TeSC: TLS/SSL-Certificate Endorsed Smart Contracts

blinded

***Abstract*—Although nearly all information regarding smart contract addresses is shared via websites, emails, or other forms of digital communication, blockchains and distributed ledger technologies are unable to establish secure bindings between websites and the corresponding smart contracts. A user cannot differentiate between a website link to a legitimate smart contract set up by a reputable business owner and that to an illicit contract aiming to defraud the user. Surprisingly, current attempts to resolve this issue are based mostly on information redundancy, e.g., displaying contract addresses multiple times in varying forms of images and text. These verification processes are burdensome because the user is responsible for verifying the accuracy of an address. More importantly, these measures do not address the core problem because the contract itself does not contain information on its authenticity. To resolve such limitations and to increase security, we propose a solution that leverages publicly issued Transport Layer Security (TLS)/Secure Sockets Layer (SSL) certificates of Fully-Qualified Domain Names (FQDN) to ensure the authenticity of smart contracts and their owners. Our approach combines on-chain endorsement storage that utilizes signatures from the respective certificate and off-chain authentication of the smart contract. The system is open and transparent because the only requirement for usage is ownership of a TLS/SSL certificate. Further, moderate deployment and maintenance costs, a widely accepted public key infrastructure, and a simple interface enable TLS/SSL endorsed smart contracts (TeSC) to bridge the gap between websites and smart contracts.**

***Index Terms*—blockchain, authentication, smart contracts, Ethereum, certificates**

I. Introduction

Users in publicly available blockchain-based systems can encounter hazardous and hostile conditions because hackers have strong incentives to defraud the user [1]. Although the underlying ledger provides immutable smart contracts, decentralized key management, and robust consensus mechanisms, malicious activities like fraudulent behavior, hacks, and identity theft continually occur [2]. Abundant research has been conducted to resolve adverse activity related to smart contract engineering, formal verification of contracts, and the design of secure programming languages [3]–[5]. However, an often ignored issue is the weak link to the information stored on blockchains. In particular, the user needs to know the exact location of the smart contract or the externally owned account in question. Because these addresses consist of characters, e.g., 64 random hexadecimal characters in Ethereum [6], users copy and paste the address information obtained online. Even in decentralized applications, the user can view only the hexadecimal address in question. Consequentially, the called address or smart contract contains no further information for verifying whether the user is interacting with the intended contract or not.

The approach of sharing addresses on web pages with no further connection between the business and the smart contract is prone to error and can be exploited by attackers [7]. We refer to this attack as an address replacement attack. The CoinDash Initial Coin Offering (ICO) is an example of such an attack [8]. At the height of token sales in 2017, CoinDash conducted an ICO to sell its newly created token to interested investors in return for Ether, the currency of the Ethereum blockchain. To advertise the token sale, the CoinDash founders set up a website that contained relevant information on investing in the company including the smart contract's address. Many people participated in the sale, with several millions of US dollars transferred to the contract during the first few hours. However, the contract was not the intended token smart contract; a different smart contract was set up by malicious actors, who collected all of the investment funds. These malicious actors had previously hacked the website and replaced the original contract address with their own, resulting in the loss of more than US$7 million. Even though WordPress content management system was compromised, the underlying web server remained intact. Since then, several other projects have been affected by broken website access control or vulnerabilities in web applications [7].

To resolve the issue of loose coupling between a website and a smart contract and to prevent address replacement attacks, we propose the use of Ethereum TLS/SSL endorsed smart contracts (TeSC). TeSC enables the endorsement of smart contracts with a private/public key pair of a previously existing TLS/SSL certificate issued by a certificate authority (CA). This approach allows wallets or other software that directly interact with smart contracts to verify their authenticity and binding to the respective domain. This technique reduces the risk of address replacement attacks by removing web applications from the threat model and enables the detection and prevention of maliciously issued TLS endorsed smart contracts (TeSC).

TeSC contains four crucial components: the endorsement, an authenticated smart contract storing the endorsement, an off-chain verifier, which authenticates the smart contract, and the TeSC registry, which prevents downgrade-attacks and offers a discovery service.

Our approach considers privacy-preserving authentication of the smart contracts. Entities can endorse smart contracts without the threat of their endorsement being retrieved by third parties crawling the respective blockchain. In addition, the system enables the discovery of contracts supporting TeSC.

Specifically, if the user journey begins at the smart contract itself rather than at the website, the user is still able to authenticate the smart contract.

In the remainder of this paper, Section II introduces the system architecture and the methodologies for asserting and authenticating smart contracts. The security of the system is discussed in Section III, and our approach is evaluated in Section IV. An overview of related work is provided in Section V and the conclusions of this study and a brief discussion of potential future work are given in Section VI.

II. System Architecture

Our system consists of the four components listed below:

- The **Endorsement** (Section II-A) holds information about the approval of a smart contract by the domain owner signed with the TLS/SSL certificate's private key. This information contains relevant data, such as the smart contract address and further information such as the expiration date or additional flags (Section II-B).
- The **On-Chain TLS Endorsed Smart Contract** (Section II-C) contains methods for storing and updating the endorsement. In addition, we provide reference implementation [3] which enables independent usage from the actual smart contract (e.g., token contract or other purpose).
- **Off-Chain Verifier** (Section II-D) is an application that runs outside of the blockchain. It is responsible for verifying endorsements stored on-chain and retrieves data from the certificate issuer, website owner, smart contract stored in the blockchain, and potential certificate authorities. This information enables authentication of on-chain information in the context of the user-defined environment.
- **TeSC Registry** (Section II-E) prevents downgrade attacks by providing a list of smart contracts for domains and enables easy discovery of smart contracts that implement this library. In addition, this component manages a list of existing smart contracts and their respective domains to be used for verification purposes.

