# ON BUILDING TRUSTWORTHY NETWORK SYSTEMS WITH BLOCKCHAIN AND TEE

LI ZECHENG

PhD

The Hong Kong Polytechnic University

2022

The Hong Kong Polytechnic University

Department of Computing

# On Building Trustworthy Network Systems with Blockchain and TEE

Li Zecheng

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
July 2022

# CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

Signature: ____________________

Name of Student: Li Zecheng

# On Building Trustworthy Network Systems with Blockchain and TEE

by

Zecheng Li

Submitted to the Department of Computing
on June 2022, in partial fulfillment of the
requirements for the degrees of
Doctor of Philosophy
and
Master of Science

## Abstract

The Internet consists of many network systems, such as Domain Name System (DNS) and Public Key Infrastructure (PKI), that work together to provide network services and connect the world. However, these network systems suffer from a number of security issues, such as cache poisoning attacks on DNS and rogue certificates on PKI. These security risks can in turn lead to the proliferation of phishing sites, man-in-the-middle attacks on encrypted connections, and a host of other severe and complex network attacks. Traditional solutions still have limitations, and as we delve into blockchain and Trusted Execution Environment (TEE) technology, we find that their benefits can be leveraged to enhance the security of these network systems.

Blockchain technology was born from the cryptocurrency Bitcoin, whose tamper-proof nature catalyzes the secure exchange of assets. The decentralized architecture and replicated storage of blockchain guarantee the integrity and consistency of the stored data. They also provide a new way of building traditional network systems with guaranteed data security. In addition, TEE ensures execution security. Its model of attested execution allows users to verify the content returned by the enclave inside TEE and decide whether to trust the execution result. The combination of blockchain and TEE provides a new computing paradigm for building trustworthy network systems.

Firstly, we note that DNS is vulnerable to many attacks such as the cache poisoning attack and DDoS attack. Records in recursive resolver are vulnerable to be modified maliciously. Facing these problems, we propose B-DNS, a secure and efficient blockchain-based domain name system. B-DNS leverages blockchain to store resource records and provide name service. The tamper-proof feature of blockchain prevents it from poisoning attacks. B-DNS also fills up two shortcomings in legacy blockchain-based DNS: computation-heavy consensus protocol and inefficient query. For the security of B-DNS, a novel way is proposed to quantitatively compare the security of B-DNS and legacy DNS in terms of attack success rate, attack cost, and

attack surface. Our experiments show that the probability of a successful attack on B-DNS is 1% of a successful attack on legacy DNS. The attack cost goes up a million times in B-DNS, and the attack surface of B-DNS is far smaller than that of legacy DNS. The query performance evaluation of B-DNS shows that B-DNS can achieve similar or even less query latency than state-of-the-art commercial DNS implementations.

Secondly, we find that current Certificate Authorities (CAs) are vulnerable to be compromised to issue unauthorized certificates. Current countermeasures can only detect unauthorized certificates rather than preventing them. Facing these problems, we propose Pistis, a framework for issuing authorized and trusted certificates with blockchain and TEE. In Pistis, TEE nodes validate whether the applicant in a certificate request passes the domain ownership validation (i.e., the domain is under the corresponding applicant's control) and submit attested results to a smart contract on the blockchain. The smart contract issues the certificate to the applicant when an attested result shows a pass. Therefore, Pistis can ensure its issued certificates are authorized because of the domain ownership validation mechanism. The security of Pistis is formally proved in the Universally Composable (UC) framework. Compared with state-of-the-art, Pistis avoids potential damages by preventing unauthorized certificates from issuing.

Thirdly, we note that smart contracts cannot be modified once they are deployed on the blockchain, so vulnerabilities in deployed smart contracts can have devastating consequences. We emphasize that current countermeasures is to thoroughly test and validate contracts prior to deployment. However, these testing methods suffer from false-negative results and do not protect against unknown contract defects. Furthermore, Decentralised Finance (DeFi) based on smart contracts has gained significant momentum and is now attractive target for attacks. Facing these problems, we propose SolSaviour to protect deployed smart contracts and DeFi. SolSaviour consists of a voteDestruct mechanism and a TEE cluster. The voteDestruct mechanism allows contract stakeholders to decide whether to destroy the defective contract and withdraw inside assets. The TEE cluster is responsible for asset escrow, redeployment of patched contracts, and state migration. Specifically, SolSaviour can destroy the defective contract, redeploy a patched contract, and migrate the funds and state variables from the destroyed contract to the patched one. Our experiment results show SolSaviour can protect smart contracts and complex DeFi protocols with feasible overhead.

# Acknowledgments

I would first like to thank Dr. Bin Xiao, who is my thesis advisor. Dr. Bin Xiao gave me timely and careful guidance on the challenges I encountered during the writing process of my thesis. He made guiding comments and suggestions on the research direction of my thesis, put forward many helpful suggestions for improvement, and expended a lot of time and effort. I want to sincerely thank Dr. Bin Xiao for his assistance and concern for me!

I would also like to thank my collaborators in the research group. They are the comrades who struggle together on the road of scientific research, and they are also the friends who play together in life. We all learn from each other and help each other in the process of pursuing the Ph.D. degree. We spend a wonderful and unforgettable time together.

In addition, I would like to thank the thesis reviewers for their hard work. I would also like to thank the authors in the references for giving me a good starting point for my research topic through their research papers.

Finally, my heartfelt thanks to my parents. My five-year Ph.D. career has had its setbacks and ups and downs. Without their encouragement and support, I could not have successfully completed this dissertation.

# Previous Published Material

1. **Zecheng Li**, Bin Xiao, Songtao Guo, and Yuanyuan Yang, “Securing Deployed Smart Contracts and DeFi with Distributed TEE Cluster”, *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (Major Revision).

2. **Zecheng Li**, Shengyuan Chen, Hongshu Dai, Dunyuan Xu, Chengkang Chu, and Bin Xiao, “Abnormal Traffic Detection: Traffic Feature Extraction and DAE-GAN with Efficient Data Augmentation”, *IEEE Transactions on Reliability (TR)*, (Accepted).

3. **Zecheng Li**, Yu Zhou, Songtao Guo, Bin Xiao, “SolSaviour: A Defending Framework for Deployed Defective Smart Contracts”, in *Annual Computer Security Applications Conference (ACSAC)*, Virtual, 2021.

4. **Zecheng Li**, Haotian Wu, Lap Hou Lao, Songtao Guo, Yuanyuan Yang, Bin Xiao, “Pistis: Issuing Trusted and Authorized Certificates With Distributed Ledger and TEE”, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, issue. 7, 2021.

5. **Zecheng Li**, Shang Gao, Zhe Peng, Songtao Guo, Yuanyuan Yang, and Bin Xiao, “B-DNS: A Secure and Efficient DNS Based on the Blockchain Technology”, *IEEE Transactions on Network Science and Engineering (TNSE)*, vol. 8, issue. 2, 2021.

6. Laphou Lao, **Zecheng Li**, Songlin Hou, Bin Xiao, Songtao Guo, and Yuanyuan Yang, “A Survey of IoT Applications in Blockchain Systems: Architecture, Consensus, and Traffic Modeling”, *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, 2020.

7. Shang Gao, **Zecheng Li**, Zhe Peng, and Bin Xiao, “Power Adjusting and Bribery Racing: Novel Mining Attacks in the Bitcoin System”, in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, London, UK, 2019.

8. Shang Gao, **Zecheng Li**, Yuan Yao, Bin Xiao, Songtao Guo, and Yuanyuan Yang, “Software-defined firewall: Enabling malware traffic detection and programmable security control”, in *Proceedings of the 2018 Asia Conference on Computer and Communications Security (ASIACCS)*, Incheon, Korea, 2018.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The Internet is the cornerstone of the efficient functioning of modern society, where different countries and people are connected through hundreds of submarine fiber optic cables that span the oceans. Current Internet is filled with network systems responsible for different functionalities. For example, the domain name system (DNS) is responsible for locating the IP address based on the host name. The public key infrastructure (PKI) is responsible for managing, distributing, renewing, and revoking domain certificates. These network systems corporate to build the foundation for the proper functioning of the Internet.

Both DNS and PKI are built in a hierarchical tree architecture, i.e., both network systems rely on a root node. In DNS tree, the root node is the root domain . of the entire domain name space. There are 13 root server clusters responsible for maintaining the root domain, which are numbered in letters `A-M`. For example, root domain `A` and `J` indicate the root domain server clusters maintained by Verisign. Root name servers store the registration information of top-level domain (TLD) servers. Typically, DNS lookup requests are first sent to the root name server for the IP address of the TLD in the corresponding domain name. The queried root server returns a list of the authoritative name servers for the appropriate TLD. In the client side, service providers maintain recursive resolvers to store IP addresses of frequently queried domains, which decreases the DNS service latency by eliminating redundant queries. In PKI, a certificate authority (CA) is an entity that issues certificates to domain owners. In cryptographic terms, a certificate links the domain's identity to a public key. Certificates are not issued independently in PKI, but start with the root certificate, which is authorized by chains of trust, making the end certificates trusted. Root certificates are usually self-signed certificates, and they are pre-installed in applications (e.g., operating system, browser). A CA often relies on the trust of the root certificate to issue certificates for end users and sign them with their own private keys. In the Internet, DNS enables us to use network resources with different addresses, while PKI protects the security of information transmission.

There are still specific problems in the current network system. For example, the cache poisoning attack against DNS can pollute the resource records stored inside the recursive resolver through continuous injection, and then lead users to previously-built phishing websites for further theft. The PKI is centralized due to its internal CA. Attackers can issue rogue certificates by compromising the intermediate CA's private key. These unauthorized certificates can be authenticated through HTTPS connections. With the information of the certificates in hand, attackers can conduct man-in-the-middle (MitM) attack against the transmission flow between the user and the website, stealing the user's account password and other confidential information.

We point that the trust paradigm of current network systems are typically based on the identity. Users trust service provided by DNS and PKI based on the identity of service provider. However, this trust paradigm still has some limitations as it cannot provide defence to the malicious behaviours of naming/certificate service providers. A new trust paradigm is deemed to enhance the security of current network system.

## 1.1 The Emergence of Blockchain and TEE

A blockchain is essentially a growing transaction ledger maintained by distributed consensus. The ledger is stored in the data structure of a linked block, with each block storing a hash of the previous block's contents to ensure that data growth is tracked. The internal data load part of a block stores the transaction data on the ledger, usually in the form of a Merkle tree. The blockchain periodically elects a node to publish new blocks. Since each block holds information about previous one, all blocks form a unique chain and the blockchain is resistant to data tampering. As the difficulty of being successfully selected as the node to publish a new block increases, tampering with the data of the blockchain is gradually considered impossible. The underlying blockchain is implemented by a peer-to-peer network, where each node maintains a list of neighbors to exchange new blocks and transactions with each other. The blockchain technology has provided a new trust paradigm for network systems. Its decentralized and tamper-proof feature has led to the belief that the data stored in the blockchain is trusted, which provides a new direction for building trustworthy network systems.

Trusted execution environment (TEE) is another promising technology for building trustworthy network systems. In TEE, CPU divides a portion of the memory area to ensure that internally loaded code and data are protected in terms of confidentiality and integrity. Integrity means that software outside the TEE (e.g., operating system) cannot tamper with the inside code and data without authorization. Confidentiality indicates that entities outside the TEE cannot be aware of information inside the TEE without permission. These two properties are achieved by implementing secure and confidential architecture, such as Intel Software Guard Extensions (Intel SGX), which provides new CPU instructions that implement hardware-level memory encryption that isolates application-specific code and data in memory, namely the enclave. Even applications with higher privilege levels cannot compromise the integrity and confidentiality of the enclave. TEE execution results can also be verified following the method called remote attestation. The TEE technology has also provided a new trust paradigm for network systems. Users can trust the results from TEE as it ensures the integrity and confidentiality of loaded code and data, which provides a new direction for building trustworthy network systems.

By combining blockchain and TEE, we build network systems that simultaneously protect data integrity and enforce execution security, i.e., a more secure DNS and PKI. During these two pieces of work, we have also identified some problems within the blockchain itself, namely that the contracts deployed on the blockchain cannot be modified. On the one hand, we can only redeploy a smart contract when we want to add new features, and on the other hand, it is difficult to fix a deployed smart contract with a patch when it has bugs. Facing this problem, we also propose a feasible solution based on TEE.

## 1.2 Thesis Contribution

In our thesis, we propose a series of security improvements for three network systems, namely DNS, PKI, and blockchain, with respect to their specific problems. We summarize our work in Figure. 1-1. From the perspective of domain, we propose a trusted domain name system, B-DNS, based on blockchain technology to provide users with trusted domain name services. Users do not have to worry about the domain name service they use suffering from problems such as cache poisoning attacks. Further, considering that a domain usually has a certificate to prove its identity and enables encrypted transmission, we analyze the vulnerabilities in the current certificate issuance process and propose a trusted certificate issuance mechanism called PISTIS. We implement the certificate issuance logic using smart contracts. In the process, we found that the blockchain itself, which is highly relied on in work 1 and work 2, also has certain security issues. Specifically, i.e., the blockchain-based smart contract is vulnerable to unknown bugs once deployed. We propose SolSaviour to protect the deployed smart contracts and build the cornerstone for establishing a trusted network system.

With the introduction of blockchain technology and TEE technology, we have significantly improved the security of current DNS and PKI. We also change the trust paradigm in traditional network systems, where users trust a service provider typically because of that service provider's identity, which has proven to be risky. Therefore, we propose a trust paradigm based on blockchain and TEE, where users trust out of knowledge of the operational logic. This allows us to build more trustworthy network systems.

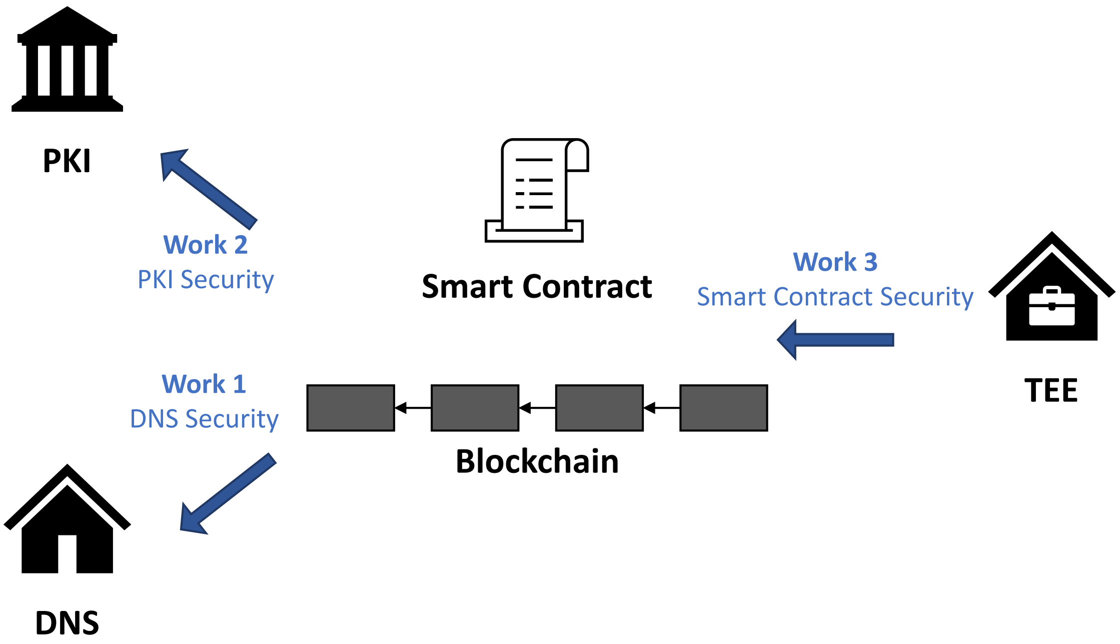


Figure 1-1: The structure of our work.

### 1.2.1 Secure and Efficient Naming System

As one of the most important basic network systems of the Internet, the domain name system has been disclosed to have flaws that make it vulnerable to cache poisoning attacks. The resource records stored within the recursive resolver are likely to be contaminated by attackers. Therefore, ensuring the data security of the domain name system has become an important research topic.

We propose B-DNS, a secure and efficient domain name system based on blockchain technology. We design an operation record data structure to store the registration, renewal, and revocation of domain names in the blockchain to be compatible with the resource records of the domain name system. To solve the problem of inefficient querying within the pure blockchain, we propose the mechanism of a dual-bloom filter to achieve efficient domain name querying. Based on our proposed B-DNS system, users can trust that the queried domain name information is accurate. We verified the security of B-DNS through quantitative security assessments. Specifically, we compared the security of B-DNS and traditional DNS from three perspectives: attack success rate, attack cost, and attack surface. From the perspective of attack success rate, we found that the success rate of attacking the B-DNS system, i.e., successfully contaminating the domain name records stored within the B-DNS system, is significantly lower than the success rate of attacking the recursive resolvers of the traditional DNS. From an attack cost perspective, the cost required to attack B-DNS is also much higher than the cost of attacking a traditional recursive resolver. For the attack surface, traditional DNS also discloses more vulnerabilities and has a larger attack surface. We can see from the above three points that B-DNS undoubtedly provides a more secure domain name service for users.

### 1.2.2 Trusted and Authorized Certificate Management

With the widespread use of HTTPS, the PKI system for managing certificates has gradually become one of the most fundamental network systems of the current Internet. The security of its issued certificates directly affects the confidentiality of encrypted transmission over the Internet. The rogue certificate incidents that occur from time to time also make how to ensure the security of issued certificates a critical research topic.

We refactor the logic of certificate issuance and propose a trusted and authorized certificate issuance mechanism, Pistis, which requires that all issued certificates start with the applicant's application, i.e., the applicant requests to issue a certificate for the domain it owns. In turn, Pistis verifies whether the applicant actually owns the domain it applies for, and if ownership is confirmed, Pistis issues the required certificate. Technically, the entire certificate issuance logic is implemented by a smart contract. At the same time, the verification of the applicant's ownership is initiated by the smart contract and implemented via TEE with a challenge-proof protocol. Successfully issued certificates are broadcast to the blockchain for confirmation. In this way, Pistis can realize that all issued certificates are trusted and authorized, and users can trust the certificates issued by Pistis.

### 1.2.3 Trusted Smart Contract and DeFi Protection Mechanism

With the development of blockchain technology and the advancement of general smart contract technology, decentralized finance is widely used, which forms the cornerstone of the next-generation Internet Web 3.0. The importance of smart contracts as a network system is also growing day by day. Considering the frequent security problems of smart contracts, The DAO hack, Parity multi-sig hack, and other attacks have caused a large amount of asset loss, protecting the security of smart contracts will become an important research topic.

Smart contracts have the property that they cannot be modified once deployed, so a deployed smart contract that discloses a bug cannot be fixed by patching it. The usual pre-deployment verification approach has the limitation that it can only detect known vulnerabilities and problems with false negatives. To address the limitations of the current solution, we propose SolSavior, a mechanism to protect deployed smart contracts. SolSaviour combines the on-chain voteDestruct mechanism with the off-chain TEE cluster, allowing users to vote on whether to destroy a buggy smart contract and redeploy it through the TEE cluster a patched smart contract. Specifically, the voteDestruct mechanism allows contract stake holders to decide whether to destroy the defective contract and withdraw inside assets. The TEE cluster is responsible for asset escrow, redeployment of patched contracts, and state migration. The experimental results show that SolSaviour significantly reduces the damage caused by the attack, effectively protects the assets within the contract, and maintains the normal operation of the contract.

## 1.3 Thesis Outline

The remainder of thesis is organized as follows. Chapter 2 introduces the background knowledge of this thesis, including the introduction to DNS, PKI, smart contract, and TEE. Chapter 3 presents our work on B-DNS. Chapter 4 proposes the design and implementation of PISTIS and elaborates on how to issue trusted and authorized certificates. Chapter 5 presents the principle and working logic of SolSaviour. We conclude our work and talk about our future directions in Chapter 6.

The primary research outputs arise from the following items:

- **Zecheng Li**, Shang Gao, Zhe Peng, Songtao Guo, Yuanyuan Yang, and Bin Xiao, “B-DNS: A Secure and Efficient DNS Based on the Blockchain Technology”, *IEEE Transactions on Network Science and Engineering (TNSE)*, vol. 8, issue. 2, pp. 1674-1686, 2021.

- **Zecheng Li**, Haotian Wu, Lap Hou Lao, Songtao Guo, Yuanyuan Yang, Bin Xiao, “PISTIS: Issuing Trusted and Authorized Certificates With Distributed Ledger and TEE”, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, issue. 7, pp. 1636-1649, 2021.

- **Zecheng Li**, Yu Zhou, Songtao Guo, Bin Xiao, “SolSaviour: A Defending Framework for Deployed Defective Smart Contracts”, in *Annual Computer Security Applications Conference (ACSAC)*, pp. 748-760, Virtual, 2021.

- **Zecheng Li**, Bin Xiao, Songtao Guo, and Yuanyuan Yang, “Securing Deployed Smart Contracts and DeFi with Distributed TEE Cluster”, *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (Major Revision).

# Chapter 2

# Preliminary

In this chapter, we give the precursor knowledge for this thesis and discuss related work. This chapter is structured as follows: we first discuss DNS-related knowledge and introduce existing work on securing DNS; secondly, we introduce PKI as well as related work on securing certificates; and finally, we discuss work on smart contract security.

## 2.1 Domain Name System

Each host has its unique IP address to designate its location on the Internet. However, IP addresses in the numeric form are complicated to remember, and the domain name system was created to solve this problem. DNS is a naming database where Internet domain names are stored and translated into IP addresses. In this way, we can access the corresponding server by simply remembering the domain name like `www.comp.polyu.edu.hk` and `www.google.com`. DNS automatically maps the domain name that people enter into their browser to the IP address of the website server.

DNS nameservers are organized as a tree, and the namespace is separated into layers. In each layer, the namespace is partitioned into non-overlapping regions called domains. A domain owner formulates the domain policy and keeps track of its subdomains. The root node of the DNS tree is called the *root zone*, which stores the delegation information of domains in the leaf node. The leaf node domains of the DNS root node are called the top-level domain (TLD). There exist different kinds of TLDs, of which the most widely-used ones are country-code TLD (ccTLD) and generic TLD (gTLD). The third level of the DNS tree is usually represented as the second-level domain (SLD or 2LD), which is under the supervision of its parent node, i.e., TLDs. The architecture of DNS is depicted in Fig. 2-1.

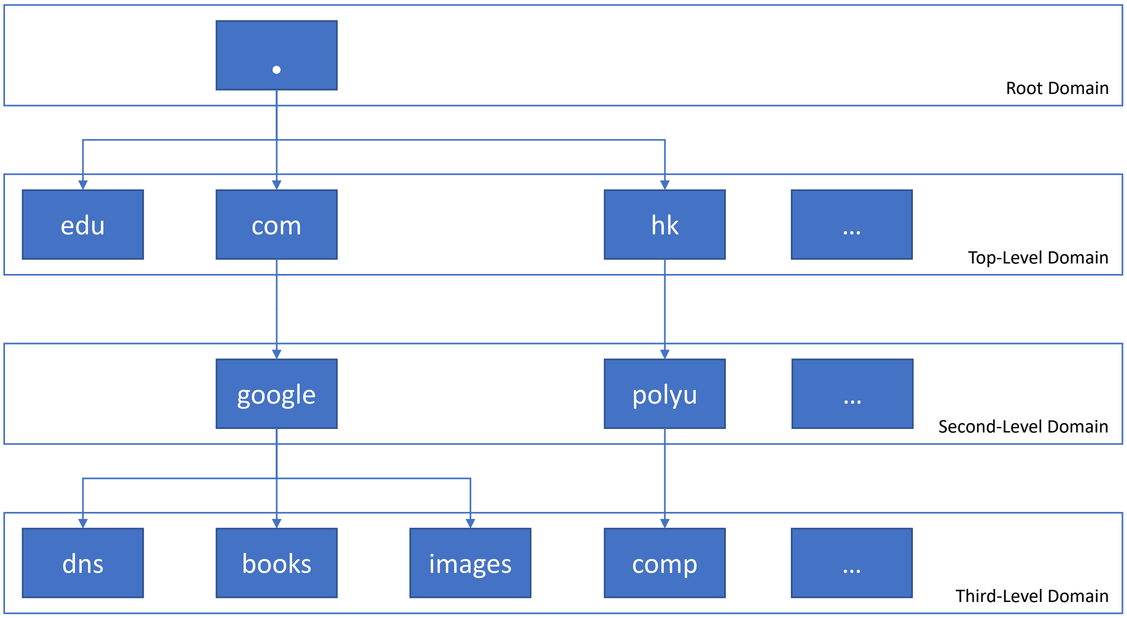


Figure 2-1: The architecture of DNS.

Each domain owner operates several authoritative servers, which store DNS entries using the extensible *resource record*. There are many types of resource record such as `A/AAAA`, `NS`, and `SOA`. The `A/AAAA` resource record is responsible for IPv4 and IPv6 address resolution respectively. The `NS` record stores the name of an authoritative server. Since DNS is hierarchical and each layer is only related to the previous layer, a query packet should travel from the root zone to the target authoritative server layer by layer, which is called *recursive resolution*.

## 2.2 Public Key Infrastructure

A public key infrastructure is a network system responsible for registering, distributing, and revoking digital certificates in the Internet. PKI provides secure information transfer for a range of network activities by issuing certificates containing cryptographic information to bind domains to a public key. In some network applications with high security requirements, simple passwords are insecure, and more stringent proofs are needed to confirm the identity of the parties involved in the communication and to verify the transmitted information, and this is where we need PKI. Through certificates issued by PKI, public keys and entities on the Internet can be cryptographically linked. The architecture of PKI is depicted in Fig. 2-2.

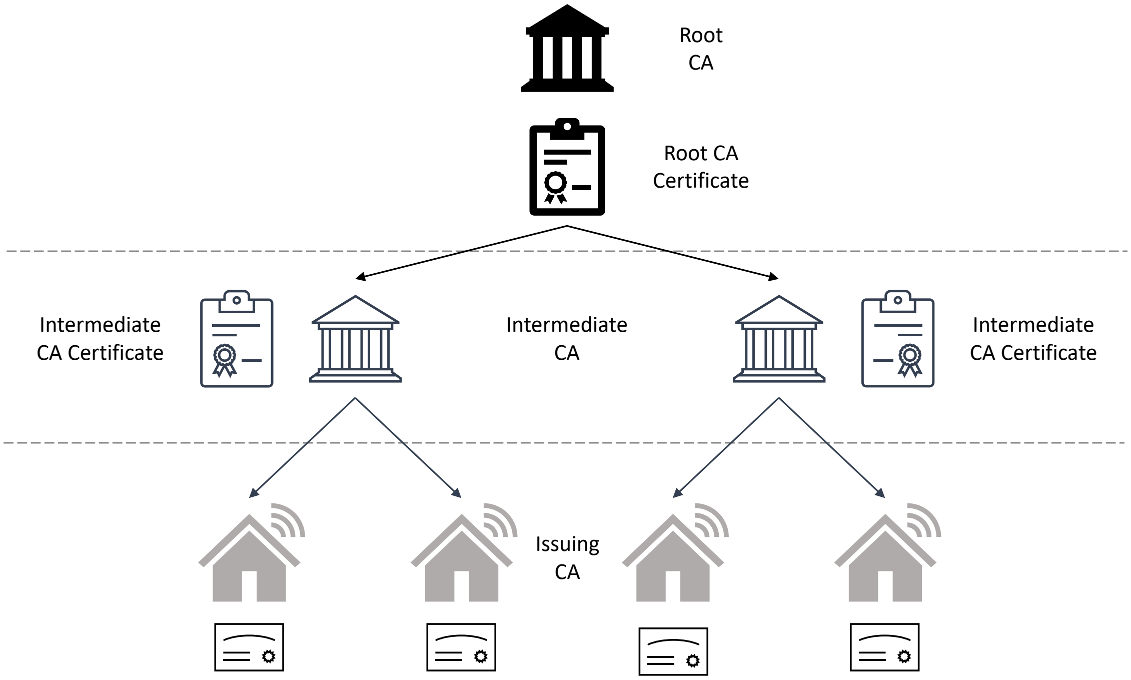


Figure 2-2: The architecture of PKI.

CA is the organization that issues digital certificates to prove that the entity and the public key listed in a certificate are bound. CA signs and issues the certificates, which prevents attackers from forging or tampering with these certificates. Users can determine the authenticity of a certificate by verifying its issuer's signature. In this case, people trust a CA and use the certificates it issues. Based on the certificates provided by the PKI, the SSL/TLS protocol was designed to preserve the integrity and confidentiality by encrypting data. This addresses the problem that the HTTP protocol originally used on the Internet was plain text and transmission content would be sniffed and tampered.

## 2.3 Blockchain

The blockchain technology is derived from Bitcoin, which was proposed in 2009 by Satoshi Nakamoto [81], the first cryptocurrency around the world. Participants collaborate to maintain the system operation and periodically elect nodes to commit new content into the blockchain through Byzantine fault-tolerant consensus protocol [17, 111]. A blockchain system is intrinsically a shared, immutable distributed ledger.

### 2.3.1 Data Structure

**Hash Chain** Tamper-proof is one of the main features provided by the blockchain. Specifically, all transactions are put in the leaf node of a Merkle Tree, and an iterative calculation process will proceed until the `MerkleRoot` is calculated, which is encapsulated in the block header so that the user can easily verify the integrity of the transaction data. Moreover, blocks are chained sequentially by calculating the hash value of the previous block's header, which is called the `prevHash`. It is very tough to calculate a valid `PrevHash` value since the target is pretty high, which requires the attacker to try millions of times. Accordingly, the combination of `MerkleRoot` and `prevHash` constitutes the cornerstone of the tamper-proof feature of the blockchain.

**Merkle Tree** Each block in a blockchain contains a summary of all transactions in that block, using a Merkle tree, a binary hash tree. Merkle tree can provide an efficient way to verify a large amount of data while summarizing it. When $n$ transactions are aggregated by hashing to compute the Merkle tree, the tree's leaf nodes will correspond to the hash value of each transaction. The root node of a leaf node is obtained through performing cascading hash computation on the hash values of the leaf nodes. By such recursive computation, we can get the value of the root node of the Merkle tree and use it as a summary of the entire block transaction information.

### 2.3.2 Network Layer

**Bootstrapping** Bootstrapping is the process by which new nodes join the network. Typically, a blockchain client stores the IP addresses of a number of "DNS seeds", namely a DNS service that provides a list of active blockchain nodes. A node that knows nothing about the blockchain must be given the IP address of an active node in the blockchain. Otherwise, this node will never be able to connect to the blockchain. Once connected to an active node, the bootstrapping node is given a list of IP addresses. The bootstrapping node can ask the newly-connected nodes for new IP addresses recursively. A blockchain node also updates its neighbor list following this way.

**Overlay Network** Though the blockchain node is connected to the Internet directly, the on-chain functionalities cannot be visited directly. Namely, they are working on a blockchain overlay network. Specifically, the smart contract can only be used by blockchain nodes. External users must install a blockchain client to invoke the smart contracts. On the contrary, smart contracts cannot visit off-chain websites and network resources directly. Considering a smart contract that acquires the NASDAQ index periodically, it cannot visit any websites that display the number. The only way it can get the value is to ask users to write it into the contract. In this case, we can say that the on-chain network is separated from the off-chain network.

## 2.4 Smart Contract

Ethereum is recognized as the second-generation blockchain system. From the science perspective, it is a globally accessible state machine with a built-in virtual machine that allows users to change its global state. From the engineering perspective, it is a distributed ledger with a built-in smart contract mechanism that enables users to develop applications atop it. In Ethereum, there are two kinds of accounts: Externally Owned Accounts (EOA) and contract accounts. End-users control EOAs with a unique key pair and blockchain address. By contrast, contract accounts are empowered by contract programs, account balances, and persistent storage in the form of key/value pairs. Ethereum has its built-in currency called ether.

The smart contract can be recognized as a constant program that runs deterministically atop Ethereum. EOAs typically create smart contracts in the form of contract creation transactions. An EOA can invoke a smart contract by sending a transaction with specified function names and parameters to the contract's address. Compared with Bitcoin's scripting system, Ethereum smart contract languages (i.e., Solidity) is Turing-complete. Ethereum introduced gas to measure the computational overhead of smart contract execution. The invoking entity should pay for the consumed gas in ether, which is calculated by multiplying *gasprice* and the number of gas used. Ethereum transactions set limit on gas with two parameters: *gasprice* and *gaslimit*. The *gasprice* indicates how expensive the user is willing to pay for each gas. The higher the *gasprice*, the faster the transaction is mined in a new block. The *gaslimit* sets a limit on how many gases a transaction can use. Once a transaction consumes more gases than *gaslimit*, all execution and state changes are reverted.

