrary, only downloads block headers and transactions filtered with requests from the client. Ethereum light client protocol is specified in [8]. As stated in the specification, an Ethereum light client can efficiently "watch" for events that are logged by TS by filtering transactions tagged only with log topics $IP.Adr$ and e.

According to the protocol specification, using a light client does not offer full security function of a blockchain. In fact, Wl cannot check if a downloaded block header $\mathcal{H}$ is completely valid or not. Wl can only make sure that $\mathcal{H}$ contains a valid POW result. Wl also does not have access to other information in a full block such as the block state tree. As Trusternity does not use that information, we do not need the state tree. Yet, the described security limitation of Wl is a great concern. Section 5 will discuss this problem in detail.

4.6 Deployment architecture

As Trusternity works over Ethereum, we want to make sure the system does not create undesirable impacts to the Ethereum ecosystem. We assume that there are thousands of W and millions of corresponding Wl connected over Ethereum P2P network. A light client does not, or cannot, relay data like a full node, yet it consumes bandwidth from other full nodes. Thus, it reduces the network throughput.

For this reason, we propose that each TS hosts a centralized proxy service Wp that relays Ethereum block headers and relevant transactions from TS to other Wl. Wp can be replicated and balanced so that there is no bottleneck in system availability. Optionally, if needed, Wl can also participate in the public Ethereum network. We depict the deployment architecture in figure 2.

From a security point of view, Wl should trust Wp in the same way as Wl trusts any other full client in the public Ethereum network. The only difference

here is that Wp is hosted by IP as a way to improve the availability of Trusternity without damaging the Ethereum ecosystem.

5 Security analysis

In this section we analyze the security of Trusternity in terms of requirements defined in section 2.

Trusternity uses S and C from CONIKS. Thus, we retain security requirement **[S1]** from CONIKS. For **[S2]**, if we assume Ethereum blockchain is trustworthy, the auditing is then similar to that in CONIKS where auditors are Ethereum clients. However, if an adversary Adv compromises TS, it is possible that the adversary presents TC a fake blockchain. We now present several scenarios where Adv can perform such attack.

5.1 Scenarios

Let us consider a scenario where $Alice$, Bob and $Charlie$ use a key server TS as in figure 2. An adversary Adv compromises TS and wants to perform MITM attack against the 3 users. Thus, besides an honest TS, Adv maintains a compromised TS' where he keeps $< Alice, PK_{Adv} >$. At epoch e, TS sends a $Publish$ transaction T_e to Ethereum blockchain ($MainNet$) for $Alice$ to monitor while sends T'_e from TS' to a fake blockchain. Adv then sends this fake chain to Bob to trick him into accepting PK_{Adv}.

As in figure 2, if Adv can compromise Wp, he will succeed in tricking Bob into accepting a compromised blockchain. Thus, Bob will have no way to detect the attack. However, if the user already has connection to other Ethereum nodes in MainNet as $Charlie$, the situation is more complex. First, Adv can compromise several Ethereum full nodes and find a way to redirect $Charlie$ to those compromised nodes. This is sometimes referred as *eclipse attack* [16]. Secondly, if Adv can hijack $Charlie$'s network connection, he can simply block all connections to honest nodes except to Wp. Lastly, Adv can try to perform Sybil attack on MainNet. Nevertheless, we see that Adv has various ways of tricking a user to connect a compromised full node and accept a compromised blockchain.

Detecting a malicious blockchain while connecting to an untrusted full node is a mandatory step to satisfy both **[S2]** and **[R1]** requirements. We now simulate scenarios when a client is fed on a malicious blockchain and present how to detect such problem.