We introduce supplementary design decisions for TeSC in Section II-F, explain revocation in Section II-G and discuss TLS Key Management in Section II-H, accounting for key rotation and multiple certificates of one domain and endorsing pre-existing smart contracts.

Figure 1 depicts the overall structure of the proposed architecture. The reference interface is given in Appendix A and its implementation follows [3], using Solidity for Ethereum. Since TeSC is blockchain-independent, all networks can adopt the underlying principle.

A. Endorsement

We first define three key components of a smart contract: the claim, signature, and endorsement.

First, we define the claim C in Eq. 1:

$$C = \{addr, domain, date_{exp}, flags\} \tag{1}$$

The claim C contains the address $addr$ of the endorsed smart contract, the FQDN $domain$, the expiration date $date_{exp}$, and further flags defined in $flags$. We discuss relevant flags in Section II-B. An FQDN $domain$ is provided as-is, and protocol information is omitted, e.g., *hq.example.org*. Furthermore, the expiration date $date_{exp}$ is defined as Unix epoch time.

We define a valid signature S in Eq. 2:

$$S = \{sign(hash(C), cert_{privKey})\} \tag{2}$$

The signature S contains the claim hashed with a cryptographic hash function signed with the private key $cert_{privKey}$ of the TLS certificate of the respective domain.

We further define an endorsement E in Eq. 3:

$$E = \{S, C, [cert_{fingerprint}]\} \tag{3}$$

The endorsement E contains the previously defined signature S, the claim C required to validate the signature, and the optional fingerprint of the signing certificate $cert_{fingerprint}$. We include the fingerprint of the certificate to facilitate its retrieval if the certificate is not present at the web server, as described subsequently in Section II-H2.

B. Endorsement Flags

We further introduce flags which enable additional functionality or restrictions in handling the endorsement. In this section, we provide a brief insight into potential flags and reasoning. In Appendix B, we display a list of all available flags.

- **ALLOW_SUBENDORSEMENT**: This flag can be set if a smart contract should be able to endorse additional addresses such as contracts or externally owned accounts.
- **EXCLUSIVE**: This flag can be set if the smart contract is the only contract endorsed by the respective domain. If other valid smart contract endorsements exist, the verification of the smart contract should be aborted.

C. On-Chain TLS Endorsed Smart Contract

A domain owner claims the ownership of a smart contract by providing an endorsement in the respective contract. Information required to authenticate the smart contract is retrieved from external data sources. The interface necessary for a smart contract to adhere to TeSC is depicted in Appendix A.

Three steps are required to enable authentication of the smart contract: smart contract creation, endorsement generation, and endorsement upload. For simplicity, we omitted the intermediate or supporting activities such as creating wallet addresses, funding the accounts, and including the library in the respective smart contract.

Initially, only newly created Smart Contracts which adhere to the interface standard can be endorsed. To allow the endorsement of previously existing smart contracts, we describe sub-endorsements as a potential solution In section II-H. The to-be endorsed smart contract is supplied with the respective endorsement data. To create the endorsement, the owner of the domain retrieves the unique contract address and signs

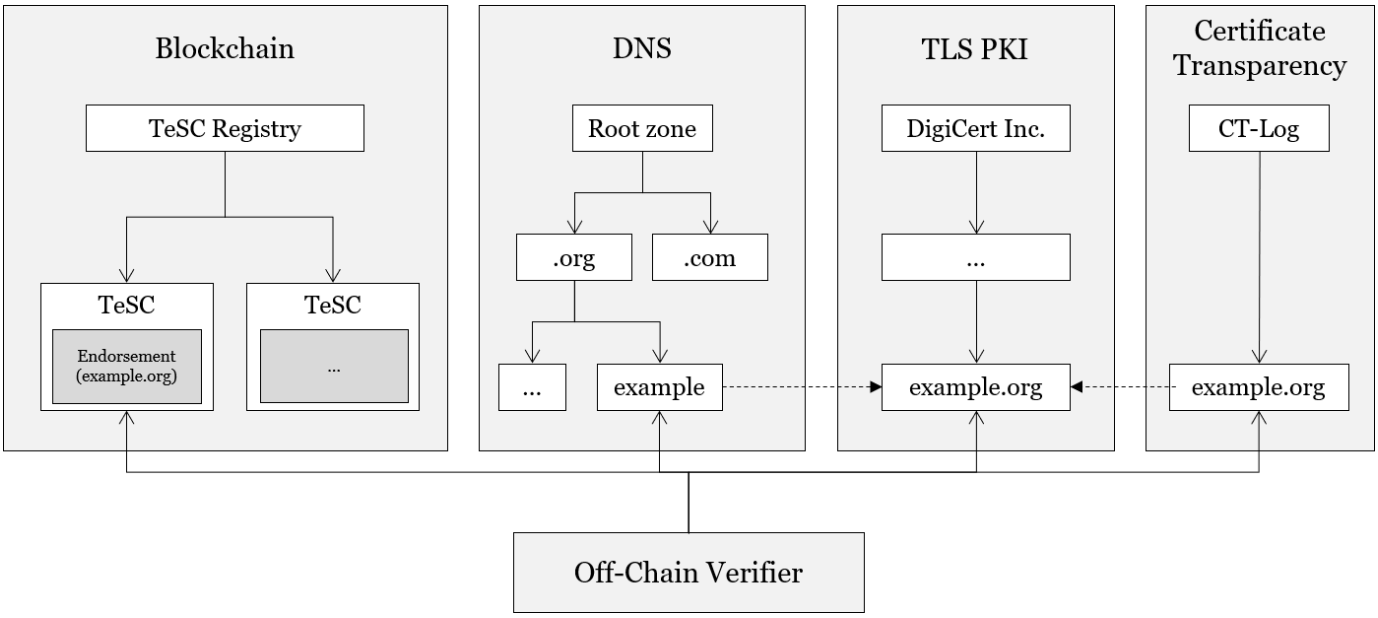

Fig. 1. High-Level Structure. The system fully relies on the TLS PKI and partly relies on the DNS and Certificate Transparency to verify TLS endorsed Smart Contracts.

a respective claim consisting of the smart contract address, domain, expiration, and potential flags with its private key of the TLS/SSL certificate. Because the endorsement contains only one specific smart contract address, other smart contracts cannot use this information for fraudulent endorsement. The endorsement is then added to the smart contract via a regular method call.