### 2.4.1 Defective Smart Contracts

Smart contracts are subject to a wide variety of defects. Defective smart contracts can be divided into two categories: exploitable smart contracts and unexpected smart contacts (i.e., may have unanticipated internal states)

For the first type, either some problems in the contract implementation create bugs (e.g., reentrancy vulnerability), or an attacker can exploit the contract's internal logic to launch attacks (e.g., front running attack). Attackers can gain benefits that do not belong to them by exploiting these bugs. For the second type, these bugs may cause a smart contract to an unexpected state, such as a locked state. For example, a jackpot may never succeed because of a strictly equal operation [26]. From this, we can see that different contract defects can lead to different problems, they have different causes and severity, but they all require practical protection work.

### 2.4.2 Contract Protection

Generally, we define the defending methods of smart contracts as repairing and recovering techniques. The repairing technique can alleviate the bugs in a smart contract, and recovering technique can save a contract from severe states.

Almost all contract repair techniques focus on repairing smart contracts before deployment. Part of the work identifies vulnerabilities by statically analyzing the contract code and generating the appropriate patches. Another aspect of the work, runtime verification, determines whether a deployed contract is vulnerable by selecting its operational state and developing the appropriate patch. These two tasks are then indistinguishable from contractual vulnerability detection techniques to protect deployed contracts. That is, they cannot fix vulnerabilities that have not been detected. Nor can they save the assets in deployed smart contracts.

One possible solution for fixing vulnerabilities in deployed contracts is the proxy model, where the smart contract is separated into a logical contract and a data contract. Message calls are made through the data contract, which is redirected to the latest deployed logical contract. Once an error is exposed, a new logical contract is deployed to replace the defective one. However, this approach is subject to the requirements of the trusted contract owner. That is, the contract developer will set its own address as the contract owner during the contract creation step, which results in the contract user needing to trust the contract creator, which is not applicable to multi-user contract scenarios.

### 2.4.3 Internal State

The concept of a contract's internal state includes the values of contract variables and the stake distribution inside the contract. When migrating and upgrading contracts, the consistency of the contract's internal state should be maintained. People could recover the value of variables inside a smart contract. The `getter` function can be used to acquire values of contract variables. The stake distribution can be recovered in a similar way as long as the contract explicitly defines variables to store stake distribution. If not, people can go over the history to determine the stake distribution. However, it is non-trivial to migrate the stake distribution from defective contracts to patched contracts, which requires actual transactions of funds. In this case, assets are transferred from defective contracts' addresses to patched contracts' addresses, in accordance with the stake distribution of the defective contracts. This is one of the main problems addressed by SolSaviour.

## 2.5 Decentralized Finance

The life cycle of a smart contract typically consists of four stages: contract creation, contract freeze, contract execution and contract finalization. The contract creation phase involves writing and deploying a smart contract. During the freeze phase, the miner incorporates the contract creation transaction into a new block, making it persistent. In execution phase, the execution of the contract is carried out via message calls. Once executed, the transaction containing the message call and the new state information are persisted with the new block on chain, which is recognized as finalization.

By definition, Decentralized Finance (DeFi) is an emerging financial technology based on a secure distributed ledger. DeFi removes the control of banks and institutions over money, financial products, and services and eliminates the fees that banks and other financial companies charge for using their services. People can keep their money in a secure digital wallet rather than in a bank. Anyone with an internet connection can use it and no approval is required. People can transfer funds in seconds or minutes. Popular DeFi protocol Uniswap offers flash loan services. DAI and Fei offer stable coin services. Though usually called protocols, the core of DeFi protocols is still the smart contract. Further, as a DeFi protocol typically consists of multiple contracts, the risk of smart contract bugs increases.

## 2.6 Trusted Execution Environment

The Trusted Execution Environment (TEE) isolates a preserved memory and conducts attested execution via dedicated CPU instructions. TEE guarantees the confidentiality and integrity of stored code and data. Through **remote attestation**, TEE ensures the authenticity of its execution. Current mainstream TEE implementations include Intel Software Guard Extensions (SGX) [7,50], ARM TrustZone [8], and AMD SEV [6]. There are also some open-source implementations such as Keystone [65].

**Intel SGX.** Intel SGX prevents stored code and data from disclosure and modification. Developers can divide an application into a CPU-enhanced *enclave* and a host application that manages it, which can improve security even in attacked platforms. Benefiting from TEE, developers can enable identity and record privacy, secure browsing and digital rights management protection, or any high-security application scenario that requires secure storage of confidential or protected data.

**Attested Execution Model.** TEE can generate attestation signatures to convince a remote verifier that the execution result is outputted by a specific program with specified inputs. The attestation requires to establish a secure communication channel between the TEE and verifier, and a software measurement method to provide information concisely. The secure communication channel can be built based on a key exchange protocol. Code and data inside TEE can be measured using the Intel TEE instructions. Verifier can check whether the measurement matches its expectation and then determine the security of execution outputs transmitted via the secure communication channel.

The attested execution model is proposed by Pass et al. in [89]. Subramanyan et al. established the formal foundation for the secure remote execution of enclaves [102]. They formalize an ideal functionality $\mathcal{G}_{att}$ to abstract the attested execution processor. Different attested processors have chosen different design paradigms. Most of the differences exist at the level of implementation details and are not reflected at the level of abstraction. Therefore a generic abstraction is a good generalization of the attested execution processor.

## 2.7 Blockchain and TEE

We find some similarities between blockchain and TEE. Among other things, blockchain-based smart contracts can be considered to a certain extent as trusted computing. As the code (in bytecode form) and the input and output of the contract are publicly available, the user can verify the execution result, and therefore the contract is executed with trustworthy results. However, the trust mechanism of a smart contract is fundamentally different from that of a TEE. The execution of a contract is based on the distributed consensus of the blockchain, i.e., the user can trust the outcome of contract execution due to the presence of incentive-based consensus mechanisms that spike the cost of malicious node creation. The trustworthy mechanism of TEE, on the other hand, is technical in the sense that users can determine whether the result of that execution is the output of a specified input executed by a specified program by verifying attestation.

There is some work on complementing blockchain with TEE. Specifically, TEE is applied to one of the blockchain layers. Resource efficient mining (REM) is proposed to eliminate the current computation-heavy Proof of Work (PoW) [123]. REM's key idea involves miners providing trustworthy reports on the CPU cycles they use for inherently useful workloads.REM is flexible enough to allow any entity to create a useful workload. Proof of Luck is another consensus layer work [77]. TeeChain is the first layer-two payment network that executes off-chain transactions asynchronously concerning the underlying blockchain [70]. TeeChain proposes treasuries, which is a transaction processing entity protected by TEEs, to build layer-2 payment channels for off-chain transactions. Current lightweight clients typically outsource most of the computation and storage workload to full blockchain nodes. However, such verification leaks critical information about clients' transactions. To address the privacy problem, Bite uses the trusted execution capability of SGX enclaves [75]. The enclaves on full nodes serve privacy-preserving requests from light clients. However, naive processing of client requests from within SGX enclaves still leaks clients' addresses and transactions. Similarly, ZLiTE work on building a privacy wallet for

Zcash with TEE is proposed in [116]. Tesseract is a secure cryptocurrency exchange that provide real-time service. Existing centralized exchange designs are vulnerable to theft of funds, while decentralized exchanges cannot offer real-time cross-chain trades [13]. All currently deployed exchanges are also vulnerable to frontrunning attacks. Tesseract overcomes these flaws and achieves a best-of-both-worlds design using a trusted execution environment. Committing the recent trade data to independent cryptocurrency systems presents an all-or-nothing fairness problem.

There is also some work on improving the security of TEE with blockchain. Kaptchuk et al. focused on using ledgers to enhance the security of TEE [55]. Zhang et al. proposed Paralysis proofs [122], which can alleviate the impasse of potential loss of digital assets. TEE-KV [61]. TEE-KV, built on blockchain, is a secure, immutable key-value store for TEE. Tran et al. proposed Obscuro [109], a Bitcoin mixer to protect the relationship between payers and payees.

# Chapter 3

# B-DNS: A Secure and Efficient Domain Name System based on Blockchain

## 3.1 Introduction

IP addresses are unique identifiers of Internet resource. Anyone who wants to visit some specific resources must know its IP address. However, it is hard to remember alphabetized domain names compared with numerical IP addresses. Accordingly, the DNS is designed to provide a domain name to IP address mapping service so that people could approach resources on Internet easily by only remembering their domain names. Unfortunately, researchers have exposed several vulnerabilities in current DNS such as the weak verification mechanism and single point failure of name servers, which causes different attacks.

The weak verification mechanism of current DNS causes the cache poisoning attack [60,101]. In cache poisoning attacks, attackers can send well-crafted response packets when a recursive resolver updates cache. Once a forged DNS entry is injected into the cache successfully, clients under the victim recursive resolver will be redirected to a phishing website when they visit the affected domain. The biggest banks of Brazil, Bandesco was reported to be attacked in this way [32].

The single point failure makes legacy DNS vulnerable to DDoS attacks. Currently, DNS domains are structured in a tree and each node stores a bunch of IP addresses of its sub-domians. Once the name server of a critical domain is under the DDoS attack, the service of its sub-domains will be disrupted. In this way, DDoS attacks that targeting key servers can significantly collapse the availability of partial legacy DNS [80]. The more essential the server, the more serious the consequence of the DDoS attack. In 2016, a DNS service provider Dyn was attacked in this way [66].

Several methods have been proposed to address these attacks. Approaches against the cache poisoning attack can be categorized into two types: employing the Domain Name System Security Extensions (DNSSEC) [114] and increasing the entropy of query packets [33, 91], which provides more information for recursive resolvers to distinguish a valid response packet. Methods to mitigate the DDoS attack mainly focus on storing more resource records in the cache [9, 87], which makes domains can still be resolved even its parent domain is under attack.

However, we observe that there are still some limitations in these countermeasures as they aim to defend attacks rather than repair the vulnerabilities. A completely DNSSEC-enabled DNS can prevent itself from the cache poisoning attack while current DNSSEC deployment rate is still low. We launch a quick scan over the Alexa top 1000 .com/.net/.org domains and find that only 3% domains support DNSSEC. Merely increasing the entropy of query packets is not a long-term solution. It can decrease the success rate of attack to some extent. However, as the development of computation power and network bandwidth, this defense becomes weaker and weaker. In DDoS defense, there exist the probability that queried domains are not cached. In addition, these methods do not work when authoritative servers are under attack. As to the T-DNS, its security depends on the TLS protocol, which employs certificates issued by the certificate authority (CA). However, the centralized CA is not secure since it suffers from the single point failure and maliciously issued certificates.

Facing these challenges and inspired by the good properties of blockchain, we intend to build a secure domain name system based on the blockchain. The blockchain is a distributed ledger secured by hash functions [17]. Blocks are chained sequentially by encapsulating the hash value of its previous block `PrevHash` into the header. Transactions are stored in blocks, and each block calculates a `MerkleRoot` to provide an easy way to verify the integrity of transaction records. In this way, blockchain can ensure the integrity of stored data, which presents good authentication mechanism. Additionally, blockchain employs the peer-to-peer network to transmit data and all peers are equal, which provide great resilience to the single point failure and DDoS attack.

However, building a blockchain-based DNS is quite tricky. It is not merely storing resource records in the distributed ledger. Many challenges exist in the way to implement a secure and efficient blockchain-based DNS system.

- **Stored Data are Immutable.** Once a resource record is written into the blockchain, it is hard to modify the content. However, there exists the need to update DNS records because domain owners may change the IP addresses of their domains. Accordingly, some new designs should be adopted to provide flexible record updates.

- **Performance is Poor.** The primary search operation in the blockchain is slow while the name service is time-sensitive. Therefore, some schemes should be designed to speed up the search process in a blockchain.

- **New Vulnerabilities May be Introduced.** Though blockchain can provide many good properties such as tamper-proof data and DDoS resilience, it may introduce new vulnerabilities such as inconsistent data across nodes and 51% attack, which are inherent problems of the blockchain. Accordingly, how to build a blockchain-based DNS without introducing new security problems of the blockchain is a big challenge.

In this chapter, we propose the Blockchain-based DNS (B-DNS), a secure and efficient domain name system. In B-DNS, DNS records are stored as transactions in the blockchain. B-DNS also equips with an index to accelerate the search speed in the blockchain to provide efficient name service. Moreover, B-DNS is compatible with the legacy DNS. Recursive resolver and users can interact with B-DNS name servers directly. Our contributions can be summarized as follows:

- We alleviate the computation-heavy PoW consensus protocol utilized in previous blockchain-based DNS. By proposing a biased-coin flipping protocol and a distributed random-number generation (DRG) protocol, B-DNS builds a Proof-of-Stake consensus protocol. The security of B-DNS PoS consensus protocol will not be affected by the amount of computation power.

- We address the problem of inefficient query in previous blockchain-based DNS. We build an index tree for B-DNS and propose a search algorithm to improve the query speed. Our experiment results show that B-DNS can provide similar query performance with current commercial DNS implementations.

- We propose a novel way to quantitatively compare the security of B-DNS and the legacy DNS according to the attack success rate, attack cost, and attack surface. To the best of our knowledge, this is the first time that researchers quantitatively compare the security of blockchain-based systems with traditional systems. Experiments show that B-DNS is much securer than legacy DNS.

## 3.2 Background

### 3.2.1 The Problems of Legacy DNS

A DNS query packet utilizes the transaction ID (TXID) to distinguish a valid response packet from forged ones. Typically, a TXID value is random and takes attackers a while to guess. But sometimes, the recursive resolver increments TXID from zero, which makes it easier to come up with a correct value. Accordingly, an attacker can first initiate a query to the recursive resolver and then forge response packets, which try all possible TXIDs, to deceive the recursive resolver that it is the valid response packet [101]. Once the recursive resolver accepts the forged packet, the attacker succeeds in poisoning the cache. Dan Kaminsky proposed an improvement to the cache poisoning attack and made it more effective [54]. Kaminsky's attack adds a non-existent sub-domain name to the victim domain. For example, it launches a query for the *ns1.example.com* when it wants to poison the resource record for *example.com*. Accordingly, if the first trial failed, a Kaminsky attacker can immediately launch another attack, which queries for *ns2.example.com*. After evolving to the Kaminsky's attack, the danger of cache poisoning attack significantly increases.

Additionally, the DDoS attack that targets legacy DNS often happens. Once a higher-level domain is under the DDoS attack, the availability of its sub-domains could be significantly degraded. In history, the root server and some TLD authoritative servers have been attacked by DDoS attack several times [80]. Some attacks did succeed in disabling the victim DNS servers and causing parts of the Internet experiencing severe domain name resolution problems.

## 3.3 From Legacy DNS to B-DNS

In this section, we describe the changes from legacy DNS to B-DNS to provide a smooth transition from legacy DNS to B-DNS. We also discuss some considerations when designing the B-DNS and give a formal definition of the B-DNS blockchain.

### 3.3.1 What's the Differences?

The difference between legacy DNS and B-DNS is mainly reflected in three aspects: the management of DNS records, the architecture that name servers structured, and the resolution path of domain names.

In legacy DNS, DNS records are managed by domain owners, who operate authoritative name servers. In this case, domain owners can update, add, or delete DNS records by changing the resource records in authoritative name servers. In B-DNS, as DNS records are stored in the blockchain, the management of DNS records are conducted by different types of transactions.

In legacy DNS, name servers are structured in a tree. The name servers in each layer store the IP addresses of their sub-domains. By contrast, B-DNS name servers are structured in a peer-to-peer network. Each name server either stores a full copy or metadata of the blockchain.

In legacy DNS, the query operations is conducted by the recursive resolver, who maintains a cache to store frequently-queried DNS records. Once a queried domain name is not cached, the resolver will conduct recursive resolution to acquire the asked DNS record. In B-DNS, as DNS records are stored in blockchain, end-users can directly query the name server with complete blockchain data.

### 3.3.2 Considerations of B-DNS Consensus Protocol

Current mainstream blockchain consensus protocols can be divided into: Proof-of-Work(PoW), Proof-of-Stake(PoS), and Practical Byzantine Fault Tolerance(PBFT).

PoW consensus protocol, as the consensus protocol of the first blockchain system Bitcoin, has been widely studied and verified for its security and performance. However, an empirical study on the Namecoin has exposed the vulnerability of using PoW consensus protocol in blockchain-based naming system. A PoW system is always exposed to 51% attack, which can only be prevented by enlarging the system's overall computation power. More computation power brings in extra resource consumption, which introduces additional system maintenance cost. Bitcoin's security is supported by its huge computing power, whose annual electricity consumption is 61.4 TWh (the same as Switzerland's annual electricity consumption), making it difficult for people to control 51% of it. But for those smaller or newer PoW blockchain systems, it may only require 5% of Bitcoin's computing power to reach 51% of its computing power, which is easy and affordable. Though adopting merged mining with Bitcoin provides a viable solution to the above problem, the high maintenance cost it introduces will become the burden on system maintenance.

PBFT consensus protocol can enable high-throughput transaction processing, low-latency confirmation, and good security properties (33% byzantine tolerance). However, its excessive use of network to transmit consensus packets makes it less scalable.

Experiments have demonstrated that the throughput of PBFT consensus protocol drops exponentially after the number of nodes exceeds 64.

The PoS consensus protocol assigns the blockchain generation right based on the number of stake controlled by each node. The proportion of stake determines the probability of being selected Stake, as an internal parameter of the system, makes it impossible for an attacker to call stake as if it is computing power, which makes the 51% attack on the PoS system difficult. In addition, compared with the PBFT protocol, PoS is more scalable because once the node to generate blocks are selected, consensus is reached. Based on the above considerations, we chose PoS as the B-DNS consensus protocol.

### 3.3.3 Formal Definition of B-DNS Blockchain

We give the formal definition of B-DNS blockchain in this section. In B-DNS, the leader election is conducted according to discrete-time units.

**Definition 1.** (Slot). *In B-DNS, time is divided into discrete units called* slot*, which is represented as* $sl_j, j \in \mathbb{Z}^+$.

All registries are equipped with clocks to show the current slot and allow them to synchronize with each other. Each slot owns a slot leader $L_j$, who is responsible for issuing a new block. However, limited by the network latency, the leader election process cannot be executed slot-by-slot. Accordingly, a larger time unit *epoch* is also defined.

**Definition 2.** (Epoch). *The* epoch *is defined as a sequence of continuous slots. Each epoch consists of* $R$ *slots and is denoted as* $e_x, x \in \{1, 2, ...\}$*. Specifically,* $e_x = \{sl_{xR+1}, sl_{xR+2}, ..., sl_{(x+1)R}\}$.

**Definition 3.** (Block). *A* block $B_j$ *issued at slot* $sl_j$ *contains the current state* $st_j \in \{0,1\}^\lambda$*, data* $d \in \{0,1\}^*$*, the slot number* $sl_j$ *and a signature* $\sigma = Sign_{sk_i}(st_j, d, sl_j)$ *signed using the private key* $sk_i$ *of the slot leader* $R_i$.

**Definition 4.** (Genesis Block). *The* genesis block $B_0$ *contains the list of registries identified by their public keys and stakes* $\mathbb{S}^0 = \{(vk_1, s_1^0), ..., (vk_n, s_n^0)\}$ *and initial randomness* $\rho^0$*, which is used to seed the leader election function.*

**Definition 5.** (Blockchain). *A* blockchain *relative to the genesis block* $B_0$ *is a sequence of blocks* $B_1, ..., B_n$ *associated with a strictly increasing sequence of slots. The length of a chain* $len(C) = n$ *is its number of blocks. The block* $B_n$ *is the head of the chain, denoted* $head(C)$*. The empty string* $\varepsilon$ *is recognized as a legal chain.*

**Definition 6.** (State). *The* state *is defined as a string* $st \in \{0,1\}^\lambda$ *that represents the balance of each account. Specifically, the state* $st_j$ *of* $B_j$ *is equal to* $H(B_{j-1})$*, where* $H$ *is a predefined collision-resistant hash function.*

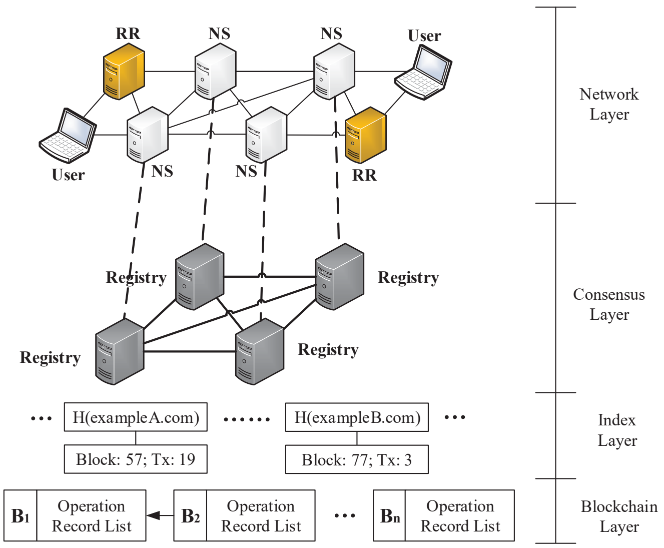


Figure 3-1: The 4-layer architecture of B-DNS. The blockchain layer contains a blockchain that stores the DNS records. The index layer maintains an index to accelerate the query service. The consensus layer ensures the consistency of stored data. The network layer provides legacy DNS-compatible query operation.

## 3.4 System Design

In this section, we introduce the detailed design information of B-DNS according to the 4-layer architecture, which is depicted in Fig. 3-1.

### 3.4.1 Blockchain Layer

In the blockchain layer, B-DNS stores DNS records as transactions in the blockchain. Since traditional resource records cannot ensure data continuity, we propose a new format called *operation records*. There are three types of operation records in B-DNS: registration, update, and revocation.

***Registration.*** In legacy DNS, the right of domain registration is controlled by official registries (e.g., Verisign) and registrars (e.g., GoDaddy). In B-DNS, to be compatible with current DNS, new domain names still should be registered with the corresponding registries. After successfully registered, the new domain record with its valid period are signed and encapsulated into a registration record by the registry. Additionally, the address (i.e., hash of public key) that is controlled by the domain owner is also added to the registration record. The registry signature enables the slot leaders to verify whether the record is legitimate, while the address of the domain owner is left for further update.

***Update.*** Dynamic update is one of the major concerns when designing the B-DNS. Considering a scenario where the IP address changes, the corresponding registration record needs to be updated to map to new address. The update operation is defined to meet these requirements. Similar to the registration record, the update record is signed and broadcasted by the registry. However, an update record needs to redeem its corresponding registration record first, which can only be conducted by its domain owner. Otherwise, it cannot pass the verification and will not be included in the blockchain.

***Revocation.*** As the name suggests, the revocation record is used to terminate the ownership of a domain name. An expired domain will be revoked automatically. The registry will issue a revocation record to terminate its ownership. Similar to the update record, the revocation record should first redeem its registration or update record.

The idea of operation record is inspired by the Bitcoin scripting system. Bitcoin scripting system only allows the change of coins' ownership in a transaction. Operation record not only enables the change of domain ownership but also can change the transaction content. B-DNS blockchain also stores the stake of each registry as well as their public keys in the block header. The stake information updates every epoch, which makes the PoS consensus protocol in accordance with the latest state.

### 3.4.2 Index Layer

In the index layer, B-DNS maintains an index to facilitate the search process. Searching DNS records in the blockchain is time-consuming as data are structured in a linked list. However, DNS service is time-sensitive. If the target record is located in the latest block, a DNS query needs to take a long time. In this case, we build an index tree to map domain names to their IP addresses, where keys are hashes of domains and values are corresponding IP addresses. B-DNS also encapsulates a `IndexHash` into the block header, which stores the hash value of the index of current block. The `IndexHash` enables B-DNS name servers to verify the correctness of generated indexes.

**Algorithm 1:** The update algorithm.

```
Data: domain name: key; IP address: val
Result: An updated index tree
initialization;
while receiving a new block do
    validate the received block;
    key = hash(domain name);
    val = IP address;
    if node.root = null then
        return a new tree with node(key, val);
    endif
    if key < node.key then
        return node = node.left;
    else if key > node.key then
        return node = node.right;
    else
        node.val = val;
    endif
end
```

**Algorithm 2:** The search algorithm.

```
Data: A DNS query
Result: Blockchain address
initialization;
key = domain name;
while receiving a new query do
    parse the DNS query;
    if $node.root == null$ then
        return null;
    endif
    if $key < node.key$ then
        return node.left.key;
    else if $key > node.key$ then
        return node.right.key;
    else
        return node.val;
    endif
end
```

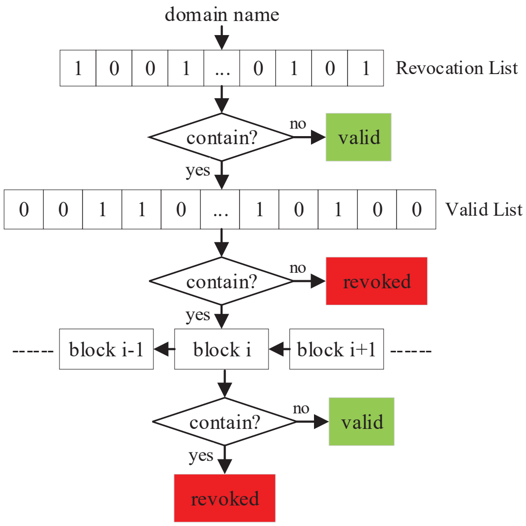


Figure 3-2: The revocation checking algorithm in B-DNS. The dual-bloom filter mode can alleviate the false positive when checking one bloom filter.

We design an update algorithm and a search algorithm for our constructed index, which are presented in Algorithm 1 and Algorithm 2, respectively. The update algorithm is used to insert new records into the tree, which works in a recursive way. It first checks the root node. Whether the root node is equal or larger or smaller than the target node affects the operation. In this way, the tree can route to the target node so that we could insert new value. The search algorithm works on an updated tree. It first checks the root node. If the root node is empty then returns error. Otherwise, it goes down according to the node's key. In this way, B-DNS can promptly find the target value in the index.

We also establish two bloom filters, which consist of a revocation list and a valid list, to provide fast domain revocation checking service. The flow chart of our revocation checking algorithm is given in Fig. 3-2. B-DNS first checks whether the domain name is in the revocation list. If it is not in the revocation list, the domain name must be valid. If it is in the revocation list, we cannot ensure that it is revoked because of the hash collision. Therefore, we check whether the domain name is in the valid list. If it is not in the valid list, the domain name must be revoked. If it is in the valid list, we need to search the latest transaction of this domain in the blockchain.

### 3.4.3 Consensus Layer

In the consensus layer, B-DNS implements a Proof-of-Stake consensus protocol to ensure the consistency of DNS records. A leader election function is executed every epoch to select block generators. The B-DNS PoS consensus protocol ensures that a registry $R_i$ holds the probability proportional to its stake $s_i$ to be elected as the block generator:

$$P(R_i) = \frac{s_i}{\sum_{m=1}^{n} s_m}$$

In B-DNS, the stake of a registry is defined as the number of domains registered with it. However, this number is zero before the system initialization. In this case, all participants' stakes are set as $s_i = \frac{1}{n}$ in the genesis block so that all the registries have equal probability to be elected as the slot leader in the first epoch. In B-DNS, the

Table 3.1: Summary of notations.

| Notation | Description |
|---|---|
| $R_i$ | The $i$-th registry |
| $vk_i$ | The verification key of registry $R_i$ |
| $sk_i$ | The secret key of registry $R_i$ |
| $s_i$ | The stake held by registry $R_i$ |
| $sl_j$ | The basic time unit, called *slot* |
| $L_j$ | The slot leader in slot $sl_j$ |
| $e_x$ | A set of continuous time slots, called *epoch* |
| $B_i$ | The block issued in slot $sl_i$ |
| $st_i$ | The state of the blockchain in slot $sl_i$ |
| $\sigma$ | The signature calculated by the slot leader |
| $\mathcal{C}$ | The current blockchain, used in $\pi_{PoS}$ |
| $\mathbb{C}$ | A set of candidate blockchain, used in $\pi_{PoS}$ |
| $\mathbb{S}^x$ | The stake distribution $\{(vk_1, s_1^x), ..., (vk_n, s_n^x)\}$ |
| $\rho^x$ | The randomness used to select slot leaders |

stake updates every epoch since the distribution of registered domains is continually changing.

The key point is to construct a progressive protocol that can select leaders according to the pre-defined probability. The B-DNS PoS consensus protocol flips a $\tilde{p}_i$-biased coin to achieve this goal where

$$\tilde{p}_i = \frac{s_i}{\sum_{m=i}^{n} s_m}$$

If the output of this biased coin is 1, then the $i$-th registry is selected as the slot leader. Otherwise, the protocol proceeds and it flips a $\tilde{p}_{i+1}$-biased coin where

$$\tilde{p}_{i+1} = \frac{s_{i+1}}{\sum_{m=i+1}^{n} s_m}$$

Note that $\tilde{p}_n = s_n/s_n = 1$ so that this protocol always outputs a deterministic slot leader.

In B-DNS, the randomness seeds are generated distributionally. All the registries follow a distributed random-number generation (DRG) protocol to generate a distributed number, which acts as the seed in the $\tilde{p}_i$-biased coin flipping. The DRG protocol contains three phases:

- **Commitment Phase.** When epoch $e_x$ starts, each registry $R_i$ samples a uniformly random string $u_i$ and randomness $r_i$ for the underlying commitment scheme, generates shares $\{\sigma^i_1, \sigma^i_2, ..., \sigma^i_N\} \leftarrow \mathit{Deal}(N, u_i)$ and encrypts these shares under the public key of registry $R_1, R_2, ..., R_N$. Finally, $R_i$ posts the encrypted shares and commitments $\mathit{Com}(r_i, u_i)$ onto the blockchain.

- **Reveal Phase.** In the reveal phase, the registry $R_i$ distributes the key to open its commitment by posting $\mathit{Open}(r_i, u_i)$ onto the blockchain.

- **Recovery Phase.** When all shares $\{\sigma^i_1, \sigma^i_2, ..., \sigma^i_N\}$ distributed by $R_i$ are available, the other registries can compute $\mathit{Rec}(\sigma^i_1, \sigma^i_2, ..., \sigma^i_N)$ to reconstruct $u_i$. Then, the randomness for the next epoch is calculated by $u_1 \oplus u_2 \oplus ... \oplus u_N$.

By proposing the DRG protocol and $\tilde{p}_i$-biased coin flipping protocol, the B-DNS PoS consensus protocol can select leaders proportional to their stakes. Then, we introduce the detailed PoS consensus protocol as below, which defines the operations each registry should follow as well as the corresponding encryption mechanisms that ensure the data consistency.

**Protocol $\Pi_{PoS}$**

$\Pi_{PoS}$ is operated by a set of registries $\{R_1, ..., R_n\}$. It proceeds as follows:

**1.Initialization.** At the very beginning, the registry $R_i$ receives its secret key and verification key $(sk_i, vk_i)$ from the key registration interface of $\mathcal{F}^{D,F}_{DLS}$.

If the blockchain is not initialized, it sends $(\mathit{GenBlc}, R_i)$ to $\mathcal{F}^{D,F}_{DLS}$, receiving $(\mathit{GenBlcGet}, \mathbb{S}^0, \rho^0, \mathsf{F})$. Then, the registry $R_i$ sets the local blockchain $C = B_0 = (\mathbb{S}^0, \rho^0)$ and the initial state $st_0 = H(B^0)$.

Otherwise, if the blockchain has been initialized, the registry $R_i$ receives a set of valid chains $\mathbb{C}$ and set current chain $C = \texttt{maxvalid}(\mathbb{C})$ and the current state $st_j = H(B_j)$.

**2.Chain Extension.** In a typical slot $sl_j$, each online registry $R_i$ performs the following jobs:

1. **Update Stake.** In epoch $e_x$, the registry $R_i$ updates the stake distribution using the data drawn from latest block with slot number less that $xR - 2k$, where $k$ is the security parameter that used to indicate that the transaction is stable under $k$ blocks. The stake distribution is updated as $\mathbb{S}^{x+1} = \{(vk_1, s_1^{x+1}), (vk_2, s_2^{x+1}), ..., (vk_n, s_n^{x+1})\}$, which means that the stake updated in the epoch $e_x$ is used to select leaders for epoch $e_{x+1}$.

2. **Update Randomness.** In epoch $e_x$, a registry $R_i$ needs to update the randomness $\rho^x$ for the upcoming epoch. It sends a message $(EpcRnd, R_i, e_x)$ to the functionality $\mathcal{F}_{DLS}^{D,F}$, receiving a message $(EpcRndUpd, \rho^{x+1})$, where $\rho^{x+1}$ is the updated randomness used to seed the leader election function for epoch $e_{x+1}$.

3. **Collecting Valid Chains.** Once a registry $R_i$ is selected as the slot leader in slot $sl_j$, it needs to accept all valid chains via broadcast and put them into a candidate chain set $\mathbb{C}$. Then, the registry $R_i$ verify whether all the candidate chains are valid. After verification, $R_i$ acquires the longest valid chain by calculating $C' = \mathsf{maxvalid}(C, \mathbb{C})$. $C'$ is set as the current chain $C = C'$ and the latest state is set as $st = H(C')$.