Recall from section 3.3 that miners run a POW algorithm to solve a computing puzzle at a given block difficulty d_n of block number $n > 0$ at t_n. d_n is calculated based on d_{n-1} and the *time interval* $\Delta_{t_n} = t_n - t_{n-1}$ [7]. The calculation function is tuned so that the average block time of the Ethereum network is around 17 seconds. For example, d_n increases if $\Delta_{t_n} < \Delta_{t_{n-1}}$ and decreases otherwise. d_n, t_n and the accumulated difficulty $\sum_d$ of all blocks in the chain is stored in each block header. Therefore, if Adv controls all of the neighbor nodes of Wl, Adv must provide a malicious, yet valid blockchain to Wl. Assume that Wl has access

to a portion of MainNet from block 0 to block $m-1$ until Adv decides to fork into a fake chain. This can be achieved by hard coding *checkpoint blocks* in Cx. If the adversary does not have enough computing power, he cannot solve POW with the honest chain difficulty as fast as MainNet. Thus, the client will observe significant increases in block time interval and drops in block difficulty. We then conduct a simulation on a private Ethereum network to simulate this scenario and propose our method to *automatically* detect the attack on the client side.

5.2 Attack simulation

We deploy a private Ethereum network of 40 miners on a testbed system. All miners are connected to a *bootnode* which helps bootstrapping the peer-to-peer network. We then study the distribution of MainNet mining pools [11] to have a brief understanding of of a potential adversary capability. There are many mining pools who process relatively large computing power (*Hashrate*) in comparison to the rest of the network. Our simulation is based on the assumption that an adversary can compromise one of the pools and use its computing power for a brief period of time to conduct the attack. We define $p > 0$ as the capability of Adv over total computing power of MainNet. Since Adv should not have more than 50% computing power of the whole network, $p < 0.5$. Our experiment consists of two phases:

1. **Stable:** We run a fresh private Ethereum network with all 40 miners from our genesis block beginning with d_0 of block 0. We run this phase for 1 hour to produce a base chain of a stable Ethereum network where all nodes are honest. Our simulation script automatically switches to the second phase after 1 hour.
2. **Malicious:** Instead of keeping running all 40 miners, we only keep m miners for an additional hour where $m/40 = p$. As in [11], the biggest mining pool has around 25% of MainNet computing power. Thus, with $m = 10$, we can simulate the situation when this pool is compromised. We repeat the simulation varying m range from 1 to 19 multiple times.

We did not choose to simulate this attack on any official Ethereum test network (e.g. ropsten[2]) because our simulated attack could cause temporary forks in the network that may harm experiments from other parties.

5.3 Results

In all experiments, we are interested in the aberration of d and Δ_n. Figure 3 shows two sample results from our experiments with $m = 4$ and $m = 10$ respectively. $m = 10$ can be interpreted as the adversary has 25% of the total network Hashrate by compromising the biggest mining pool, i.e. ethpool [12] while $m = 4$ is when the secondary biggest pool with 10% Hashrate is compromised. d is collected

[2] `https://ropsten.etherscan.io/`

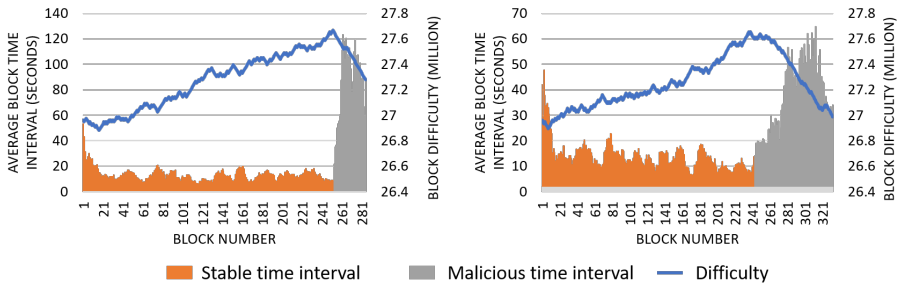

Fig. 3. Average block time interval and difficulty where $m = 4$ ($p = 10\%$) and $m = 10$ ($p = 25\%$) respectively

directly from the chain header and is presented with a blue line. The average block time interval $\bar{t}$ is calculated as in equation 1 where we set $l1 = 10$. We will explain the rationale of choosing $l1$ in the next section.

$$\bar{t}_n = \frac{\sum_{i=0}^{l1-1} \Delta_{t_{n-i}}}{l1} \tag{1}$$

In both cases, we observe the immediate change in the trend of d and $\bar{t}$ when the malicious phase kicked in. $\bar{t}$ is kept below 20 seconds in the stable time then increased significantly after block 241. On the other hand, d experienced a dropping trend after block 241 due to the increase in time between blocks. However, it is not trivial to *automatically* distinguish between malicious attempt and an occasional fluctuation of the result.