D. Off-Chain Verifier

The verification of the smart contract endorsement occurs on the client-side. The software itself (e.g., a wallet in a browser) needs to access the following data sources:

- the address of the relevant smart contract, which is usually obtained via the internet; alternatively it is obtained by the TeSC registry or proprietary discovery services;
- the endorsement of this smart contract, e.g., domain name, signature, and expiration date;
- the signed certificate, its public key alongside the information of the certificate authority, which is obtained by requesting the domain and retrieving the TLS/SSL certificate[1]; and
- the list of trusted certificate authorities of the user.

The list of trusted certificate authorities is defined by the user, either by directly providing a list or by reviewing the CA list stored in the user's computer or browser.

Smart contract authentication involves four steps. Again, we omit intermediate steps and assume that the user journey starts on the website, such that the domain is known. Also, we assume that the contract address is known.

1) **Smart Contract Endorsement Retrieval:** The application first retrieves the endorsement for later authentication. It contains information about the claim and the signature data of the smart contract.
2) **Certificate Retrieval:** Afterward, it connects to the respective domain given in the smart contract (or that previously known by the website) and obtains the certificate. If the optional $cert_{fingerprint}$ is set, the certificate is directly retrieved from Certificate Transparency.
3) **Smart Contract Endorsement Verification:** The software validates whether the private key of the certificate signed the smart contract claim and checks any additional properties of the claim, such as those listed below:
   a) It first verifies that the address stored in the claim is identical to the smart contract address. Otherwise, the process is aborted.
   b) The domain in the claim has to be identical to the domain from which the information was obtained. If not, the process is aborted[2].
   c) The smart contract is endorsed only until the $date_{exp}$. If $date_{exp} < date_{today}$, then the smart contract is expired and should not be trusted.
   d) The flags described in Section II-B can modify the off-chain verifier's behavior and need to be considered.
4) **Certificate trust:** After successfully verifying the endorsement, the software checks whether a trusted CA has (indirectly) signed or issued the certificate found

[1]As we discuss in Section II-H2, the certificate can be obtained by other methods, e.g., Certificate Transparency.

[2]The explicit statement of the domain is important because certificates exist that are valid for multiple domains or contain domain wildcards. If the domain would not be stored within the claim, the owner of the smart contract could decide on its own to which domain it belongs.

in the smart contract and the web server. If a trusted CA signs the certificate and its public key, the identity is successfully validated. Otherwise, the program aborts with an error. It executes the certification path validation algorithm as defined in *RFC 5280* [9]. To further verify the authentiticy of the certificate, the verifier requires a proof for inclusion of the certificate into Certificate Transparency in form of a *signed certificate timestamp* (SCT) [10].

E. TeSC Registry

For the client-side verification it is relevant to know 1) whether a contract exists for a given domain and 2) under which address it is deployed. To find such contracts or check its existence, we propose extending this architecture by introducing a smart contract registry that enables the user to find all domains registered on the blockchain. This registry lists all contracts that adhere to this interface standard and claim to be the identity contract for a specific domain. We allow multiple endorsed smart contracts for the same domain because one domain can endorse multiple smart contracts. Even though it is possible to search the complete blockchain for such contracts [11], it is easier and faster to use an on-chain smart contract.

The registry smart contract is in place to map domains to smart contract addresses by storing these relationships. The usage of such a contract involves the following steps.

1) **Insertion of Contract Information:** All parties in control of endorsed smart contracts submit information about their contract and optionally the respective domain to the registry smart contract.
2) **Domain Lookup:** A user searching for a domain queries the registry smart contract and asks for all contracts assigned to that specific domain. The contract returns all relevant contract addresses.
3) **Contract authentication:** The client can execute the previously described authentication method for each of the returned contracts, which ensures that the correct smart contract is found, if it exists.

This registry smart contract relies on owners submitting their endorsed contracts. Thus, the design for registry should strive for two properties: every endorsed contract is added, and the amount of incorrect data is minimized. First, the entities creating contracts have a strong incentive to be found because it enhances the security by preventing downgrade attacks, and invites users and other parties to interact with their smart contracts. The registry smart contract enables owners to advertise their service and helps the users to find them. Second, malicious entities that do not own the respective TLS certificate should be discouraged from linking irrelevant contracts. Spaming the registry smart contract can cost a lot of money; further, spaming contracts will hardly impact the verification process as modern computation power is sufficient for processing hundreds of such contracts in milliseconds. Given this (dis)incentive structure, we have no restrictions in place for adding data to the registry.

F. Endorsement of pre-existing Smart Contracts

It is not possible to update previously existing smart contracts to adhere to the TeSC interface. To circumvent this limitation, we decide to allow endorsed smart contracts to subendorse other accounts, irrespective of being a smart contract or externally owned account. These subendorsements are stored in the Smart Contract, alongside the endorsement. The owner of the TLS certificate needs to consent to the subendorsement functionality by setting the **ALLOW_SUBENDORSEMENT**-flag (see Section II-B and Appendix B) in the endorsement.