4. **Issuing New Blocks.** If $R_i$ is the slot leader determined by function $\mathsf{F}(\mathbb{S}^x, \rho^x, sl_{xR+j})$ in the slot $sl_{xR+j}$ of epoch $e_x$, it generates a new block $B_{xR+j} = (st, d, sl, \sigma)$, where $st$ is the state of the former block (i.e., $st = H(head(B_{xR+j-1}))$), $d \in \{0,1\}^*$ is the stored operation records data and $\sigma = Sign_{sk_i}(st, d, sl)$ is a signature on $(st, d, sl)$. $R_i$ appends the newly-generated block to the current chain $C' = C|B$, broadcasts block $B_{xR+j}$, sets $C'$ as the new current chain and sets state $st = H(B_{xR+j})$.

**3. Broadcasting Operation Records.** Once a registrant registers a domain with registry $R_i$, the registry creates a operation record $op$ according to the template. Given that this operation record is consistent with the state of the record chain, the registry $R_i$ returns $\sigma = Sign_{sk_i}(op)$.

Then we describes how to construct a leader election function $\mathsf{F}$ using the PVSS scheme. The main problem in implementing the functionality $\mathcal{F}_{DLS}^{D,F}$ is how to generate uniform randomness for leader election function in a distributed way.

**Protocol $\pi_{DLS}$**

$\pi_{DLS}$ is run by a set of elected leaders during an epoch $e_x$ that lasts $N$ slots, without loss of generality denoted by $R_1, R_2, ..., R_N$ (Actually, these elected leader are not necessarily different).

**1.Commitment Phase.** When epoch $e_x$ starts, each registry $R_i, 1 \leq i \leq N$ samples a uniformly random string $u_i$ and randomness $r_i$ for the underlying committment scheme, generates shares $\{\sigma_1^i, \sigma_2^i, ..., \sigma_N^i\} \leftarrow Deal(N, u_i)$ and encrypts these shares under the public key of registry $R_1, R_2, ..., R_N$. Finally, $U_i$ posts the encrypted shares and commitments $Com(r_i, u_i)$ onto the record chain.

**2.Reveal Phase.** In the reveal phase, the registry $R_i, 1 \leq i \leq n$ distributes the key to open its commitment by posting $Open(r_i, u_i)$ onto the record chain.

**3.Recovery Phase.** When all shares $\{\sigma_1^i, \sigma_2^i, ..., \sigma_N^i\}$ distributed by the registry $R_i$ are available, the other registries can compute $Rec(\sigma_1^i, \sigma_2^i, ..., \sigma_N^i)$ to reconstruct $u_i$. Then, the randomness for the next epoch is calculated by $u_1 \oplus u_2 \oplus ... \oplus u_N$.

### 3.4.4 Network Layer

In the network layer, B-DNS provides three interfaces to enable different types of communication with different entities. In this case, B-DNS not only communicate with peers, but also provides domain name service to resolvers and end-users.

The interface between B-DNS name servers is similar to current blockchain systems. B-DNS supports transmission of transactions and blocks. The interface between B-DNS name servers and resolvers & end-users supports direct DNS query, which makes B-DNS compatible with current DNS. A B-DNS name server can respond the query directly by fetching the required DNS record from the blockchain.

**B-DNS Name Servers to B-DNS Name Servers:** The interactions between B-DNS name servers are mainly responsible for data transmission.

*inv.* It allows a node to advertise its knowledge about the blockchain.

*getblock.* A name server sends this message with its highest block number to get a list of unstored blocks from its peers.

*getdata.* This is used to respond to *inv* message. After receiving an *inv* message, a B-DNS name server checks if there are any unstored DNS records. If so, it sends a *getdata* message to require the lost records.

*getmerklepath.* When a light node wants to verify whether a record is valid, it needs to query a random full node the corresponding Merkle path.

**B-DNS Name Servers to Recursive Resolvers:** In this interface, B-DNS acts as an authoritative name server that reply DNS queries. On receiving a DNS query, a B-DNS name server directly fetches the queried DNS record from the stored blockchain and respond it. In addition, if a recursive resolver wants to use the B-DNS name service, it simply sends a DNS query to a B-DNS name server directly.

**B-DNS Name Servers to Users:** B-DNS is designed to be compatible with the legacy DNS. Accordingly, it could directly interact with the clients. A client wants to use the B-DNS service can modify its *resolv.conf* with the IP address of a B-DNS name server. If the queried B-DNS name server is a full node, it could easily fetch the required DNS record from the blockchain. By contrast, if the queried B-DNS name server is a light node, which only stores the headers of the blockchain, the query will be redirected to a full node without delay.

We finally discuss the differences of query operation in B-DNS. Traditionally, the recursive resolver conducts query operation with recursive resolution. The query packets need to traverse the DNS tree to obtain the IP address stored in the authoritative

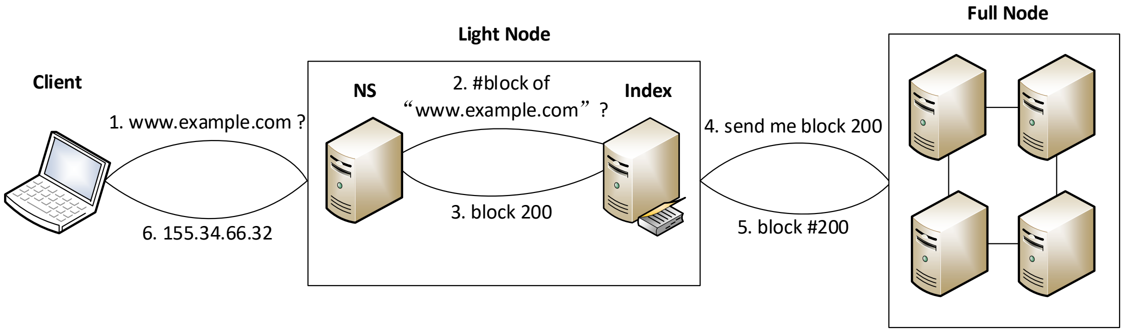


Figure 3-3: The query operations in B-DNS. The light node will randomly choose a full node to obtain the queried record and its merkle proof. For more security, the light node can query several full nodes and compare their answers.

name server. However, in B-DNS, as name servers are structured in a peer-to-peer way, the query process is different. As a full node, the B-DNS name server can respond to the client directly by fetching the required DNS record from the blockchain. By contrast, as a light node, the B-DNS name server needs to check whether the queried DNS records in stored locally. If so, it responds directly. Otherwise, the light node needs to query a full node to acquire the requested DNS record as shown in Fig. 3-3. The light node can verify the correctness of the DNS record in the response packet using the `MerkleRoot` and its corresponding Merkle path.

## 3.5 Experiment

We implement a prototype of B-DNS in *Golang* according to our 4-layer architecture. We also establish a testbed for B-DNS on an i9-9900k server. We set up eight B-DNS nodes and each acts as a registry that stores a full copy of the blockchain. Each node is a 2 GB memory, 2 CPU, Ubuntu 18.04 virtual machine and the hypervisor is Vmware Workstation 15.0.4. We also set up a commercial DNS implementation PowerDNS Recursor 4.1.10 as a comparison. As to the DNS record dataset, we use publicly accessible DNS traces provided by the CAIDA to generate transactions [22]. Our blockchain consists of around 100,000 DNS entries. In our experimental blockchain, all operation transactions are signed using ECDSA scheme and encapsulated in the form of registration records. We also craft a query dataset, which consists of 20,000 domain names, to test the lookup performance and resilience of B-DNS name servers.

We design two sets of experiments to evaluate the security and performance of B-DNS, respectively. In the security evaluation, we compare the security properties between legacy DNS and B-DNS from three dimensions: the probability of successful attack, the attack cost, and the attack surface. In the performance evaluation, we test whether B-DNS provides acceptable performance.

### 3.5.1 Security Evaluation

In security evaluation, we conduct three experiments to compare the security between legacy DNS and B-DNS from the probability of successful attack, the attack cost, and the attack surface.

#### The Probability of Successful Attacks

In this experiment, we compare the probability of successful attacks against legacy DNS and B-DNS. In legacy DNS, a successful attack implies that an attacker has generated a response packet with identical transaction ID and port number as the query packet. Its probability can be calculated as:

$$P_{success} = \frac{b * t}{\alpha * (\beta - \gamma) * \theta * s}$$

The meaning of these notations are listed in Table 3.2. As the attack time $t$ increases, more forged packets the attacker sends, higher the probability a attack succeeds.

In B-DNS, the attack methods against B-DNS are different because of a different architecture, where all records are stored in the blockchain. In this case, if an attacker wants to tamper the stored data, it needs to rewrite all the blocks after the one that stores the target record. In the experiment, we model a successful poisoning attack against B-DNS as a catch-up problems, where attackers need to generate forge blocks to replace blocks generated by honest users.

We introduce security parameter to restrict only data in $k$-depth block can be

Table 3.2: The notations used to calculate the probability.

| Notation | Description |
|---|---|
| $\alpha$ | The range of transaction IDs (universally $2^{16}$, or 65536 values) |
| $\beta$ | The range of source ports (conceptually $2^{16}$) |
| $\gamma$ | The number of reserved ports (usually $2^{10}$) |
| $\theta$ | The number of authority name servers. Many domain operate several authority servers with independent public IP address. A recursive server normally queries the closest one. Accordingly, $\theta$ is the product of all public facing addresses used by recursive resolver and authority servers. |
| $b$ | the bandwidth between the attacker and the victim recursive resolver |
| $t$ | The time that the attacker is able to send forged response packets |
| $s$ | The size of a response packet |
| $p$ | The probability that an honest node finds the next block |
| $q$ | The probability that an attacker finds the next block |
| $p_z$ | The probability that an attacker will never catch up from $z$ blocks behind |
| $q_z$ | The probability that an attacker will catch up from $z$ blocks behind |

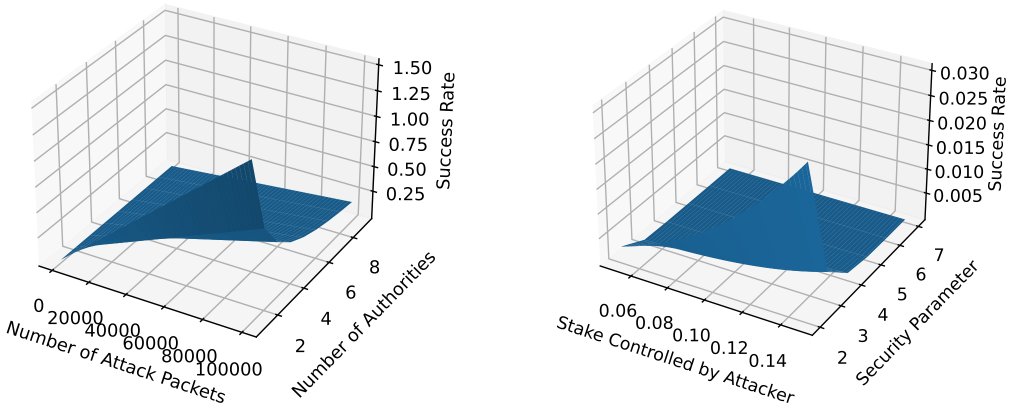
recognized as trusted. Therefore, when the amount of stake controlled by attackers is $s_i$ and there are $n$ nodes in B-DNS, the probability of a successful attack is:

$$P_{success} = (\frac{s_i}{\sum_{m=1}^{i-1} s_m + \sum_{m=i+1}^{n} s_m})^z$$

In Fig. 3-4, we draw the probability of cache poisoning attacks against legacy DNS and B-DNS, respectively. We can see that the probability of attacks against current DNS increases linearly with the number of packets sent. We can also see that the highest probability of attack against current DNS reaches 1, which means an attacker will always succeed as long as it sends enough forged packets to poison the cache.

On the contrary, the probability of attacks against B-DNS increases with the amount of controlled stake and decreases with the security parameter, which can be fine-tuned to modify the security of B-DNS. Usually, the attacker's controlled stake is a constant. Even a small increase in the stake is costly. In this case, the probability of a successful attack against B-DNS can hardly exceed 3% in current setting.

We also conduct experiments to evaluate the attack success rate against B-DNS



(a) Legacy DNS

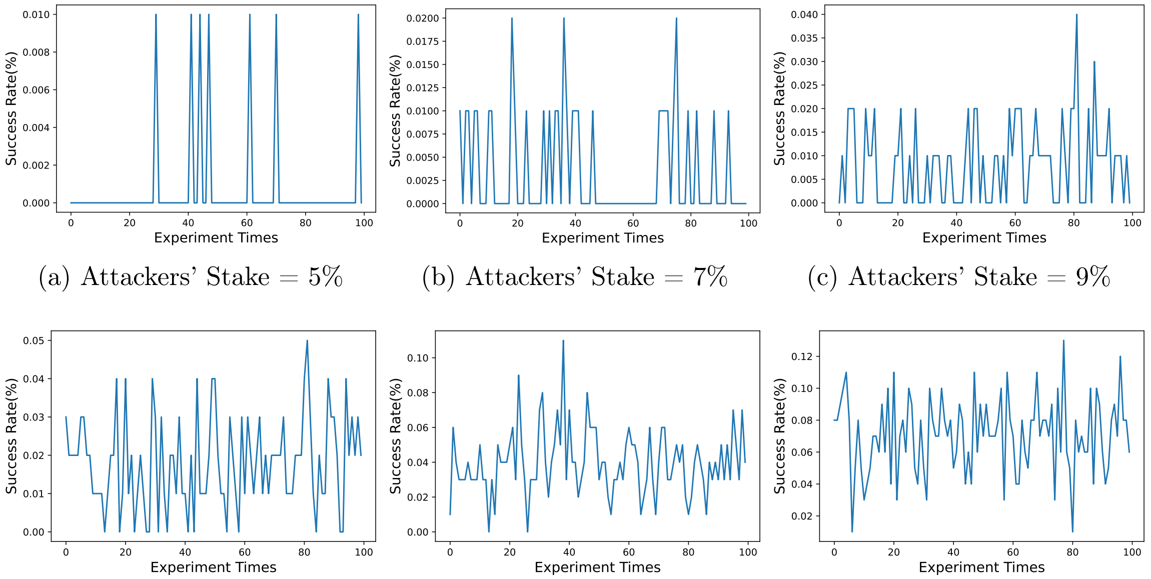
(b) BDNS

Figure 3-4: The theoretical success rates of cache poisoning attacks against DNS and B-DNS. In legacy DNS, the number of authorities means how many authoritative name servers the queried domain owns, typically 3. The success probability increases as the number of attack packets increases and will finally succeed. In B-DNS, the depth of target block means the gap between the attacker and honest nodes. The larger the gap, the less the probability of succeeding. The stake controlled by the attacker affects its probability of being selected to issue a new block.

in different scenarios. The results are shown in Fig. 3-5. We simulate an environment with 100 B-DNS nodes, each with 1% stake. Then we adjust the portion of attackers from 5% to 15%, which means the number of malicious nodes is from 5 to 15. We also assume that the honest nodes' chain is 4 blocks longer than attackers', which is the optimistic assumption for the attacker. Then we start the system to generate 100 blocks. If the attackers' chain catches up with honest nodes' chain, we record that attackers succeed. Otherwise, honest nodes win. We conduct the experiment 10000 times to calculate the success rate of attack against B-DNS according to the times that attackers win. We get 100 possible success rates for each stake distribution scenario. As we can see, even with 15% stake, the success rate of cache poisoning attack against B-DNS is at most 0.12%, which is tremendously small.

### Attack Cost

In this experiment, we compare the attack cost for attackers to launch attacks against legacy DNS and B-DNS. We think a fair way to compare the attack cost in different



(a) Attackers' Stake = 5%

(b) Attackers' Stake = 7%

(c) Attackers' Stake = 9%

(d) Attackers' Stake = 11%

(e) Attackers' Stake = 13%

(f) Attackers' Stake = 15%

Figure 3-5: The experimental success rates of cache poisoning attacks against B-DNS. In each scenario, the experiment was conducted 100 times. Though the success probability increases with the number of attackers' stakes, it is still negligible compared with the success probability in legacy DNS.

systems is when their success rates are equal. In this case, we consider the case that the probability of successful attack is 1%. How much should an attacker pay to attack legacy DNS and B-DNS.

For legacy DNS with 3 authority name server, an attacker needs to continuously send 126,835,750 packets to reach the probability of 1%, which requires 12216.9 MB traffic. For a network with 10 Mbps bandwidth, this attack lasts for 9780 seconds, less than three hour. The cost to use 10 Mbps for 2.7 hours is just several dollars.

For B-DNS, if an attacker wants to succeed with 1% probability, the stake it should own is shown in Table 3.3. If B-DNS possesses 1,000,000 domains, and registering one domain requires 10 dollars, the attack cost against B-DNS ranges from 3,160,000 dollars to 4,320,000 dollars, which is far more than the cost of attacking legacy DNS.

Table 3.3: The stake an attacker should own to launch an attack with 1% success probability.

| Stake | Depth | Stake | Depth | Stake | Depth |
|---|---|---|---|---|---|
| 31.6% | 6 | 38.6% | 10 | 41.8% | 14 |
| 34.1% | 7 | 39.6% | 11 | 42.4% | 15 |
| 35.9% | 8 | 40.5% | 12 | 42.8% | 16 |
| 37.4% | 9 | 41.2% | 13 | 43.2% | 17 |

### Attack Surface

In this experiment, we compare the security of legacy DNS and B-DNS with respect to their attack surfaces. We define the attack surface as the system's actions that are externally visible to its users and the system's resources that each action accesses or modifies. We first identify all resources of the system that are potential targets of attacks. For current DNS and B-DNS, the stored records and provided name service are vulnerable to different kinds of attacks. Then, we define attack class as a set of attacks that employ similar attack methods. In our experiment, we categorize the common attacks against DNS as spoofing, denial-of-service, hijacking, injection, and poisoning. Finally, we counte the number of instances of each attack classes for DNS and B-DNS, respectively. The results are concluded in Table. 3.4.

We can conclude that legacy DNS has more vulnerabilities in all types of attack classes. In spoofing class, current DNS has exposed 48 vulnerabilities while B-DNS only has one. The closest attack class between current DNS and B-DNS is the denial-of-service attack, where current DNS has detected 24 vulnerabilities, double of B-DNS's vulnerabilities. As to the hijacking and poisoning classes, B-DNS does not have such kind of vulnerabilities. Even the only one injection vulnerability, the affected component of B-DNS is its debug log, which does not affect the core parts of B-DNS such as the name service. In a word, we can see the attack of B-DNS is much smaller than that of current DNS, which makes attackers more difficult to attack B-DNS.

Table 3.4: The comparison of attack surface between legacy DNS and B-DNS.

| Attack Class | DNS | | | B-DNS | | |
|---|---|---|---|---|---|---|
| | Number | Example | Description | Number | Example | Description |
| Spoofing | 48 | CVE-2020-6412<br>CVE-2018-6175<br>CVE-2017-5106 | insufficient validation<br>incorrect handling of URL characters<br>insufficient policy enforcement | 1 | CVE-2018-10831 | incorrect verifier accepts spoof mining shares |
| Denial-of-Service | 24 | CVE-2020-6079<br>CVE-2018-8304<br>CVE-2018-19118 | resource allocation vulnerability<br>DNS response<br>stack overflow | 12 | CVE-2018-17144<br>CVE-2016-10724 | duplicate inputs<br>memory exhaustion |
| Hijacking | 2 | CVE-2015-4020 | insufficient validation on SRV records | - | - | - |
| Injection | 15 | CVE-2019-5168<br>CVE-2011-5276 | iochecked service vulnerability<br>SQL injection vulnerability | 1 | CVE-2018-20586 | injection to debug log |
| Poisoning | 4 | CVE-2018-5532<br>CVE-2015-4641 | cached BIG F5 IP addresses<br>directory traversal vulnerability | - | - | - |

Table 3.5: The search performance of B-DNS with a constructed index tree.

| Records Number | Route Times | Search Time (ms) |
|---|---|---|
| 136780 | 17.9 | 63.7 |
| 1467032 | 19.7 | 74.9 |
| 1803284 | 20.3 | 80.3 |
| 23190410 | 21.9 | 89.6 |
| 160403846 | 23.4 | 91.3 |

### 3.5.2 Performance Evaluation

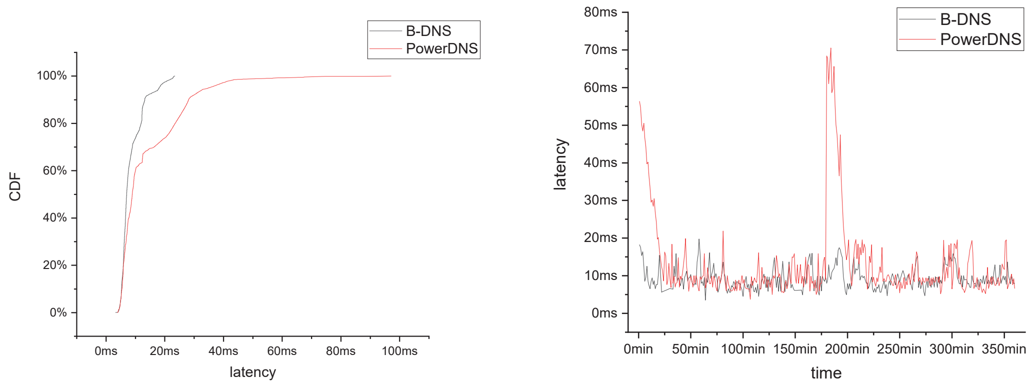
In performance evaluation, we conduct four experiments to evaluate the performance of B-DNS.

#### Search Speed

In this experiment, we examine to what extent can index speed up searching in the blockchain and whether adding an index affects the overall performance. Specifically, we test the search time in the blockchain with and without an index, respectively. We tested the search speed of our index with different record sets. The experiment results is illustrated in Table 3.5. As we can see, the search time is very limited in our constructed index. We also notice that the distribution of search speed without an index is approximately linear, which is because the search in a single chain needs to travel from the very beginning to the target block. Therefore, as the blockchain grows, the search time increase linearly. However, in B-DNS, the search time grows logarithmically.

#### Space Cost

In this experiment, we investigate the space cost of the B-DNS full node and light node, respectively. We use 31,535,998 DNS entries provided by CAIDA. We first test the volume of a full node. The result is 887.41GB, which is quite an affordable result. Considering the price of disk nowadays, a registry can easily afford hundreds of drives. Then, we test the space cost of a light node, which only keeps the block header of a blockchain. The result shows that it only needs 4.12 GB. Apparently, it is feasible



(a) Legacy DNS

(b) BDNS

Figure 3-6: The performance of B-DNS. In figure 3-6a, almost all B-DNS query latency is less than 20ms, while some PowerDNS's latency exceeds 100ms. In figure 3-6b, we can see that B-DNS is not affected by the flash-crowd effect while PowerDNS takes around thirty seconds to recover.

for most current DNS servers to operate a B-DNS name server.

### Query Latency

In this experiment, we examine the query latency of B-DNS and compare it with PowerDNS. Both of them are set up in the lab without any cache warming up. B-DNS name server is equipped with a full blockchain and an empty cache. Correspondingly, the PowerDNS server is initialized with an empty cache as well. We continuously query two servers using pre-generated query packets and measur the corresponding latency. The results are illustrated in Fig. 3-6a. Explicitly, we find that B-DNS could achieve approaching or even better lookup performance than PowerDNS. We remark that this is because the PowerDNS server obtains the queried IP address by recursive resolution while B-DNS can fetch records from the locally-stored blockchain directly. In recursive resolution, PowerDNS may suffer from the network congestion and packet loss. By contrast, B-DNS can provide more stable and efficient name service as long as the record has been stored in the blockchain.

#### Flash-crowd Effect

In this experiment, we test the resilience of B-DNS when it faces the flash-crowd effect. Specifically, the flash-crowd effect in DNS refers to sudden upheavals in the frequency of queried domain names. The server setting in this experiment is the same as the former one. We start by continuously sending DNS query packets to B-DNS server and PowerDNS server for three hours. Then, we flipped the popularity of queried domain names. Specifically, the most popular domain name becomes the least popular and the second popular domain name becomes the second least popular, and so on. We measure the corresponding query latency and use the median in each minute to illustrate the trend as shown in Fig. 3-6b. We notice that the latency of PowerDNS is much higher than that of B-DNS at the very beginning and between *180*-th to *210*-th mins. This is because of the uncached query, which requires PowerDNS to launch recursive resolution. Specifically, when facing flash-crowd effect, the query latency of PowerDNS increases substantially while that of B-DNS remains stable.

## 3.6 Security Analysis

In this section, we discuss how B-DNS handles the DDoS attack and two other potential attacks: the sybil attack, the index attack, and the hash collision attack.

### 3.6.1 DDoS Attack

B-DNS can provide great resistance against the DDoS attack. The underlying blockchain distributes content to a large number of nodes. In addition, the peer-to-peer structure of B-DNS makes it hard to attack all the B-DNS name servers. Though some name servers may be compromised by the DDoS attack, it will not affect the overall name service.

### 3.6.2 Sybil Attack

Sybil Attack is a type of attack seen in peer-to-peer networks in which a node in the network operates multiple identities actively at the same time and undermines the authority/power in this system. In B-DNS, the sybil attack can be launched by one registry mispresenting the number of domain names registered by it and so pretending to control a huge amount of stake. We argue that in B-DNS the cost to launch sybil attack is tremendously high. A registry cannot arbitrarily claim the amount of its registered domains. The other nodes can easily check its real stake by quickly traversing the whole blockchain. In this case, the only way to increase your stake is to register as many domain as possible, which is costly and tardy. Additionally, once a registry is caught to lie in its stake, B-DNS can eliminate its domain registration power in a soft-fork, which in turn warn the other registries to behave honestly.

### 3.6.3 Index Attack

The index attack is conducted by forging an incorrect attack and send it to honest peers. This stems from the fact that an adversary may try to create a fork when it generates a new block. However, the same as Bitcoin, B-DNS sets a security parameter 6, which represents that a block is stable when it is 6-block deep in the blockchain. B-DNS can ensure the correctness of the index by only converting the stable portion of the blockchain. In addition, B-DNS encapsulates the parameter `IndexHash` into the header. The parameter `IndexHash` is the hash value of the latest index. It allows each node to verify whether their index has been updated to the latest state of the blockchain.

### 3.6.4 Hash Collision Attack

The hash collision attack against DNS is that different domain names have identical hash value. The B-DNS index is implemented using a hash map instead of a hash table. A hash value corresponds to a bucket, which can accommodate eight operation records. As a result, if two domain names have identical hash value, they can be stored in the bucket without causing a collision. Then, we calculate the possibility of a hash collision. Given that there are $N$ possible hash values and $k$ ($k < N$) domain names. The possibility of $k$ domain names have different hash value is

$$\frac{N-1}{N} \times \frac{N-2}{N} \times ... \times \frac{N-(k-2)}{N} \times \frac{N-(k-1)}{N}$$

Accordingly, the collision possibility of a hash table is

$$P = 1 - e^{\frac{-k(k-1)}{2N}}$$

We then investigate the hash collision rate of B-DNS. Assuming that tens of millions domain names have been registered in B-DNS, the index utilizes a hash function whose hash range is $2^{52}$. The collision rate is illustrated in Fig. 3-7. When the number of stored domain names is less than 60,000,000, the collision rate is almost zero. The probability is less than 3% even when we store 100,000,000 records. If we need to store more records, we can maintain a low collision rate by increasing the hash range. Also, B-DNS index can be improved by using separate chaining with linked lists to entirely resolve the collisions. Thus, B-DNS is resistant to hash collision attack.

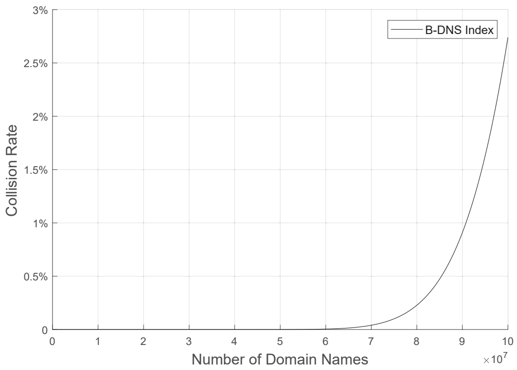


Figure 3-7: The collision rate of the B-DNS index.

## 3.7 Related Work

In this section, we introduce the related work on enhancing the security and performance of legacy DNS, and efforts that have been devoted to implementing blockchain-based domain name systems.

### 3.7.1 DNS Security and Performance

Several methods have been proposed to defend against the cache poisoning attack. Dagon et al. proposed to mix the upper and lower case spelling of the domain name in the query packet so that the adversary cannot succeed in poisoning unless he guesses the right combination of upper and lower case letters [33]. Perdisci et al. utilized the wildcard domain names [35], which are in the form of `*.example.com` where `*` can be recognized as any combination of characters [91]. Accordingly, a recursive resolver can prepend random strings to the queried domain name and still obtain the correct answer. In this way, a recursive resolver can distinguish a valid response by checking whether the combination of upper/lower cases or random strings matches its query packet. In addition, the cache can also be used to track user behaviors [59]. Klein et al. proposed a user tracking technique to track user behaviors even if they use "privacy mode" browsing, or use multiple browsers.

DNSSEC creates a trust chain from the root server to the authoritative server, in which way a recursive resolver can check the query route of the response packet by verifying the signatures [47]. An attacker can hardly tamper with a DNSSEC-deployed domain because they cannot forge signatures. However, though DNSSEC has been proposed for decades, its deployment rate is still meager nowadays [69]. Recent survey reveals that only 1% of `.com`, `.net`, and `.org` domains have enabled DNSSEC [12]. Several reasons account for this phenomenon: the sophisticated deployment procedure of DNSSEC, additional cost [31], and political reason that some countries may be hostile to the country where the root server is located [91]. For example, Cuba may deny DNS packets originated from the USA. In addition, Shulman et al. conducted an Internet study of the cryptographic security of DNSSEC-signed domains [99]. They collected 2.1 million DNSSEC keys and found that 35% are singed with RSA keys that share their modulo with some other domain and 66% use keys that are too short. They conclude that this problem arises from the poor key generation practices.

Additionally, researchers proposed T-DNS [125], a TCP-based DNS, to enhance the security of legacy DNS and defend against these former mentioned attacks. It uses the TLS to establish secure communication channels from clients to resolvers and from resolvers to authoritative servers.

Facing the DDoS attack, Pappas et al. noticed the importance of `NS` resource record [87]. They mentioned that prolonging the TTL of `NS` record could ensure a certain degree of availability of these domains when their father domain is under a DDoS attack. Similarly, Ballani et al. proposed to build a separate "stale cache" in the recursive resolver to store expired records [9]. In this way, if a recursive resolver does not receive the response from the authoritative server, it could use stored records in the stale cache to complete the query process. Besides, some efforts were devoted to evaluating the performance of root servers under DDoS attacks [80]. Their work demonstrated that massive attacks could overwhelm some letters (letters are used to represent root servers, e.g., root server `A-M`) so that securer mechanisms should be developed to improve the DNS security.

There are also some work on improving the performance of DNS. Park et al. proposed CoDNS [88], a lightweight and cooperative DNS lookup service that can be independently and incrementally deployed to existing nameservers. They demonstrate that CoDNS can reduce the lookup latency by 28-82%, which greatly improves the performance of legacy DNS. Gao et al. focused on the update performance of managed DNS, which consumes dozens of seconds to complete, and proposed feasible improvement techniques [46]. Alouf et al. introduced an analytical models to study expiration-based caching systems based on renewal arguments and found that no distribution maximizes the hit probability anywhere in a network of caches. [4]. Liu et al. proposed ContainerDNS [71], a scalable high-performance DNS for large-scale container cloud platforms, which maximizes DNS's performance and scalability by optimizing packet processing and using efficient memory and cache management.

### 3.7.2 Blockchain-based DNS

Namecoin was proposed to build a blockchain-based namespace [82]. It was forked from the Bitcoin so that it shares lots of similarities with the Bitcoin, including the block size, mining interval, and scripting system (with a few additions). Namecoin adopts the merged mining to ensure its data consistency. A user could register unlimited domains as long as it has enough coins. However, an empirical study on Namecoin shows that most registered domains are inactive and squatted [52], which is of great danger to a naming system [5]. Blockchain-DNS [15] provided a browser-side name resolution service for Namecoin. However, as Namecoin has many intrinsic problems, the usage of Blockchain-DNS is limited.

Ali et al. proposed Blockstack [3], a blockchain-based naming and storage system. Blockstack separates its control plane and data plane. In this case, it can introduce new functionalities without forking the underlying blockchain. Blockstack also has some advantages such as the cross-chain migration ability and fast bootstrapping. These properties make it easier to deploy the Blockstack system. Blockstack can also provide good read/write performance with limited computation overhead.

## 3.8 Conclusion

In this chapter, we propose B-DNS, a secure and efficient blockchain-based DNS. B-DNS is compatible with current DNS and can provide better defense against the cache poisoning attack and the DDoS attack. The experiment demonstrated the good security of B-DNS by comparing it with legacy DNS according to attack success rate, attack cost, and attack surface. B-DNS can also provide efficient name service compared with legacy DNS. Our work actively explored the construction of the next-generation DNS infrastructure and provides a potential solution for building domain name systems for wide area network, local area network, or intranet.