5.4 Detection of Malice

Given the presented results, we want to detect the block when the adversary forks the chain as soon as possible. Our general idea is to detect anomalies in the change of d and $\bar{t}$ over time. Several approaches of anomaly detection in multi-time series data have been proposed [14]. However, due to **R2.3**, we follow a simple approach in anomaly detection. We first analyze 4 million Ethereum blocks in MainNet. For a block i on MainNet, we observe that $\bar{t}_i$ is less than 20 seconds in 92% of the time. We also find that d_i never decreases continuously more than 20 times.

Assuming we start monitoring blockchain header at block i, all previous blocks are trusted. We then design an adaptive algorithm to detect the malicious in real-time manner as presented in algorithm 1. We introduce 4 parameters:

$l1$: Number of blocks to compute average block time.
$l2$: Number of blocks to compute difficulty decrease streak.
$\langle t \rangle$: Mean of Δ_t from block 0 to the most recent trusted block.
λ: Longest decrease times of Δ_d over $l2$

Algorithm 1: Detection of Malice

input : A block header H_i contains t_i and d_i
output : `true` if H_i is malicious, else `false`

1 *Constant:* $l1$, $l2$, λ, $\langle t \rangle$;
2 $\bar{t}_i \leftarrow$ equation 1 ;
3 $S1 \leftarrow \sum_{k \leftarrow 0}^{l1-1} \bar{t}_{i-k} / \langle t \rangle$; `//` $l1$ `block before` i
4 $\Delta_{d_i} \leftarrow (d_i - d_{i-1}) < 0$; `// 1 iff true, 0 iff false`
5 $S2 \leftarrow \sum_{k \leftarrow 0}^{l2-1} \Delta_{d_{i-k}}$;
6 **return** $(S1 > 2 * l1) \vee (S2 > l2 * \lambda)$;

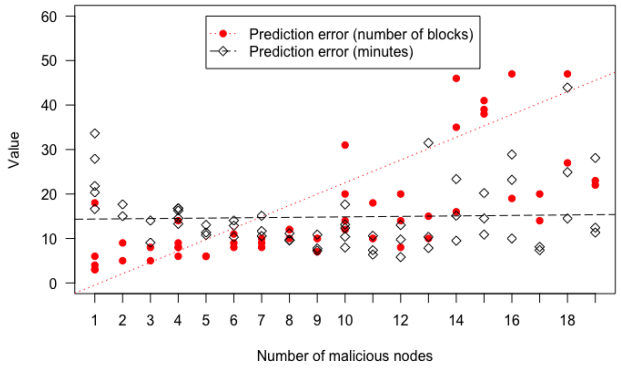

Fig. 4. Prediction error.

The idea of $l1$ and $l2$ is to smoothen out the block time and difficulty value. If they are too small, we cannot eliminate the risk of true negativity for detection. Meanwhile, large values mean that we might not detect the attack early enough after it happens. Our aim is to keep the time required for detection at a reasonable length, i.e. $\approx$ 15 minutes. We find those values by permuting a set of possible parameters and rerun the algorithm. We achieve a result where $l1 = 10$ and $l2 = 20$. In MainNet we can set $\langle t \rangle$ at 17.5 seconds and 13 in our private net.

Figure 4 presents our detection result when the number of malicious miners change. The figure shows the prediction error, i.e. how much time or number of block we need after the malicious miner appears to detect the attack. Two dashed lines in Figure 4 represent the regression line, i.e. to show the trends of prediction errors when the number of malicious participants change.

We see that the number of needed blocks increases almost linearly when the number of malicious miners increases, yet the needed time shows almost no difference ($\approx$ 15 minutes as intended) due to the decrease in average block time when there are more malicious miners in the network. We note that our prediction results are acquired in conjunction with analyzing past 4 million Ethereum blocks on MainNet. Thus, while we cannot guarantee 100% confidence

in detecting future attacks, we have a strong base of trust in the method if the network continues to behave as in the past. Obviously, in case there are events that significantly affect block time and difficulty such as a hard fork or natural disasters, our algorithm will yield true negative results. However, it makes sense to notify users when such events happen since Trusternity depends on Ethereum.