The verification of directly endorsed Smart Contracts differs from the verification of subendorsements. The subendorsed smart contract does not contain any information about its endorsement. Therefore, the verifier first needs to find and verify potential endorsing smart contracts. To obtain these parent smart contracts, the verifier retrieves all smart contracts that belong to one domain from the TeSC registry, verifies them, and examine if the subendorsed Smart Contract is stored in the `subendorsement` array.

The downsides of subendorsements are that a discovery service like TeSC registry is mandatory. Additionally, the smart contract cannot be authenticated if the user journey directly starts at the subendorsed smart contract and no domain information is available. In direct endorsed smart contracts it is possible to obtain the domain information and authenticate the smart contract by retrieving certificate information from the domain or, if a certificate fingerprint is available, directly from Certificate Transparency.

G. Revocation

There are several reasons to revoke an endorsed smart contract. We describe two scenarios in which a revocation becomes necessary. First, the entity wanting to reskind the endorsement might do so voluntarily, e.g., as the smart contract is no longer in use. Second, the entity might need to involuntarily revoke the endorsement, e.g., as it has lost access to the smart contract or the respective private keys controlling the smart contract.

In the first case, if the owner of the domain still has access to the smart contract or the respective account that administers the smart contract, the endorsement can just be removed by adding an invalid or empty endorsement. The off-chain verifier only accesses the latest information added to the smart contract, therefore, the verification of the endorsement will fail and users should be discouraged from interacting with the Smart Contract.

In the second case, an entity does not have access anymore to the smart contract with the endorsement. To invalidate the contract, the respective TLS certificate has to be revoked. If the certificates are no longer valid, the validation will fail for the respective smart contract. As it is not possible to publish a contradicting statement with the same certificate in the smart contract, the certificate has to be revoked and a new one has to be used.

H. TLS Key Management

Private keys used in TLS certificates have characteristics that need to be kept in mind when designing a system like TeSC. First, we need to address the expiry and rotation of key material. Second, we need to care for revoked TLS certificates. Last, we need to account for multiple existing TLS-certificates, thus multiple existing keys for one domain.

1) Key Expiry and Rotation: The key material in the X.509 certificates expires regularly. The *Certificate Authority/Browser Forum* has decided that TLS certificates are valid for a maximum time of 825 days[3]. Therefore, the public key and signature information must be regularly updated; otherwise, the verifying entity cannot assert the validity of the endorsed contract. To update the smart contract, a new endorsement has to be created with the respective certificate and inserted in the contract. This procedure ensures that smart contract endorsement remains valid.

This procedure bears the risk, that users get error messages for a brief time in which the smart contract endorsement does not match the certificate provided by the webserver. This can be the case, when the endorsement is updated before the TLS certificate or vice versa. In case the smart contract experience high demand, such an error message, even when only displayed for a short period of time, might not be acceptable to some parties. In this case, the usage of the certificate fingerprint variable is highly recommended. In that case, the verifying entity will look up the respective certificate in Certificate Transparency if it is not presented by the web server. With that, the certificate and endorsement can be updated independently from each other without short "downtimes" in the verification process.

2) Certificate Key Revocation: Key revocation becomes mandatory if the key material of a certificate is compromised. Such key material could be abused for creating additional endorsed smart contracts, potentially tricking users into believing that the new smart contract does indeed belong to the compromised entity. However, because our approach relies on the web server's public TLS key, revocation of the respective TLS certificate also renders potentially fraudulent smart contracts invalid. The software is able to evaluate whether the certificates not yet verified are included in certificate revocation lists (CRL) or if requests made through Online Certificate Status Protocol (OCSP) are valid.

Multiple Certificates for a Single Domain: Having multiple valid certificates assigned to one domain at the same time is common. Especially larger enterprises have many certificates, e.g., *facebook.com* has 521 valid certificates assigned to it at the time of this writing[4]. For this reason, we include the fingerprint $cert_{fingerprint}$ as a unique identifier of the respective certificate within the endorsement. This fingerprint enables us to retrieve the certificate that signed the endorsement via Certificate Transparency. Browsers such as Google Chrome require a certificate to be included in Certificate Transparency in order to be accepted. The retrieval from Certificate Transparency allows us to verify the endorsement independent of the web server's current certificate. This also allows the owner of the Smart Contract to update the server's certificate or the smart contract's endorsement with no concern of the smart contract suddenly becoming unverified.

[3]https://cabforum.org/2017/03/17/ballot-193-825-day-certificate-lifetimes/, accessed 03rd May 2021.

[4]https://crt.sh/?Identity=*.facebook.com&exclude=expired&match==, accessed 07th January 2021.

III. Security Analysis

In this section, we analyze the security and discuss risks regarding the usage of TeSC. Namely, we cover risks associated with TLS as a base protocol in section III-A, examine cross-protocol attack vectors in section III-B and discuss other attacks on TeSCin section III-C and III-D.

A. TLS as a Base Protocol

TLS/SSL certificates are often deemed unreliable and prone to fraud. Owing to the centralized nature of the system with multiple certificate authorities, potential points of failure exist: Certificate authorities might maliciously issue certificates for domains, revoke valid certificates without reason, or leak private keys, resulting in possible eavesdropping or man-in-the-middle attacks.