# Chapter 4

# Pistis: Issuing Trusted and Authorized Certificates With Blockchain and TEE

## 4.1 Introduction

SSL/TLS certificates issued by Certificate Authorities (CAs) form the security foundation of HTTPS connection. Clients establish HTTPS connections with a website only when its server provides a valid certificate to prove its identity. However, current CAs are vulnerable to be compromised to issue *unauthorized certificates*, which arises from the fact that a compromised CA might issue certificates without domains' permission. As an example, in 2011, attackers compromised the private key of DigiNotar and maliciously issued unauthorized certificates for Google, which were used to launched Man-in-the-Middle (MitM) attacks against Google services [115]. Similar incidents happened dozens of times [48, 57].

To mitigate this problem, some countermeasures such as HTTP Public Key Pinning (HPKP) [39] and Certificate Transparency (CT) [64] have been proposed. HPKP is a straightforward way through which a server provides a list of trusted public keys to clients. CT builds a log system to monitor the operation of certificates, which enables clients to detect unauthorized certificates. In addition, researchers proposed some other countermeasures, such as Accountable Key Infrastructure (AKI) [58] to disperse centralized trust, Attack Resilient Public Key Infrastructure (ARPKI) [11] & PoliCert [105] to log certificate operations, and Certificate Issuance and Revocation Transparency (CIRT) [96] to provide an efficient certificate verification method.

However, these countermeasures have some common limitations. What they can do is to detect the issuance of rogue certificates or reduce the probability of unauthorized certificate problems. In most cases, attacks have already caused damage before unauthorized certificates are detected. In addition, when an unauthorized certificate is reported, browser vendors typically add the corresponding CA to the blacklist. However, some compromised CAs are too big to fail. For example, although Symantec was reported to issue unauthorized certificates to `Google.com` in 2015 [100], it was unrealistic to block Symantec immediately since it had controlled more than 10% of active certificates by that time. Blacklisting Symantec will block millions of websites at the same time.

We observe that three reasons account for the problem of unauthorized certificates: 1. traditional centralized CAs might be compromised by attackers; 2. traditional centralized CAs can issue certificates without domains' permission (i.e., unauthorized); 3. operations of traditional centralized CAs are opaque. Furthermore, we find that people typically trust a CA based on its identity. However, this trust relationship is fragile especially when the trusted CA might be compromised. In this case, we consider addressing this issue through a complete certificate issuance process design, which restricts only the domain owner can apply for a certificate related to that domain. A certificate is issued only when its applicant passes domain validation. We also ensure that CAs are hardly compromised. The advent of blockchain makes our idea feasible.

Blockchain reproduced smart contract, whose execution is immutable, transparent, and deterministic. Considering building a CA based on smart contract, we find that: 1. the immutable feature makes it difficult for an attacker to compromise a CA based on smart contract, which eliminates the problem of traditional CAs being manipulated in the event of key breaches; 2. the transparent feature let people trust a CA based on its logic rather than its identity, and the blockchain becomes the log of certificates' operation history naturally; 3. the deterministic feature allows CA developers to implement autonomous and complete CA logic through sound contract design. In PISTIS, we hard coded that each certificate applicant should prove the ownership of the domain related to its requested certificate to the smart contract before certificate issuance. This process is called domain (ownership) validation. However, it is challenging to conduct domain validation in smart contracts solely.

Smart contracts work on overlay networks (i.e., blockchain P2P network), which are only accessed by blockchain nodes, and cannot communicate with domain servers directly. In this case, traditional domain validation methods via DNS or email cannot work. It is also infeasible for external validators to pass validation results to a smart contract directly. Due to the public nature of blockchain, anyone can pass information to smart contracts so that a smart contract cannot tell the authenticity of provided validation results. In this case, we need a mechanism that can conduct domain ownership validation and transfer domain validation results to smart contracts in a trusted way. Inspired by Town Crier [121], which employs TEE as smart contract's trusted information source, we intend to address this challenge with the TEE technology.

A TEE isolates a memory space to prevent other software applications from learning or tampering with the data inside enclaves. The attested execution model of TEE enables others to check whether an attested result is outputted by an expected program. We leverage TEE to validate whether a domain is under the control of its corresponding certificate applicant. Once the domain related to a requested certificate passes ownership validation, TEE nodes pass validation results to the smart contract. Then, after verifying the attested validation results, the smart contract can issue a certificate as request to the applicant.

In this chapter, we present PISTIS[1], a framework for issuing authorized and trusted certificates with blockchain and TEE. PISTIS sets blockchain as the root of trust. Clients can decide whether to trust a smart contract on the blockchain based on its

[1] In Greek mythology, PISTIS was the personification of trust and reliability.

logic. Pistis employs two contracts: authority contract $\mathcal{C}_{\text{AC}}$ and verification contract $\mathcal{C}_{\text{VC}}$. $\mathcal{C}_{\text{AC}}$ is for certificate issuance and revocation, and $\mathcal{C}_{\text{VC}}$ is for certificate validity verification.

Pistis is designed to be TEE-agnostic (i.e., the underlying TEE implementation is changeable and upgradeable). Pistis combines the security of blockchain system with TEE-based execution. Pistis's design supports rigorous analysis of its security properties under cryptographic ideal functionality. The main challenges of this solution, such as TEE failures and DNS failures are discussed in Section 4.2.1.

Our main contributions are summarized as follows:

- **Authorized and trusted certificate issuance:** Pistis can ensure all its issued certificates are authorized and trusted. The issuance is authorized because Pistis leverages TEE nodes to validate the actual control of registrants over domains related to requested certificates. The attested validation results provided by TEE nodes allow the Pistis smart contract to issue certificates to registrants who have passed the domain validation. As these Pistis-issued certificates are recorded on the blockchain, they are trusted. Details are discussed in Section 4.3.3.

- **Efficient certificate verification:** Pistis maintains a Merkle Patricia Tree (MPT), which stores latest certificates' states, to enable efficient certificate verification. In this tree, the key is a Fully Qualified Domain Name (FQDN), and the value field stores the state of the corresponding certificate. For the client, Pistis provides web3.js scripts, which are executed in browsers and can verify the validity of Pistis-issued certificates. Detailed information are discussed in Section 4.3.3.

- **Formal modeling and security analysis:** We formally model the Pistis protocol as $\mathbf{Prot}_{\text{Pistis}}$ in Section 4.3 and give its ideal functionality $\mathcal{F}_{\text{Pistis}}$ in Section 4.4.1. We formally prove the security of Pistis in the universally composable (UC) framework [23] in Section 4.4.2. By showing that $\mathbf{Prot}_{\text{Pistis}}$ UC-realizes the ideal functionality $\mathcal{F}_{\text{Pistis}}$, Pistis can be seamlessly integrated in more complex cryptographic protocols such as HTTPS.

- **Implementation and evaluation:** We implemented a prototype of Pistis based on Ethereum and Intel SGX, which could issue X.509 certificates according to the specification. We tested the gas consumption, storage overhead, and verification latency of Pistis. The experiment results demonstrate that Pistis can provide an efficient certificate verification service to end users.

## 4.2 System Design

### 4.2.1 Challenges

Before diving into the specifics of Pistis, we first describe and address the fundamental pitfalls that arise when hybridizing smart contracts and TEE. Note that designing such a protocol that integrates smart contract with TEE to issue certificates is nontrivial, which requires us to resolve the following challenges.

#### TEE Failures

Though ensuring the integrity and confidentiality of enclave execution, TEE is not a panacea. We first consider the **availability** of TEE. In this chapter, we do not make honest host assumptions. By contrast, hosts may be malicious. A malicious host can drop messages, abort execution, or exhaust the hardware resource (e.g., conduct computation-heavy work). Furthermore, even honest hosts may encounter power loss, which makes the TEE unavailable. Pistis resolves this problem by employing a cluster of TEE nodes, where a TEE is easily replaced. We also assume that the adversary can compromise all but one TEE.

We then consider the breach of TEE **confidentiality**. Recent work has demonstrated the feasibility of extracting secrets from TEE enclaves via side-channel attacks [113]. Pistis addresses this problem in two ways. First, Pistis is designed to be TEE-agnostic so that vulnerable TEE can be upgraded with patched version promptly to resolve the discovered vulnerabilities. Second, Pistis leverages constant memory consumption and execution time enclaves to defend against side-channel attacks.

We finally consider the replay attack that may be launched by malicious hosts. Replay attack is to confuse the TEE states by re-sending previous messages. Pistis is fault-tolerant to this problem since all states are stored on the blockchain. As long as Pistis smart contract maintains a correct state, the stateless TEE will not confuse smart contract execution.

#### DNS Failures

Before issuing certificates, the domain ownership should be validated via DNS, which, however, has been demonstrated insecure [19]. In Pistis, we address this problem by conducting ownership validation from multiple vantage points. Specifically, TEE nodes are deployed in different ASes. Our approach is based on the assumption that the adversary cannot control the majority of the Internet, which was also made in [19].

### 4.2.2 Blockchain as a Root of Trust

In traditional PKI, root CAs act as the root of trust. They issue intermediate certificates for commodity CAs to issue certificates. In Pistis, we adopt the blockchain as a root of trust. We construct two contracts, an authorization contract $\mathcal{C}_{\text{AC}}$ and a verification contract $\mathcal{C}_{\text{VC}}$. $\mathcal{C}_{\text{AC}}$ takes charge of validating the domain and issuing certificates. Specifically, $\mathcal{C}_{\text{AC}}$ is a stateful smart contract that allows concurrent state transitions of different registrants. In the stateful contract, an execution either reaches the next state or revert, which protects Pistis from malicious certificate squatting. The state transition is depicted in the bottom left of Fig. 4-1. $\mathcal{C}_{\text{VC}}$ provides two interfaces: tree update and certificate verification. The tree update allows the authorization contract $\mathcal{C}_{\text{AC}}$ to update certificates' states, and the certificate verification enables clients to verify certificates.

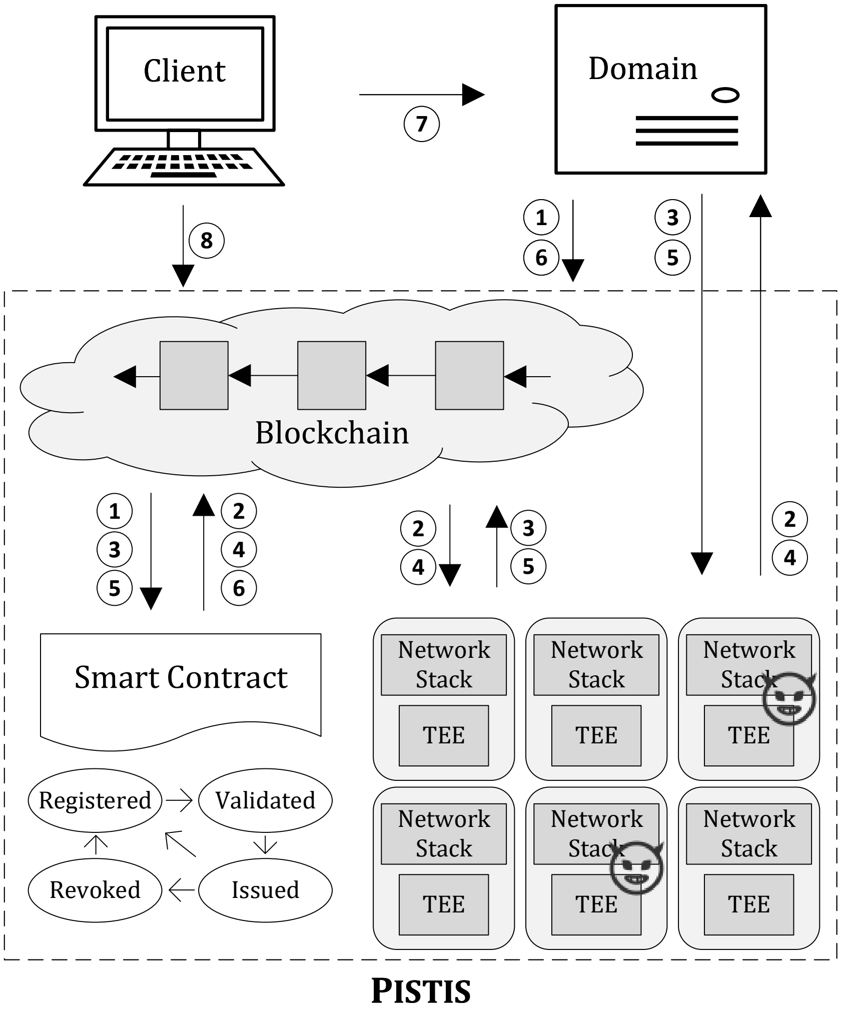


Figure 4-1: Architecture and workflow of PISTIS. PISTIS consists of three parts: blockchain, smart contract and TEE nodes. A domain server can request a certificate from PISTIS. A client can verify a certificate provided by the domain server via the blockchain in PISTIS.

### 4.2.3 Architecture and Workflow

Fig. 4-1 illustrates the architecture of Pistis. As it shows, Pistis consists of blockchain, smart contract, and TEE nodes (some of them might be malicious). A domain, or accurately its owner can request a certificate from Pistis via the blockchain. A client can verify the authenticity of Pistis-issued certificates by querying the blockchain when connecting to a domain. Then, we discuss the workflow:

**Certificate Request.** Similar to certificate signing request (CSR), an applicant should first register an entry in Pistis smart contract (Step ①), which contains a fully qualified domain name, an entity name, and a public key.

**Domain Validity Validation.** Pistis only issues certificates to domains with `Expiry Date` longer than $n$ seconds, which is an adjustable parameter that equals to the validity of Pistis-issued certificates. The smart contract invokes the TEE to check the validity of requested domain by posting an invocation transaction onto the blockchain. When a TEE node receives latest block and parses the invocation transaction, it conducts the validation (Step ②). A TEE node typically queries the whois database for the corresponding `Expiry Date` and converts it to a Unix timestamp. By subtracting the timestamp of latest block from `Expiry Date`, the TEE node can get the remaining validity of this domain. The smart contract processes the certificate issuance request only when domain's valid period is longer than a predefined length (i.e., 3 months). The validation results will be put on the blockchain as a transaction that calls Pistis smart contract to update the corresponding domain's state (Step ③).

**Domain Ownership Validation.** Pistis smart contract invokes TEE nodes to validate whether an applicant controls the provided domain. Specifically, smart contract posts an invocation transaction onto the blockchain until TEE nodes receive it (Step ④). Pistis adopts a secure 2-party computation mechanism to conduct the validation (Step ⑤). We require that the domain server should be equipped with a TEE processor because the secure 2-party computation can only be achieved when both parties are equipped with TEE, which has been proved in [89]. Once a registrant passes the domain ownership validation, TEE nodes can upload the validation result onto the blockchain as transactions that calls the smart contract (Step ⑤). Then, Pistis smart contract can confirm that this domain is valid and under the control of the registrant. Following above steps, Pistis can ensure its issued certificates are authorized.

**Certificate Issuance and Revocation.** Once a domain passes the validity and ownership validation, Pistis smart contract can issue a certificate to it. The issuance (Step ⑥) and revocation (Step ⑥) are two publicly available functions that can be invoked by valid registrants. Pistis smart contract publishes issued certificates onto the blockchain, which allows clients to verify their validity and therefore trust these certificates (Step ⑥).

**Certificate Verification.** Pistis provides an efficient way to verify its certificates. When a client connects to a website that is protected by a Pisits-issued certificate (Step ⑦), it can leverage the web3.js script to query the authenticity of provided certificate via blockchain (Step ⑧).

### 4.2.4 Smart Contract Design

In this section, we present the implementation details of Pistis contracts, namely the certificate authorization contract $C_{AC}$ is illustrated in Fig. 4-2 and the certificate verification contract $C_{VC}$ is presented in Fig. 4-3.

### 4.2.5 Threat Model

We assume applicants are semi-honest, namely they behave honestly only when registering certificates for their own domains. However, during other domain owners' application, they may behave maliciously. We assume TEE hosts are malicious. They may delay, abort, relay arbitrary messages, or replay previously transmitted messages to enclaves. In addition, we assume that most active TEE enclaves are trustworthy, but a small set of enclaves might suffer from integrity or confidentiality problems. They may be compromised by other parties (i.e., malicious domain owners)

```
The Program of Authorization Contract $C_{AC}$

Functions
  Account Registration:
    On receiving Account_Request:
      If $pk_{acct}$, address $\notin$ account list;
      Insert {$pk_{acct}$, address} into account list;
  Identifier Authorization
    On receiving Identifier_Authorization:
      Parse {$pk_{acct}$, address}:
      If $pk_{acct}$ $\cup$ address $\notin$ account list: revert;
      If identifier $\notin$ account list: revert;
      Else:
        Invoke $\mathcal{R}$ to generate a challenging token $\rho$
      On receiving a response $R$:
      If $R = proof(\rho)$:
        Add the identifier to {$pk_{acct}$, address};
      Else: revert;
  Certificate Issuance
    On receiving Certificate_Issuance:
      Parse the {$pk_{acct}$, address, identifier};
      If $pk_{acct}$ $\cup$ address $\cup$ identifier $\notin$ account list:
        revert;
      Else:
        Generate a X.509 certificate cert;
      Send the cert to the domain owner via $\mathcal{R}$;
      Broadcast $cert_{id}$ on chain;
      Invoke $C_{VC}$ with {$cert$,$cert_{id}$,valid};
  Certificate Revocation
    On receiving Certificate_Revocation or $cert$ expires:
      If {$pk_{acct}$, address, identifier} $\in$ account list:
        Invoke $C_{VC}$ with {$cert$,$cert_{id}$,invalid};
```

Figure 4-2: The Implementation of Certificate Authorization Contract $C_{AC}$.

```
The Program of Verification Contract $C_{VC}$
Functions
    Certificate Verification
    On receiving Verification_Request:
        Parse {$cert_{id}$, identifier}:
        For MPT: search identifier;
        If not found: return false;
        If found: compare $cert_{id}$;
            If not equal: return false;
        Else: parse {status}:
            If valid: return true;
            Else: return false;
    MPT Update
        On receiving the MPT_update:
        Parse the {$cert_{id}$, identifier, status};
        For MPT: search identifier;
        If not found:
            Add a node {$cert_{id}$, identifier, valid};
        Else: Compare the $cert_{id}$;
        If equal: Update the {status};
        Else: Revert;
```

Figure 4-3: The Implementation of Certificate Verification Contract $C_{VC}$.

or external attackers who want to violate the certificate issuance security.

We assume the underlying blockchain satisfies three properties: liveness, consistency, and immutability. Liveness means a transaction will be included in the blockchain in a fixed time period. Consistency guarantees that all blockchain participants have the same view on the state of the blockchain eventually. Immutability implies that once a transaction is confirmed $k$ times, it cannot be reverted.

We consider two types of adversaries: internal byzantine adversary and external adversary. For byzantine adversary, they may control the operating system and network stacks of TEE nodes, which can reorder, replay, drop transmitted messages, and schedule processes arbitrarily. For external adversary, they observe global network traffic, and may reorder or delay messages arbitrarily.

## 4.3 The Pistis Protocol

In this section, we specify $\mathbf{Prot}_{\text{Pistis}}$, which aims to realize a Universal Composable (UC) [23] ideal functionality $\mathcal{F}_{\text{Pistis}}$. $\mathbf{Prot}_{\text{Pistis}}$ utilizes digital signature scheme $\Sigma$(Gen,Sign,Verf), a symmetric encryption scheme $\mathcal{SE}$(Gen,Enc,Dec), and an asymmetric encryption scheme $\mathcal{AE}$(Gen,Enc,Dec).

### 4.3.1 Blockchain Model

We define the underlying blockchain as a general-purpose append-only ledger $\mathcal{F}_{\mathbf{B}}$ that maintained by common blockchain protocols. The blockchain is comprised as a chain of blocks that store transactions. We let the overlay semantics of blockchain as follows:

- $\mathcal{F}_{\mathbf{B}}$.account: it is used to generate a new account with an address on the blockchain.
- $\mathcal{F}_{\mathbf{B}}$.latest($n$): it is used to download the latest $n$ blocks. By default, $n = 1$.
- $\mathcal{F}_{\mathbf{B}}$.post(tx): it is used to broadcast a transaction onto the blockchain. The broadcasted transaction will be included in $\delta$ blocks.

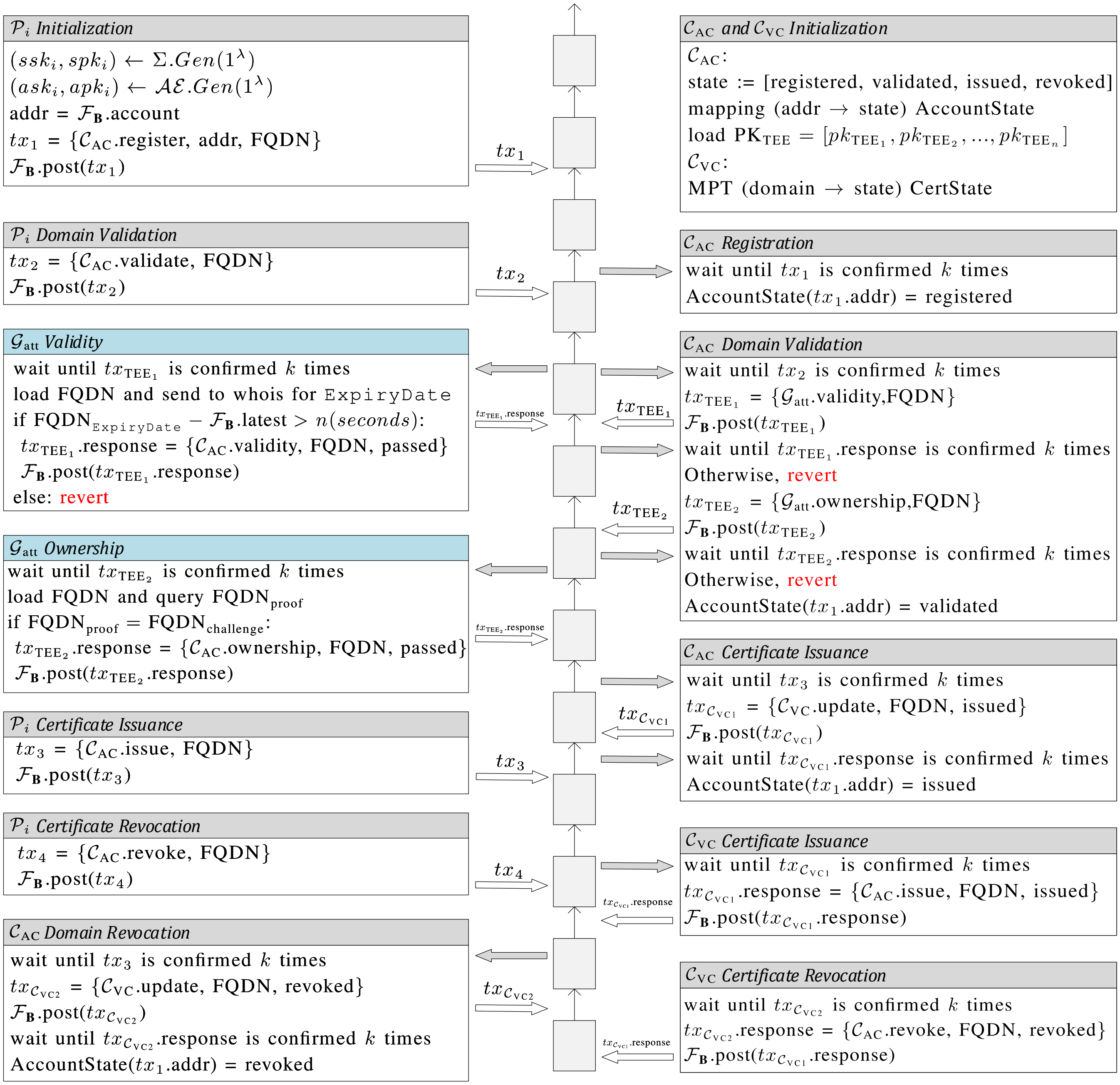
### 4.3.2 TEE Model

We adopt the attested execution model formalized in [89] and define our TEE as an ideal functionality $\mathcal{G}_{att}$. Similar to the notation in [89], a party that loads en enclave into TEE with an "install" message. A party that invokes the TEE with a "resume" call.

### 4.3.3 Formal Specification of the Protocol

In this section, we give the formal protocol of Pistis, which is depicted in Fig. 4-4.

**Pistis Registration:** An applicant $\mathcal{P}_i$ who wants to request a certificate from Pistis should first create a blockchain account and register its domain in the $\mathcal{C}_{\text{AC}}$.



*$\mathcal{P}_i$ Initialization*

```
$(ssk_i, spk_i) \leftarrow \Sigma.Gen(1^\lambda)$
$(ask_i, apk_i) \leftarrow \mathcal{AE}.Gen(1^\lambda)$
addr = $\mathcal{F}_\mathbf{B}$.account
$tx_1$ = {$\mathcal{C}_{\text{AC}}$.register, addr, FQDN}
$\mathcal{F}_\mathbf{B}$.post($tx_1$)
```

*$\mathcal{P}_i$ Domain Validation*

```
$tx_2$ = {$\mathcal{C}_{\text{AC}}$.validate, FQDN}
$\mathcal{F}_\mathbf{B}$.post($tx_2$)
```

*$\mathcal{G}_{\text{att}}$ Validity*

```
wait until $tx_{\text{TEE}_1}$ is confirmed $k$ times
load FQDN and send to whois for ExpiryDate
if FQDN$_{\texttt{ExpiryDate}}$ $- \mathcal{F}_\mathbf{B}$.latest $> n(seconds)$:
 $tx_{\text{TEE}_1}$.response = {$\mathcal{C}_{\text{AC}}$.validity, FQDN, passed}
 $\mathcal{F}_\mathbf{B}$.post($tx_{\text{TEE}_1}$.response)
else: revert
```

*$\mathcal{G}_{\text{att}}$ Ownership*

```
wait until $tx_{\text{TEE}_2}$ is confirmed $k$ times
load FQDN and query FQDN$_{\text{proof}}$
if FQDN$_{\text{proof}}$ = FQDN$_{\text{challenge}}$:
 $tx_{\text{TEE}_2}$.response = {$\mathcal{C}_{\text{AC}}$.ownership, FQDN, passed}
 $\mathcal{F}_\mathbf{B}$.post($tx_{\text{TEE}_2}$.response)
```

*$\mathcal{P}_i$ Certificate Issuance*

```
 $tx_3$ = {$\mathcal{C}_{\text{AC}}$.issue, FQDN}
 $\mathcal{F}_\mathbf{B}$.post($tx_3$)
```

*$\mathcal{P}_i$ Certificate Revocation*

```
 $tx_4$ = {$\mathcal{C}_{\text{AC}}$.revoke, FQDN}
 $\mathcal{F}_\mathbf{B}$.post($tx_4$)
```

*$\mathcal{C}_{\text{AC}}$ Domain Revocation*

```
wait until $tx_3$ is confirmed $k$ times
$tx_{\mathcal{C}_{\text{VC2}}}$ = {$\mathcal{C}_{\text{VC}}$.update, FQDN, revoked}
$\mathcal{F}_\mathbf{B}$.post($tx_{\mathcal{C}_{\text{VC2}}}$)
wait until $tx_{\mathcal{C}_{\text{VC2}}}$.response is confirmed $k$ times
AccountState($tx_1$.addr) = revoked
```

*$\mathcal{C}_{\text{AC}}$ and $\mathcal{C}_{\text{VC}}$ Initialization*

```
$\mathcal{C}_{\text{AC}}$:
state := [registered, validated, issued, revoked]
mapping (addr → state) AccountState
load PK$_{\text{TEE}}$ = $[pk_{\text{TEE}_1}, pk_{\text{TEE}_2}, ..., pk_{\text{TEE}_n}]$
$\mathcal{C}_{\text{VC}}$:
MPT (domain → state) CertState
```

*$\mathcal{C}_{\text{AC}}$ Registration*

```
wait until $tx_1$ is confirmed $k$ times
AccountState($tx_1$.addr) = registered
```

*$\mathcal{C}_{\text{AC}}$ Domain Validation*

```
wait until $tx_2$ is confirmed $k$ times
$tx_{\text{TEE}_1}$ = {$\mathcal{G}_{\text{att}}$.validity,FQDN}
$\mathcal{F}_\mathbf{B}$.post($tx_{\text{TEE}_1}$)
wait until $tx_{\text{TEE}_1}$.response is confirmed $k$ times
Otherwise, revert
$tx_{\text{TEE}_2}$ = {$\mathcal{G}_{\text{att}}$.ownership,FQDN}
$\mathcal{F}_\mathbf{B}$.post($tx_{\text{TEE}_2}$)
wait until $tx_{\text{TEE}_2}$.response is confirmed $k$ times
Otherwise, revert
AccountState($tx_1$.addr) = validated
```

*$\mathcal{C}_{\text{AC}}$ Certificate Issuance*

```
wait until $tx_3$ is confirmed $k$ times
$tx_{\mathcal{C}_{\text{VC1}}}$ = {$\mathcal{C}_{\text{VC}}$.update, FQDN, issued}
$\mathcal{F}_\mathbf{B}$.post($tx_{\mathcal{C}_{\text{VC1}}}$)
wait until $tx_{\mathcal{C}_{\text{VC1}}}$.response is confirmed $k$ times
AccountState($tx_1$.addr) = issued
```

*$\mathcal{C}_{\text{VC}}$ Certificate Issuance*

```
wait until $tx_{\mathcal{C}_{\text{VC1}}}$ is confirmed $k$ times
$tx_{\mathcal{C}_{\text{VC1}}}$.response = {$\mathcal{C}_{\text{AC}}$.issue, FQDN, issued}
$\mathcal{F}_\mathbf{B}$.post($tx_{\mathcal{C}_{\text{VC1}}}$.response)
```

*$\mathcal{C}_{\text{VC}}$ Certificate Revocation*

```
wait until $tx_{\mathcal{C}_{\text{VC2}}}$ is confirmed $k$ times
$tx_{\mathcal{C}_{\text{VC2}}}$.response = {$\mathcal{C}_{\text{AC}}$.revoke, FQDN, revoked}
$\mathcal{F}_\mathbf{B}$.post($tx_{\mathcal{C}_{\text{VC1}}}$.response)
```

$tx_1$
$tx_2$
$tx_{\text{TEE}_1}$.response
$tx_{\text{TEE}_1}$
$tx_{\text{TEE}_2}$
$tx_{\text{TEE}_2}$.response
$tx_{\mathcal{C}_{\text{VC1}}}$
$tx_3$
$tx_4$
$tx_{\mathcal{C}_{\text{VC1}}}$.response
$tx_{\mathcal{C}_{\text{VC2}}}$
$tx_{\mathcal{C}_{\text{VC1}}}$.response

Figure 4-4: A formal specification of PISTIS protocol. Gray arrows indicate reading blockchain data. White arrows indicate broadcasting a transaction onto the blockchain.

In this phase, $\mathcal{P}_i$ can invoke $\mathcal{F}_{\mathbf{B}}$.post to broadcast a transaction to invoke $\mathcal{C}_{\text{AC}}$'s registration function. $\mathcal{C}_{\text{AC}}$ will mark $\mathcal{P}_i$'s blockchain address as `registered` and store its provided domain in the contract.

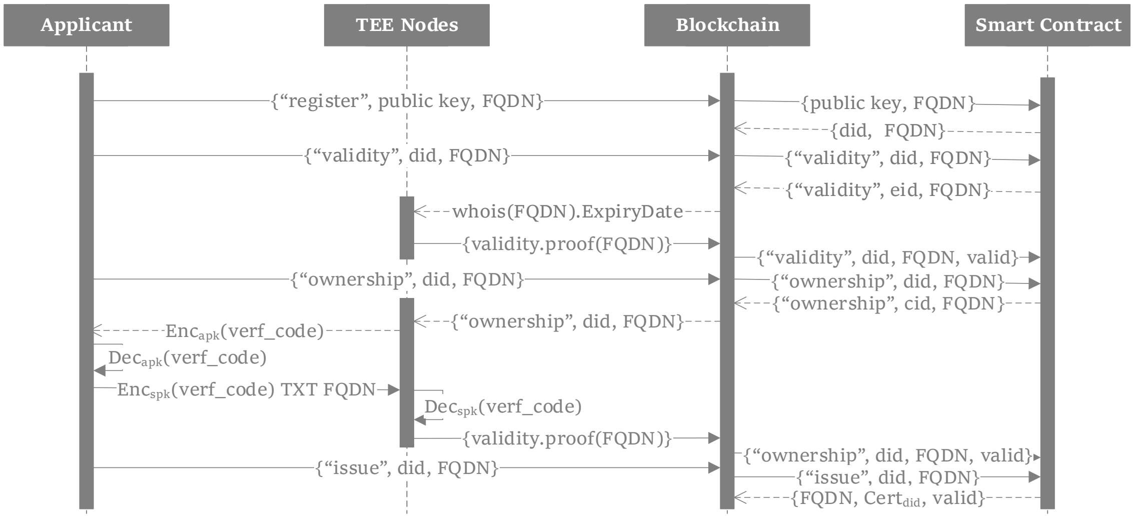


Figure 4-5: The *challenge-proof* protocol. Under TEE nodes, dashed line means the communication does not involve the enclave while the thick solid line represents that enclaves are involved.

