Trusternity: Auditing Transparent Log Server with Blockchain

Hoang-Long Nguyen, Claudia-Lavinia Ignat, Olivier Perrin

Trusternity: Auditing Transparent Log Server with Blockchain

Hoang-Long Nguyen, Claudia-Lavinia Ignat, 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, transparent log systems such as CONIKS employ 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 not scalable. This paper presents *Trusternity*, an auditing scheme relying on Ethereum blockchain that is easy to implement, scalable and inexpensive to operate.

KEYWORDS

blockchain, key transparency, auditing, Ethereum

1 INTRODUCTION

End-to-end encryption (*E2EE*) has become more popular over the years due to the increase in public awareness about online privacy and the dangers of digital snooping or identity and data theft. A major challenge in E2EE system is to prevent Man-in-the-Middle (*MITM*) attack where an adversary impersonates a legitimate communication participant. Currently, popular E2EE systems (e.g. WhatsApp) adopt the use of trusted *key servers* to distribute and authenticate public keys among clients to prevent MITM. However, such servers can be vulnerable to attacks from adversaries or surveillance requests from authorities.

While it is difficult to preemptively protect a key server, it is possible for the clients to, later on, check if the key server behaved correctly during communication. Such technique is called *Key transparency* [5] [8]. The general idea is that the key server maintains a *transparent log* using a Merkle Tree that is append-only and can be efficiently *audited*. To audit the server, clients request a compact proof from the server to show that their uploaded public keys are not modified. Any attempt to modify client keys will be recorded on the server log and a client can check if the server is behaving maliciously. Also, to prevent a compromised server to present different keys and proofs to different clients, the auditing process includes a *separated gossiping protocol* among clients to cross-validate the proofs.

WWW '18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3186938

However, such gossiping mechanism is hard to implement in practice. It is vulnerable to certain classes of failures when adversaries are present in the network [4]. It is also hard to incentivize clients to participate in gossiping. A similar effort in Certificate Transparency [7] is being standardized though after several years it is not yet finished. Rather than using a separate gossiping protocol, EthIKS [2] implements the transparent log server on Ethereum blockchain [9]. 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.

In this paper we present ***Trusternity***, a practical transparent log auditing scheme using blockchain that addresses shortcomings of state-of-the-art approaches. We use blockchain as an immutable storage for proofs required in the auditing process. We also optimized cost by avoiding computation on-chain. Therefore, our scheme is secure, easy to implement, suitable for large scale key servers, as well as lightweight for clients.

2 APPROACH

We developed Trusternity as an extension to CONIKS [6], the first transparent log key management solution for end users. In CONIKS, a user runs a client software to upload their public keys to a CONIKS server. Our Trusternity extension allows the CONIKS server to disseminate a public verifiable proof to the blockchain. When a client downloads public keys from the server, it can cross-check this proof value from the blockchain with the downloaded data to ensure the integrity of the key server. We depict Trusternity architecture in Figure 1. Our system contains four modules: *Server*, *Smart Contract*, *Storage* and *Client*.

(1) 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. As originally designed in CONIKS, when a user registers his public key with TS, S uses a *Merkle radix tree* to map the user-key binding at a leaf. After a fixed period of time (an *epoch*), S signs the *root* hash of the tree to make a *Signed Tree Root* (STR). A STR_t at epoch t is also hashed together with STR_{t-1} to form an ever growing hash chain to commit the entire history of the key server.

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 an RPC API. At every *epoch* t, TS sends on the blockchain network an Ethereum *transaction* embedded with STR_t to a smart contract.

(2) Trusternity Smart Contract: We develop a Trusternity smart contract TSC on Ethereum. The smart contract exposes two main functions: *Register* that accepts TS registration and *Publish* that stores STR on each epoch. TSC enforces policy on how a TS