Nonetheless, this public key infrastructure (PKI) is a crucial part of the internet. Without proper security mechanisms to reliably identify, connect, and communicate with a remote server, most current business and communication would not be possible. In particular, because these certificates have become omnipresent on most websites, other technologies (with potentially better security) lack the adoption for broad usage (see also section V). The security model of websites assumes and relies on the proper functioning of this PKI. Thus, every system attempting to establish a secure connection between websites and smart contracts (which is the aim of this study) implicitly relies on this system. In TeSC, we make this trust assumption explicit, remove vulnerable intermediaries (such as the web application), and directly depend on this PKI by using private keys in these certificates. It is also important to mention that the PKI is under active enhancement by systems such as Certificate Transparency [10].

B. TeSC-TLS Cross-Protocol Attack Vectors

It is of high importance that one protocol does not endanger the functionality or security of the other. As TeSCdepends on TLS, there are two potential ways how cross-protocol attacks can occur: 1) An attacker abusing existing systems such as the web server or web application to attack TeSC and 2) using the signature created for the specific endorsed contract to launch attacks on existing systems, e.g., the web server or users. The first attack scenario requires that a process or communication protocol unintentionally generates a valid endorsement (defined by the attacker) for a smart contract. The second scenario discusses if the signature in the endorsement can be used to allow an attacker to impersonate the server.

In scenario 1) it is beneficial for an attacker that the TLS certificate is frequently used while communicating with the

server. Potentially, every new request to the server leads to a response signed with the TLS certificate. To better understand the attack scenario, we have to look at the TLS handshake protocol, as outlined in RFC8446 for TLS 1.3 [12]. The `CertificateVerify` message *"is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate"* [12]. It contains the used signature scheme and the signature that signs the hash of the data transferred previously to the handshake. From there on, two potential attack vectors exist: Either the hash function is not collision-resistant, or the same data is hashed. The first case can be discarded, as both TLS 1.3 and TeSC use cryptographic hash functions that are collision-resistant. For the second case, the attacker would need to define the contents that are hashed. As the server generates 32 random bytes in the message `SERVER HELLO` with a secure random number generator, it can be assumed that it is not possible to generate a valid endorsement from the information of a TLS handshake.

In scenario 2), the identical mechanisms apply. As a difference, endorsements have a longer timespan and are not created as often as TLS handshakes. As the data signed in the endorsement creation process in no way resembles messages exchanged between client and server in a TLS handshake, they cannot be reused in the TLS handshake protocol. Therefore, the signature stored within the endorsed smart contract does not open up attack vectors on the TLS protocol or the communication with the server.

C. Downgrade Attacks

In a downgrade attack, an adversary tricks a party in a communication protocol to assume that the other party is incapable of adhering to newer (and more secure) versions of the communication protocol, which leads to the usage of an older and less secure version of the protocol. Because older versions are susceptible to further attacks or have no protection at all (e.g., plain text), an adversary can further exploit the communication. The same applies to TeSC. Considering the base case and our introductory example of CoinDash, it becomes apparent that the user needs to know that a website uses TeSC to protect the users from sending transactions to malicious contracts. However, we cannot expect a user to know whether the counter-party actually implements TeSC, and in the case of CoinDash, an address swap is still possible without the user noticing.

To account for downgrade attacks, the off-chain verifier (or other software implementing the verification mechanism) needs to know whether a contract exists for a given domain. If the verifier obtains a contract address from a website, it checks if this contract supports the TeSC interface. If this is not the case, it checks if another valid smart contract exists for that domain. If such a contract is encountered without the current smart contract adhering to the interface standard, a warning is emitted to the user stating that the current contract has no additional protection against impersonation and that other contracts with such protection exist.

The TeSC registry is asked about previously existing smart contracts for a domain. If the original and endorsed smart contract is registered within this registry, downgrade attacks are prevented.

D. TLS Private Key Compromise

In the case of a downgrade attack (and similarly to the CoinDash incident), the attacker is assumed to have access to the contents of the webpage and can manipulate them, such as replacing an address. However, this might not always be the case. Attackers might have higher privileges on the victim's machine, allowing them to obtain the TLS certificate's private keys or the access to the file system allows them to create new certificates by using Automatic Certificate Management Environment (ACME) [13] or Certificate Signing Requests (CSR) [14][5]. In such cases, the attackers are able to create new smart contracts with valid endorsements, allowing them to trick users into believing that they are interacting with a legit smart contract.

TeSC is not able to prevent these attacks but allows *detecting* them. Similar to Certificate Transparency [10], TeSC enables owners of TLS endorsed smart contracts to detect whether their domain new smart contracts are issued. As all contracts are publicly available on the blockchain, the network can be monitored for newly created smart contracts that contain an endorsement for a specific domain. With this information, companies are able to implement monitoring services and act accordingly by reclaiming possessions of their servers and revoking any compromised key material as outlined in II-G. Further risk mitigation is placed in the hand of the legitimate owners of the website respectively the smart contract, as the `EXCLUSIVE`-flag (Section II-B) prevents the creation of further valid smart contracts.

IV. Evaluation

In this section, we first discuss potential usages of TeSC and further assess the costs of deploying TeSC on Ethereum Mainnet.

A. Usage Scenarios

Besides for allowing the verification of the relationship of a smart contract to a domain, we briefly introduce further usages for TeSC.

1) Consortia Member Identification: Companies and other entities often opt for permissioned settings concerning blockchain systems: Either they preselect nodes that are allowed to produce blocks in the network (public permissioned blockchains), or limit the accounts allowed to interact with a given smart contract architecture [15]. Often, these entities (smart contracts or externally owned accounts) are white-listed after an off-chain proof; usually, teams from different companies communicate their account information outside of the blockchain network. However, this process is expensive, as for every new connection, trust has to be established.