**Domain Validation:** In this phase, the contract $\mathcal{C}_{\text{AC}}$ needs to ensure that the registered domain is valid (i.e., its validity is longer than $n$ seconds) and under the control of applicant $\mathcal{P}_i$.

For the **validity** validation, $\mathcal{C}_{\text{AC}}$ triggers TEE nodes by invoking the $\mathcal{F}_{\mathbf{B}}$.post function to broadcast a transaction that contains domain name and validity validation instructions.

For the **ownership** validation, we refer to the Automated Certificate Management Environment (ACME) protocol [19] and propose a *challenge-proof* protocol, which is illustrated in Fig. 4-5.

First, the applicant $\mathcal{P}_i$ invokes the ownership validation function by posting a transaction with a triple tuple *{address, pk$_{acct}$, domain}*. Contract $\mathcal{C}_{\text{AC}}$ checks whether the sender has registered an account. If so, $\mathcal{C}_{\text{AC}}$ generates a verification code $\sigma_{\mathcal{P}_i}$ and sends it to the applicant $\mathcal{P}_i$ as a return value.

Then, the applicant $\mathcal{P}_i$ puts the *proof* as a `TXT` resource record in its authoritative

DNS server in the form of `challenge.<domain> IN TXT` *`proof`*. $\mathcal{P}_i$ can invoke the PISTIS to check whether the *proof* matches the *challenge*.

The $\mathcal{C}_{\mathrm{AC}}$ then invokes TEE nodes to generate DNS query packets to `challenge.<domain>`. After receiving the `TXT` response, TEE nodes can extract the *proof* and respond to the PISTIS with a transaction. The PISTIS checks whether the received *proof* matches its corresponding *challenge*. If so, the domain passes the validation, and $\mathcal{C}_{\mathrm{AC}}$ mark the domain as `validated`.

**Certificate Issuance:** After passing the domain validation, applicant $\mathcal{P}_i$ can invoke the $\mathcal{C}_{\mathrm{AC}}$ to issue a certificate. $\mathcal{P}_i$ should generate a public key for its certificate and post this public key onto the blockchain. Then, $\mathcal{C}_{\mathrm{AC}}$ generates a certificate for $\mathcal{P}_i$ and broadcasts it to the blockchain.

$\mathcal{C}_{\mathrm{VC}}$ maintains a MPT to store the states of its issued certificates. Specifically, we set the domain name as the *key* of MPT. States of domain certificates (i.e., issued, revoked) are stored as *value*.

We make some adjustments to traditional MPT by indexing from the top-level domains (TLD). This is because domain names show similarity in their TLDs, and it is better to group similar domain names in a sub-tree. For example, `google.com` and `gmail.com` have identical TLD so that they can be categorized into one sub-tree. For the value part, there are two types. A valid certificate is given a concatenation in the form of `CertID||valid`. For a revoked certificate, the value part is in the form of `CertID||revoked`.

PISTIS $\mathcal{C}_{\mathrm{VC}}$ contract provides three operation functions `Insert`, `Update`, and `Get`. The first two functions enable $\mathcal{C}_{\mathrm{AC}}$ to update certificates' states, and the last function allows clients to verify certificates.

`Insert.` The insert function is operated as follows. First, we should find the node with the same top-level domain. Then, the TLD node's pointer will lead us to its branch node with 26 characters. After that, we check the first letter of the required domain and find its sub-tree. If this sub-tree is empty, we could insert the left letter as a leaf node after it. Otherwise, we go to the second letter and its sub-tree. We follow the letters one by one recursively until we find an empty sub-tree or run out of all letters in the domain name.

`Update.` The update operation is similar to the insert operation. The difference is when we update the state of a certificate, it has been inserted into the MPT. Accordingly, we could search the corresponding node from TLD nodes until we find the same longest prefix node of the domain name. We conduct this operation recursively until we match all letters in the domain. Finally, we could update the state of the found domain.

**Certificate Revocation:** In Pistis, certificate revocation can be triggered in three ways:

1. A certificate expires once the latest block's timestamp exceeds its validity. In this case, $\mathcal{C}_{\text{VC}}$ will update the state of this revoked certificate as invalid automatically.

2. A domain owner who owns the private key can sign a revocation request to Pistis to invalidate the certificate. In this case, the domain owner $\mathcal{P}_i$ should invoke the certificate revocation function directly by sending a transaction that contains its address, public key, domain, and certificate id.

3. A new applicant $\mathcal{P}_j$, who could prove its ownership of the domain related to a valid certificate, can invalidate that certificate. We consider this scenario because the ownership of a domain may change, while the previously-issued certificate is still valid. In this case, the new domain owner should explicitly invoke the Pistis certificate revocation function. Then, Pistis invokes the domain authorization function, and once the new domain owner passes the ownership validation, $\mathcal{C}_{\text{AC}}$ issues a new certificate and invokes $\mathcal{C}_{\text{VC}}$ to update certificate state.

**Certificate Verification:** When a client connects to a website protected by a Pistis-issued certificate, he/she could verify whether the certificate is valid before establishing a HTTPS connection. Only when the certificate passes verification, the client believes that the server is the authentic one. Otherwise, the browser should halt the connection. In PISTIS, certificate verification is conducted by the `Get` operation provided by contract $\mathcal{C}_{\text{VC}}$.

`Get`. The `Get` operation is a getter function provided by the underlying blockchain, which enables clients to acquire data directly. Specifically, the underlying blockchain implementation traverses the MPT to acquire the required state for users.

## 4.4 Security Analysis

In this section, we formally prove the security of PISTIS in the Universally Composable (UC) framework [23].

### 4.4.1 Ideal Functionality

The idea functionality of PISTIS is specified in Fig. 4-6 as $\mathcal{F}_{\text{PISTIS}}$. $\mathcal{F}_{\text{PISTIS}}$ allows applicants (each applicant is denoted by a unique id $\mathcal{P}_i$) to request for certificates. Following the convention in [23], we set an information leakage function $\ell$ to capture the allowed information leakage from the encryption. We also use the standard delayed output [23] to model the power of network adversary.

Applicants can send messages to $\mathcal{F}_{\text{PISTIS}}$ to invoke PISTIS registration, domain validation, certificate issuance, and certificate revocation, which will update the corresponding domain's state. Clients (i.e., environment $\mathcal{Z}$) can also query $\mathcal{F}_{\text{PISTIS}}$ for the state of a domain's certificate.

### 4.4.2 Security Proof

Intuitively, a PISTIS-issued certificate being authorized and trusted means that an adversary cannot convince the PISTIS to accept a response that differs from the expected content obtained from the specified domain.

**Theorem 1.** *(Security of $\boldsymbol{Prot}_{\text{PISTIS}}$). Assume $\Sigma_{TEE}$ is existentially unforgeable under chosen message attacks (EU-CMA), and $\mathcal{AE}$ is INC-CPA secure. Then $\boldsymbol{Prot}_{\text{PISTIS}}$ securely realizes $\mathcal{F}_{\text{PISTIS}}$ in the ($\mathcal{G}_{att}$,$\mathcal{F}_{\mathbf{B}}$)-hybrid model, for static adversaries.*

```
$\mathcal{F}_{\textsc{Pistis}}(\ell, \mathcal{P}_i)$
Parameter: leakage function $\ell : \{0,1\}^* \to \{0,1\}^*$
On receive (“register”, FQDN) from $\mathcal{P}_i$:
  did $\leftarrow \{0,1\}^\lambda$
  notify $\mathcal{A}$ of (“register”, $\mathcal{P}_i$, did, FQDN)
  Storage[did] := (FQDN, $\vec{0}$)
  send a public delayed output (“receipt”, did) to $\mathcal{P}_i$
On receive (“validity”, did, eid) from $\mathcal{P}_i$:
  notify $\mathcal{A}$ of (“validity”, $\mathcal{P}_i$, did, eid)
  (FQDN, st) := Storage[did]; abort if not found
  (outp, st) := whois(FQDN, did)
  notify $\mathcal{A}$ of ($\ell(outp)$, did, eid)
  update Storage[did] := (FQDN, st)
  send a public delayed output (“receipt”, did) to $\mathcal{P}_i$
On receive (“ownership”, FQDN, did, eid) from $\mathcal{P}_i$:
  notify $\mathcal{A}$ of (“ownership”, $\mathcal{P}_i$, did, eid)
  (FQDN, st) := Storage[did]; abort if not found
  send a secret delayed message (challenge, did) to $\mathcal{P}_i$
  notify $\mathcal{A}$ of ($\ell$(challenge), did, eid)
  update Storage[did] := (FQDN, st)
  send a public delayed output (“receipt”, did) to $\mathcal{P}_i$
On receive “issue”, FQDN) from $\mathcal{P}_i$:
  notify $\mathcal{A}$ of (“issue”, $\mathcal{P}_i$, did, eid)
  (FQDN, st) := Storage[did]; abort if not found
  update Storage[did] := (FQDN, st)
  send a public delayed output (“issued”, did) to $\mathcal{P}_i$
On receive “revoke”, FQDN) from $\mathcal{P}_i$:
  notify $\mathcal{A}$ of (“revoke”, $\mathcal{P}_i$, did, eid)
  (FQDN, st) := Storage[did]; abort if not found
  update Storage[did] := (FQDN, st)
  send a public delayed output (“revoked”, did) to $\mathcal{P}_i$
On receive (“read”, did) from $\mathcal{Z}$:
  (FQDN, st) := Storage[did]; abort if not found
  send st to $\mathcal{Z}$
```

Figure 4-6: The ideal functionality of Pistis.

*Proof.* Let $\mathcal{Z}$ be an environment and $\mathcal{A}$ be a "dummy adversary" (i.e., acts as a "transparent channel" between the environment $\mathcal{Z}$ and protocol [23]). To prove that $\mathbf{Prot}_{\textsc{Pistis}}$ UC-realizes $\mathcal{F}_{\textsc{Pistis}}$, we define a simulator Sim. In this case, no environment $\mathcal{Z}$ can distinguish an interaction between $\mathbf{Prot}_{\textsc{Pistis}}$ and $\mathcal{A}$ from an interaction between $\mathcal{F}_{\textsc{Pistis}}$ and Sim. That is, Sim satisfies

$$\forall \mathcal{Z},\ \mathrm{EXEC}_{\mathbf{Prot}_{\textsc{Pistis}},\mathcal{A},\mathcal{Z}} \approx \mathrm{EXEC}_{\mathcal{F}_{\textsc{Pistis}},\mathrm{Sim},\mathcal{Z}}$$

### Construction of Sim

Sim works as follows: if a message is sent by an honest applicant to $\mathcal{F}_{\textsc{Pistis}}$, Sim emulates appropriate real world "network traffic" for $\mathcal{Z}$ with information obtained from $\mathcal{F}_{\textsc{Pistis}}$. If a message is broadcasted by a corrupted party, Sim first extracts the input. Then, Sim interacts with the corrupted party with $\mathcal{F}_{\textsc{Pistis}}$.

**Pistis Registration:**

- If applicant $\mathcal{P}_i$ is honest, Sim obtains ($\mathcal{P}_i$, FQDN, eid) from $\mathcal{F}_{\textsc{Pistis}}$ and emulates an execution of the "create" call.
- If applicant $\mathcal{P}_i$ is corrupted, Sim extracts FQDN from $\mathcal{Z}$. On behalf of $\mathcal{P}_i$, Sim sends {"register", FQDN} to $\mathcal{F}_{\textsc{Pistis}}$ and instructs $\mathcal{F}_{\textsc{Pistis}}$ to deliver the output.
- In both cases, Sim acts as the adversary and honest parties to interact between $\mathcal{F}_{\mathbf{B}}$ and $\mathcal{G}_{\mathrm{att}}$, respectively.

**Domain Validity Validation:**

- If applicant $\mathcal{P}_i$ is honest, Sim obtains ($\mathcal{P}_i$, FQDN, did, eid) from $\mathcal{F}_{\textsc{Pistis}}$ and emulates an execution of the "validity" call. Specifically, Sim extracts the whois database and checks FQDN's validity. Then, Sim updates $\mathcal{F}_{\textsc{Pistis}}$ with (FQDN, eid, st).
- If applicant $\mathcal{P}_i$ is corrupted, Sim extracts FQDN from $\mathcal{Z}$. On behalf of $\mathcal{P}_i$, Sim sends {"validity", FQDN} to $\mathcal{F}_{\textsc{Pistis}}$ and instructs $\mathcal{F}_{\textsc{Pistis}}$ to extract the whois database.

**Domain Ownership Validation:**

- If applicant $\mathcal{P}_i$ is honest, Sim obtains ($\mathcal{P}_i$, FQDN, did, eid) from $\mathcal{F}_{\textsc{Pistis}}$ and emulates an execution of the "validity" call.

- If applicant $\mathcal{P}_i$ is corrupted, Sim extracts FQDN from $\mathcal{Z}$. On behalf of $\mathcal{P}_i$, Sim sends {"ownership", FQDN} to $\mathcal{F}_{\textsc{Pistis}}$ and instructs $\mathcal{F}_{\textsc{Pistis}}$ to deliver the output.

**Certificate Issuance:**

- If applicant $\mathcal{P}_i$ is honest, Sim obtains ($\mathcal{P}_i$, "issue", FQDN, eid) from $\mathcal{F}_{\textsc{Pistis}}$ and emulates an execution of the "revoke" call.

- If applicant $\mathcal{P}_i$ is corrupted, Sim extracts FQDN from $\mathcal{Z}$. On behalf of $\mathcal{P}_i$, Sim sends {"issue", FQDN} to $\mathcal{F}_{\textsc{Pistis}}$ and instructs $\mathcal{F}_{\textsc{Pistis}}$ to deliver the output.

**Certificate Revocation:**

- If applicant $\mathcal{P}_i$ is honest, Sim obtains ($\mathcal{P}_i$, "revoke", FQDN, eid) from $\mathcal{F}_{\textsc{Pistis}}$ and emulates an execution of the "revoke" call.

- If applicant $\mathcal{P}_i$ is corrupted, Sim extracts FQDN from $\mathcal{Z}$. On behalf of $\mathcal{P}_i$, Sim sends {"revoke", FQDN} to $\mathcal{F}_{\textsc{Pistis}}$ and instructs $\mathcal{F}_{\textsc{Pistis}}$ to deliver the output.

**Public Read:** On receiving any call ("read", eid) from party $\mathcal{P}_i$, Sim executes a "read" message to $\mathcal{F}_{\mathbf{B}}$. If $\mathcal{P}_i$ is corrupted, Sim sends to $\mathcal{F}_{\textsc{Pistis}}$ a "read" message on $\mathcal{P}_i$'s behalf and respond to $\mathcal{A}$.

**Corrupted Enclaves:** When $\mathcal{Z}$ corrupts enclaves, Sim obtains the id of them. In real world, $\mathcal{Z}$ could make corrupted enclaves unavailable at any time by terminating it. Sim relays all messages between a corrupted enclave and $\mathcal{Z}$. Sim stops $\mathcal{F}_{\textsc{Pistis}}$ once $\mathcal{Z}$ aborts the execution.

**Validity of Sim**

We show that, in hrbrid settings, no environment $\mathcal{Z}$ can distinguish an interaction with $\textbf{Prot}_{\textsc{Pistis}}$ and $\mathcal{A}$ from an interaction with $\mathcal{F}_{\textsc{Pistis}}$ and Sim. We consider the following sequence of hybrid settings, starting with the real protocol execution.

- Hybrid $H_1$ lets Sim to emulate $\mathcal{G}_{\text{att}}$ and $\mathcal{F}_{\mathbf{B}}$.
- Hybrid $H_2$ filters out the forgery attacks against $\Sigma_{\text{TEE}}$.
- Hybrid $H_3$ lets Sim emulate the issuance phase.
- Hybrid $H_4$ replaces the encryption of challenge with encryption of $\vec{0}$.

**Hybrid** $H_1$ proceeds as real world. To emulate $\mathcal{G}_{\text{att}}$, a key pair ($\text{pk}_{\text{TEE}}$ is generated by Sim, namely $\text{sk}_{\text{TEE}}$) for $\Sigma_{\text{TEE}}$. On the occasion that $\mathcal{A}$ communicates with $\mathcal{G}_{\text{att}}$, Sim stores $\mathcal{A}$'s information and emulates $\mathcal{G}_{\text{att}}$'s behavior. As to emulate $\mathcal{F}_{\mathbf{B}}$, Sim stores blockchain data internally.

From the perspective of $\mathcal{A}$', $H_1$ is simulated the same as the real world. In this case, $\mathcal{Z}$ cannot distinguish between $H_1$ and the real world execution.

**Hybrid** $H_2$ proceeds as in $H_1$, except for the following differences. If $\mathcal{G}_{\text{att}}$ is called by $\mathcal{A}$ with correct message, Sim receives state that is the output and $\sigma_{\text{TEE}}$ that is the attested result. Let $\Omega$ denote the set of all such tuples. Sim aborts on the occasion that $\mathcal{A}$ sends an attested result (st, $\sigma_{\text{TEE}}$) $\notin \Omega$ to $\mathcal{F}_{\mathbf{B}}$. The indistinguishability between $H_1$ and $H_2$ can be reduced to the EU-CMA property of $\Sigma$.

**Hybrid** $H_3$ is the same as $H_2$ but has Sim to emulate the certificate issuance. Sim emulates messages from $G_{att}$ to $F_B$ as described above. If $P_i$ is corrupted, Sim sends ("issue", FQDN) to $F_{\textsc{Pistis}}$ as $P_i$. From the perspective of $\mathcal{A}$, Sim emulates $G_{att}$ and $F_B$ successfully so that $\mathcal{Z}$ cannot distinguish between $H_3$ and the real world.

**Hybrid** $H_4$ is the same as $H_3$ except that honest applicants also send messages to $\mathcal{F}_{\textsc{Pistis}}$. If $\mathcal{P}_i$ is corrupted, Sim generates messages with $\mathcal{F}_{\textsc{Pistis}}$ as real-world. The indistinguishability between $H_3$ and $H_4$ can be recognized as a reduction to IND-CPA property of $\mathcal{AE}$. Adversary $\mathcal{A}$ cannot distinguish the encryption of $\vec{0}$ from the encryption of other messages.

□

## 4.5 Experiments and Evaluation

In this section, we conduct experiments to explore the feasibility of Pistis by evaluating the performance of contracts and TEE nodes.

### 4.5.1 Contract Evaluation

We implement Pistis contracts in Solidity and deploy them on the Ropsten testnet. We operate three nodes: a domain node, a TEE node, and a client node. All nodes are equipped with a blockchain endpoint. For client node, we install a MetaMask wallet, which is a web-based Ethereum wallet. The Geth wallet operates as the endpoint for the TEE node. It exchanges data between Pistis contracts and the TEE node. On the client side, we inject certificate verification script into the Pistis-protected websites. Only when Pistis returns a valid answer, the browser recognizes the certificate as valid and establishes a HTTPS connection.

We first evaluate the performance of Pistis contract. Three experiments are conducted to test the gas consumption, storage cost, and verification latency.

#### Gas Consumption

In this experiment, we test the gas consumption of most operations provided by Pistis. The gas consumption is closely related to the sustainability and feasibility of Pistis. Specifically, we measure the approximate computational steps (in Ethereum gas) and money cost (in USD) for each operation supported by the Pistis contract. During the writing of this paper (i.e., December of 2020), an ether costs around 500 USD. For the gas price, we adopt 40 Gwei (1 Gwei = $10^{-9}$ ether), which is 0.00002 USD. For testing, we assume all strings are a maximum of 32 bytes, which is the basic storage unit in Ethereum. We also assume that the public keys for certificate verification are 2048-bit RSA keys. Table 4.1 shows the costs of various operations in Pistis.

Table 4.1: The consumption of PISTIS operations.

| Operation | Gas Cost (Unit) | Gas Cost (USD) |
|---|---|---|
| Account_Registration | 38500 | 0.77 |
| Domain_Registration | 41800 | 0.836 |
| Domain_Validation | 52900 | 1.058 |
| Challenge_Generation | 65050 | 1.31 |
| Challenge_Verification | 3600 | 0.072 |
| Certificate_Issuance | 52500 | 1.05 |
| Certificate_Revocation | 4100 | 0.082 |
| Certificate_Verification | 0 | 0 |
| CertID_Broadcast | 1800 | 0.036 |

As we can see, the issuance of a certificate with PISTIS may cost a domain owner around 5 dollars, which is less than most commercial certificate authorities' certificate issuance service. For end clients, the certificate verification costs nothing so that they do not need to pay to visit PISTIS-protected websites.

### Storage Overhead

In this experiment, we investigate the storage overhead of PISTIS. Specifically, we investigate the on-chain storage for MPT, the account list, and transactions required for data transmission. For the MPT, we store the states of 11,239 certificates, which costs about 104 MB. For the account list, we insert 11,239 account entries, whose storage requirement is 8 MB. As the storage overhead increases linearly with the number of certificates, we can infer that the required storage space is in the TB level when there are hundreds of millions of certificates in the system. We think this requirement can be reached easily on current consumer computers.

### Verification Performance

In this experiment, we aim to test the performance of certificate verification in PISTIS and compare it with the latency of OCSP and CRL, two widely-used certificate verification methods. Typical certificate verification can be categorized into two parts. First, we need to verify the signature provided by the certificate and check whether

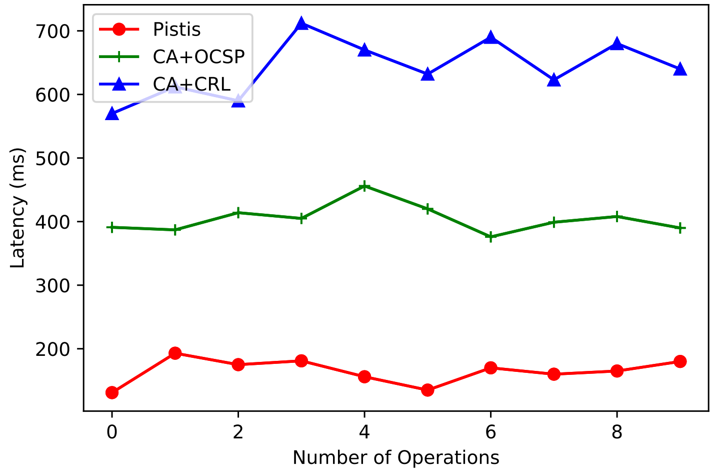


Figure 4-7: The comparison of verification latency between PISTIS, traditional CA + OCSP, and traditional CA + CRL.

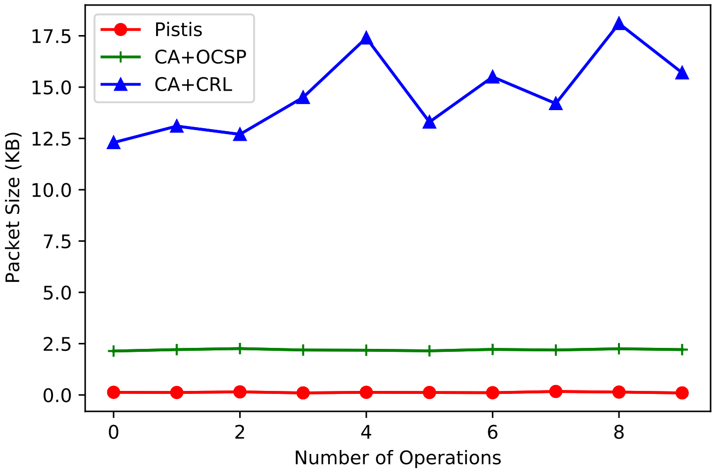


Figure 4-8: The comparison of verification packet size between PISTIS, traditional CA + OCSP, and traditional CA + CRL

it is assigned by a trusted CA. Second, we need to verify whether this certificate is revoked. A certificate will not pass the verification if either step fails.

The first step is conducted on the browser side. The main difference in verification latency lies in the certificate state checking. For PISTIS, we measure the latency from the time that a client sends a state verification transaction to the time that a response transaction is parsed, and the certificate's state is confirmed. For the OCSP and CRL, we use OpenSSL to send OCSP and CRL requests and record the corresponding latency.

We conduct the measurement ten times on PISTIS, OCSP, and CRL, respectively. Results are summarized in Fig. 4-7. We use the circle dots to represent the delay of PISTIS, the vertical lines to represent the delay of OCSP, and the triangle dots to represent the delay of CRL. As depicted in Fig. 4-7, the verification latency of PISTIS is the lowest, which fluctuates around 150 ms. The delay of OCSP ranks secondly with 400 ms certificate verification service. The CRL has the longest delay, nearly 700 ms, which results from the biggest packet they transmit during verification. In addition, we also measure the packet size during each operation and illustrate them in Fig. 4-8. The CRL needs to transmit around 15 KB data for certificate verification, while OCSP needs to transmit 2 KB. By contrast, PISTIS only needs to transmit a transaction for verification, whose size is around 0.1 KB.

### 4.5.2 TEE Evaluation

We evaluate the performance of PISTIS TEE on a server with i9-9900k CPU and 32GB memory. Our experiment results prove that PISTIS can easily meet the peak throughput of Ethereum network and can be deployed on current commodity servers.

#### Throughput

We first evaluate the throughput of a TEE node. We aim to explore to what extent can a TEE node processes transactions. The experiment results are illustrated in Fig. 4-9. We can see that a server with at most 30 enclave instances can handle up to 58 transactions per second. We also notice that when the number of enclave instances is less than 16, the throughput increases linearly.

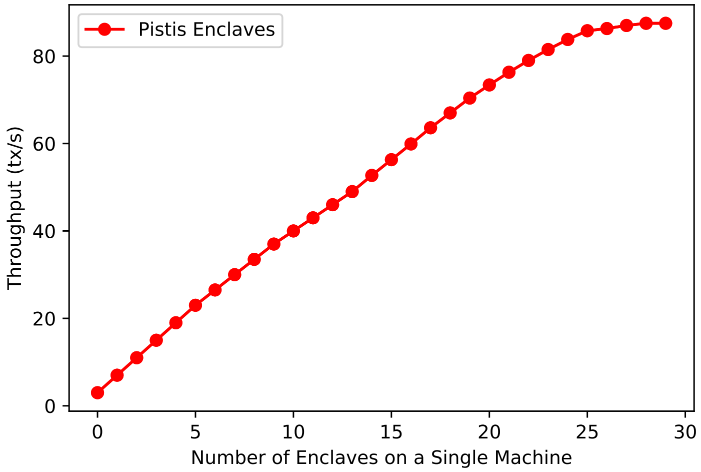


Figure 4-9: The throughput of TEE node on a single machine with different numbers of enclave instances.

### Response Time

We define the response time of a TEE node as the interval between the time that a Pistis Contract/Domain Owner sends a request to the enclave and the time that a Pistis Contract/Domain Owner receives a response from the TEE node. The experiment results are presented in the Table 4.2. As we can see, the interaction between a enclave and Pistis contracts are fast, which is because the TEE node is equipped with a blockchain endpoint. In this case, the interaction between the Pistis contract and the enclave omit the network latency. By contrast, the DNS query operation has the longest response time, which is because the communication latency between TEE nodes and domains.

## 4.6 Discussion

In this section, we discuss how to deploy, maintain, and upgrade Pistis contracts and TEE nodes. We also discuss the limitations of the underlying infrastructure of

Table 4.2: The response time of TEE node operations.

| Operation | $t_{mean}$ (ms) | $t_{max}$ (ms) | $t_{min}$ (ms) |
|---|---|---|---|
| Protocol Trigger | 1.1 | 4.21 | 0.36 |
| DNS Query | 1.06 | 4.17 | 0.28 |
| DNS Response | 86.4 | 280.2 | 46.7 |
| Protocol Response | 1.13 | 4.25 | 0.41 |
| Total | 89.69 | 292.83 | 47.75 |

Pistis.

### 4.6.1 Deployment, Maintenance, and Upgrade

As Pistis adopts a new way to issue and revoke certificates, its deployment, maintenance, and upgrade methods are different from traditional CAs. Pistis contracts are written by its developers (e.g., a traditional CA company or open-source community that wants to issue certificates using Pistis). These developers can deploy Pistis contracts onto a chosen blockchain platform and operate TEE nodes to establish the communication channels between Pistis and domains. Developers also monitor the state transition of Pistis and check if there are any problems. If they want to add new functions, they can use a new contract to upgrade the Pistis service. The older contracts will be destructed by calling the `selfdestruct` function and the new contract address will be advertised to target domain owners. We want to emphasize that these upgrades will not increase the possibility that Pistis is manipulated by its developers. Since Pistis is transparent, any deviation from its design principle will cause people not to trust it, which conflicts with the interests of developers.

Once developers deploy the Pistis contract on the blockchain, domain owners can interact with it to request a certificate. After getting a Pistis-issued certificate, a domain owner can set up its HTTPS-protected service. Clients can verify whether a certificate is valid by querying Pistis blockchain. Specifically, a client needs to inject the verification script into the website to interact with the certificate verification contract $\mathcal{C}_{\text{VC}}$. Once a certificate passes verification, the client can establish a HTTPS connection with the Pistis-protected website.

In this process, Pistis will not be affected by whether people trust its developers. If developers leave a back door in the contract program, domain owners can discover it and reject Pistis-issued certificates. Even developers are compromised during Pistis's execution, the hybrid architecture of blockchain and TEE prevents their malicious behaviors as long as the Pistis contract has no loopholes.

### 4.6.2 Infrastructure Limitations

Pistis is constructed based on smart contract and TEE, which are both emerging technologies and have some limitations.

**Blockchain Limitations.** Blockchain has demonstrated to be vulnerable in some aspects. For example, the mining policy is not secure since attackers can withhold a newly-mined block and broadcast it until another miner finds a new block in the same height, which makes the computation power devoted by most honest miners invalid. In addition, bribery attack [45] demonstrates the possibility that attackers can manipulate the inclusion of transactions.

Blockchain platforms, especially those support smart contract, can only provide limited-throughput transaction processing service. In this case, Pistis can only issue certificates at a limited rate. In our future work, we aim to extend Pistis to some performant blockchain systems such as Hyperledger, Libra, or some experimental sharding systems.

**Smart Contract Limitations.** The smart contract also has some vulnerabilities and some even lead to serious consequences such as the DAO attack. Since the authorization of certificates constructs the foundation of secure web connections, we should alleviate the exposed vulnerabilities and protect Pistis from the other potential vulnerabilities.

**TEE Limitations.** The Intel SGX was proposed to provide trusted computation by isolating some memory parts and encapsulating programs in enclaves securely. However, some researchers have successfully launched side-channel attacks against the Intel SGX platforms [113]. Moreover, enclaves can offload some computation overhead from the Pistis contract, which might lead to new attack vectors such as

SgxPectre [24]. In this case, mitigation should be located in the Pistis framework. We also consider the case that some TEE hosts are malicious. They can delay or drop correct DNS responses but cannot forge a valid proof of ownership validation. In this case, malicious hosts can only cause a small DoS attack. As we have assumed that at least one TEE is working, such attacks cannot affect the overall operation of Pistis.

**MitM Attack.** We consider the possibility that MitM attacker may affect the system. We discuss attacks launched by two kinds of attackers: passive attackers that controls a large ISP, and active attackers that attempts to attract traffic from other networks. We run simulations with different number of TEE nodes. If the attacker is in the victim domain's network, it can hijack requests from the domain owner and spoof responses. Most domains may have multiple nameservers and these nameservers are usually placed in different networks. This is following the best practice to avoid a single point of failure for domains. Furthermore, as nameservers of the same domain are hosted in different networks, an attacker can hardly hijack or spoof all responses. We also quantify the ability of an on-path attacker to intercept majority of DNS requests sent to a TEE node from the victim domain. The simulation evaluates all possible scenarios for an on-path attacker to cover almost all routes between the victim domain and the TEE node. Results show that it is impossible for a MitM attacker to acquire challenge value during the validation process.

## 4.7 Related Work

In this section, we discuss the related work on protecting PKI, including traditional countermeasures against unauthorized certificates, and blockchain-based PKI/CA.

### 4.7.1 Traditional Countermeasures

Traditional countermeasures that aim to address the unauthorized certificate problem can be categorized into two types: client-side and server-side.

On the client-side, proposals such as HPKP [39] and Trust Assertions for Certificate Keys (TACK) [74] aim to establish a solid connection between the public key and the domain name. These solutions, however, require a domain to inform clients which keys are valid so that clients can distinguish valid certificates from unauthorized ones. Researchers also proposed community of trust [85], which acts as the basis of certificate verification. Syta et al. proposed CoSi [103], which employs a witness cosigning protocol to ensure that every statement is verified and publicly logged by a diverse group.