6 Evaluation

In this section, we show a thorough evaluation of Trusternity with regard to network bandwidth overhead and operating costs in comparison to CONIKS and EthIKS. We also suggest various options to apply Trusternity into other systems.

6.1 Network overhead

We reuse most of setup and assumptions from CONIKS and EthIKS. In particular, Trusternity client uses the eliptic-curve based VUF and signature scheme. There are total $U = 2^{32}$ users, $u = 2^{21}$ users update their keys per epoch and $k = 24$ epochs per day. Ethereum block time average is set at a lower bound of 12 seconds.

However, our calculation [3] shows that an Ethereum client has to download $\approx 0.6KB$ for each header instead of only $0.2KB$ per block header as in EthIKS scenario. We slightly modify EthIKS calculation to reflect this change and calculate our result for Trusternity in table 1.

Overall, we see there are no change in look up and monitor cost compared to CONIKS since the calculation separates blockchain into auditing section. An EthIKS full client has to download all α transactions related to EthIKS. This assumption is rather complicated since we do not have the source code of EthIKS smart contract, however, α should be proportional to the number of updates to the contract per epoch. Thus, in a naive assumption, we can have $\alpha \approx u$. We also cannot compare to a EthIKS light client since the author assumes that the light version has a trusted source to query for Ethereum block header and data.

In summary, we can see that Trusternity only adds a fixed amount of bandwidth overhead per epoch of 200 KB, result in less than 5MB per day to operate in comparison to the original CONIKS client. We calculate this number entirely based on Ethereum formal specification so the actual amount might be slightly different due to encoding, extra protocol messages or bloom filter false positive result. However, our calculation shows a clear advantage of our approach to EthIKS over network bandwidth.

6.2 Gas cost

The transaction costs of operating Trusternity only come from the two listed functions on the smart contract. Overall, *Register* costs 63,000 *gas* and *Publish*

[3] `https://github.com/coast-team/trusternity-contract/blob/master/appendix/calculation.md`

Table 1. Client bandwidth requirements (KB) with α is the number of transaction in each epoch

Operation	CONIKS	EthIKS	Trusternity
lookup (per binding)	1.2	7.9	1.2
monitor (per epoch)	0.7	5.2	0.7
monitor (daily)	17.6	1405	17.6
audit (per epoch)	0.1	$200.4 + \alpha * 2$	201.6
audit (daily)	2.3	$4809.6 + \alpha * 48$	4838.5

costs 44,000 gas. We take the assumption from section 4.1 which is $0.0004 ETH$ per 20,000 gas and each ETH costs €500. This results in *Register* costs €0.63 which an IP only has to pay once when he installs Trusternity and €0.44 for each *Publish* call per epoch or ≈ €10.56 per day. Comparing our results to EthIKS, assume that we only take into account 2^{21} update mapping transactions of 12,000 gas per epoch, this costs ≈ €6 million per day.

6.3 Final result

We compare our implementation to the pre-defined requirements from section 2. **S1** and **S2** are satisfied as we discussed in section 5. Regarding **R1**, our anomaly detection algorithm can effectively detect a fork attack from a sub network of malicious Ethereum miners in less than 15 minutes, assuming that the malicious network only has less than 50% of the main network computing power. Thus, even with untrusted auditors, Trusternity is still able to detect malicious behaviors after a short period of time.

We also show above the improvement over network bandwidth and gas cost overhead of Trusternity over EthIKS. Our scheme adds a flat amount of €10 per day for an Identity Provider to operate regardless of the client amounts (**R2.2**). Each Trusternity client only has to download a merely extra 5 MB every day. Although our anomaly detection algorithm requires clients to continuously monitor the downloaded Ethereum block header, the algorithm is simple enough to not cause any noticeable overhead for clients. As a result, **R2.3** is satisfied. Lastly, Trusternity operates on Ethereum, any Ethereum client can be considered an auditor, including W. Thus, **R2.1** is trivial to achieve.