[5]These systems usually require a proof of control for the respective domain, e.g., placing files in specific paths

Proofs can be generated with TeSC that endorse accounts and smart contracts, allowing the "administrative" party to add new entities to the network after sufficient proof automatically. This not only reduces costs but allows for novel filtering, e.g., any domain with a TLD from ,e.g., Sweden is allowed to participate in the network. Proofs can be stored in public permissionless blockchains (e.g., the Ethereum Mainnet) and used in private blockchains, allowing an even further adoption of SSL/TLS-certificates within these networks.

2) Information attribution and authentication in public blockchains: The properties of immutability, transparency, and longevity of data and information are often used as arguments for blockchain technology, as many companies value such properties for their use cases, e.g., reduce fraud or increase transparency towards their users. Data stored on blockchains with the intention of transparency, however, lack the attribution or the connection to the company, such that a proof that data comes from a company or institution is bothersome, as prior communication is required as evidence. For these companies, it is much easier to authenticate this smart contract, store their data in it, and let other parties validate the stored data alongside authenticating the smart contract. To give an example, this approach could be used for allowing online-authentication of digital certificates, e.g., degrees or references. Hashes of such documents are stored on the blockchain, similar to other blockchain-based approaches [16]. The authenticated smart contract ensures the correctness of the identity of the issuer. Third parties (e.g., potential employers) can compare the hashes of received documents to the hashes contained in the smart contract. This software additionally authenticates the smart contract and validates the asserted identity. With such a solution in place, a software can automatically scan if uploaded documents by the applicant are valid or not.

B. Costs

To assess the system's overall costs, we execute a series of transactions, measure the respective gas usage, apply a realistic gas price and multiply it with the cost of Ethereum's cryptocurrency Ether. We find that the overall costs of our approach are moderate. Excluding the costs of requesting a certificate, a company (or an entity creating an endorsed contract) faces one primary cost driver: The **deployment** of the smart contract on the blockchain costs about 15 times more gas than the update of key material, the creation of subendorsements or the recording of the contract within the TeSC registry.

We deploy a TeSC-compliant contract [3] with disabled compiler optimization and a domain length of 9 characters and provide all endorsement information upfront. The contract creation transaction requires 1,550,980 gas, with included fingerprint 1,570,564 gas. Updating the signature alongside the expiration date costs 104,614 gas. Adding a subendorsement to the contract costs 67,105 gas, and adding the smart contract to the registry costs 123,661 gas. The end price is determined by the current network utilization and the market price for Ethereum. While Ether cost around 233 USD at the beginning of March 2020 with a recommended gas price of 1 - 8 GWei, the price rose to 412 USD (21 August 2020) with a recommended gas price between 30 and 150 GWei. In the meantime, the price rose to about 1,300 USD (20 January 2021), and gas price stabilized around 50 GWei. Table 1 provides detailed costs for all previously mentioned transactions in the respective time frame.

Overall, an investment of 120 USD is sufficient to set up a smart contract that includes TeSC. Maintaining the smart contract and updating endorsement information is much cheaper compared to the initial storage of data. The additional costs compared to a smart contract without TeSC are lower, as e.g., the gas costs for transactions and smart contract creation exist nonetheless. Comparable approaches, such as ENS [17] relying on DNSSEC (Section V-B), use a similar amount of gas (1,224,628)[6].

V. Related Work

This section outlines concepts that partly overlap with our research or try to solve similar issues. As our approach lies within multiple domains, we categorize related work in two distinct categories: 1) verification and validation of Ethereum addresses and 2) name services.

A. Verification of Ethereum addresses

We focus on the verification of Ethereum addresses, however, these approaches remain identical for addresses of other blockchains or public-key address schemes.

Address Checksums: In EIP55, an address scheme was introduced which capitalizes parts of the characters in the address to allow for an easy verifiable checksum [18]. It helps to prevent mistakes (e.g., left out characters) while copying the addresses. It is often used inside wallets (e.g., MetaMask). However, as an attacker can quickly generate such checksums or the tools used to create them automatically, the checksum does not help the user distinguish between valid and illicit smart contracts.

Vanity addresses: Vanity addresses are specially crafted addresses to enable a partly readability. For example, the author of Profanity, a tool for producing these vanity addresses, has given `0x`**`000dead000`**`ae1c8e8ac27103e4ff65f42a4e9203` as a donation address. The software computes private keys with a respective public key until an address has been found, which satisfies the predefined requirements [19]. Alternative approaches include the usage of account nonces (specific to Ethereum) to increase speed, however, the result remains identical. An attacker can also generate such an address with similar patterns, given she/he has the computing power.

Account address images: An approach used by some wallets is blockies. Blockies is a library that allows generating unique images of addresses, called identicons [20]. These images should help users to identify single addresses and better recognize if addresses have been swapped. However, also this

[6]Transaction *0xe3f845faf95ca3cafff155a8e0ca25f6396dba4afe899beaed7a3c31f78c3736*, available on https://etherscan.io/, accessed 7th January 2021.

TABLE I
COSTS OF SMART CONTRACT DEPLOYMENT AND INTERACTION

	Deploy Contract	Update Signature	Add Subendorsement	Add to Registry
Gas usage	1,550,980 gas	104,614 gas	67,105 gas	123,661 gas
03/2020 (8 GWei)	2.89 USD	0.20 USD	0.13 USD	0.23 USD
08/2020 (30 GWei)	19.17 USD	1.29 USD	0.83 USD	1.53 USD
01/2021 (50 GWei)	103.17 USD	6.97 USD	4.47 USD	8.24 USD

approach suffers from potential brute force to recreate almost identical images, as work by Austin Griffith shows [21].