On the server-side, most countermeasures aim to establish log servers, which allows domain owners to record operations on their certificates. This also provides public accountability to clients. Sovereign Keys (SK) [37], Certificate Transparency (CT) [64], AKI [58], ARPKI [11], DTKI [119], and PoliCert [105] fall in this category. Researchers also focused on the notification of certificate revocation to prevent attackers from further involvement [72].

### 4.7.2 Blockchain-based PKI/CA

There are some proposals aiming to address the unauthorized certificate problem utilizing blockchain technology.

Fromknecht et al. proposed CertCoin [44], whose core idea is letting the public ledger as a "bulletin boards" for domains and their associated public keys. Mustafa proposed SCPKI [2], which is an alternative PKI system based on a decentralized and transparent design using the web-of-trust model and smart contracts. Catena [107] leverages Bitcoin as the log server and generates a transaction chain to prevent CAs from issuing contradicting certificates. By contrast, PISTIS not only preserves transactions to trace certificate state transitions, but also maintains a MPT to store latest states of certificates. IKP [76] aims to mitigate unauthorized certificates by incentivizing the CA, domain owner, and clients to report unauthorized certificates. IKP is designed to be compatible with current PKI so that certificate issuance and revocation are both conducted by traditional CAs. Cecoin [92] employs a Bitcoin-like blockchain to provide irreversible unforgeability and public verifiability in the CAs. Specifically, certificates are treated as currency and are circulated on the blockchain.

BlockPKI [36] uses blockchain as the log server to make CA operations publicly visible and accountable.

Yakubov et al. introduced smart contracts to establish a blockchain-based PKI [117]. Each smart contract acts as a CA that takes charge of issuing and revoking certificates. CertChain [28] aims to enhance the security of PKI by recording certificate operations on the blockchain. However, it does not provide a feasible domain validation function, which leaves a door of unauthorized certificates. PBCert [118] explores the way to enable privacy-preserving in querying the latest states of certificates, which protects users from eavesdropping. CertLedger [63] utilizes the blockchain to implement certificate transparency and provides an efficient certificate verification method. SmartCert [104] generates smart contracts for certificates to automate the certificate validation.

### 4.7.3 Comparison

Differences between Pistis and state-of-the-art are summarized in Table 4.3. The comparison is conducted in six dimensions: certificate authority's construction and type, certificate issuance, domain ownership validation, certificate validity verification, certificate revocation request and checking, and security analysis of the proposed system. The related work is sorted chronologically in the table, while we separate Pistis, traditional solutions and blockchain-based solutions with dual horizontal lines.

For traditional solutions, all work relies on external trusted CA to issue certificates. As to the request initiation side, all systems can initiate registration request from both domain and CA side. We have emphasized that requests initiated from CA side enable attackers to compromise a CA and maliciously issue unauthorized certificates. By contrast, Pistis uses smart contracts to build CAs, which are essentially source code that cannot initiate a certificate issuance request locally, namely from CA side. For ownership validation, most work are left empty because they rely on external trusted CA to conduct ownership validation except DV++, which proposes a multiple vantage points domain validation mechanism. Allowing CA-side certificate issuance means that malicious or compromised CAs may issue certificates without domain permission, i.e. unauthorized certificates. PISTIS ensures that malicious or compromised CAs cannot issue certificates by combining this certificate issuance limitation. A certificate request can only be initiated from domain side and the domain must pass the ownership validation. In addition, all systems except DV++ proposed the corresponding certificate validity validation mechanism, that is, to detect whether a certificate is valid, whether it has been revoked or beyond its valid period. As to certificate revocation, DV++ and CoSi do not provide mechanism to revoke certificates and only PISTIS and PoliCert provide revocation checking mechanism. For security analysis, either formal or informal, most work provides security analysis. In general, traditional approaches were based on a trusted CA, with enhanced protection to avoid corruption caused by unauthorized certificates. However, as the CA might still be compromised to issue rogue certificates, traditional methods cannot prevent the issuance of unauthorized certificates as PISTIS.

For blockchain-based solutions, we found that most of them rely on external trusted CAs to issue certificates. This reflects their log server based ideas, which are similar to traditional log server based solution. In these blockchain-based log server solutions, blockchain is used as a data carrier similar to the traditional log server to chain the history of certificate operations. By contrast, SCPKI relies on smart contracts to issue certificates. SCPKI does not issue full certificates, but binding relationships between public keys and identities. In addition, SCPKI cannot conduct domain ownership validation on its applicants, which indicates that SCPKI is incomplete. CertCoin and Cecoin are based on Namecoin and therefore use blockchain as their certificate authority. Furthermore, solutions that rely on external trusted CAs still offload certificate issuance and ownership validation to them. Only BlockPKI emphasizes that it carries out ownership validation through ACME method. In CertCoin and Cecoin, as the registration of certificates is based on the validity of relevant domains, it is only necessary for a domain owner to broadcast a certificate registration transaction based on its previous domain registration transaction. At this step, ownership validation of domains is simply a matter of looking up the transaction history on blockchain. For certificate validity validation methods, all listed solutions are equipped with a proper one. As to certificate revocation, BlockPKI issues short-lived certificates so that omits it while Cecoin does not provide revocation mechanism. In addition, only CertChain and PBCert provide revocation checking mechanism. For the security analysis, only CertLedger provides a formal analysis. The rest either only provide an informal analysis or no analysis at all. Generally speaking, the idea behind most blockchain-based work is similar to that of traditional work, namely log server mode. They simply use the blockchain as a substitute of the log server to record operations of certificates.

In summary, Pistis focuses on fundamental issues of certificate issuance, i.e., mandatory domain ownership validation for applicants and disallowing CA-side initiated requests. By contrast, most existing work still focus on monitoring issued certificates and providing accountability in the form of log server, which cannot prevent issuing unauthorized certificates and even further MitM attacks.

## 4.8 Conclusion

In this chapter, we propose Pistis to address the problem that traditional CAs are vulnerable to be compromised to issue unauthorized certificates. Pistis leverages TEE and smart contract to ensure that only the domain owner can request a certificate for its domain, which guarantees all issued certificates are authorized. Previous work either only detects unauthorized certificate or decreases its possibility. Pistis not only ensures its issued certificates are authorized, but also provides a new trust paradigm. Users can trust a CA based on its execution logic rather than its identity, and Pistis-issued certificates are trusted as they are recorded on blockchain. To the best of our knowledge, this is the first attempt to prevent issuing unauthorized certificates. Our security analysis and experiment results prove the security and feasibility of Pistis.

Table 4.3: Comparison between Pistis and related work.

| Name | Certificate Authority | | Issuance | | Ownership[2] | | Validity | | Revocation | | Security |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Construction | Type | Domain | CA[1] | Validation | Method | Validation | Method | Request | Checking | Analysis |
| Pistis | Smart Contract + TEE | Decentralized | ✓ | N/A | ✓ | TEE | ✓ | MPTP[3] | ✓ | web3.js | Formal |
| AKI [58] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | ILS Proof[4] | ✓ | N/A | Informal |
| ARPKI [11] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MTP[3] | ✓ | N/A | Formal |
| PoliCert [105] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | Log Proof | ✓ | Log Proof | Informal |
| DTKI [119] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | Master Key | ✓ | N/A | Formal |
| DV++ [19] | External Trusted CA | Centralized | ✓ | ✓ | ✓ | Multi-Path | N/A | N/A | N/A | N/A | N/A |
| CoSi [103] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MTP | N/A | N/A | N/A |
| CertCoin [44] | On-Chain Transaction | Decentralized | ✓ | N/A | ✓ | Passive | ✓ | Passive | ✓ | N/A | N/A |
| SCPKI [2] | Smart Contract | Decentralized | N/A | N/A | N/A | N/A | ✓ | MTP | ✓ | N/A | N/A |
| IKP [76] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MTP | ✓ | N/A | N/A |
| Cecoin [92] | On-Chain Transaction | Decentralized | ✓ | N/A | ✓ | Passive | ✓ | Passive | N/A | N/A | Informal |
| Certchain [28] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MTP | ✓ | Bloom Filter | Informal |
| PBCert [118] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MTP | ✓ | OCSP | Informal |
| BlockPKI [36] | External Trusted CA | Centralized | ✓ | ✓ | ✓ | ACME | ✓ | Multi-Sig | N/A | N/A | Informal |
| CertLedger [63] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MPTP | ✓ | N/A | Formal |
| SmartCert [104] | External Trusted CA | Centralized | ✓ | ✓ | - | - | ✓ | MTP | ✓ | N/A | Informal |

[1] This column indicates that a malicious or compromised CA can issue certificates for any domains without their permission.

[2] For systems that rely external trusted CA to issue certificates, we use "-" to indicate not covering ownership validation.

[3] MPTP is short for Merkle Patricia Tree Proof while MTP is short for Merkle Tree Proof.

[4] ILS is short for Integrity Log Server, which is proposed in AKI [58].

# Chapter 5

# SolSaviour: A Defending Framework for Protecting Defective Deployed Smart Contracts and DeFi

## 5.1 Introduction

As the global market size of smart contract grows year by year, the security of smart contracts has raised huge concerns. Due to the fact that the logic of a smart contract cannot be modified once deployed and vulnerabilities cannot be fixed by means of updates like traditional applications, a great deal of work has been focused on fully validating smart contracts before they are deployed. However, there is no good way to protect smart contracts from vulnerabilities once they have been deployed.

Vulnerabilities in smart contracts as well as derived attacks have led to significant losses in recent years due to the lack of relevant methods to protect smart contracts. One of the most notorious incidents was the DAO hack [21], in which attackers exploited the reentrancy vulnerability in the DAO contract to withdraw ethers wantonly. During the attack, honest contract users can do nothing but to withdraw ethers to secure accounts as fast as possible. The DAO hack caused the loss of approximately 3.6 million ethers. The Ethereum community eventually decided to reduce the impact of the DAO hack via a hard fork. Another infamous incident was the Parity Multisig Wallet, which was hacked twice. In the first time, attackers exploited the vulnerability of `delegatecall` in the fallback function to change the contract ownership and stole 153,037 ethers [93]. In the second time, attackers managed to destroy the Parity Wallet library contract. 587 related contracts were blocked and 513,774.16 ethers were locked [1]. Furthermore, with the increased popularity of decentralized applications (DAPP) and decentralized finance (DeFi) applications (e.g., UniSwap and flash loan), new types of attacks are appearing. For example, a DeFi application Fei has experienced malicious trading attacks caused by price manipulation [78].

Motivated by these attacks, the community started to conduct research on detecting vulnerabilities in smart contracts before deployment. Many software testing techniques are utilized such as symbolic execution [73], formal verification [14], static analysis [20, 40], dynamic analysis [110], and fuzzing testing [83]. However, these detection methods still have certain limitations. They cover limited types of vulnerabilities, have restricted detection efficiency (i.e., sensitive to some vulnerabilities, but insensitive to others), and suffer from possible false negative cases. There are also some work on repairing smart contracts such as EVMPatch [95], SCRepair [120] and sGUARD [41]. We point out that for high-net-worth smart contracts, pre-deployment detection methods are not fully effective, and the possibility of post-deployment vulnerabilities still exists. Though detecting and fixing contract bugs has been extensively studied, how to repair and recover defective deployed contracts remains an unsolved problem. However, for current smart contracts, pre-deployment detection approaches do not address the problem that a vulnerability is found in a smart contract after deployment. This is partly due to limitations in the detection tool itself, and partly due to the possibility of vulnerabilities that are not covered by the detection tool, or even currently unknown vulnerabilities. Therefore, how to protect deployed smart contracts remains a problem that requires urgent research.

The community has come up with a number of ways to safeguard deployed smart contracts, such as the proxy pattern. A smart contract deployed in proxy pattern is divided into a data contract and a logical contract. The data contract is used to store corresponding state variables, while the logical contract is used to store the execution logic of the contract. The data contract can call the logical contract via the `delegatecall` opcode. The `delegatecall` executes the code of the logical contract in the context of the data contract. This allows the user to simply redeploy a new patched logic contract to replace the defective contract once a bug is found in a logic contract. The proxy pattern also has the drawback of fixing vulnerabilities and not protecting the assets in the contract. And assets are to some extent the most important thing to protect.

Our core idea is to replace a defective smart contract with a patched contract, and maintain the consistency of contracts' states at the same time. Specifically, the state variables and contract stake distribution remain same in the newly deployed smart contract. This inspires a decentralized control on a smart contract to multiple parties through a secure and principled combination of blockchain and trusted hardware. We first propose the voteDestruct mechanism to enable the decentralized control of a smart contract. Contract participants (i.e., stakeholders) can vote on the future of the contract. They can lock it, destroy it, or even unlock the locked contract and continue the execution. The weight of their votes depends on the number of ethers (i.e., stake) they have deposited. The more stake a stakeholder controls the weightier its vote. We then establish a TEE cluster to take charge of temporary asset escrow after destroying a defective smart contract and stake migration. Once a patched smart contract is provided, the TEE cluster deploys it and conducts the state migration to transfer all internal assets to the newly-deployed contract. Assuming the integrity of the blockchain, users do not need to trust the validity, persistence, confidentiality, or correctness of smart contract creators, miners, or TEE nodes. SolSaviour thus can provide self-sustaining service even when some miners, contract creators, contract participants, or TEE nodes are unavailable.

The main contributions of this work are summarized as follows:

- We propose the voteDestruct mechanism to allow the decentralized control over a deployed smart contract.

- We build a TEE cluster based on Intel SGX to take charge of asset escrow and contract state migration. The TEE cluster can authenticate the identity of caller, preserve trusted execution of contract invocation, patching, and deployment. We also provide complete APIs for clients to invoke the TEE cluster. We provide identity authentication and patch tester in SolSaviour, which address the problem in [68].

- We give a thorough security analysis of SolSaviour from three perspectives: balance security, correctness, and fairness.

- We collect smart contracts and DeFi protocols that were attacked in the past and use them to evaluate the effectiveness and performance of SolSaviour. Experiment results show that SolSaviour can effectively mitigate the loss caused by smart contract vulnerabilities with little overhead.

The remainder of this chapter is organized as follows. In Section 5.2, we present the overview, workflow, and building blocks of SolSaviour. The detailed implementation is presented in Section 5.3. We thoroughly analyze the security of SolSaviour in Section 5.4. We discuss potential improvement of SolSaviour in 5.5. The effectiveness and performance of SolSaviour are evaluated in Section 5.6. We discuss the related work in Section 5.7 and conclude our work in Section 5.8.

## 5.2 SolSaviour

The architecture of SolSaviour is depicted in Fig. 5-1. SolSaviour consists of two core components: voteDestruct and TEE cluster. The voteDestruct mechanism is embedded in smart contracts before deployment. It allows smart contracts to be destroyed in a voting manner. TEE cluster allows contract stakeholders to replace the defective contract with a patched contract via invoking the voteDestruct mechanism. Specifically, TEE cluster can deploy a patched contract onto the blockchain and migrate all assets as well as the stake distribution to it. Since then, stakeholders can continue to execute the contract without the vulnerability. In this case, even a bug is

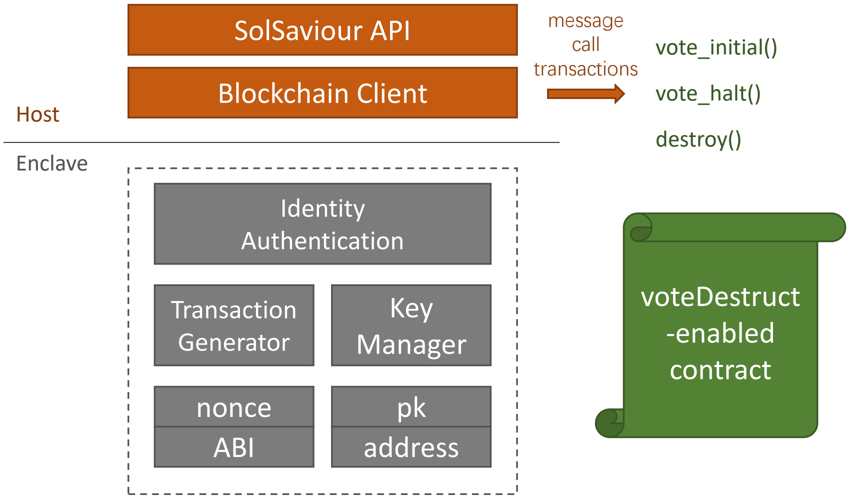


Figure 5-1: The architecture of SolSaviour. Contract stakeholders can invoke the TEE cluster via SolSaviour APIs to generate message call transactions to instruct the voteDestruct-enabled contract.

exposed during the contract execution, they can fix it without worrying about asset loss.

### 5.2.1 Workflow

First of all, stakeholders should prepare a TEE cluster for protecting smart contracts. Contract stakeholders collect a cluster of SGX-capable computers and launch enclaves into them. The architecture of a TEE node is depicted in Fig. 5-1. Then, following the bootstrapping process, these TEE nodes establish a TEE cluster. After constructing the TEE cluster, users can invoke it to deploy a voteDestruct-enabled smart contract.

We emphasize here that the initial deployment of a contract should call the TEE cluster. Otherwise, the deployed smart contract, even with voteDestruct mechanism, cannot be protected by the SolSaviour. Once the contract passes the initial deployment phase, it can work under the protection of SolSaviour.

During the contract execution (i.e., blockchain growth), an unknown bug may be disclosed. Then, contract stakeholders can check whether this exposed bug is a false positive. If not, stakeholders can invoke APIs provided by SolSaviour to save the contract. The detailed workflow is summarized below as shown in Fig. 5-2:

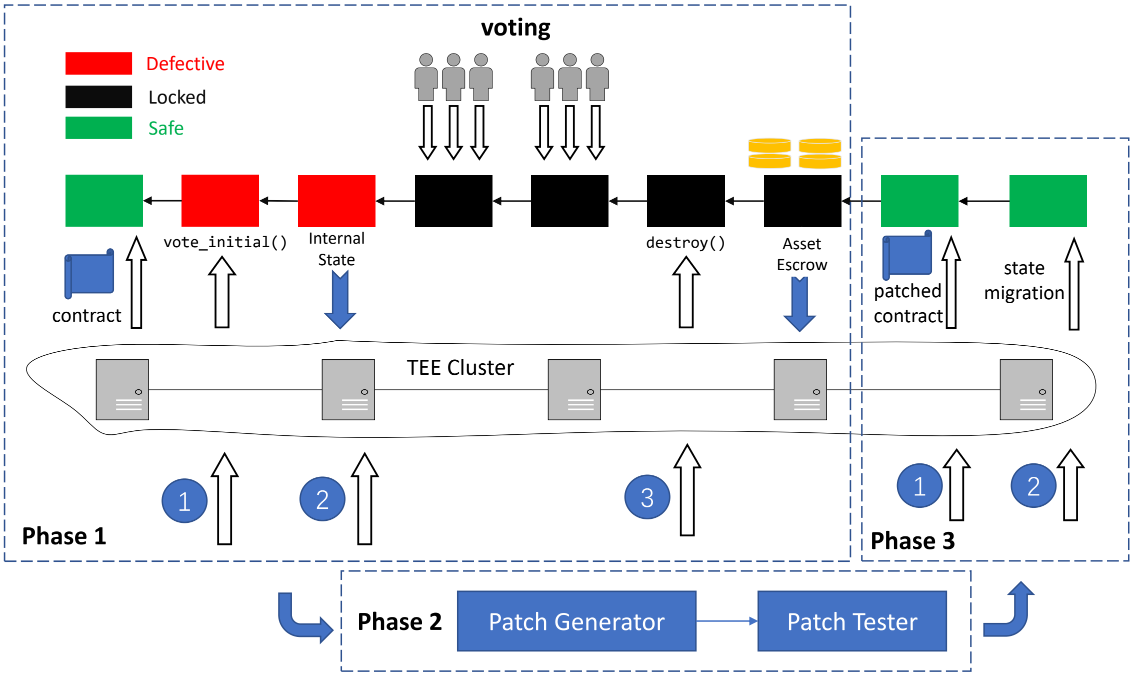


Figure 5-2: The workflow of SolSaviour framework.

**Phase 1: Destroying the Defective Contract.**

① In the destroying phase, stake holders first lock the defective smart contract when identifying a bug. They can invoke the TEE cluster to lock the defective smart contracts, which prevent deployed smart contracts from further attacks.

② Then, stakeholders invoke the TEE cluster to extract the internal state of the defective smart contract. In this step, the TEE cluster extracts the internal state of the defective contract for future state migration. Specifically, it stores the values of state variables and the stake distribution at the time that the contract is locked.

③ Thirdly, stakeholders who have assets inside a smart contract can decide whether to destroy the defective smart contract via a cumulative voting way, where people's stake means how much they have in the contract. They simply invoke the `vote()` function provided by the voteDestruct mechanism. After completing voting, stakeholders can invoke the TEE cluster to destroy the defective contract.

**Phase 2: Preparing a Patched Contract Offline.**

In this phase, stakeholders generate and test patch for the defective contract discovered in Phase 1. Stakeholders can prepare a patch for the located vulnerability and integrate it with the original contract. After that, stakeholders employ patch tester to validate whether the vulnerability is resolved and whether the functionalities of patched contract remain same. Then, contract stakeholders can upload a patched contract into the TEE cluster.

**Phase 3: Redeploying the Patched Contract.**

① In this phase, stakeholders first invoke the TEE cluster to deploy the patched contract prepared in Phase 2. The TEE cluster generates a contract creation transaction for deploying the patched contract and records its address.

② Then, stakeholders migrate the previously-extracted state as well as assets into the patched contract. After destroying the defective smart contract, all inside assets are temporarily held by the TEE cluster. The temporarily-held assets are transferred into the deployed patched contract according to the stake distribution and previously-extracted values are written into the state variables of the newly-deployed patched contract.

### 5.2.2 Building Blocks

#### Exception Detector

We propose a policy-based contract exception detector. In this we design two exception policies first. One is to broadcast a warning message when a contract withdrawal exceeds a certain percentage of the contract's total assets, and the other is to warn when a contract withdrawal is made to an address that does not belong to a contract stakeholder. This is achieved by posting an event message on the blockchain. Considering that an attacker may have a large number of accounts which can circumvent the above two exception detection rules through a large number of small withdrawal transactions, we further propose that a warning event message is sent when a certain percentage of assets are lost from an account within a specified period of time.

The design of the exception detection is based on the smart contract level and is implemented through Solidity. The warning messages exist on the blockchain network and are not actively communicated to the contract stakeholders, who therefore need to

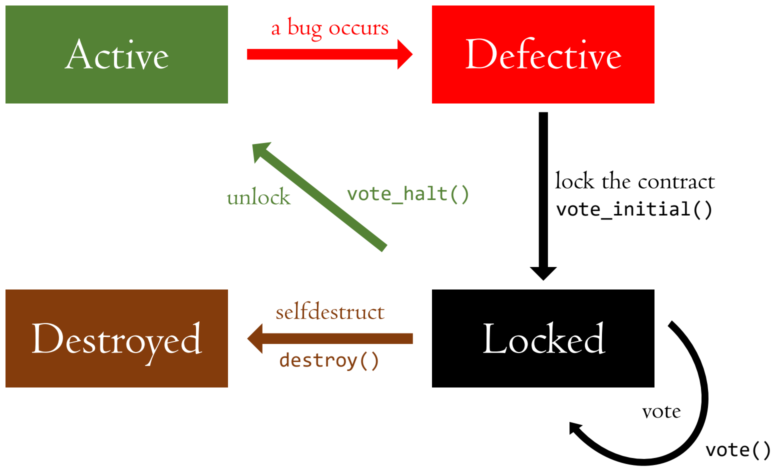


Figure 5-3: The state transition diagram of a voteDestruct-enabled smart contract.

implement an active crawler outside the contract that automatically and continuously crawls for event information to monitor the contract. This way, when an exception occurs in a monitored contract, stakeholders can be aware of it and take further action as soon as possible.

### voteDestruct Mechanism

Currently, the typical method of destroying a contract is to have a privileged contract destruction function that only privileged accounts are eligible to invoke. This approach is not applicable to multi-user smart contract scenarios. For example, the assets stored in a DeFi contract may belong to different users. Furthermore, as an autonomous community, stakeholders of a smart contract can reach a consensus on whether to destroy the smart contract among themselves. Therefore, we propose the voteDestruct mechanism, which enables decentralized control of smart contracts.

In voteDestruct-enabled contracts, stakeholders can vote on whether to destroy the smart contract and withdraw all internal funds. The voteDestruct mechanism is based on the contract stake distribution, which is recorded by state variables. The more stake a stakeholder controls, the greater its vote weight. After completing the vote, stakeholders can destroy the defective contract if the support rate exceeds a predefined threshold. Specifically, The voteDestruct is processed in three steps.

The contract first forms the stake distribution during its execution. Then, once a vulnerability is exposed, stakeholders can invoke the contract via TEE cluster to vote whether to destroy the contract. After completing the voting, the contract stakeholders can invoke the TEE cluster to destroy the defective contract.

**Stake Distribution:** Each depositing transaction will be recorded. Specifically, the address of depositor and the amount of deposited assets will be recorded.

**Voting:** During the voting process, the smart contract is locked so that no external users can deposit or withdraw assets. This ensures that the stake distribution remains constant throughout the voting process. Once the vote is completed, the contract stores the percentage of stake on supporting and opposing respectively via state variables and in turn determines whether stakeholders are allowed to destroy the contract.

**Destroying:** The contract can only be destroyed if the cumulative number of votes in favour of destruction exceeds the threshold. Contract stakeholders can instruct the TEE cluster to invoke the destroy function once the voting process completes.

**State Transition:** The state transition of a voteDestruct-enabled smart contract is depicted in Fig. 5-3. A smart contract is initially in an active state. Then, an unknown bug is discovered and the contract enters a defective state. Stakeholders can now lock the defective contract with the `vote_initial()` function. For false positive cases, they could unlock the contract via `vote_halt()` function. For true bugs, stakeholders could vote on whether to destroy the defective contract. When the support rate exceeds the threshold, the user can call the `destroy()` function to destroy it.

### TEE Cluster

In the TEE architecture, the security container ensures that the party maintaining the SGX will follow the enclave's implementation of the execution procedure. attested execution can help untrusting parties establish trust at a lower cost. The limitation of enclave memory is a natural barrier that affects the scalability of solutions using

SGX. In particular, it will result in a performance load that oversubscribes the secure memory. To this end, multiple SGX nodes maintained by the consistent protocol can be employed to improve the solutions. Availability failures are another factor that could sway SGX-based schemes. These failures can result in the SGX-base nodes becoming completely or partially disabled. To tolerate such failures, the scheme may need to store the persistent state information in the trusted role, such as the blockchain or a committee composed of SGX-based nodes.

To this end, the TEE cluster is established to mitigate the availability of single TEE node. We present the working logic of TEE cluster in Phase 1 and 3 in Algorithm 3 and Algorithm 4, respectively. Each enclave is denoted as $\sigma_i$. We use $c$ to represent a potentially defective smart contract and $C_p$ to denote a patched contract. To distinguish the difference between intra-TEE cluster communication and TEE cluster-blockchain communication, we use “broadcast” to indicate broadcasting messages inside TEE cluster and “upload” to represent broadcasting transactions onto the blockchain.

**Key Generation:** Let $\mathbb{G}$ be an Elliptic curve group of order $q$ with generator (base point) $\mathrm{G}$. Each TEE node $P_i$ has the information of $(\mathbb{G}, \mathrm{G}, q)$. A TEE node $P_i$ first chooses a random $x_i$ from $\mathbb{Z}_q^*$ and computing $Q_i = x_i \cdot G$. Each TEE node stores $(\mathbb{G}, \mathrm{G}, q, x)$. Then, each TEE node $P_i$ broadcasts its generated $x_i$ to $P_j$ for every $j \in [n] \backslash \{j\}$. After receiving $x_j$ ($j \in [n] \backslash \{j\}$) from other parties, $P_i$ locally computes $x = \sum_{l=1}^{n} x_l$ and $Q = x \cdot G$. Each party $P_i$ locally stores $Q$ as the ECDSA public key.

**Signing:** A TEE node $P_i$ locally generates $k_i$ and $\rho_i$ randomly. Then, $P_i$ broadcasts $k_i$ and $\rho_i$ to $P_j$ for every $j \in [n] \backslash \{j\}$. After receiving $k_j$ and $\rho_j$ ($j \in [n] \backslash \{j\}$). Each party can locally compute $k = \sum_{l=1}^{n} k_l$ and $\rho = \sum_{l=1}^{n} \rho_l$. Then, TEE node $P_i$ computes $\tau = k \cdot \rho$ and $R = k \cdot G$. The TEE node $P_i$ can now compute $R = (r_x, r_y)$ and $r = r_x \mod q$. For a raw transaction $m$, node $P_i$ generates $\beta = \rho \cdot (m + x \cdot r) \mod q$. Then, $P_i$ computes $s' = \tau^{-1} \cdot \beta \mod q$ and $s = min(s, q - s)$. Finally, TEE node $P_i$ outputs $(r, s)$ as the ECDSA signature of the transaction.

**Identity Authentication:** After bootstrapping, we first discuss identity authentication, which happens before each invocation from stakeholders. Identity authentication allows only eligible stakeholders invoke the TEE cluster. The TEE cluster first checks whether the invoking stakeholder is valid. Therefore, when a stakeholder invokes the TEE cluster, SolSaviour needs to verify that the identity of the stakeholder who initiated the invocation is reliable to eliminate the possibility that attackers can invoke the TEE cluster to conduct dangerous operations. TEE Cluster uses membership proof for identity authentication. It first extracts the address of the stakeholder that initiated the call, which in turn polls the membership proof integrated with the voteDestruct mechanism. If the membership proof is true, the TEE cluster determines that the invoking stakeholder is valid and proceeds with the corresponding operations, while if it is false, all operations are simply interrupted.

**Algorithm 3:** The working logic of TEE cluster in Phase 1.

```
procedure BOOTSTRAPPING($\sigma_i, i \in [0, n-1]$)
    load $\sigma_i$
    broadcast $\{$sgx_quote$_i$, $K_i\}$
    verify sgx_quote$_j$
    generate $\{K$,addr$\}$
end procedure
procedure IDENTITY AUTHENTICATION
    on receiving a message call tx
    require(st.map(msg.sender()).st_amount)
    revert()
end procedure
procedure LOCKING DEFECTIVE CONTRACT
    goto line 8
    on receiving address *addr*
    tx.payload(addr, vote_initial)
    Sign$_K$(tx)
    upload tx
    broadcast(++nonce)
end procedure
```

**Destroying:** TEE cluster allows stakeholders to lock the defective contracts. Once a bug is discovered, the stakeholder can call the TEE cluster to lock the contract. The TEE cluster generates a message call transaction to invoke the `vote_initial()` function and signs it using the key generated in the bootstrap step. Afterwards, the signed transaction is uploaded to the blockchain for processing and the incremental nonce is broadcast inside the TEE cluster.

Before destroying the defective smart contract, the TEE cluster extract the internal states from defective contract. Internal states including values of state variables and the stake distribution of stakeholders at the time of locking. As stake distribution is stored as state variables in the voteDestruct mechanism, the TEE cluster obtains internal states via `getter` functions and save them locally.

After completing the voting process in the blockchain level, contract stakeholders can instruct the TEE cluster to invoke the `destroy()` function. The TEE cluster generates a signed message call transaction destined to the address of the defective smart contract onto the blockchain. TEE cluster also broadcasted an incremental nonce. Provided the amount of stake in favour of destruction exceeds a specified threshold, the contract is allowed to be destroyed.

**Redeploying:** Contract stakeholders first generate the patch offline. Then, they could invoke the TEE cluster to redeploy a patched smart contract. After receiving a patched contract from stakeholders, the TEE cluster generates a contract creation transaction with compiled contract as payload. Then, the TEE cluster upload the signed transaction onto the blockchain and broadcasted an incremental nonce.

For the previously-extracted internal states, stakeholders can call the TEE cluster to migrate them to the patched contract. TEE cluster generates signed message call transactions to invoke the patched contract for stake distribution as well as assets. In this process, the TEE cluster ensures that the states are consistent. During the state migration, the TEE cluster also guarantees the atomicity of execution, i.e. any intermediate state resulting from the call, and the need to provide an effective rollback mechanism in the event of a failed call operation, allowing the system to revert to the state before the call, eliminating the impact of intermediate state resulting from the call.

### 5.2.3 SolSaviour APIs

We summarize the APIs of SolSaviour in Table. 5.1. Contract stakeholders can invoke the functionalities provided by SolSaviour via these APIs.