7 Conclusion

We presented Trusternity, an auditing mechanism for Transparent-log key server using Ethereum which is significantly more efficient and budget than state-of-the-art approach. Our solution scales with an unbound number of log clients, cheap to operate (€10 per day for the server) and does not require huge network bandwidth or storage of clients. Our solution is also independent of any trusted third party by being able to detect malicious sudden change in the network. Trusternity is also easy to extend for other purposes. Other transparent log based approaches such as Certificate Transparency [15] can also benefit from our

proposal. CONIKS client and server components are also replaceable with similar components, i.e. Key Transparency [13] server and clients.

References

1. Go Ethereum: Official Go implementation of the Ethereum protocol. https://geth.ethereum.org/ (2017)
2. Benet, J.: Ipfs-content addressed, versioned, p2p file system. arXiv preprint arXiv:1407.3561 (2014)
3. Birman, K.: The promise, and limitations, of gossip protocols. ACM SIGOPS Operating Systems Review **41**(5), 8–13 (2007)
4. Bonneau, J.: Ethiks: Using ethereum to audit a coniks key transparency log. In: International Conference on Financial Cryptography and Data Security. pp. 95–105. Springer (2016)
5. Boyko, V., MacKenzie, P., Patel, S.: Provably secure password-authenticated key exchange using diffie-hellman. In: Advances in Cryptology - Eurocrypt 2000. pp. 156–171. Springer (2000)
6. Brito, J., Castillo, A.: Bitcoin: A primer for policymakers. Mercatus Center at George Mason University (2013)
7. vitalik buterin: Homestead hard-fork changes. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md (2015)
8. Buterin, V.: Ethereum light client protocol. https://github.com/ethereum/wiki/wiki/Light-client-protocol (2016)
9. COAST team: The Trusternity Project. https://github.com/coast-team/coniks-go (2017)
10. Douceur, J.R.: The sybil attack. In: International Workshop on Peer-to-Peer Systems. pp. 251–260. Springer (2002)
11. etherchain.org: Mining statistics. https://etherchain.org/statistics/miners (2017), accessed on 28.08.2017
12. ethpool: The Ethereum Solo Mining Pool . http://ethpool.org (2017), accessed on 28.08.2017
13. Google: Key Transparency. https://github.com/google/keytransparency (2017)
14. Jones, M., Nikovski, D., Imamura, M., Hirata, T.: Exemplar learning for extremely efficient anomaly detection in real-valued time series. Data Min. Knowl. Discov. **30**(6), 1427–1454 (2016)
15. Laurie, B.: Certificate transparency. Queue **12**(8), 10:10–10:19 (Aug 2014). https://doi.org/10.1145/2668152.2668154, http://doi.acm.org/10.1145/2668152.2668154
16. Marcus, Y., Heilman, E., Goldberg, S.: Low-resource eclipse attacks on ethereum's peer-to-peer network. http://www.cs.bu.edu/~goldbe/projects/eclipseEth.pdf (2018)
17. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: Coniks: Bringing key transparency to end users. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 383–398 (2015)
18. Miller, A., Hicks, M., Katz, J., Shi, E.: Authenticated data structures, generically. In: ACM SIGPLAN Notices. vol. 49, pp. 411–423. ACM (2014)
19. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)

20. Nordberg, L.: Gossiping in CT. https://tools.ietf.org/html/draft-linus-trans-gossip-ct-00 (2014)
21. WhatsApp: WhatsApp Messenger. https://www.whatsapp.com/ (2017), accessed on 28.08.2017
22. Whitten, A., Tygar, J.D.: Why johnny can't encrypt: A usability evaluation of pgp 5.0. In: USENIX Security Symposium. vol. 348 (1999)
23. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**, 1–32 (2014)
24. Wood, G.: Chain Reorganisation Depth Expectations. https://blog.ethereum.org/2015/08/08/chain-reorganisation-depth-expectations/ (2015), accessed on 25.09.2017
25. Wüst, K., Gervais, A.: Do you need a blockchain? https://eprint.iacr.org/2017/375.pdf (2017)
26. Yahoo: Yahoo End-To-End. https://github.com/yahoo/end-to-end (2017)
27. Zimmermann, P.R.: The official PGP user's guide. MIT press (1995)