Multitude of address displayed: Some popular tokens, such as Token-as-a-Service (TaaS), approached the problem by introducing redundancy by displaying the address in multiple settings (website, mail, and others) and formats (image, text, and others) [22]. This approach increases resilience against attacks, as attackers might not be able to change addresses in already sent emails. However, it shifts the responsibility to the user, who is responsible for the tiresome comparison and verification of the address. If the user does not compare these addresses, an attack could still be successful.

B. Name Services

Name services are systems that allow attributing human-readable names to complex and changing information, e.g., IP-addresses. This is a well-known approach and is also the basis for the currently used DNS [23], as it maps (among other parameters) between FQDN and IP-addresses.

Ethereum Name Service (ENS): ENS is a well-known service that allows the registration of domains for the Top-level domain (TLD) .eth [24]. It started in 2018, and currently, over 700,000 domain auctions have happened. Users bid for domains in a sealed auction, and after a predefined time, the ownership is transferred to the bidder with the highest offer. The funds are locked for the time the domain is registered. It can be refunded once the ownership is abandoned. The approach is promising and finds some users within the Ethereum community, however, the overall adoption compared to the TLS ecosystem is practically non-existent. There is no judiciary system that allows reassigning domain names, e.g., in cases of trademark issues. Likely, popular domains registered in ENS do not link to respective company accounts. Further, the "root zone" is managed by a multi-signature wallet owned by seven single individuals. The team behind ENS has expanded its efforts to map FQDNs of existing TLDs (e.g., .xyz) to ENS, allowing "real world" domains being used in a blockchain setting [25]. They rely on DNSSEC (DNS Security Extension), which is used for ensuring the authenticity of DNS zone files [26]. By generating proofs from DNSSEC and providing them on-chain, the authenticity of smart contracts is ensured. The downside of their approach is that the TLD, as well as the hoster, has to support DNSSEC. As DNSSEC is relatively new, it lacks significant adoption worldwide.

Other PKI-implementations: A series of approaches are trying to replicate public key infrastructures on blockchains, creating such systems from scratch. The key difference between these approaches and ours is that we include preexisting infrastructure in our approach, eliminating the need for bootstrapping. Out of brevity, we do not cover single approaches, but link to already existing research conducted in [27], which provides an overview of several implementations and their key differences.

VI. CONCLUSION AND FUTURE WORK

In this study, we introduce a novel approach to combat address replacement attacks. We use established forms of certificate authorities and signature schemes to allow for a secure binding between smart contract-based systems and the web. Compared with other approaches, our method does not rely on the user to verify the accuracy of the addresses. Furthermore, it does not depend on the adoption rate or the solution's network effect because TLS/SSL certificates and the cryptographic mechanisms are omnipresent and are currently in heavy usage on the internet. The enhanced threat model and moderate costs make the usage of TLS/SSL certificates reasonable in the context of blockchain.

We identify two primary directions of future work: 1) extending the model of TeSC and 2) proper integration of TeSC into wallets.

1) The proposed model of endorsements is highly flexible. However, we did not discuss a) further applications and b) additional dimensions in which we can use endorsements: Internal vs. external endorsements, information required for verification on-chain versus off-chain, and endorsement verification on-chain versus off-chain. In this study, we describe the internal storage of endorsements in smart contracts while verifying that these endorsements take place off-chain.

2) Currently, no wallet integrates TeSC. An integration into tools like MetaMask is necessary, as it allows users to access the features and help establish trust quickly. In this case, we need to focus on risks and potential warnings displayed to the user, as the user's danger varies depending on the on-chain and off-chain situation. In particular, mismatches between certificates, domain names, expiration of endorsements, and set flags need to be considered.

APPENDIX

A. TeSC Interface

The interface describes the functions that are exposed by an endorsed smart contract. It is important to understand that the implementation contains additional functions to set the variables. As an interface cannot define access restrictions for functions, we decided to not integrate these functions in the interface. They are however included in the reference implementation [3].

```
pragma solidity ^0.7.0;

interface ERCXXX /* is ERC165 */ {

    // @dev This emits when the Domain of the endorsement changes by any mechanism
    event DomainChanged(string domain);

    // @dev This emits when the Expriy Date of the endorsement changes by any mechanism
    event ExpiryChanged(uint64 expiry);

    // @dev This emits when the Flags of the endorsement changes by any mechanism
    event FlagsChanged(bytes24 flags);

    // @dev This emits when the Signature of the endorsement changes by any mechanism
    // Signature signed the claim {addr|domain|expiry|flags}
    event SignatureChanged(string signature);

    // @dev This emits when the certificate-fingerprint of the endorsement changes by any mechanism
    event FingerprintChanged(bytes32 fingerprint);

    // @dev This enum specifies whether an array-element was added or removed
    enum EventType {Add, Remove}

    // @dev This emits when an address was added or removed from the subendorsements
    event SubendorsementsChanged(address indexed contractAddr, EventType eventType);

    // @notice Returns the domain for the given Smart Contract
    // @dev The domain is a fully-qualified domain name (e.g. "domain.example.org")
    // @return The domain of the contract
    function getDomain() external view returns (string memory);

    // @notice Returns the expiry of the endorsement
    // @dev The timestamp is given in seconds since the epoch
    // @return The expiry of the contract
    function getExpiry() external view returns (uint64);

    // @notice Returns the flags as bytes24
    // @return The flags as bytes
    function getFlags() external view returns (bytes24);

    // @notice Returns the signature for the given Smart Contract
    // @dev The signature contains the claim {addr|domain|expiry|flags} and is signed with the private key from the certificate
    // @return The signature of the contract
    function getSignature() external view returns (string memory);

    // @notice Returns the sha256-fingerprint of the certificate that was used to sign the claim
    // @return The certificate fingerprint of the contract
    function getFingerprint() external view returns (bytes32);

    // @notice Returns subendorsements
    // @return The array of subendorsed contract addresses
    function getSubendorsements() external view returns (address[] memory);
}
```

Listing 1. On-Chain Smart Contract Interface in Solidity

B. TeSC Flags

In the smart contracts, we store flags in a `bytes24` variable. This allows us to store up to 192 flags, addressing them from f_0 to f_{191}. Each flag can be set either to *true* or *false*, resulting in $f_i \in \{0, 1\}$.