**Algorithm 4:** The working logic of TEE cluster in Phase 3.

```
procedure EXTRACTING INTERNAL STATE
    goto line 8
    verify_tx(tx,$pub_1$,$pub_2$)
    d ← create(tx)
    mapping stake_dist(addr→stake_amount)
    stake_dist ← addr.st_map_getter()
end procedure
procedure DESTROYING DEFECTIVE CONTRACT
    goto line 8
    on receiving address $addr$
    tx.payload(addr, destroy)
    $\text{Sign}_K$(tx)
    upload tx
    broadcast(++nonce)
end procedure
procedure REDEPLOYING PATCHED CONTRACT
    goto line 8
    on receiving $C_p$
    tx.payload($C_p$)
    $\text{Sign}_K$(tx)
    upload tx
    broadcast(++nonce)
end procedure
procedure MIGRATING TO PATCHED CONTRACT
    goto line 8
    tx.payload(addr_$C_p$, stake_dist), go to line 15
    values[]←stake_dist[]
    balance[c_$id_n$]←values[c_$id_n$]
end procedure
```

SolSaviour APIs are categorized into 3 types: bootstrap, deployment, and message call. In the bootstrap type, stakeholders can invoke the `new_address` function to generate a new key pair for the agreed account address utilized in the TEE cluster. In the deployment type, there are two functions: `new_contract` and `patched_contract`. The former one is used to deploy a compiled new smart contract, and the later one is used to deploy a bytecode-version smart contract. Both functions return the id of the contract generation transaction that utilized to deploy the contract. stakeholders can invoke the first function to deploy a new smart contract and the second function to

Table 5.1: SolSaviour APIs.

| SolSaviour APIs | Inputs | Outputs | Description |
|---|---|---|---|
| ***Bootstrapping*** | | | |
| new_address | N/A | $\top\|\bot$ | Generate a public key as well as the blockchain address for the TEE cluster |
| ***Deployment*** | | | |
| new_contract | $\mathcal{C}_b$ | tx_id | Deploys a compiled smart contract and returns a transaction id |
| patched_contract | $\mathcal{C}_b$ | tx_id | Receives the bytecode-version smart contract, deploys it and returns a transaction id |
| ***Message Call*** | | | |
| lock | $\mathcal{C}_{addr}$ | tx_id | Receives the address of contract and returns the id of transaction that invokes `vote_initial()` |
| unlock | $\mathcal{C}_{addr}$ | tx_id | Receives the address of contract and returns the id of transaction that invokes `vote_halt()` |
| destroy | $\mathcal{C}_{addr}$ | tx_id | Receives the address of contract and returns the id of transaction that invokes `destroy()` |

deploy a patched contract when aiming to recovering from a defective smart contract.

For message call APIs, stakeholders can invoke them to generate message call transactions to instruct the voteDestruct mechanism. stakeholders should provide the address of the defective contract as inputs so that TEE cluster know which contract should call. SolSaviour provides 3 message call functions: `lock`, `unlock`, and `destroy`. `lock` function can invoke the `vote_initial` function in voteDestruct mechanism to lock a defective smart contract, `unlock` can invoke `vote_halt` function to unlock a falsely locked smart contract, `destroy` function can invoke the `destroy` function in voteDestruct mechanism to destroy a defective smart contract and let all assets transferred to TEE cluster. After receiving the invocation from contract stakeholders, the TEE cluster transfer the parameters to the transaction generator to generate a signed message call transaction. The transaction generator fetches the locally stored nonce value as well as the parameter and generates a raw transaction.

```
contract sample {
    struct st_holder { uint key_index; uint st_amount; bool voted; }
    address TEE_addr;
    uint contract_stake; uint support_stake;
    mapping(address => st_holder) st_map;
    enum  State {Active, Locked};
    State public state;

    modifier inState(State _state) {
        if (state != _state) revert InvalidState(); _;}

    constructor { TEE_addr = msg.sender; }

    function any_payable_function() inState(State.Active) public payable {
        st_map[msg.sender].st_amount += msg.value;
        contract_stake += msg.value;}

    function vote_initial() inState(State.Active) public {
        state = State.Locked;}

    function vote_halt() inState(State.Locked) public {
      require (msg.sender == TEE_addr);
      if((2 * support_stake) < contract_stake) { state = State.Active; }}

    function vote(bool choice) inState(State.Locked) public {
        require(!st_map[msg.sender].voted);
        st_map[msg.sender].voted = true;
        if (choice) { support_stake += st_map[msg.sender].st_amount; }}

    function destroy() public {
        if ((2 * support_stake) > contract_stake) {
            selfdestruct(TEE_addr); }}}
```

Figure 5-4: A sample of voteDestruct-enabled contract.

Then, the TEE cluster invoke the key manager to sign it. For generated signed message call transaction, the enclave transmit it to the host, i.e., a blockchain client. The blockchain client broadcast the transaction onto the blockchain for further operations.

# 5.3 Implementation

## 5.3.1 Destroying

### Destroying Defective Smart Contract

Once a contract stakeholder notices a potential vulnerability in the contract, it can broadcast the vulnerability to attract other stakeholders' attention. Then, all contract stakeholders can vote whether to lock the contract. If the support rate exceeds 1/3 (i.e., lock threshold), the contract enters a locked state and no external calls can be executed, except for calls that unlock or destroy the contract. During the locking phase, the contract stakeholders can discuss the exposed vulnerability and develop corresponding patches.

If most stakeholders think that this vulnerability is a false positive case, they can vote to unlock the smart contract and continue to work with it. The voting threshold for unlocking a locked smart contract is the same as the threshold for locking one. Afterwards, the contract can continue to operate normally. If the discussion thinks the vulnerability may lead to serious consequences, stakeholders need to develop a valid patch and test it. Then, they could vote whether to destroy the defective contract. When the support rate exceeds 2/3 (i.e., destroy threshold), the vote is passed and the contract can be destroyed by TEE cluster. All internal assets are transferred to the TEE cluster for temporary escrow.

Currently, we can use `selfdestruct` primitive to destroy deployed smart contracts and refund all inside assets [25]. When writing a smart contract, there are usually restrictions set to limit that only privileged owners can invoke the `selfdestruct` primitive, otherwise this contract can be destroyed by anyone. However, this method requires contract stakeholders to trust the privileged owner since he/she is able to withdraw all inside assets. In this case, we introduce voteDestruct mechanism, which allows contract stakeholders to destroy a deployed smart contract in a decentralized way. Fig. 5-4 shows the sample of voteDestruct mechanism. We emphasize that its implementation does not require new EVM instructions. It is constructed based on pure Solidity language. Moreover, the voteDestruct mechanism can be implemented in different versions Solidity with minor modifications.

In the life cycle of a smart contract, contract participants may deposit ethers before a bug is exposed. The voteDestruct mechanism records these participants as stakeholders `st_holder` and the amount of their deposited ethers as `st_amount`. Once a stakeholder found that this smart contract is vulnerable to some newly discovered bugs, it can invoke the `vote_initial` function via the TEE cluster. The smart contract then starts the cumulative vote in which each stakeholder chooses whether to destroy this contract and return all funds.

### Safe Exit

Through voteDestruct mechanism, SolSaviour enables contract stakeholders to safely exit from a defective smart contract. SolSaviour utilizes a cumulative voting algorithm, which calculates votes based on stake. Cumulative voting is the procedure followed by electing whether to destroy the smart contract. Typically, each stakeholder should choose to support or oppose the contract destruction. Once the vote completes, different choice will be counted according to the amount of stake. If the contract stakeholders vote not to destroy the contract to resolve the defect, they can instruct TEE cluster to invoke `vote_halt` function to unlock contract. Otherwise, the contract executes its `selfdestruct` operations and send all inside funds to an account controlled by the TEE cluster. In this way, contract stakeholders can safely exit from a contract that is exposed to some critical bugs or under attacks. Compared with traditional `selfdestruct` operations, safe exit not only saves all preserved funds, but also avoids the requirement to trust a privileged owner.

Once completing voting, the old vulnerable smart contract has been destroyed. All assets inside the defective contract are transferred to an account controlled by the TEE cluster. Before redeploying a patched smart contract, TEE cluster extracts the internal state of the old defective smart contract, namely values of state variables and stake distribution. By analyzing the internal state of defective contract, the TEE cluster is able to migrate the state of defective contract to the patched one.

### 5.3.2 Patching

In SolSaviour, patches for defective smart contracts are provided by the contract stakeholders. This is because the main purpose of SolSaviour is to provide a framework for repairing and recovering defective deployed smart contracts, rather than providing a system that can automatically generate patches. Smart contract patches can be generated manually or using existing tools such as sGuard [41] and SCRepair [120]. Once a patched contract is prepared, contract stakeholders can pass it to the TEE cluster for redeployment.

For known bugs such as reentrancy and integer overflow, stakeholders can leverage existing tools to generate patched contracts. For unknown bugs, patched contracts should be developed by experts, which are trusted by contract stakeholders. After patching, the contract should be tested thoroughly before deploying by re-executing all previous related transactions. This can test the whether the patched contract functions functions well and has fixed all related bugs. The patch tester re-executes all non-malicious historical transactions on the patched smart contract and verifies that the execution results of the old contract and the patched contract are consistent. Any execution discrepancies are scrutinised to determine whether the patch has caused the patched smart contract to function inconsistently with the defective contract. Detailed implementation on patching and testing is provided in Fig. 5-5.

### 5.3.3 Redeploying

#### Redeploy a Patched Contract

In this step, stakeholders can redeploy a patched contract and migrate the internal state from the defective contract to the patched one.

During redeployment, the TEE cluster takes charge of injecting the initial state of the patched contract and generating a contract creation transaction. The TEE cluster injects a list of stakeholder addresses and the amount of their stakes to the patched contract, which indicates the amount of assets they deposited before contract destruction. Then, the TEE cluster generates a contract creation transaction for the

```
static bytes C_patched[];

void patchGeneration(bytes C_defective, bytes& C_patched){
    C_patched[0] << sGuard(C_defective);
    C_patched[1] << SCRepair(C_defective);
    C_patched[3] << C_patched;}

bytes patchTest(bytes& C_patched[]){
    string Tx[];
    while(web3.eth.addr){ Tx += web3.eth.addr.transaction; }
    uint length = Tx.length;
    uint index;
    uint success[] = 0;

    for(int i =0; i<3; i++){
        for(index=0, index<length; index++){
            if(C_patched[i].exe(Tx[index])){
                success[i]++;}}}

    if(success[0]>success[1]){
        if(success[0]>success[2]) return C_patched[0];
        else return(C_patched[2]);}
    else{
        if(success[1]>success[2]) return(C_patched[1]);
        else return(C_patched[2]);}}
```

Figure 5-5: The implementation on patching a defective contract.

patched contract and broadcast it onto the blockchain. For contract stakeholders, the internal state of the redeployed contract remains the same as the previous defective contract, but SolSaviour has already fixed the vulnerabilities.

### State Migration

For migrating states to the newly-patched smart contract, the TEE cluster first extracts required variables from the blockchain. Then, the TEE cluster modifies the patched smart contracts provided by the contract stakeholders. The purpose of this modification is to migrate the internal state from the old, vulnerable contract to the new, patched smart contract. TEE cluster ensures that variables in the patched contract are the same with before by initializing these variables. Then, TEE cluster directly transfers all the escrow assets to the newly deployed contract. Since the stake distribution has been injected by TEE cluster, the ownership of these assets is certain and consistent, as well as their corresponding voting rights. Detailed implementation

```
void stateVariableGetter(string addr){
    struct stateVariable{ string name; bytes value; };
    stateVariable states[];
    uint length = addr.ABI.length;
    uint index;

    for (index=0, index<length; index++){
        states[index].name = addr.ABI[index].name;
        states[index].value = addr.ABI[index].value;}}

void stakeDistribution(string addr){
    struct stakeDist{ string addr; uint amount;};
    stakeDist stakes[];
    uint length = addr.st_map.length;
    uint index;

    for (index=0; index<length; index++){
        stakes[index].addr = addr.st_map[index].key;
        stakes[index].amount = addr.st_map[index].st_amount;}}

void stateMigration(string addrFrom[], string addrTo[], uint256 values[]){
    require(addrFrom.length == values.length);
    uint256 length = values.length;
    uint i;
    for (i=0; i<length; i++){
        balances[addrTo[i]] = values[i];
        emit Transfer(0x0, addrTo[i], values[i]);}}
```

Figure 5-6: The implementation on recovering a patched contract.

is listed in Fig. 5-6.

## 5.4 Security Analysis

### 5.4.1 System Model

We assume that parties who do not trust each other use the blockchain to execute smart contracts and mine new blocks. We assume most machines are equipped with TEE, which is based on the observation that most computers have SGX-capable Intel CPU. We assume that the TEE (i.e., programs inside enclaves) on a machine is trusted, but some TEE nodes may suffer from integrity and confidentiality problem. These TEE nodes might be compromised by external parties and attackers. All parties are rational and potentially malicious. When there are benefits, they may try to steal funds that belong to others and force the TEE to modify the stake distribution. All parties are connected via the network, and they can discard and replay the information. Malicious hosts can delay or prevent others from accessing the blockchain for an unlimited time, but we assume that this will not happen indefinitely. We also assume that an adversary $\mathcal{A}$ can corrupt up to $t$ of $n$ hosts in the TEE cluster. We consider the adversary can cause corrupted hosts to deviate from the specified protocol, namely drop or delay messages between enclaves. Our adversary $\mathcal{A}$ is static, namely chooses the corrupted hosts at the beginning of the protocol.

### 5.4.2 Trusted Execution as a Root-of-Trust

In SolSaviour, participants can monitor the blockchain to detect deviations from the protocol and react appropriately. Here, we propose a new trust mechanism with a TEE cluster as the root of trust. The TEE cluster is independent of the blockchain and can ensure the faithful execution of SolSaviour. TEEs are encrypted memory regions that can protect the security of internal execution. Contents in protected memory are separated from other applications even higher-privilege system software. By using the TEE cluster as an independent root of trust, SolSaviour can ensure secure exit, redeployment, and effective state migration of defective smart contracts. In addition, the cluster architecture improves the overall fault tolerance of the system. Single point failures will not affect the overall availability of SolSaviour.

### 5.4.3 Threat Analysis

In this section, we analyse the security of SolSaviour from three perspectives: balance security, correctness and fairness.

#### Balance Security

The first security property of SolSaviour is defined as **balance security**, which says that honest stakeholders cannot lose money as long as they behave honestly.

**Definition 7.** *(Balance Security) The SolSaviour protocol run by parties satisfies* *balance security if for a contract $\mathcal{C}$ executed by (n,m) parties, for every adversary $\mathcal{A}$ corrupting only parties from $\mathcal{P}$, the resulting stake distribution of the protocol execution equals to the initial stake distribution.*

For balance security, it is important that contracts repaired by SolSaviour do not result in a loss of assets, except for the necessary gas consumption. If during the setup phase, the participants do not agree on a patched contract $\mathcal{C}$, then the defective smart contract remains locked as well as all deposits and hence does not lose any more coins. If the parties agree on a patched contract during the setup phase, the TEE cluster proceeds to redeployment phase, then the enclave generates a contract creation transaction.

In this case, the balance of patched contract and defective contract remains same. First, assets are held by TEE cluster after destroying defective smart contracts. As assumed before, attackers can only control no more than $t$ of $n$ TEE nodes and cannot derive the account private key. Thus, attackers cannot perform asset transfer directly. Since the execution logic of enclaves is fixed once encapsulated, there is no way for attackers to tamper with the internal logic of the TEE cluster. In addition, the history of defective smart contracts is stored on the blockchain and publicly available. TEE cluster can crawl the contract history to determine the values of defective contract's internal variables. The stake distribution can also be calculated by querying the transaction history of the contract and thus acquiring the internal state of the defective contract. In this way, TEE cluster can ensure safe and successful migration of contract internal state from the old vulnerable smart contract to the new patched one. In summary, an attacker who cannot tamper with the logic inside the TEE cannot prevent the TEE from transferring assets to a patched contract. at the same time, due to the tamper-evident nature of the blockchain, once an asset is identified in a new contract, it cannot be stolen.

### Correctness

The second property is **correctness**. Intuitively, correctness states that in case all parties behave honestly, every party outputs the correct result and successfully transmit the assets from defective smart contracts to the deployed patched contracts.

**Definition 8.** *(Correctness) Protocol run by parties satisfies correctness property if for a contract $\mathcal{C}$ executed by (n,m) parties, the patched contract outputted by SolSaviour functions identical to the original contract.*

For the proof of correctness, we divide it into several phases: bootstrapping of the TEE cluster and destruction of the contract.

Firstly, we prove that the bootstrapping process of TEE cluster satisfies correctness with threshold $t$, which indicates that TEE nodes can reach an agreement on a key when there are at most $t$ corrupted parties among $n$ TEE nodes. During the bootstrapping process, once a node $U_i$ receives other node's $K_j$, it can verify it. If the check fails for an index $i$, $U_i$ can broadcast a complaint against node $U_j$. If more than $t$ nodes complain about a node $U_j$, that node is recognized as disqualified. Each node stores a node set $QUAL$ for all qualified nodes. In this case, they generate the key based on nodes inside $QUAL$. As all honest nodes construct identical $QUAL$, they can generate the same key and derive the same blockchain address. Then, we prove that TEE cluster can reach an agreement on an identical key as long as more than $t$ out of $n$ nodes are honest. Thus, we can show that TEE cluster can correctly complete the bootstrapping process for a given attacker threshold.

Secondly, we prove that the destruction of defective smart contracts satisfies correctness. When all parties are honest, the protocol starts with the setup phase, where the parties from $\mathcal{P}$ vote to decide whether to destroy the defective smart contract, followed by a message call transaction generated by TEE cluster. The message call transaction invokes the `destroy()` function in the voteDestruct mechanism to conduct the destruction of defective smart contract. Once the `destroy()` transaction is confirmed on the blockchain, then the destruction completes and cannot be reverted.

### Fairness

Finally, we define the **fairness** property of SolSaviour.

**Definition 9.** *(Fairness) A protocol run by parties satisfies the fairness property if* *for every (n-m)-contract $\mathcal{C}$, for every adversary $\mathcal{A}$ corrupting parties such that at least one party is honest, the output of protocol execution is true.*

For the proof of fairness, we prove that the voteDestruct mechanism is fair. That is, even if an attacker holds a certain stake in the defective contract and has the right to vote, he cannot influence the final outcome of the vote and prevent the destruction of the defective smart contract.

We first analyze the case where there are some malicious stakeholders. As malicious stakeholders are profit oriented, what they want is to acquire the assets of honest stakeholders. We prove the security of voteDestruct mechanism by showing that honest stakeholders can always safely exit a smart contract as long as their cumulative stake amount exceeds the specified destroy threshold. In SolSaviour, a smart contract has three statuses, active but potentially defective, locked, and destroyed. In locked status, a contract is protected by blockchain miners that reject all function calls except those initiated from the TEE cluster. In this case, malicious stakeholders cannot steal assets. In destroyed status, assets in a contract are held in custody by the TEE cluster, so that malicious stakeholders cannot profit either. The only chance for malicious stakeholders to profit is during the active but potentially defective status. In SolSaviour, the threshold of required stake amount to lock a contract is $1/3$. Only when the amount of stake held by malicious stakeholders exceeds $2/3$, they can prevent the contract from entering the locked status.

During the active but potentially defective status, when a hidden vulnerability is exposed, malicious stakeholders can prevent the contract from entering the locked status and exploit the vulnerability. However, due to the unknown nature of the vulnerability, it may simply causes the contract to an inexecutable state, which would also be unprofitable for malicious stakeholders. Therefore, the only feasible way for attackers to exploit a contract is to inject an exploitable vulnerability during the initial contract deployment or redeployment of patched contract. However, as the deployed contract needs to pass the checks of all stakeholders, honest stakeholders can reject a potentially vulnerable contract. Even if an attacker possesses an unknown vulnerability and successfully deploys a malicious contract with it, SolSaviour can increase the attack cost and reduce the losses of honest stakeholders by lowering the threshold for entering the locked state. For example, if the threshold is lowered to 1/10, the attacker must have 9/10 of the stake to guarantee that this contract will not enter the locked stake, which will also reduce the losses of honest stakeholders.

## 5.5 Discussion

### 5.5.1 Limitations and Security Risks

In this section, we discuss the limitations and security risks of SolSaviour. One of the main limitations of SolSaviour is that it can only protect contracts that have integrated the voteDestruct mechanism. Due to the tamper-proof feature of the blockchain, SolSaviour cannot provide the defence mechanism for active smart contracts that have already been deployed. As TEE is a technology still under development, there may be unknown vulnerabilities. TEE is therefore at risk and newly discovered TEE vulnerabilities could compromise the security of the entire system and the in-contract assets.

### 5.5.2 Recovering Patched Contract without TEE Escrow

#### Implementation

Considering the risks to the security of assets temporarily held in the TEE cluster, we introduce a new way to recover a defective smart contract to a patched contract directly.

**Setup Phase.** In this way, SolSaviour first locks a potentially defective contract to prevent it from further attacks. Then, stakeholders develop a patched smart contract and provide it to the TEE cluster. The TEE cluster first deploys the patched contract onto the blockchain. After deploying the patched smart contract, the voteDestruct mechanism could set the parameter of the `destroy` function to the address of the patched contract. In this way, the assets are directly transferred from the defective smart contract to the patched contract without interaction of the TEE cluster.

In this way, even the serious problem such as key leakage of the account controlled by the TEE cluster, the assets are still under protection.

**Recovering Phase.** Unlike reverting to a TEE cluster, this approach eschews the use of TEE and therefore its state transfer behaviour cannot be implemented through TEE. We then need to implement complex state migration logic in the smart contract. Specifically, we need to store the complete state variables of the smart contract and the take distribution of the assets stored in the take holders, and we need to have transfer logic that acts on top of these state variables. We consider using a bridging contract to accomplish this step, i.e. deploying a smart contract dedicated to state migration, in which the state variables and take distribution associated with the defective smart contract are stored, and in the constructor of the patched contract, allowing it to read and write the state variables in the contract. In this way, we can achieve state transfer for smart contracts without the need for TEE intervention. In contrast, this approach requires significant gas consumption to maintain the various operations and data stores. The cost required to implement contract recovery on the public chain is significantly higher due to the addition of bridging contracts and the corresponding storage of state variables. However, the advantages of this approach are clear: by avoiding temporary asset hosting of TEE clusters, this approach significantly reduces the attack surface of the system and eliminates the risk of security issues that may arise from the TEE itself.

### Comparison

In summary, there are advantages and disadvantages to both recovering to the TEE cluster and recovering to the patched contract, the former being more gas efficient but less secure than the latter, and the latter being secure, but the cost of protecting the smart contract is greatly increased by the high gas consumption it entails. The latter is secure, but the high gas consumption associated with it can add significantly to the cost of protecting smart contracts. Therefore, when faced with a specific smart contract, the contract developer needs to make a trade-off between security and cost and choose the best method to protect the smart contract.

Table 5.2: The list of contracts that have been attacked or exposed to serious bugs.

| Contract | Address | Vulnerability Type | Caused Damage |
|---|---|---|---|
| King of Ether | 0x2464d1d97f8D0180CFaD67BdB19bc30ccA69DdA0 | Unchecked Return Values | Ownership Loss |
| GovernMental | 0xF45717552f12Ef7cb65e95476F217Ea008167Ae3 | Timestamp Dependence | DoS |
| Rubixi | 0xe82719202e5965Cf5D9B6673B7503a3b92DE20be | Bad Constructor | Ownership Loss |
| ENS Name Wrapper | 0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e | Access Control | Domain Ownership Loss |
| $1^{st}$ Parity Multisig | 0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4 | Delegatecall | 153,037 ETH Loss ($31M) |
| $2^{nd}$ Parity Multisig | 0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4 | Denial of Service | 513,774.16 ETH Locked ($300M) |
| The DAO | 0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413 | Reentrancy | 3.6M ETH Loss ($150M) |
| PoWH Coin | 0xA7CA36F7273D4d38fc2aEC5A454C497F86728a7A | Integer Underflow | 866 ETH Loss ($800k) |
| Bancor Exchange | 0x1F573D6Fb3F13d689FF844B4cE37794d79a7FF1C | Front Running | Economic Earns ($150) |
| SushiSwap | 0x6B3595068778DD592e39A122f4f5a5cF09C90fE2 | Supply Chain Attack | 864.8 ETH |
| Fei | 0x956F47F50A910163D8BF957Cf5846D573E7f87CA | Price Manipulation | 60k ETH at risk |
| Uniswap Hack | 0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984 | ERC777 Reentrancy Exploit | $320M |

## 5.6 Experiment

In our prototype of SolSaviour, the voteDestruct mechanism is implemented in Solidity and the TEE cluster is implemented based on Intel SGX with around 2000 LOC. Four nodes are set up in the TEE cluster. The experiments are conducted in two aspects: effectiveness and performance.

### 5.6.1 Dataset Preparation

To accurately evaluate the effectiveness and performance of SolSaviour, we have collected a number of contracts that have emerged from real-world applications, including some from early applications and some DeFi contracts that have emerged in recent years. Some contracts have experienced real attacks that have caused asset losses; others have not been attacked, but their vulnerability has been verified. Our collected contracts are the DAO [21], PoWH Coin [10], 1st [93] and 2nd [1] Parity Multisig Wallet, King of Ether [106], Bancor Exchange [16], GovernMental [112], and Rubixi [56]. We also collected some DeFi contracts that were exposed to some severe bugs such as SushiSwap [98], ENS Name Wrapper [97], Fei [78], and Uniswap [90]. We list these contracts in Table. 5.2, accompanying with contract vulnerability type and caused damage as well as accurate loss (if so).

For our collected contracts, we also prepare corresponding voteDestruct-enabled contracts and patched contracts. The voteDestruct mechanism is injected on the source code level. As collected contracts are written in different versions of Solidity, we make minor modifications to make our voteDestruct mechanism compatible in all versions of Solidity. For patched contract, our collected contracts are also patched manually by modifying the code. We also ensure that the compiled voteDestruct-enabled contracts and patched contracts following the same version of Solidity as original contracts.

SolSaviour is then applied to these generated comparative smart contracts so that we could verify the effectiveness and performance by validating the results. We can verify whether the voteDestruct mechanism is effective and that the patched smart contract redeployed by SolSaviour fixed vulnerability.

### 5.6.2 Effectiveness

The effectiveness of SolSaviour is evaluated from two perspectives: qualitative and quantitative.

For qualitative part, we check whether we can leverage SolSaviour to safe exit from all collected defective contracts, refund locked assets back to stakeholders, and redeploy a patched contract. To test whether SolSaviour can recover a buggy smart contract, we generate a large and representative evaluation dataset by collecting transactions sent to the collected contracts from the Ethereum. Replaying those transactions and observing outcomes can check the functionality and defence of patched contracts. Specifically, we test whether SolSaviour can successfully destroy a defective smart contract with voteDestruct mechanism and redeploy a patched one with TEE cluster. In addition, we test whether the TEE cluster can successfully migrate the previous state to the new contract to ensure the state consistency.

For the quantitative part, we set up two contract instances for each collected defective contract: an original contract instance and a SolSaviour-protected instance. We compare the loss between the original one and SolSaviour-protected one. For the original one, we also record the loss when taking traditional defence measures and doing nothing. We test to what extent can SolSaviour save loss when facing different vulnerabilities.

Table 5.3: Comparison of losses/damages in the event of an attack against our collected contracts. “Actual” indicates that no action is taken; “Traditional” indicates that traditional defence methods are used; “SolSaviour” indicates that the contract is protected by SolSaviour.

| Contract | Actual | Traditional | SolSaviour |
|---|---|---|---|
| King of Ether | Lose Onwership | No Mitigation | Fix |
| GovernMental | Lose Ownership | No Mitigation | Fix |
| Rubixi | Lose Ownership | No Mitigation | Fix |
| ENS Name Wrapper | Lose Ownership | No Mitigation | Fix |
| $1^{st}$ Parity Multisig | 100 | 100 | 0 |
| $2^{nd}$ Parity Multisig | 100 | 100 | 0 |
| The DAO | 100 | 48.6 | 6.5 |
| PoWH Coin | 100 | 100 | 0 |
| Bancor Exchange | 100 | 69.6 | 0 |
| SushiSwap | 100 | 100 | 3.3 |
| Fei | 100 | 61.2 | 2.3 |
| Uniswap Hack | 100 | 54.3 | 4.6 |

### Qualitative

We evaluate the effectiveness of SolSaviour from a qualitative perspective in three aspects: successful state migration, identical functionalities, and successful defence. For each contract, we use Ganache to simulate 10 accounts, who play the role of contract stakeholders and each has deposited 100 ethers. Then, a random stakeholder initializes the `vote_initial` and provides a patched contract to the TEE cluster. In qualitative experiments, we omit the security assumption of potential malicious stakeholders and assume all of them vote to destroy the defective contract. Once the voting completes, the TEE cluster destroys the defective contract, redeploys a patched one, and conducts state migration.

By checking the patched contracts deployed by the TEE cluster, we can evaluate whether state migration successes. A successful state migration means the internal states of buggy contract and patched contract are identical. Not only the stake distribution, but also the ownership. We check this by letting each stakeholder withdraw their previously-deposited ethers. We found that stakeholders can withdraw their assets successfully from all contracts. Then, we check the functionality and defence of patched contracts by replaying collected transactions. We compare the execution results of patched contract with defective contract history state transition. Our experiment results show that SolSaviour can successfully migrate the values of state variables and stake distribution to the redeployed patched contracts. The contracts redeployed by SolSaviour are functionally identical to the original contracts and have fixed vulnerabilities of the original contracts.

#### Quantitative

We evaluate the effectiveness of SolSaviour from a quantitative perspective by attacking and recovering defective contracts simultaneously. We evaluate to what extent can SolSaviour reduce loss. Since different contracts are tested in different scenarios, the amount of loss is different.

For the DAO contract, we simulate a scenario, where the defective deployed contract contains 100 ethers. Then, we start to attack and recover it at the same time. Attackers can arbitrarily withdraw ethers until honest stakeholders lock the contract. Then, we follow the safe exit way to refund all locked ethers to stakeholders and calculate the loss. Similar steps are conducted to evaluate the loss when using SolSaviour. For the PoWH coin contract, since the real attack transactions are limited, we simply replay these attack transactions and check the execution results. We also test the loss when doing nothing and taking traditional defence. The results are listed in Table. 5.3.

### 5.6.3 Performance

#### Contract Size Increase

In this section, we evaluate the additional code required to use SolSaviour. On Ethereum, deploying smart contracts consumes gases, which are proportionally to the size of the deployed contract. In SolSaviour, as the voteDestruct mechanism is

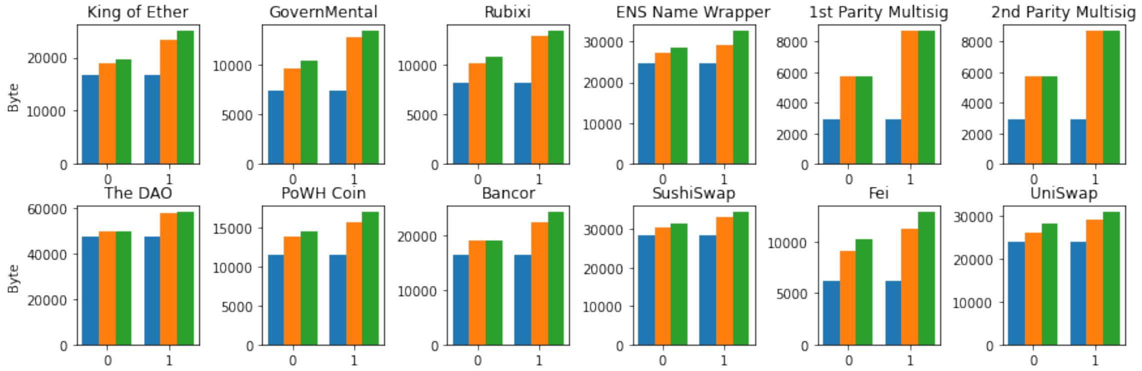


Figure 5-7: Contract size increase in SolSaviour. The left three bars represent using TEE cluster for asset escrow while the right three bars using patched contract. Blue bar indicates original contract, orange bar indicates voteDestruct-enabled contract, and green bar indicates patched contract.

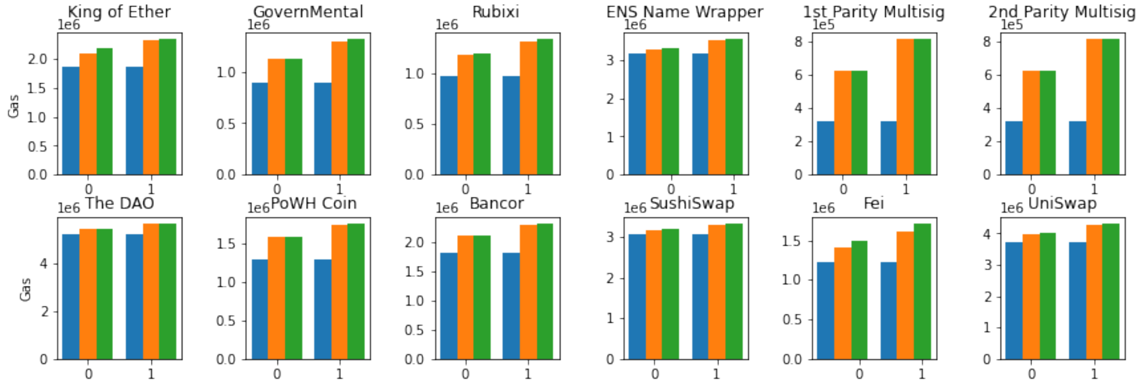


Figure 5-8: Gas consumption of SolSaviour. The left three bars represent using TEE cluster for asset escrow while the right three bars using patched contract. Blue bar indicates original contract, orange bar indicates voteDestruct-enabled contract, and green bar indicates patched contract.

implemented inside contracts, extra codes are introduced. Moreover, the method on recovering patched contract without TEE escrow also introduces extra code in the contract. Results are summarized in Fig. 5-7. The code size of the original collected defective contracts are listed as the baselines. Each subplot has six bars. The left three bars represent the size of contract when recovering with TEE, while the right three bars represent the size of contract when recovering without TEE escrow. In each subplot with three bars, from left to right are the size of the original contract, the size of voteDestruct-enabled contract and the size of the patched contract.