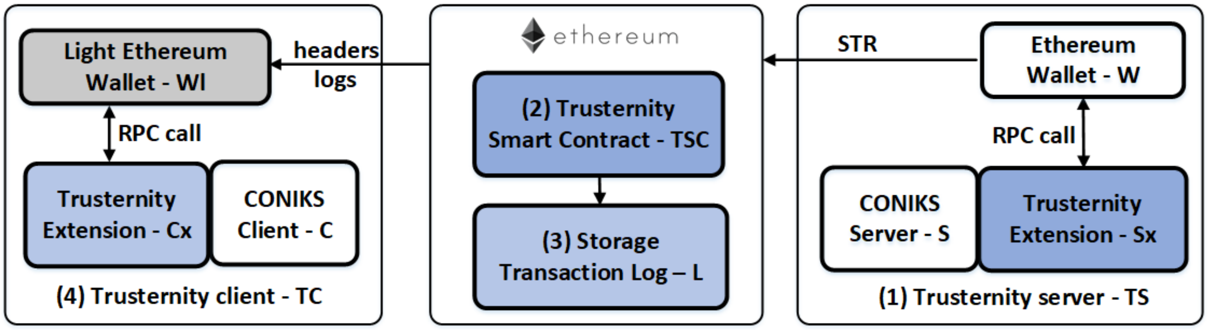

Figure 1: Trusternity architecture

can publish data such as epoch number uniqueness or sequence order.

(3) Storage: TSC does not store STR in the default smart contract storage. Instead, we use Ethereum *transaction log* for this purpose. A transaction log L is a collection of *log entry* l representing the result of the code execution in Ethereum virtual machine. L can be recomputed at anytime by re-executing the code stored in the blockchain. According to Ethereum yellow paper [9], the cost of storing a byte in the transaction log is 8 *gas* (an internal unit) compared to 625 *gas* for storing it inside the smart contract.

(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 synchronizes epoch time with the server, 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. While a light wallet cannot offer full security model as a full one, it is able to “watch" for *Publish* events with efficient verifiable proofs [3].

3 EVALUATION

We evaluated Trusternity and compared the results with EthIKS.

Security: Trusternity is an extension of CONIKS. Thus, we retain the proven security of the CONIKS client/server data structures and protocols. After each *Publish* call, TSC holds STR_e of epoch e on Ethereum transaction logs. If TS maliciously updates PK_{Alice} without informing $Alice$, $Alice$ can query the log to discover the change in STR. Even when TC uses a light wallet, the adversary can neither trick TC into accepting a fake transaction nor modify any published STR due to the immutability property of the blockchain.

Network overhead: We theoretically computed the client network overhead by reusing assumptions from EthIKS and Ethereum yellow paper. In particular, we assume a total number of users $U = 2^{32}$ and consider that $u = 2^{21}$ users update their keys every epoch. We also assume that $k = 24$ epochs per day. In our case, for each epoch, TC downloads $\approx 201.6KB$, resulting in less than 5MB per day to operate while an EthIKS client has to download the full blockchain in order to avoid relying on a trusted party.

Server cost: In order to send a transaction to the Ethereum network, TS must pay a transaction cost in ETH. Trusternity operation cost consists of the transaction costs of the two smart contrat functions *Register* and *Publish*. Using the ETH price from January 1^{st} 2018 ($\approx$ € 500), *Register* costs € 0.63 which TS only has to pay once during registration to TSC and € 0.44 for each subsequent *Publish* call per epoch or $\approx$ € 10 per day regardless of the size of the key server. This is because we only publish a 256-bit hash of STR at every epoch on Ethereum. Meanwhile, as EthIKS stores the whole Merkle tree data structure on the smart contract, EthIKS costs will increase proportional to the size of the tree.

4 CONCLUSION

We have presented Trusternity, an auditing mechanism for Transparent log key server using Ethereum which is significantly more efficient and budget than state-of-the art approaches. Our solution scales with an unbound number of log clients. It is efficient and does not require any trusted third-party. Trusternity is also easy to extend for other purposes. Other transparent log based approaches such as Certificate Transparency [7] can benefit from our proposal. For example, we can replace the CONIKS clients and servers with similar components, i.e. Key Transparency [5] servers and clients.

REFERENCES

[1] 2017. Go Ethereum: Official Go implementation of the Ethereum protocol. https://geth.ethereum.org/. (2017).
[2] Joseph Bonneau. 2016. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In *International Conference on Financial Cryptography and Data Security*. Springer, 95–105.
[3] Vitalik Buterin. 2016. Ethereum Light client protocol. https://github.com/ethereum/wiki/wiki/Light-client-protocol. (2016).
[4] John R Douceur. 2002. The sybil attack. In *International Workshop on Peer-to-Peer Systems*. Springer, 251–260.
[5] Google. 2017. Key Transparency. https://github.com/google/keytransparency. (2017).
[6] http://coniks.org. 2018. A CONIKS implementation in Golang. https://github.com/coniks-sys/coniks-go. (2018). Accessed on 08.01.2018.
[7] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. *Certificate transparency*. Technical Report.
[8] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*. 383–398.
[9] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.