- f_0 `SANITY`: The sanity flag is always set to 1 to check if the flag variable is uninitialized or if all flags are actually set intentionally to 0.
- f_1 `DOMAIN_HASHED`: This flag is set if a domain is stored as a hash for privacy reasons. The hash is constructed as $h = hash(domain)$. This flag is set if an owner does not want the smart contract to be easily attributed to the domain by crawling the blockchain. We rely on `keccak256` as hash function.
- f_2 `ALLOW_SUBENDORSEMENT`: This flag is set if a smart contract is able to endorse further addresses such as contracts or externally owned accounts. The referenced smart contracts are stored in an array of the respective endorsing smart contract. If this flag is not set, the verification of the subendorsements fails.
- f_3 `EXCLUSIVE`: If this flag is set, one contract equipped with a valid endorsement can exist; if multiples exist, no contract is considered to be valid as long as the owner resolves the issues by either invalidating the endorsements, or removing the EXCLUSIVE flag from the contracts.
- f_4 `PAYABLE`: If a domain owner wants to allow users to send funds to the domain (owner), the owner sets this flag to let users know that this contract accepts funds.
- f_5 `ALLOW_SUBDOMAIN`: If this flag is set, smart contract addresses that are displayed in a subdomain context (the smart contract only being endorsed by the regular domain) can be verified. This is similar to a wildcard in TLS certificates and requires the certificate being issued for the respective domain.
- f_6 `TRUST_AFTER_EXPIRY`: Data that has been entered while the endorsement was valid can still be considered as valid after the endorsement or the TLS certificate expires. Because this information is time-stamped and no one can add or modify the data (without being noticed), the data is considered to be valid if this flag is set. This flag is especially useful for cases in which the blockchain is used to store data for public verification.
- f_7 `STRICT`: If this flag is set, the certificate returned via the web server must be identical to the certificate that signed the endorsement; otherwise, the verification fails.
- f_x `reserved`: All other flags are reserved.

REFERENCES

[1] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107:841–853, 2020.

[2] Ledger.com. Message by LEDGER's CEO – Update on the July data breach., 2020.

[3] https://github.com/ieeedappssubmission50/code. TeSC Source Code, 2021.

[4] Tesnim Abdellatif and Kei-Léo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.

[5] Reza M Parizi, Ali Dehghantanha, et al. Smart contract programming languages on blockchains: an empirical evaluation of usability and security. In *International Conference on Blockchain*, pages 75–91. Springer, 2018.

[6] Dr. Gavin Wood. Ethereum: a Secure Decentralized Generalized Transaction Ledger. 2017.

[7] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[8] Jen Wieczner. Ethereum: CoinDash ICO Hacked, $7 Million in Ether Stolen — Fortune, 2017.

[9] David Cooper, Stefan Santesson, S Farrell, Sharon Boeyen, Rusell Housley, and W Polk. RFC 5280: Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile. *IETF, May*, 2008.

[10] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. *ACM Queue*, 12(8):10–19, 2014.

[11] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on ethereum. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 93–112, Cham, 2019. Springer International Publishing.

[12] Eric Rescorla and Tim Dierks. Rfc 8446: The transport layer security (tls) protocol version 1.3. 2018.

[13] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. Automatic certificate management environment (acme). *Internet-Draft draft-ietf-acme-acme-09, IETF Secretariat*, 2017.

[14] Garret Grajek, Stephen Moore, and Mark Lambiase. Method and system for generating digital certificates and certificate signing requests, June 3 2010. US Patent App. 12/326,002.

[15] Joost De Kruijff and Hans Weigand. Understanding the blockchain using enterprise ontology. In *International Conference on Advanced Information Systems Engineering*, pages 29–43. Springer, 2017.

[16] Philipp Schmidt. Blockcerts—an open infrastructure for academic credentials on the blockchain. *MLLearning (24/10/2016)*, 2016.

[17] Nick Johnson, Virgil Griffith, and Chris Remus. Ethereum Name Service, 2017.

[18] Vitalik Buterin and Alex Van de Sande. Mixed-case checksum address encoding, 2016.

[19] johguse. Profanity: Vanity address generator for Ethereum, 2019.

[20] Erin Dachtler. Blockies: ¡1k library that generates blocky identicons, 2018.

[21] Austin Thomas Griffith. Vanity Blockie Miner for Ethereum, 2018.

[22] TaaS home. https://taas.fund/, 2020. Accessed: 2020-02-01.

[23] Paul Mockapetris and Kevin J Dunlap. *Development of the domain name system*, volume 18. ACM, 1988.

[24] Ethereum name service. https://ens.domains/, 2020. Accessed: 2020-03-01.

[25] Nick Johnson. ENS Root Change Will Allow Easy Integration of More Than 1300 DNS TLDs, 2019.

[26] Giuseppe Ateniese and Stefan Mangard. A new approach to dns security (dnssec). In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 86–95, 2001.

[27] Clemens Brunner., Fabian Knirsch., Andreas Unterweger., and Dominik Engel. A comparison of blockchain-based pki implementations. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 333–340. INSTICC, SciTePress, 2020.