However, since patches are generated manually, it is impossible to determine the performance of the system in terms of the number of lines of code added by the patch. The only additional code introduced by the system is the voteDestruct framework that makes defective contracts have the ability to iterative upgrades under SolSaviour. In this case, we compare the size of compiled contract. We found that voteDestruct mechanism introduces limited size to the original contract. These code size increases are worth compared to the security enhancement that SolSaviour brings. For Parity Miltisig contract, we note that injecting voteDestruct mechanism naturally resolves the vulnerability so that the patched contract and voteDestruct-enabled contract have the same size.

### Gas Consumption

In this section, we evaluate the additional gas consumption incurred by SolSaviour, which mainly arises from two aspects: the voteDestruct mechanism and the redeployment of the patched contract. We also evaluate the gas consumption on redeploying a patched contract without TEE asset escrow. Results are summarized in Fig. 5-8. The gas consumption to deploy the original version of collected defective contracts are evaluated as the baselines. Each subplot has six bars. The left three bars represent the gas consumption of original contract, voteDestruct-enabled contract, and patched contract respectively when letting TEE conduct asset escrow. By contrast, the right three bars represent the gas consumption when recovering without TEE asset escrow.

For voteDestruct mechanism, the evaluation is conducted by deploying our prepared voteDestruct-enabled contracts. From the results, we can see the voteDestruct mechanism introduced minimal gas overhead. The gas cost are mainly introduced by additional storage of state variables and corresponding logic. Storing data on Ethereum is expensive, which leads to a lot of gases to be consumed. For the redeployment of patched contract, the extra gas consumption are mainly introduced by the patch. As the gas consumption depends on the contract size, namely the size of original contract plus the size of the patch as well as the voteDestruct mechanism for future protection. As shown in the results, the overhead introduced by the patch is not stable. This is because different contracts require different type of patches.

Table 5.4: The overhead of TEE Cluster, which is counted in the number of blocks mined by different combinations of TEE nodes.

| Node | | A | AB | ABC | ABCD |
|---|---|---|---|---|---|
| All | 5780 | 5685(-1.6%) | 5507(-4.7%) | 5469(-5.3%) | 5391(-6.7%) |
| A | 1451 | 1368(-5.7%) | 1312(-9.5%) | 1340(-7.6%) | 1341(-7.6%) |
| B | 1446 | 1439 | 1325(-8.3%) | 1339(-7.4%) | 1335(-7.7%) |
| C | 1438 | 1441 | 1439 | 1351(-6.1%) | 1349(-6.2%) |
| D | 1445 | 1437 | 1431 | 1439 | 1366(-5.5%) |

**TEE Cluster Overhead**

In SolSaviour, state migration and asset escrow are conducted by TEE cluster. We therefore evaluate the overhead introduced by TEE cluster. We build a Ethereum private network with four nodes (i.e., node A, B, C, and D), each is installed with an Ethereum endpoint. We record the number of blocks mined by them in one day. Then, we initialize the enclave in one node and monitor the blocks mined by each node. After that, we sequentially initialize enclaves in the other three nodes and add them to the TEE cluster. During this time, we continuously monitor the number of blocks mined by each node. The mining difficulty remains the same during this experiment. We summarize the results in Table. 5.4. As we can see, for nodes without TEE cluster, they can mine around 1440 blocks per day, which satisfies the Ethereum blockchain generation speed, namely a block per 15 seconds. For nodes with TEE cluster, the mining rate is slightly affected. The impact is greatest when only half nodes participate the TEE cluster, and tends to become smaller when all nodes initialize TEE.

## 5.7 Related Work

### 5.7.1 Smart Contract Vulnerability Detection

The infamous reentrancy bug in “TheDAO” contract [21] has spurred community to work on detecting smart contract vulnerabilities. Luu et al. first proposed Oyente [73] based on symbolic execution, which automates the reentrancy bug detection. Then, a lot of symbolic execution tools are proposed such as Osiris [108], teEther [62], Maian [84], and Manticore [79]. Furthermore, Frank et al. proposed EthBMC [43], a bounded model checker based on symbolic execution. Kalra et al. presented Zeus [53], which leverages both abstract interpretation and symbolic model checking. Chen et al. identified and defined 20 types of contract defects [26] and proposed the corresponding defect detection tools to find them on the bytecode level [27].

There are also some work on developing smart contract static analysis tools. Feist et al. proposed Slither [40] to analyze the contract on source code level, and Tsankov et al. presented Security [110] to analyze the contract on bytecode level. Furthermore, Brent et al. proposed Ethainter [20], which conducts the information flow analysis and data sanitization to reveal composite vulnerabilities. There are also work on building modular dynamic analysis frameworks for protecting smart contracts. Chen et al. proposed SODA [29], which accepts user-defined vulnerability pattern. Furthermore, method like formal verification has been introduced to smart contracts [53] and the semantics of Solidity have been formalized [51]. Pan et al. proposed ReDefender [86], which detects reentrancy vulnerabilities with fuzz testing.

However, these proposed detection tools still have limitations. For example, teEther [62] and Maian [84] cannot locate integer overflow bugs since they mainly focus on generating exploits for smart contracts. In addition, aforementioned detection tools cannot identify unknown vulnerabilities.Therefore, there exists a requirement to develop a framework for protecting deployed smart contract like SolSaviour.

### 5.7.2 Smart Contract Defence and Patch

For contract defence, Rodler et al. proposed *Sereum* [94] to defend against reentrancy exploits through analyzing the specific sequence of transactions and the EVM execution traces. Ferreira et al. proposed ÆGIS [42] to defend deployed contracts against known attacks. Specifically, it records a number of known contract attack execution traces through a proposed domain-specific language and integrates within the EVM to revert the execution of transactions that match the attack traces. Ellul et al. proposed a runtime verification mechanism [38] to ensure that violating party provides insurance for correct behavior. Li et al. proposed SOLYTHESIS [67] to address the high overhead in runtime validation. Grossman et al. proposed ECFChecker [49], which defines the notion of Effectively Callback Free (ECF) objects and can detect live reentrancy attacks on vulnerable contracts.

For contract patch, Yu et al. proposed SCRepair [120], which can automatically detect and repair bugs in smart contracts before deployment. Similarly, Zhang et al. proposed SMARTSHIELD [124], which leverages the bytecode rewriting technique to fix contract vulnerabilities automatically. Bytecode rewriting technique was also used in the EVMPatch [95] proposed by Rodler et al. which can automatically repair vulnerable smart contracts.

### 5.7.3 TEE Related Work

In this section, we first present work related to the prevalent smart contract vulnerability detection tools. Then, we discuss some work on protecting deployed smart contracts and generating patches for vulnerable smart contracts automatically. Finally, we present work on combining blockchain with TEE.

### 5.7.4 Smart Contract and TEE

Town Crier [121] was proposed by Zhang et al. to address the problem that smart contracts running on the blockchain cannot access information in a trusted way. In combination with TLS, smart contracts can trust information from HTTPS sites passed by the Town Crier. Matetic et al. proposed BITE [75], which is a SGX-based lightweight node, to address the privacy issue in lightweight nodes. In BITE, full nodes load SGX enclaves to process requests from lightweight nodes. Thus, BITE can ensure the privacy of lightweight nodes compared to traditional lightweight nodes that have the risk for privacy breaches caused by Merkle proof requests.

There is a range of work focusing on offloading the execution of smart contracts to TEE, which enables privacy-preserving smart contracts on the one hand and delivers some performance gains on the other. Bowman et al. proposed Private Data Objects (PDOs) [18], in which contracts are deployed in the enclaves and mutually untrusted participants can invoke the contracts in the enclave to execute on their own private data. Cheng et al. proposed Ekiden [30], whose execution of smart contracts is also deployed inside enclaves. With the enclave's public key, users can encrypt the data in their message call transactions and call the contract in the enclave in a method called confidential transaction. In addition, consensus is decoupled from execution in Ekiden, so miners do not need to verify private contract execution, preserving confidentiality. Das et al. proposed FastKitten [34] to enable the execution of complex, high-performance smart contracts on Bitcoin. FastKitten employs TEE to execute arbitrarily complex smart contracts efficiently on cryptocurrencies, which are originally designed without smart contract support. In FastKitten, enclaves take charge of executing smart contracts and generating state transitions, which are recorded by the Bitcoin. FastKitten can extend its work to execute smart contracts on more cryptocurrencies which were designed to only support naive transactions.

In addition, the combination of blockchain and TEE shows promise in many other areas. Zhang et al. proposed REM (Resource-Efficient Mining) [123], a blockchain mining algorithm that work on useful computation. Considering the problem that traditional PoW consensus protocols consume a lot of power, REM proposes to use SGX to convert the traditional meaningless hash computation into meaningful computation workload, accompanied by a trusted verification mechanism. Lind et al. leveraged TEE as trusted nodes in payment network and proposed Teechain [70], which is a layer-two network that can processes off-chain transactions asynchronously. TeeChain leverages TEE as the entry and exit point for off-chain payment channels and enables asynchronous execution of off-chain transactions. Teechain can prevent parties from misbehaving by formalizing off-chain payment transactions with TEE. The combination of blockchain and TEE shows great potential due to their superior characteristics and complementary nature in areas such as privacy preserving and attested execution. Kaptchuk et al. explores the computational properties of using blockchain to store the state of stateless TEE [55].

## 5.8 Conclusion

In this chapter, we present SolSaviour for protecting deployed smart contracts and DeFi protocols. SolSaviour enables the offline patching of defective smart contracts through the decentralized control provided by voteDestruct mechanism and the state migration provided by TEE cluster. Compared with existing work that requires a trusted third party to redeploy patched contract and can only migrate contract data, SolSaviour can achieve effective migration of contract assets and does not require the involvement of a trusted third party. For all collected contracts and DeFi, our experiment results demonstrate that SolSaviour can effectively repair and recover all of them with affordable overhead.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

Blockchain technology guarantees data security through a distributed consensus mechanism, which ensures that the data on the chain cannot be tampered. In contrast, TEE is a technology that guarantees the security of program execution. It secures the correct execution of programs even under untrustworthy hosts by encrypting memory and enables users to verify program execution results by attested execution.

In the work of securing DNS data, we propose B-DNS based on blockchain technology, which is a secure and efficient naming system. B-DNS transforms the resource record in the traditional DNS into operation record and enables data management, update, and revocation in the DNS under the blockchain data structure by registering records, updating records, etc. The proposed dual bloom filter algorithm enables efficient data querying under the blockchain. In the face of the shortcomings of traditional blockchain-based DNS, such as Namecoin being limited by the security of the PoW consensus algorithm, B-DNS proposes a PoS consensus algorithm to improve the security against mining attacks. In the subsequent experimental results, we found that the attack cost of B-DNS is significantly improved compared to traditional DNS. When facing cache poisoning attacks on B-DNS and traditional DNS measured with the same probability of success, we found that the success rate of attacks on B-DNS tends to decrease exponentially. At the same time, due to the efficient query mechanism, B-DNS avoids the problem of inefficient queries caused by the direct adoption of blockchain technology and can achieve query speeds comparable to those achieved by the current mainstream blockchain.

In the work of PKI security, we propose a trusted and authorized certificate issuance mechanism Pistis from the logic of certificate issuance. We found that the essence of the traditional rogue certificate problem is that CA issues certificates without the domain's permission. If CA issues certificates when and only when the domain applies, we can guarantee that the certificates issued by CA are trusted and authorized. This way of thinking requires that we can ensure that the logic executed by CA is fixed, autonomous, and non-manipulable. Smart contracts based on blockchain technology naturally become the technology of choice. Smart contracts can implement CA certificate issuance, renewal, revocation, and other operations, and the security of smart contracts is guaranteed by the underlying blockchain. However, the pure smart contract cannot verify whether the applicant controls the domain. In this case, Pistis introduces TEE to link the smart contract and the domain applied by the applicant to achieve ownership validation by a challenge-proof mechanism. The certificate issued by the smart contract will be uploaded to the chain to guarantee its trustworthiness. In this way, Pistis implements a trusted and guaranteed certificate issuance mechanism. Our security analysis demonstrates the security of the Pistis mechanism. Our experiments verify the efficiency of Pistis in verifying the certificate state, which has faster latency and smaller packet size compared to traditional CRL and OCSP methods, i.e., higher efficiency.

In the work of securing smart contracts, we focus on the security of smart contracts that have been deployed to the blockchain. Since smart contracts are stored in the form of compiled bytecode, and the transactions are not tamperable once they are included in the newly mined blocks on the blockchain, the deployed smart contracts cannot be modified. This leads to the fact that if any vulnerability occurs in the deployed smart contract, we cannot fix the vulnerability by patching it, and the smart contract will become a live target. Traditional solutions can only perform static checks before smart contract deployment, and this approach cannot cope with unknown bugs and has false negatives resulting in false positives. Therefore, we propose a TEE-based, trusted protection mechanism for smart contracts and DeFi, SolSaviour. SolSaviour consists of two parts, an on-chain Solidity-based voteDestruct mechanism and an off-chain TEE cluster mechanism that can invoke the voteDestruct mechanism. Once the defective deployed smart contracts are destroyed, the user can call the TEE cluster to redeploy a smart contract through the TEE cluster. At the same time, the TEE cluster automatically transfers assets and internal variables from the destroyed defective smart contract to the new patched smart contract. The experimental results show that SolSaviour can significantly reduce the potential damage caused by smart contract attacks compared to traditional defence methods and inaction. At the same time, the additional overhead is still very manageable.

## 6.2 Future Work

Our current work certainly has some limitations, as blockchain technology and TEE technology still have some limitations, such as the throughput limitation of blockchain technology. At the same time, TEE technology also has some security risks, such as data leakage. In the future work, we focus on further improving the current work's security and guaranteeing the proposed method's completeness through verification and other means. At the same time, we also hope to improve further the effectiveness of the current work, such as implementing high-capacity, high-throughput blockchain-based DNS and PKI.

### 6.2.1 Building Network Systems based on High Throughput Blockchain

The performance of traditional single-chain blockchain systems is still the bottleneck to achieving large-scale DNS and PKI. Current blockchain throughput remains at a few or a dozen transactions per second, which is difficult to scale due to security constraints. The current high-performance blockchain system has several directions, such as sharding-based multi-chain structure, directed acyclic graph (DAG) based tree graph structure, etc. We plan to build DNS and PKI based on these high-performance blockchain systems in the future. We plan to develop DNS and PKI based on these high-performance blockchain systems in the future to ensure global consistency and high fault tolerance.

### 6.2.2 Integration of DNS and PKI

In the Internet, DNS and PKI provide different services independently but work closely together. DNS is responsible for the conversion from domain names to IP addresses, and is also responsible for the registration, renewal (i.e., updating the IP address corresponding to the domain name), and revocation of domains. In PKI, the subject of a certificate application is usually the domain. This is because the domain needs a cryptographic certificate to prove its identity. Thus, DNS and PKI are actually two systems that are closely linked. In the work of B-DNS and Pistis, we propose blockchain-based solutions for some problems in DNS and PKI, respectively. But our solutions are independent and not combinable. Therefore, in our future work, we consider organically combining DNS and PKI to propose practical solutions oriented to the combination of the two systems.

### 6.2.3 Automated Smart Contract Protection Mechanism

Our proposed smart contract protection mechanism, SolSaviour, still has some limitations. It still requires a manual introduction, i.e., patches for smart contracts still need to be generated and deployed manually. We consider implementing a stronger automated mechanism to minimize human intervention and improve the security of the system. This requires us to explore automated smart contract vulnerability detection research, smart contract patch generation research, and research on rewriting smart contracts from bytecode level and source code level. By exploring the above related directions, we can further address the shortcoming that SolSaviour can only provide a fix for deploying smart contracts.

Among the three directions mentioned above, smart contract patch generation and smart contract rewriting currently have a certain research base. A lot of work has also been done on vulnerability detection for smart contracts, but the significant problem is that current vulnerability detection for smart contracts focuses mainly on detecting known vulnerabilities. Through symbolic execution, fuzzy testing, and other methods, the contract is analyzed for known vulnerabilities from the source code or bytecode perspective. None of the current vulnerability detection can verify the detection of unknown smart contract vulnerabilities. Therefore, we consider exploring AI-based vulnerability detection. Specifically, we plan to explore pseudo-anomaly-based vulnerability detection techniques. That is, we artificially generate some anomalous data points to train the model so that the classifier only learns the behavior patterns of normal contracts. This is used as a basis to discriminate unknown vulnerabilities, i.e., abnormal behavior patterns of contract execution.

### 6.2.4 Securer TEE Support

We have directly applied TEE techniques to achieve the required functionality in our current work, such as ownership validation for domains and message call transaction generation for voteDestruct. Our work is based on Intel SGX implementation. However, Intel SGX also has many security issues that affect the security of its implementation and the confidentiality of data. Therefore, we are considering further improving the system's security by combining the latest security research about TEE. We have shown this tendency in our current work, such as in the work [68], the keys of blockchain accounts are stored in all TEE nodes. On the contrary, we have adopted a new key management mechanism in the subsequent work, i.e., all TEE nodes do not store the keys but only the partial verification key. In this way, we circumvent the possible data privacy issues of TEE. In addition, we consider refactoring our system architecture based only on the integrity and verifiability properties of TEE. Specifically, we lower our security assumption that TEE is a black box that cannot guarantee data security and only guarantees program integrity and verification of program execution results. We explore feasible mechanisms for securing DNS, PKI, and smart contracts based on such a framework.

# Bibliography

[1] Anthony Akentiev. Parity Multisig Hacked. Again, 2017.

[2] Mustafa Al-Bassam. SCPKI: A Smart Contract-based PKI and Identity System. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 35–40, 2017.

[3] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *2016 USENIX Annual Technical Conference*, pages 181–194, 2016.

[4] Sara Alouf, Nicaise Choungmo Fofack, and Nedko Nedkov. Performance Models for Hierarchy of Caches: Application to Modern DNS Caches. *Performance Evaluation*, 97:57–82, 2016.

[5] Sumayah Alrwais, Kan Yuan, Eihal Alowaisheq, Zhou Li, and XiaoFeng Wang. Understanding the Dark Side of Domain Parking. In *23rd USENIX Security Symposium*, pages 207–222, 2014.

[6] AMD. AMD Secure Encrypted Virtualization (SEV), 2020.

[7] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, pages 1–7, 2013.

[8] ARM. Arm TrustZone Technology, 2020.

[9] Hitesh Ballani and Paul Francis. Mitigating DNS DoS Attacks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 189–198, 2008.

[10] Eric Banisadr. How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight, 2018.

[11] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack Resilient Public-Key Infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 382–393, 2014.

[12] Jason Bau and John C. Mitchell. A Security Evaluation of DNSSEC with NSEC3. In *Network and Distributed Systems Security*, pages 1–18, 2010.

[13] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time Cryptocurrency Exchange Using Trusted Hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1521–1538, 2019.

[14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM workshop on Programming Languages and Analysis for Security*, pages 91–96, 2016.

[15] Blockchain-DNS. Blockchain-dns, 2019.

[16] Ivan Bogatyy. Implementing Ethereum trading front-runs on the Bancor exchange in Python, 2017.

[17] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE, 2015.

[18] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private Data Objects: An Overview. *arXiv preprint arXiv:1807.05686*, 2018.

[19] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain Validation++ for MitM-Resilient PKI. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076, 2018.

[20] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020.

[21] Vitalik Buterin. CRITICAL UPDATE Re: DAO Vulnerability, 2016.

[22] CAIDA. DNS Names for IPv4 Routed Topology Dataset, 2017.

[23] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy*, pages 142–157. IEEE, 2019.

[25] Jiachi Chen, Xin Xia, David Lo, and John Grundy. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. *ACM Transactions on Software Engineering and Methodology*, 31(2):1–37, 2021.

[26] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering*, 2020.

[27] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering*, 2021.

[28] Jing Chen, Shixiong Yao, Quan Yuan, Kun He, Shouling Ji, and Ruiying Du. CertChain: Public and Efficient Certificate Audit Based on Blockchain for TLS Connections. In *IEEE Conference on Computer Communications*, pages 2060–2068. IEEE, 2018.

[29] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. SODA: A Generic Online Detection Framework for Smart Contracts. In *Network and Distributed Systems Security*. The Internet Society, 2020.

[30] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *2019 IEEE European Symposium on Security and Privacy*, pages 185–200, 2019.

[31] Taejoong Chung, Roland van Rijswijk-Deij, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. Understanding the Role of Registrars in DNSSEC Deployment. In *Proceedings of the 2017 Internet Measurement Conference*, pages 369–383, 2017.

[32] Lucian Constantin. DNS poisoning attack against major brazilian isp, 2009.

[33] David Dagon, Manos Antonakakis, Paul Vixie, Tatuya Jinmei, and Wenke Lee. Increased DNS Forgery Resistance Through 0x20-Bit Encoding. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 211–222, 2008.

[34] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. FastKitten: Practical Smart Contracts on Bitcoin. In *28th USENIX Security Symposium*, pages 801–818, 2019.

[35] Kun Du, Hao Yang, Zhou Li, Haixin Duan, and Kehuan Zhang. The ever-changing labyrinth: A large-scale analysis of wildcard dns powered blackhat seo. In *25th USENIX Security Symposium*, pages 245–262, 2016.

[36] Lukasz Dykcik, Laurent Chuat, Pawel Szalachowski, and Adrian Perrig. BlockPKI: An Automated, Resilient, and Transparent Public-Key Infrastructure. In *2018 IEEE International Conference on Data Mining Workshops*, pages 105–114. IEEE, 2018.

[37] Peter Eckersley. Sovereign Keys: A Proposal to Make HTTPS and Email More Secure, 2012.

[38] Joshua Ellul and Gordon J Pace. Runtime Verification of Ethereum Smart Contracts. In *2018 14th European Dependable Computing Conference*, pages 158–163. IEEE, 2018.

[39] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469, IETF, April 2015.

[40] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15. IEEE, 2019.

[41] Josselin Feist, Gustavo Grieco, and Alex Groce. sGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. In *2021 IEEE Symposium on Security and Privacy*. IEEE, 2021.

[42] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. Ægis: Shielding Vulnerable Smart Contracts Against Attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 584–597, 2020.

[43] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium*, pages 2757–2774, 2020.

[44] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. Certcoin: A Namecoin based Decentralized Authentication System. Technical report, MIT, 2014.

[45] Shang Gao, Zecheng Li, Zhe Peng, and Bin Xiao. Power Adjusting and Bribery Racing: Novel Mining Attacks in the Bitcoin System. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 833–850, 2019.

[46] Zhaoyu Gao and Arun Venkataramani. Measuring Update Performance and Consistency Anomalies in Managed DNS Services. In *IEEE Conference on Computer Communications*, pages 2206–2214. IEEE, 2019.

[47] Sharon Goldberg, Moni Naor, Dimitrios Papadopoulos, Leonid Reyzin, Sachin Vasant, and Asaf Ziv. NSEC5: Provably Preventing DNSSEC Zone Enumeration. In *Network and Distributed Systems Security*, pages 1–15, 2015.

[48] DAN GOODIN. Google warns of unauthorized TLS certificates trusted by almost all OSes, 2013.

[49] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017.

[50] Intel. Intel® Software Guard Extensions, 2020.

[51] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy*, pages 1695–1712. IEEE, 2020.

[52] Harry A Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design. In *WEIS*, pages 1–21, 2015.

[53] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *Network and Distributed Systems Security*, pages 1–12, 2018.

[54] Dan Kaminsky. Black ops 2008: It's the end of the cache as we know it. Technical report, Black Hat USA, 2008.

[55] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers. In *Network and Distributed Systems Security*, pages 1–15, 2019.

[56] Katatsuki. Re: Hi! My name is Rubixi. I'm a new Ethereum Doubler. Now my new home - Rubixi.tk, 2016.

[57] Sean Michael Kerner. Google Hit Again by Unauthorized SSL/TLS Certificates, 2015.

[58] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 679–690, 2013.

[59] Amit Klein and Benny Pinkas. DNS Cache-Based User Tracking. In *Network and Distributed Systems Security*, pages 1–15, 2019.

[60] Amit Klein, Haya Shulman, and Michael Waidner. Internet-Wide Study of DNS Cache Injections. In *IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[61] Atsushi Koshiba, Ying Yan, Zhongxin Guo, Mitaro Namiki, and Lidong Zhou. TEE-KV: Secure Immutable Key-Value Store for Trusted Execution Environments. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 535–535, 2018.

[62] Johannes Krupp and Christian Rossow. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium*, pages 1317–1333, 2018.

[63] Murat Yasin Kubilay, Mehmet Sabir Kiraz, and Hacı Ali Mantar. CertLedger: A new PKI model with Certificate Transparency based on blockchain. *Computers & Security*, 85:333–352, 2019.

[64] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, IETF, June 2013.

[65] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[66] Dave Lewis. The ddos attack against dyn one year later, 2017.

[67] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020.

[68] Zecheng Li, Yu Zhou, Songtao Guo, and Bin Xiao. SolSaviour: A Defending Framework for Deployed Defective Smart Contracts. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 748–760, 2021.

[69] Wilson Lian, Eric Rescorla, Hovav Shacham, and Stefan Savage. Measuring the Practical Impact of DNSSEC Deployment. In *22nd USENIX Security Symposium*, pages 573–588, 2013.

[70] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 63–79, 2019.

[71] Haifeng Liu, Shugang Chen, Yongcheng Bao, Wanli Yang, Yuan Chen, Wei Ding, and Huasong Shan. A High Performance, Scalable DNS Service for Very Large Scale Container Cloud Platforms. In *Proceedings of the 19th International Middleware Conference Industry*, pages 39–45, 2018.

[72] Wei Liu, Hiroki Nishiyama, Nirwan Ansari, Jie Yang, and Nei Kato. Cluster-Based Certificate Revocation with Vindication Capability for Mobile Ad Hoc Networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):239–249, 2012.

[73] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, 2016.

[74] Moxie Marlinspike. Trust assertions for certificate keys, 2013.

[75] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: Bitcoin Lightweight Client Privacy using Trusted Execution. In *28th USENIX Security Symposium*, pages 783–800, 2019.

[76] Stephanos Matsumoto and Raphael M Reischuk. IKP: Turning a PKI Around with Decentralized Automated Incentives. In *2017 IEEE Symposium on Security and Privacy*, pages 410–426. IEEE, 2017.

[77] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of Luck: An Efficient Blockchain Consensus Protocol. In *proceedings of the 1st Workshop on System Software for Trusted Execution*, pages 1–6, 2016.

[78] Brianna Montgomery. Fei Bonding Curve Bug Post Mortem, 2021.

[79] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 1186–1189. IEEE, 2019.

[80] Giovane CM Moura, Ricardo de O Schmidt, John Heidemann, Wouter B de Vries, Moritz Muller, Lan Wei, and Cristian Hesselman. Anycast vs. DDoS: Evaluating the November 2015 Root DNS Event. In *Proceedings of the 2016 Internet Measurement Conference*, pages 255–270, 2016.

[81] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, Bitcoin, 2008.

[82] Namecoin. Namecoin, 2014.

[83] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[84] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663, 2018.

[85] Eric Osterweil, Dan Massey, Danny McPherson, and Lixia Zhang. Verifying Keys through Publicity and Communities of Trust: Quantifying Off-Axis Corroboration. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):283–291, 2013.

[86] Zhenyu Pan, Tianyuan Hu, Chen Qian, and Bixin Li. ReDefender: A Tool for Detecting Reentrancy Vulnerabilities in Smart Contracts Effectively. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security*, pages 915–925. IEEE, 2021.

[87] Vasileios Pappas, Dan Massey, and Lixia Zhang. Enhancing DNS Resilience against Denial of Service Attacks. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 450–459. IEEE, 2007.

[88] KyoungSoo Park, Vivek S Pai, Larry L Peterson, and Zhe Wang. CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 199–214, 2004.

[89] Rafael Pass, Elaine Shi, and Florian Tramer. Formal Abstractions for Attested Execution Secure Processors. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 260–289. Springer, 2017.

[90] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis, 2020.

[91] Roberto Perdisci, Manos Antonakakis, Xiapu Luo, and Wenke Lee. WSEC DNS: Protecting Recursive DNS Resolvers from Poisoning Attacks. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 3–12. IEEE, 2009.

[92] Bo Qin, Jikun Huang, Qin Wang, Xizhao Luo, Bin Liang, and Wenchang Shi. Cecoin: A decentralized PKI mitigating MitM attacks. *Future Generation Computer Systems*, 107:805–815, 2020.

[93] Haseeb Qureshi. A hacker stole $31M of Ether — how it happened, and what it means for Ethereum, 2017.

[94] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Network and Distributed Systems Security*, pages 1–15, 2018.

[95] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts. In *30th USENIX Security Symposium*, pages 1289–1306, 2021.

[96] Mark Dermot Ryan. Enhanced Certificate Transparency and End-to-End Encrypted Mail. In *Network and Distributed Systems Security*, pages 1–14, 2014.

[97] samczsun. The Dangers of Surprising Code, 2021.

[98] samczsun. Two Rights Might Make A Wrong, 2021.

[99] Haya Shulman and Michael Waidner. One Key to Sign Them All Considered Vulnerable: Evaluation of DNSSEC in the Internet. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 131–144, 2017.

[100] Stephan Somogyi. Google security blog: Improved digital certificate security, 2015.

[101] Sooel Son and Vitaly Shmatikov. The Hitchhiker's Guide to DNS Cache Poisoning. In *International Conference on Security and Privacy in Communication Systems*, pages 466–483. Springer, 2010.

[102] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, 2017.

[103] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *2016 IEEE Symposium on Security and Privacy*, pages 526–545. IEEE, 2016.

[104] Pawel Szalachowski. SmartCert: Redesigning Digital Certificates with Smart Contracts. *arXiv preprint arXiv:2003.13259*, 2020.

[105] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. PoliCert: Secure and Flexible TLS Certificate Management. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 406–417, 2014.

[106] KoET Team. King of Ether Throne Post-Mortem Investigation, 2016.

[107] Alin Tomescu and Srinivas Devadas. Catena: Efficient Non-equivocation via Bitcoin. In *2017 IEEE Symposium on Security and Privacy*, pages 393–409. IEEE, 2017.

[108] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.

[109] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. Obscuro: A Bitcoin Mixer Using Trusted Execution Environments. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 692–701, 2018.

[110] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[111] Florian Tschorsch and Björn Scheuermann. Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016.

[112] u/ethererik. GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas, 2016.

[113] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, pages 991–1008, 2018.

[114] Roland van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. DNSSEC and its Potential for DDoS attacks: A Comprehensive Measurement Study. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 449–460, 2014.

[115] Wikipedia. DigiNotar Unauthorized Certificates, 2011.

[116] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Čapkun. ZLiTE: Lightweight Clients for Shielded Zcash Transactions using Trusted Execution. In *International Conference on Financial Cryptography and Data Security*, pages 179–198. Springer, 2019.

[117] Alexander Yakubov, Wazen Shbair, Anders Wallbom, David Sanda, et al. A Blockchain-based PKI Management Framework. In *The First IEEE/IFIP International Workshop on Managing and Managed by Blockchain (Man2Block) colocated with IEEE/IFIP NOMS 2018*, 2018.

[118] Shixiong Yao, Jing Chen, Kun He, Ruiying Du, Tianqing Zhu, and Xin Chen. PBCert: Privacy-Preserving Blockchain-Based Certificate Status Validation Toward Mass Storage Management. *IEEE Access*, 7:6117–6128, 2018.

[119] Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: a new formalized PKI with verifiable trusted parties. *The Computer Journal*, 59(11):1695–1713, 2016.

[120] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart Contract Repair. *ACM Transactions on Software Engineering and Methodology*, 29(4):1–32, 2020.

[121] 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*, pages 270–282, 2016.

[122] Fan Zhang, Philip Daian, Iddo Bentov, Ian Miers, and Ari Juels. Paralysis Proofs: Secure Dynamic Access Structures for Cryptocurrency Custody and

More. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 1–15, 2019.

[123] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert Van Renesse. Rem: Resource-efficient mining for blockchains. In *26th USENIX Security Symposium*, pages 1427–1444, 2017.

[124] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. SMARTSHIELD: Automatic Smart Contract Protection Made Easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, pages 23–34. IEEE, 2020.

[125] Liang Zhu, Zi Hu, John Heidemann, Duane Wessels, Allison Mankin, and Nikita Somaiya. Connection-Oriented DNS to Improve Privacy and Security. In *2015 IEEE Symposium on Security and Privacy*, pages 171–186. IEEE, 2015.